Lightness

Low latency and high throughput dynamic network infrastructures for high performance datacentre interconnects

Small or medium-scale focused research project (STREP)
Co-funded by the European Commission within the Seventh Framework Programme
Project no. 318606
Strategic objective: Future Networks (ICT-2011.1.1)

Start date of project: November 1st, 2012 (36 months duration)

SEVENTH FRAMEWORK PROGRAMME

Deliverable D2.3

Simulation framework description and preliminary results

Due date:	31/10/14
Submission date:	15/11/14
Deliverable leader:	BSC
Author list:	Jose Carlos Sancho (BSC), Hugo Meyer (BSC), Yolanda Becerra (BSC), and Montse Farreras (BSC)

Dissemination Level

☒	**PU:**	Public
☐	**PP:**	Restricted to other programme participants (including the Commission Services)
☐	**RE:**	Restricted to a group specified by the consortium (including the Commission Services)
☐	**CO:**	Confidential, only for members of the consortium (including the Commission Services)

Abstract

One of the main objectives of WP2 is to provide key performance predictions of the proposed intra-DCs network architecture. This was the main focus during this second year in WP2. In order to achieve this it was necessary to build a simulation infrastructure that model the behaviour of the Lightness network. This deliverable describes the main components of this simulation infrastructure. It is based on a detailed flit-level simulator of electrical networks linked together with a very precise replaying tracing tool. The electrical simulator was based on the well-know InfiniBand network, but extended to support all the optical components of the Lightness network. These optical components were properly tuned to capture the important delays occurring in these optical devices such as TOR, OCS, and OPS. These delays were gathered through real experiments conducted in a preliminary test bed of the Lightness network.

In addition, in order to assess the impact to data centre workloads using our simulation framework it was also built during this second year a tracing tool specifically targeting to workloads that are being widely run in Data centres which are BigData applications. In this deliverable, it is described the tools that was needed in order to be able to trace these kind of applications.

Table of Contents

Figure Summary

Table Summary

0. Executive Summary

This deliverable describes the two important tools in order to evaluate the impact of the Lightness network on Data Centre workloads described as following,

- Lightness simulator: It is an accurate flit-level simulation framework that models both electrical and optical network components. This simulator is a complex simulation framework that is composed on two tools. First, it uses a replaying tool called Dimemas that allows us to replay in very detail both the computation and communication events of High Performance Computing Applications. And second, this replaying tool is connected to our Lightness simulator. This is powerful tool that enable us to quickly assess the impact of the Lightness network and also it will allow us to identify potential aspects of this network that could be improved. The main optical components that have been built during the second year are the TOR, OPS, and OCS. Additionally, it is planned that additional components will be also included during the third year such as the AoD and the optical network interface. A validation of the optical components was also conducted and results are shown in this deliverable. In addition, it is provided a comparison with their electronic counterparts such as the InfiniBand network using simple microbenchmarks. For this purposes, traces of these microbenchmarks were obtained using the Marenostrum supercomputing at BSC.
- A tracing tool for BigData: It is described a tracing tool in order to collect the main computation and communication events of BigData applications running on top of Hadoop. It is shown a general overview of the processes and types of communication involved in this kind of applications. And also, it was described the monitoring infrastructure that we have implemented and deployed in a small data centre. This infrastructure is based on three main components. First, the Hadoop internal modifications in order to basically identify process that sends and process communications. Second, the *sniffer* that extracts the necessary information from all the network packets leaving or arriving at each node of the Hadoop application. And third, the *Dumpingd* tool that associates communications and processes in Hadoop.

1. Introduction

In this chapter we introduce the simulation framework, which consists of a new simulator developed within WP2 of the project, but makes use of existing tools as well.

The general context in which the simulation framework can be employed is shown in Figure 1.

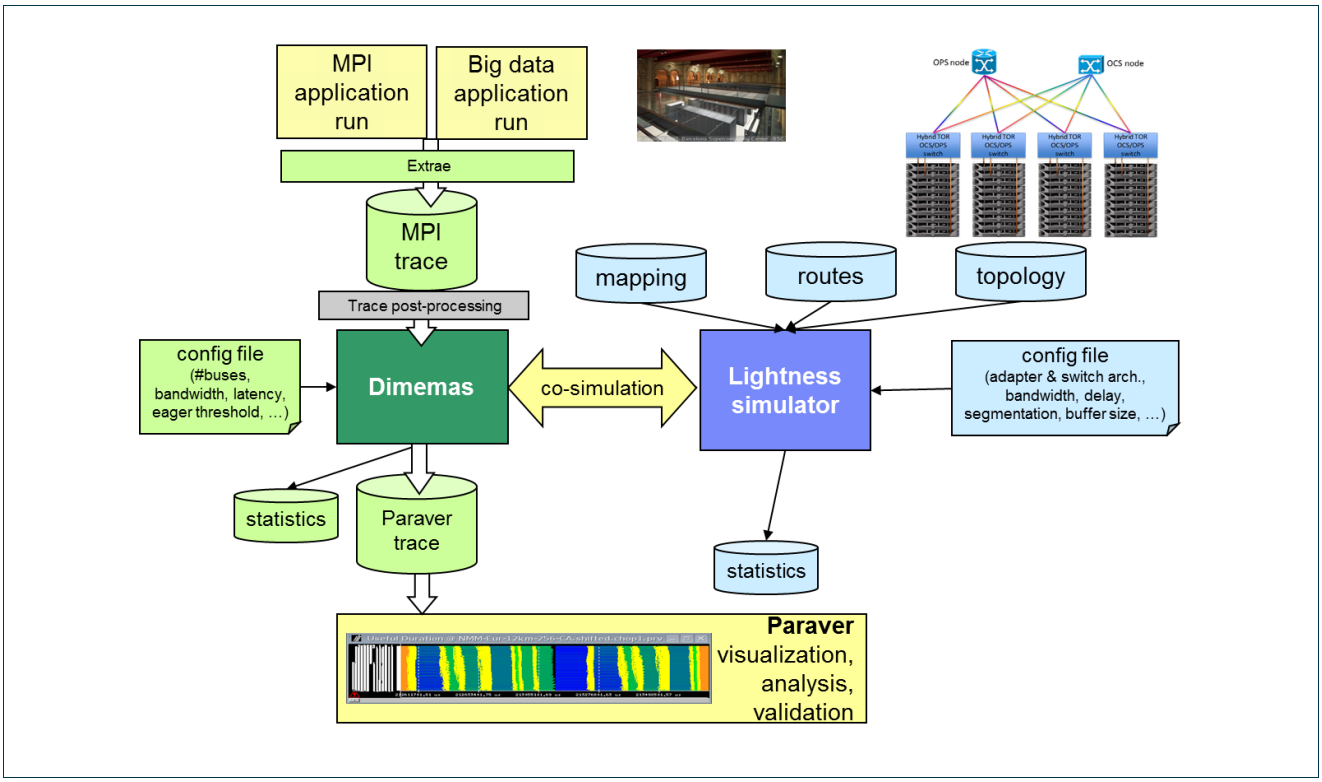

Figure 1: General context for the use of the simulation framework

Extrae [Ext] is a package developed at BSC, which can instrument applications based on MPI, OpenMP, pthreads, CUDA, etc. The information gathered by Extrae typically includes timestamps of events of runtime calls, performance counters and source code references. Additionally, Extrae provides its own API to allow the user to manually instrument the application of interest. In the case of our simulation framework, apart from tracing applications from the HPC domain, the Extrae API has been used to trace BigData applications as well.

Dimemas [Dim] is a performance analysis tool for message-passing programs, developed at BSC. The main modelling concepts and configuration files for the tool have been described with more details in deliverable

D2.2 [del-d22] of the project. Dimemas can replay traces collected by Extrae and perform prediction studies and “what-if” analysis for various system architectures, specified by the user through configuration files.

Paraver [Prv] is a visualization and analysis tool, developed at BSC as well. Paraver is very flexible and can be easily extended to support new performance data or new programming models, without changes to the visualizer. The tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines, which allows displaying a huge number of metrics with the available data.

The **LIGHTNESS simulator** is based on the OMNEST framework and has been developed within WP2 simulation activities. This simulator models the network architecture proposed by the LIGHTNESS project. More information on the implementation of the simulator is presented in Chapter 2.

In the context shown above, Extrae is used to instrument HPC applications based on the MPI standard and BigData applications. The resulting traces can be further processed with Dimemas, in order to identify performance issues. However, for network communications Dimemas uses a linear performance model, and although some non-linear effects such as network conflicts are taken into account, this approach may be too simplistic. Therefore, Dimemas has been integrated with the LIGHTNESS simulator, which models network functions accurately.

2. LIGHTNESS simulator

This simulator models the main components of the network architecture proposed within the LIGHTNESS project, namely Top-of-Rack (ToR) Switch, Optical Packet Switch (OPS) and Optical Circuit Switch (OCS). It has been integrated with Dimemas in order to conduct studies on performance prediction and to use realistic traffic from applications in the HPC and BigData domains. Additionally, it has been integrated with the InfiniBand simulator [IBSim], since InfiniBand is the most popular interconnect technology for HPC systems (more than 40% of the Top500 supercomputers use it – November 2013 list [IBTop500]).

This chapter is organized as follows. In Section 2.1, we give an overview of the OMNEST framework, on which the LIGHTNESS simulator is based, focusing on the aspects relevant to the LIGHTNESS simulator. We present the simulator implementation in Section 2.2 and helper tools to run simulations in Section 2.3. Section 2.4 illustrates a small example of how to run a simulation and of the workflow between Dimemas and the LIGHTNESS simulator.

2.1. OMNEST framework

OMNEST [Omnest] is the commercial version of OMNeT++ [Omnet], a framework for discrete-event network simulation, which is popular in the community for building network simulators. This framework provides the basic tools to write simulations, as follows:

- a C++ class library which consists of the simulation kernel and utility classes (for example, for random number generation, statistics collection, topology discovery, etc);
- the infrastructure to assemble simulations (NED language for describing network components) and configure them, through the initialization files;
- runtime user interfaces or environments for simulations (Tkenv, Cmdenv);
- an Eclipse-based simulation IDE for designing, running and evaluating simulations;
- extension interfaces for real-time simulation, etc.

2.1.1. OMNEST models

The OMNEST models are based on **simple modules**, written in C++, using the simulation class library. These modules can have several parameters in order to specify their behavior and they implement the algorithms to define the model's operation. Simple modules can be further grouped into **compound modules**, as shown in Figure 2. This approach enables the user to reuse and combine components in a flexible manner. The input and output interfaces of modules are represented by **gates**. An input gate and output gate can be linked by a **connection**. Connections can be characterized by parameters such as: propagation delay, data rate, bit error rate, packet error rate, etc.

The modules in a simulation can communicate with each other through **messages**. These are sent via gates, or directly to destination modules and can represent frames or packets in a computer network, jobs or customers in a queuing network, or other types of mobile entities. Messages can contain arbitrarily complex data structures, defined by the user, such that to support the desired operation for the model.

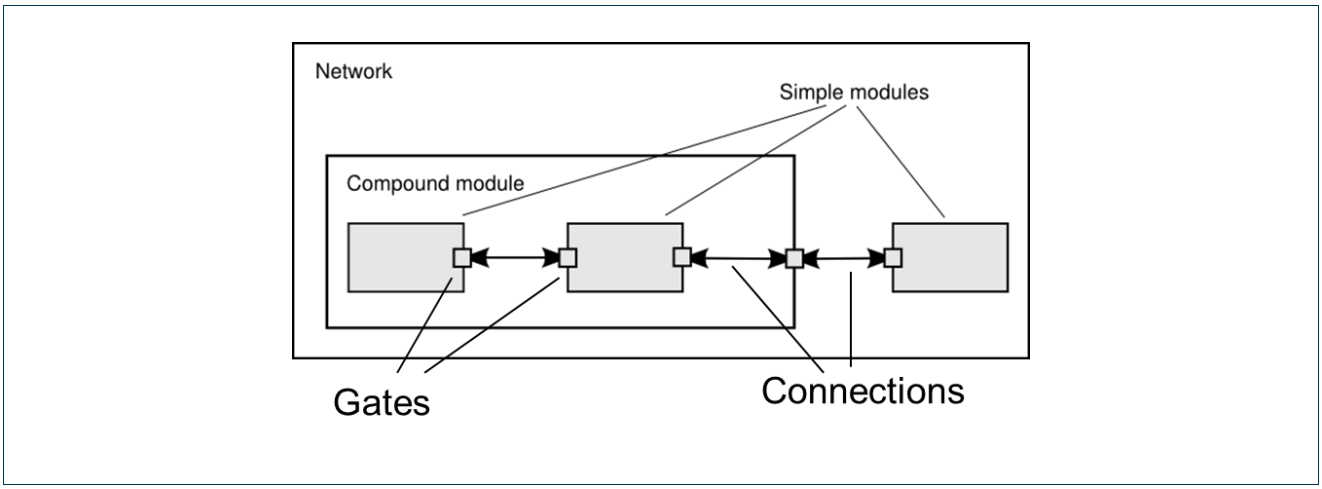

Figure 2: OMNEST modelling concepts

An OMNEST model consists of the following parts:

- NED language topology description(s) (.ned files) that describe the module structure with parameters, gates, etc. NED files can be written using either the provided IDE or any other text editor.
- Message definitions (.msg files). Users can define various message types and add data fields to them. OMNEST will translate message definitions into full-fledged C++ classes.
- Simple module sources. They are C++ files, with .h/.cc suffix.

2.1.2. Simulation with OMNEST

In order to develop a simulator in OMNEST, the following steps should be carried on, as illustrated in Figure 3:

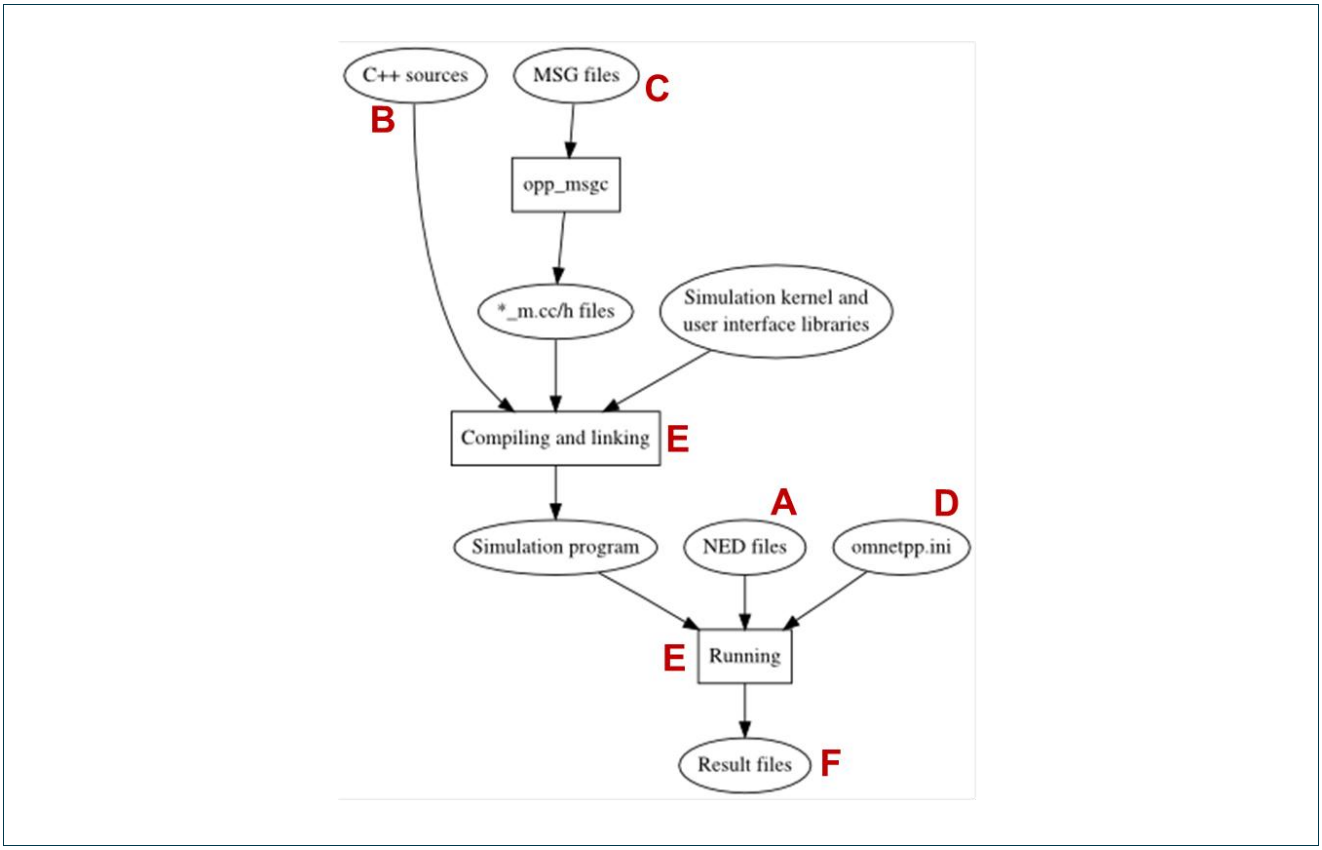

Figure 3: OMNEST workflow

A. Define the model structure in the NED language. NED files are loaded dynamically in their original text forms when the simulation program is ran.

B. Implement the behaviour of the simple modules in C++, using the simulation kernel and class library.

C. Define message types and message structure (.msg files). At compilation, these files are automatically translated into C++ code using the opp_msgc program.

D. Provide configuration parameters for the model (omnetpp.ini file).

E. Build the simulation program and run it. At this step, all C++ sources are compiled and linked with the simulation kernel and a user interface library to form a simulation executable or shared library.

F. Simulation results are written into output vector and output scalar files, which can further be processed with R, Matlab or other tools.

In general, when implementing module behaviour (Step B from above), the user should define the following functions for each simple module:

- ***initialize()***: this function performs all initialization tasks, such as read module parameters, initialize class variables, allocate dynamic data structures with new. The user should also allocate and initialize self-messages (timers) if needed in this function.

- ***handleMessage(cMessage *msg)***: this function should hold the code to process the messages arriving at the module. The function will be called for every message and return immediately after processing it. No simulation time elapses within a call to ***handleMessage()***.
- ***finish()***: this function is for recording statistics, and it only gets called when the simulation has terminated normally. It does not get called when the simulation stops with an error message.

As a discrete event simulation, OMNEST maintains the set of future events in a data structure called FES (Future Event Set), ordered by timestamps. A simulation run starts by the initialization step, when the network model is built and the initial events are inserted into the FES; this means that the ***initialize()*** function of each module in the network is called. Next, as long as there are events in the FES or until a specified simulation time limit is reached, the first event is extracted and processed. Processing of an event involves a call to the code implemented by the user (***handleMessage()*** function), which may add new events or remove existing events from the FES.

2.2. Implementation details

The LIGHTNESS simulator and Dimemas do not run in parallel, but they take turns when running a simulation. The interaction between the two tools is based on the Null Message Algorithm [Chandy], which exploits knowledge of the time when events should occur. In particular, Dimemas informs the LIGHTNESS simulator of the timestamp of the next event in its queue, thus allowing the LIGHTNESS simulator to execute the events from the FES and advance its simulation time until that timestamp.

2.2.1. Dimemas interface

Dimemas already provides a socket-based interface which we used for the interaction with our simulator. This interface should be enabled when configuring Dimemas for installation. Then, when replaying a trace, the computation events are handled within Dimemas, while communication events are sent in human-readable form to the simulator, using the sockets.

The commands sent by Dimemas to the LIGHTNESS simulator are as following:

- *SEND timestamp src dest tag size (other parameters)*

  This command represents a message sent at the specified timestamp, from a source node to a destination node. The size parameter is given in bytes and the tag represents the MPI tag of the message. Other parameters are included as well, however these should be echoed back to Dimemas.

- *STOP timestamp*

  This command informs the simulator of the timestamp when the next event occurs in Dimemas. It represents the “null” message used for synchronization between Dimemas and the LIGHTNESS simulator.

- *END*

  It represents the end of the current batch of commands sent from Dimemas to the simulator.

- *PROTO_OK_TO_SEND, PROTO_READY_TO_RECV*

  These two commands are used to implement the rendez-vous control messages for MPI.

- *FINISH*

  This command informs the simulator that Dimemas finished replaying the trace and therefore no more communication events will be sent.

This set can be extended, however the above commands were enough to fit the needs for the current version of the LIGHTNESS simulator.

2.2.2. OMNEST modules

In order to model the LIGHTNESS network architecture, we implemented several compound modules in OMNEST, such as Top-of-Rack (ToR) Switch, Optical Packet Switch (OPS) and Optical Circuit Switch (OCS). Two modules, Socket Scheduler and Dispatcher, provide the functionality needed for integrating the LIGHTNESS simulator with Dimemas. In Figure 4, we show these modules and the interactions between them.

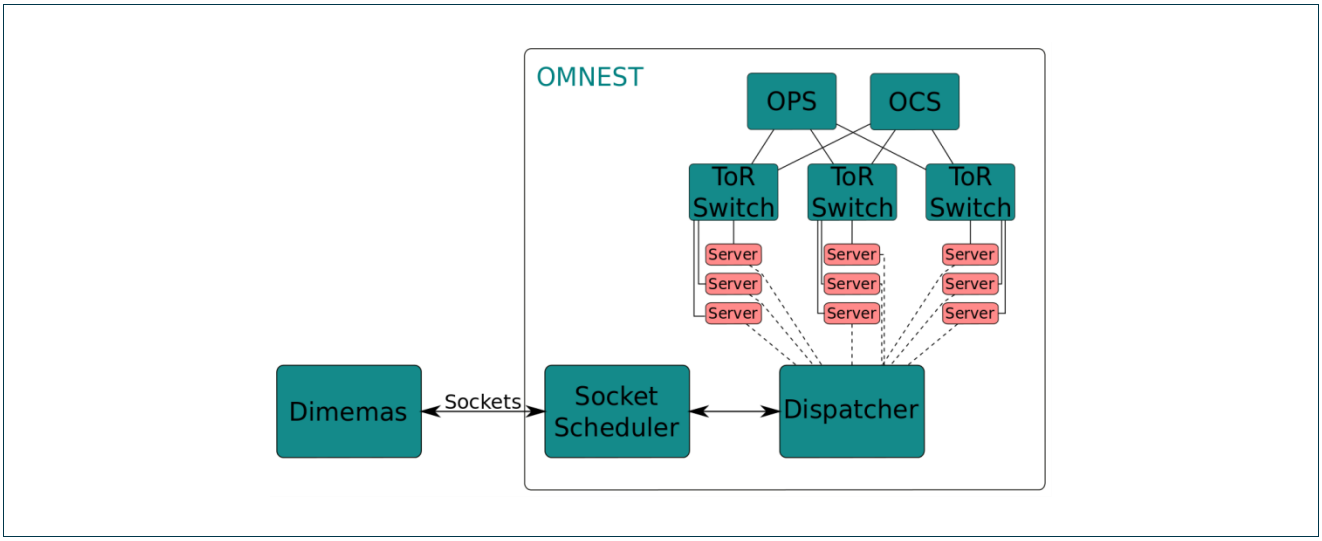

Figure 4: Overview of the LIGHTNESS simulator

2.2.2.1. *Socket Scheduler*

By default OMNEST provides a sequential scheduler which extracts events in the FES and handles them in increasing order of their timestamps. However, in our case, we needed a custom scheduler to take into account events received from Dimemas. Thus we implemented Socket Scheduler, the module which handles both events generated by OMNEST modules and events received through the socket. The main functionality of the Socket Scheduler is:

- open/close socket connections;
- send and receive data through the sockets;
- find next event to be executed from the FES.

Before extracting the next event in the FES, the scheduler first checks if any message was received on the socket, and if so, it schedules an event to process that message immediately.

2.2.2.2. *Dispatcher*

This module receives data from the socket, processes it and handles replies sent back to Dimemas. The Dispatcher processes the commands received from Dimemas in batches; that means it only handles them when an "END" command is received. In particular, for each command it takes the following actions:

- END: process current batch.
- STOP: schedule a stop event at the specified timestamp. When this stop event arrives, send back to Dimemas the message: "STOP timestamp", such that Dimemas can continue executing events from its queue.
- SEND: instruct the corresponding source node in the network to schedule a message to be sent, as specified by the received information (source and destination tasks/nodes, message size and timestamp).
- PROTO_OK_TO_SEND: if the corresponding PROTO_READY_TO_RECV command has been received, schedule a message to be sent, as above. Otherwise, store this command.
- PROTO_READY_TO_RECV: schedule a message to be sent and erase the corresponding PROTO_OK_TO_SEND command.
- FINISH: no action. The Socket Scheduler then continues with executing the remaining OMNEST events from the FES, until the simulation time is over or no more events are present in the FES.

For the messages sent in the network, by default we assume that each task from the Dimemas trace corresponds to the node with the same index in the LIGHTNESS simulator. For example, task j in Dimemas is node j in the simulated network. However, it is possible to specify a different mapping for the simulator, through a text file which is read as part of the initialization step of the Dispatcher module. This file should be given in the .ini file, by setting parameters *taskMapping* and *taskFilename*, and for each task it should contain a line with the task id and the node to which it is mapped, separated by space (" ").

The replies sent by the LIGHTNESS simulator to Dimemas are as follows:

- STOP timestamp: sent at the arrival of a stop event, scheduled previously.
- COMPLETED SEND nTimestamp src dest size (other parameters): sent when a message transmitted in the simulated network arrived at the destination node. nTimestamp represents the timestamp when the destination node received the message.
- END: sent after each of the above replies, to indicate the end of the batch.

In order to support the communication between the above modules, we defined the following message type:

- DimemasPkt: it contains the following fields: type, timestamp, src, dest, size, srcApp, destApp, seqNo. The type can be "DimCommSend" for Dimemas SEND commands or "DimReply" for replies to be sent back to Dimemas. The seqNo represents a global identifier for the packets, which is modified by the Dispatcher.

2.2.2.3. *TOR Switch*

This module handles to the OPS and OCS from the servers in a rack. It is implemented as a compound module, with its functionality enclosed in various submodules. Figure 5 shows the different modules and their connectivity between them. As you can see, the connectivity among the servers in a rack can be implemented using an IBA switch. Intra-rack communication is purely performed using electronic packets. However, when packets are addressed to a server located in another rack then the IBA switch forwards these packets to the TOR that performs the electronic to optical conversion and forwards the optical packets to the optical network.

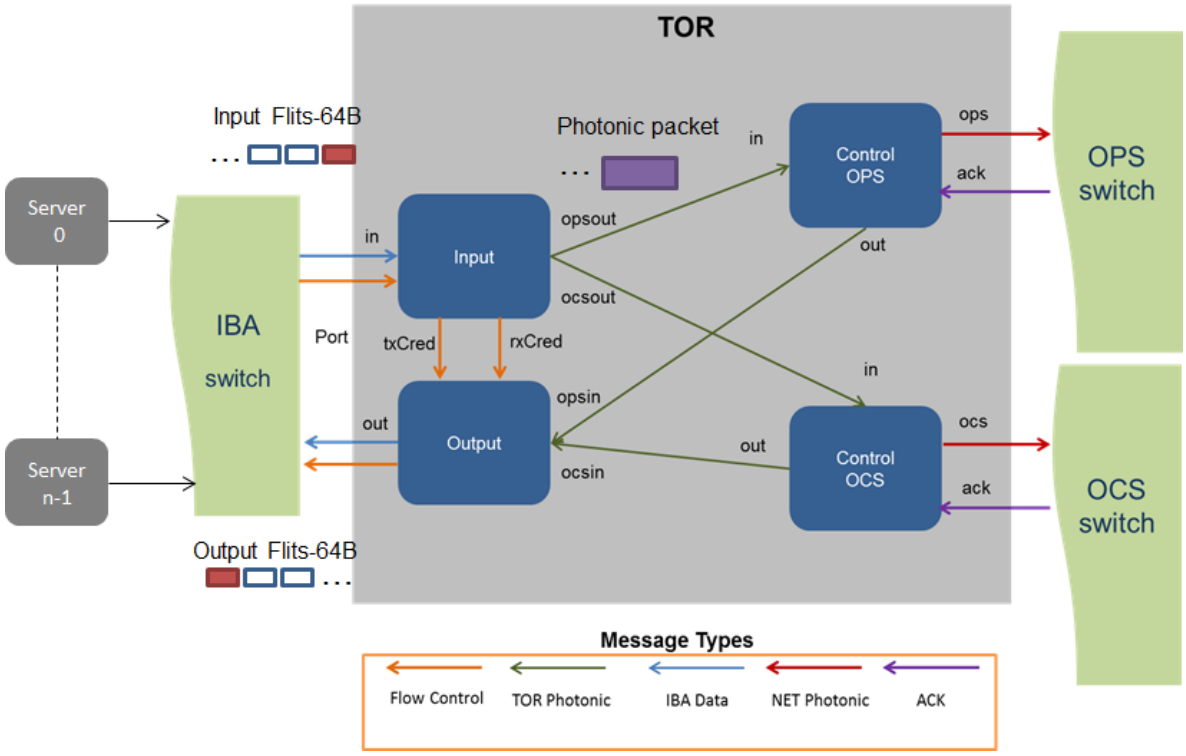

Figure 5: TOR functional modules and their connectivity among themselves, the electronic, and optical networks

The TOR is implemented using the following submodules:

- *Input:* It is directly connected to the IBA switches receiving both flow control and data packets. It creates the Photonic packets from the data packets and forwards those packets to either the Control OPS or Control OCS submodules. A decision algorithm is implemented here that decides which submodule to forward packets. It is assumed no delay for the conversion from electrical to optical. The size of Photonic packets could be as minimum as one flit or as maximum as the whole application message. In case that the Photonic packet size is larger than a flit then this module has to wait until receiving all the flits before forwarding the packet to the optical domain. This waiting time could add too much delay if the message is too large.
- *Output:* This submodule is receiving packets from the optical network from and converts them to IBA packets. It is assumed no delay for the conversion from optical to electrical.

- *Control OPS:* It manages all the communication between the TOR and OPS. The TOR optical packets are translated to NET photonic packets which are delivered to the OPS through the *ops* port. It handles all the retransmissions for packets that have been dropped in the OPS. There is an internal buffer that stores all the photonic packets that have been sent through the optical network just in case they need to be retransmitted. In the *ack* port, it is receiving ACK and NACK optical packets from the TOR. When it receives an ACK then it deletes the corresponding packet in the buffer. On the other case, when it receives a NACK it will retransmit the dropped packet when the channel will become available. All the packets are retransmitted in the arrival order.
- *Control OCS:* It manages all the communication to the OCS. Before a packet is forwarded to the OCS it has to check if there is a connection already established in the OCS for the packet's destination. In case the connection exists it directly forwards the packet. Otherwise, it has to inform to the control plane in order to set up this connection. The control plane configures the OCS if the connection is currently available and notifies the Control OCS that connection succeeds or not. In the simulator model we are assuming different fixed delays to model all of these communications with the control plane and the actions of the control plane as well.

These are the messages that have been defined to handle the optical communication:

- TOR Photonic: This is the packet created inside the TOR to communicate between the submodules within TOR.
- NET Photonic: This packet is the one that it is forwarded outside the TOR to either OCS or OPS.
- ACK: These are the control packets coming from either the OCS or OPS to inform that the packet or connection in the case of OCS succeed or not (ACK or NACK).

These are the parameters in order to configure the TOR in the Input submodule:

- maxVL: Number of maximum virtual lanes used in InfiniBand swiches.
- Width: The number of InfiniBand channels per link.
- maxStatic: The number of credits available for each virtual link.

These are the parameters in order to configure the TOR in the Ouput submodule:

- AckDelay: The time corresponding to process the ACK/NACK packet received from either OCS or OPS.
- TORMaxMgs: The size of the output buffer.
- flitSize: The size of an InfiniBand flit.
- maxVL: Number of maximum virtual lanes used in InfiniBand swiches.
- credMinTime: Time between VL update and injection of an update in InfiniBand.
- flit2FlitGap: Extra delay between flits.
- pkt2PktGap: Extra delay between packets.

These are the parameters in order to configure the TOR in the Control OPS submodule:

- TORBW: The bandwidth at the output port that connects to the OPS.
- TORMaxMgs: The size of the output buffers that holds packets for retransmission if needed.
- CableDelay: The time of propagation of the signal through the cable. Assuming a 5m/s for the speed of the lasers in each wavelength.
- TorStartupDelay: The time spent by this component in order to prepare the emission of a new packet.

- TorPer3ByteDelay: The transmission time of 3 bytes to the OPS.

These are the parameters in order to configure the TOR in the Control OCS submodule:

- TORBW: The bandwidth at the output port that connects to the OCS.
- TORMaxMgs: The size of the output buffers that holds packets for retransmission if needed.
- CableDelay: The time of propagation of the signal through the cable. Assuming a 5m/s for the speed of the lasers in each wavelength.
- maxDstLids: The maximum number of destinations in the network.
- retryReqDelay: The time to re-transmit a request of connection from the OCS.

2.2.2.4. *OPS*

This module models the behaviour of the OPS switch. According to the functionality of the OPS packets may be dropped in this switch in the following cases: (1) the output port of the OPS is already being used by another packet transmission, (2) there are various packets coming from the same fiber that wants to go to the same output port, then only one packet can go through. Once the decision process is done, the OPS sends back to the corresponding TOR an ACK or NACK packet to notify the TOR that the packet has been successfully or not forwarded. This module is implemented as a compound module, with its functionality enclosed in various submodules. Figure 6 shows the different modules and their connectivity between them. It is assumed that *m* fibers are connected to the OPS and each fiber contains *n* wavelengths. Therefore, the OPS has a total *m x n* ports.

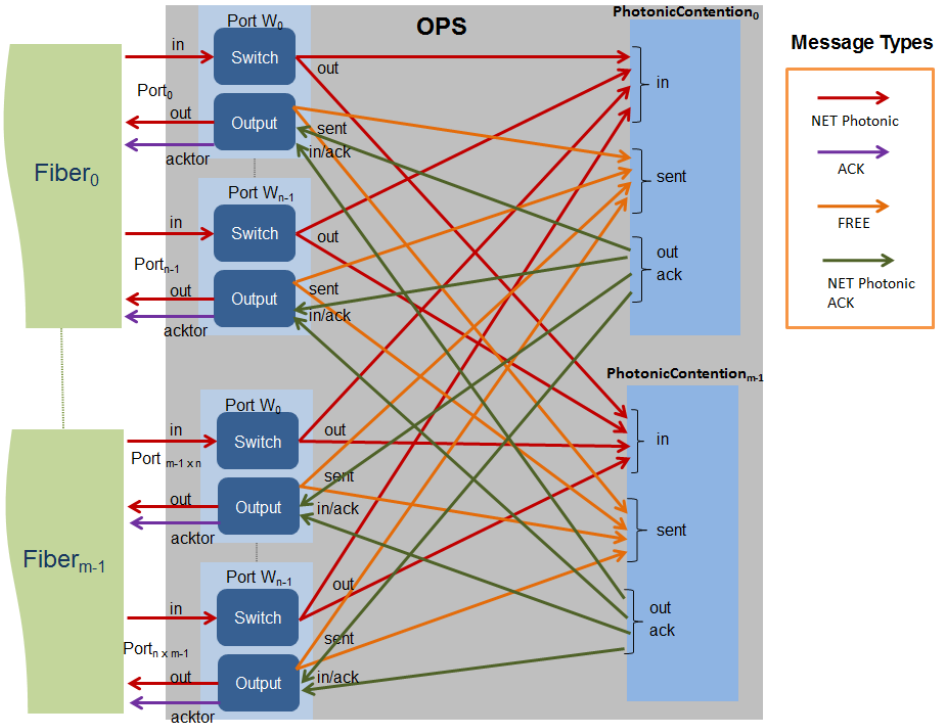

Figure 6: OPS functional modules and their connectivity among themselves considering m fibers and n wavelengths per fiber

The OPS is implemented using the following submodules:

- *Switch:* It receives Photonic packets directly from the TOR switches. It does the routing decision and forward packets to their corresponding Photonic Contention module. There is a delay corresponding to the routing processing.
- *Output:* It delivers Photonic packets to the TOR. Packets can be either photonic packets that transports data or the ACK/NACK control packets trough the output *acktor* port. The last packets are received from the Photonic Contention once it processes all the requests. In addition, it notifies back the Photonic Contention through the *sent* port that the corresponding output port is available after a transmission is finished.
- *Photonic Contention:* It decides which packet is finally delivered to the requested output port based on the availability of the output port and fiber that the packets were coming from. It sends ACK/NACK notifications to the corresponding output ports. In order to model multiple simultaneous requests to the same output port the Protonic Contention process requests only every one nanosecond. This is not delaying significantly any packet, but allows us to arbitrate well multiple requests. Sending the ACK/NACK packets has no delay. This delay is already included in the delay corresponding to the routing processing in the Switch.

These are the messages that have been defined to handle the optical communication in the OPS:

- ACK: Control packets that notifies the success or not of the packet transmission (ACK or NACK).
- NET Photonic: This packet is the one that it is received or sent from/to the TOR.
- FREE: These are internal control packets coming from Output to the Photonic Contention to notify the corresponding output port is available again.

These are the parameters in order to configure the OPS in the Switch submodule:

- numFibers: Number of fibers that connect to the OPS ports.
- numWavelengths: Number of wavelengths per fiber.
- OPSDelay: Time to route a photonic packet.

These are the parameters in order to configure the OPS in the Ouput submodule:

- AckDelay: Time to create an ACK or NACK packet.
- CableDelay: The time of propagation of the signal through the cable. Assuming a 5m/s for the speed of the lasers in each wavelength.
- OPSBW: Bandwidth available to transmit a packet.

These are the parameters in order to configure the OPS in the Photonic Contention submodule:

- numFibers: Number of fibers that connect to the OPS ports.
- numWavelengths: Number of wavelengths per fiber.
- maxDstLids: The maximum number of destinations in the network.

2.2.2.5. *OCS*

This module models the behaviour of the OCS switch. A connection has to be established before between the input and output ports before it can forward packets to the output port. This connection setup is performed in reality by the control plane upon receiving a request from the TOR. For the sake of simplicity, the part of the control plane that manages the OCS is performed in the OCS rather than using a separate module. Here, the TOR contacts directly to the OCS to request connections as it was contacting directly to the control plane. All the corresponding delays are modelled correctly in our simulator assuming that we are contacting directly the control plane. The architecture of the OCS is fully distributed unlike the OPS which is centralized per each fiber. In the OCS there is no this contention from multiple packets coming from the same fiber and thus the contention resolution can be performed individually per each output port. Figure 7 shows the functional modules and their connectivity of the OCS.

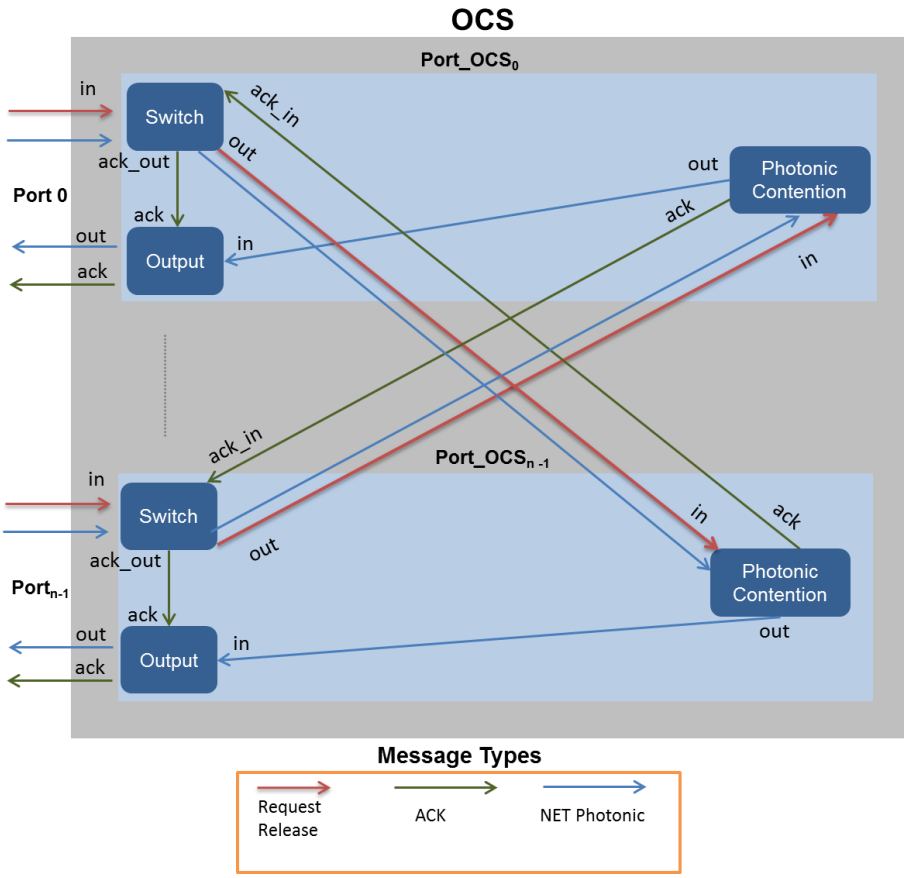

Figure 7: OCS functional modules and their connectivity among themselves

The OCS is implemented using the following submodules:

- Switch: It receives Photonic packets directly from the TOR switches. It does the routing decision and forward packets to their corresponding Photonic Contention module that has been already set up or sends the request to set up to the Photonic Contention module.

- Output: It delivers Photonic packets to the TOR. In addition, it informs the TOR that the connection has been successfully or not set up through the *ack* port. Sending the ACK/NACK packets has no delay. This delay is already included in the delay corresponding to the routing processing in the Switch.
- Photonic Contention: It checks and set up the connection between the requested input and output ports. It sends back an ACK or NACK to the input in order to notify that the requested connection succeeded or not. Connections are granted when the output port is available. A round-robbin fashion arbitration is used when multiple simultaneously connections are received. There is a processing delay in this module to process connection requests and establishes new connections. Also, for already established connections it delivers the data from input to the output ports. There is no delay for transfering data once a connection has been established.

These are the messages that have been defined to handle the optical communication in the OPS:

- ACK: Control packets that notifies the success or not of a connection request (ACK or NACK).
- NET Photonic: This packet is the one that it is received or sent from/to the TOR.
- Request: Request a connection from an input port to an output port.
- Release: Release a connection that was previously successfully established by the Request.

These are the parameters in order to configure the OCS in the Switch submodule:

- numWavelengths: Number of wavelengths per fiber.
- OCSDelay: Time to transfer a photonic packet through the OCS.

These are the parameters in order to configure the OCS in the Output submodule:

- AckDelay: Time to create an ACK or NACK packet.
- OCSBW: The bandwidth for the output ports.
- OCSOutputSize: The size of the buffer in the output port.
- CableDelay: The time of propagation of the signal through the cable. Assuming a 5m/s for the speed of the laser wavelength.
- OCSFirstMessageDelay: Time spent in setting up the channels before transmitting the first message.

These are the parameters in order to configure the OCS in the Photonic Contention submodule:

- numFibers: Number of fibers that connect to the OCS ports.
- numWavelengths: Number of wavelengths per fiber.

2.2.3. Integration with the InfiniBand simulator

InfiniBand is the most popular technology for interconnect networks for HPC systems. In the Top500 list from November 2013, there are more than 40% of supercomputers which rely on it for fast and reliable communications [IBTop500]. The MareNostrum supercomputer at BSC uses it as well; therefore we integrated the InfiniBand simulator [IBSim] with the LIGHTNESS simulator. Modelling the MareNostrum supercomputer with the InfiniBand simulator represents a benchmark for the LIGHTNESS network architecture, as well.

The InfiniBand simulator has been made available to the community as open source and has been updated in July 2013. In order to be able to use the traffic information from Dimemas, we made several modifications to the InfiniBand code, mainly related to the following modules:

- Allow the application to send messages as specified in Dimemas SEND and READY_TO_RECV commands.
- Update the sink module to inform the Dispatcher that a message has been received.
- Update the input buffer module to allow the Opt2IB module to generate flits at the correct credit rate.

These changes were made by deriving from the corresponding InfiniBand objects and implementing needed behaviour in the derived classes. Next, the InfiniBand model has been compiled as a shared library and linked with our simulator.

2.3. Helper tools

For the simulation studies to be performed we will use networks with a large number of nodes and writing the input files for each setup can be tedious. To make the task of preparing simulation setup easier, we provide several scripts which take some parameters as input and generate the needed files. Additionally, we provide an R script to process result files and plot some metrics, once a simulation is completed.

There are two scripts to generate the input files:

- generateTopoNED: generates the NED file containing the network description (i.e. OMNEST modules and the connections among them). The topology of the generated network is considered to be full fat-tree, as is the case of the MareNostrum supercomputer.
- generateRoutingInfo: generates the file containing the routing information (which port each switch should use in order to reach a certain destination). The file contains one line per switch, with the port numbers ordered by destination id, and separated by space (“ “).

Both of these scripts take as parameters the number of nodes in the network (n), the number of racks (r) and the number of core switches (c). Note that we assume all racks hold the same number of nodes, thus the scripts return an error if n is not a multiple of r.

OMNEST provides an R package which can be used to process simulation results. The package is available at [Rpkg] and it supports loading the contents of OMNEST result files into R, organizing the data and creating various plots and charts. We used this package in an R script to plot the metrics of interest.

2.4. Dimemas – LIGHTNESS example simulation

In order to run a simulation with the two tools, Dimemas and the LIGHTNESS simulator, the following commands must be executed in a terminal:

1. Start the LIGHTNESS simulator:

Path/to/simulator/dimlight -u Cmdenv -r 0 -c omnCfg -n nedPath -l path/to/ib/src/ib_flit_sim omnetpp.ini &

The flags used in the above command have the following meaning:

- -u Cmdenv: run the simulation in command line. The other option is Tkenv, with the graphical interface, which is much slower.
- -r 0: specifies the id of the run.
- -c omnCfg: specifies which configuration to be run. Several configurations can be included in the initialization file (omnetpp.ini), however just one of them can be used for a simulation run.
- -n nedPath: specifies the location of the NED files holding the definitions of the objects used in the simulation.
- -l path/to/ib/src/ib_flit_sim: links with the InfiniBand shared library
- omnetpp.ini: this is the initialization file for the simulation run, which contains configuration parameters for the model.

2. sleep 20s

3. Start Dimemas:

   Path/to/Dimemas/Dimemas -pa prvTrace -venusconn 127.0.0.1:4242 dimemasCfg

   The options in the above command have the following meaning:

   - -pa $prvTrace: the name of the Paraver trace file to be generated.
   - -venusconn 127.0.0.1:4242: this option informs Dimemas about which server address and port number to use for the socket communications.
   - dimemasCfg: this file holds the definition of the target architecture for which the Dimemas prediction is run.

When the first command is executed, the network model is built and the initial events are inserted into the FES in the LIGHTNESS simulator, which then listens for incoming connections. When Dimemas is run, it creates the socket to connect to the LIGHTNESS simulator and starts replaying an MPI trace, as specified through the configuration file (***dimemasCfg***). The computation events from the trace are executed by Dimemas, while the communication events are sent to the LIGHTNESS simulator.

An example of processing a communication event is shown in Figure 8. As explained previously, Dimemas sends its commands to the LIGHTNESS simulator in human-readable form, through the socket. On the LIGHTNESS simulator side, the Socket Scheduler observes that a message arrived on the socket and then schedules an event at the Dispatcher to process that message. In the illustrated case, this processing means that the Dispatcher informs Node 2 that it should send a message with size 512 bytes to Node 1, when simulation time reaches 100ns. At the specified timestamp, Node 2 sends the message as instructed. Next, when the transmission through the simulated network finishes (in our case, at 500 ns), Node 1 informs the Dispatcher that it received the message from Node 2, which further sends a reply to Dimemas, containing the

timestamp when the message arrived at destination. Then Dimemas updates the communication event with the new timestamp and continues simulating the events in its queue.

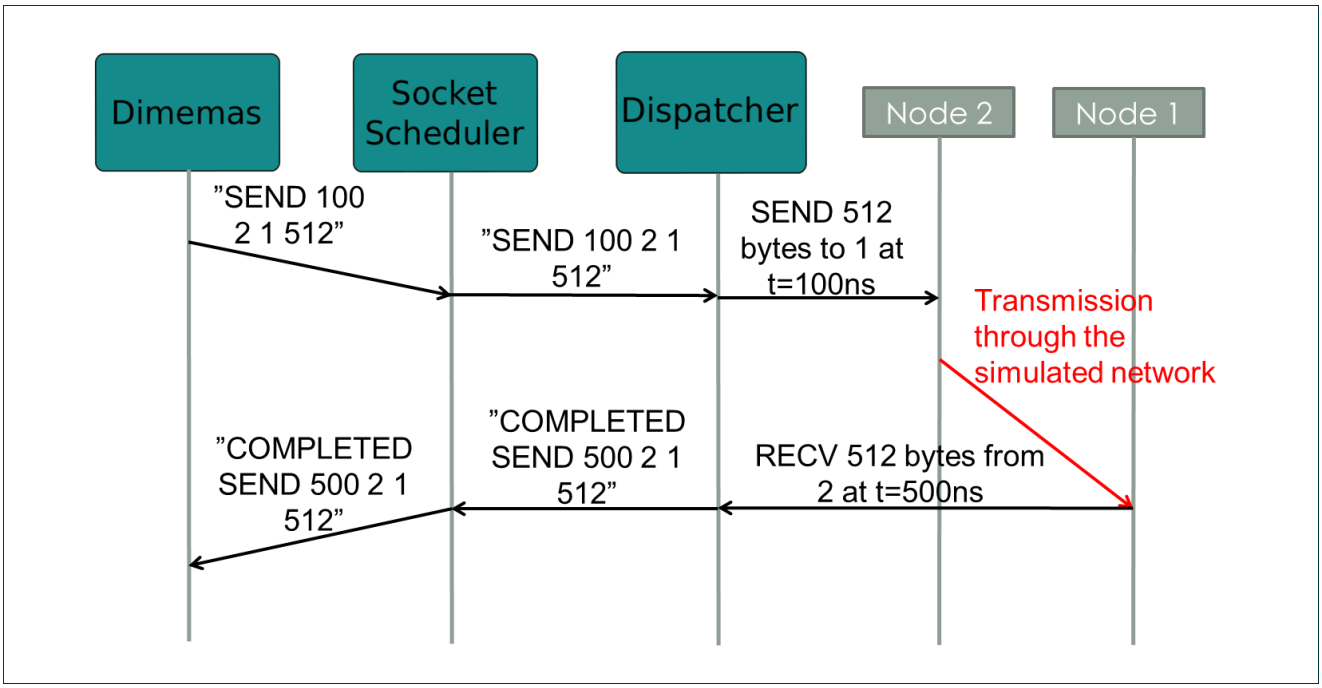

Figure 8: Dimemas – LIGHTNESS simulator workflow

3. Tracing BigData Applications

This chapter is organized as follows. In Section **Errore. L'origine riferimento non è stata trovata.** we show a general overview of the processes and types of communication involved in applications running on apache hadoop. In Section 3.2 we describe the collected tracing data that will feed the simulator. Finally, in Section 3.3 we describe the monitoring infrastructure that we have implemented and deployed to get tracing data that characterized communication in hadoop.

3.1. Hadoop processes and communication

Apache Hadoop is a software framework to support the execution of MapReduce applications, which includes the implementation of a distributed file system, HDFS, to support data management. It follows master-slave architecture both for controlling data processing and for implementing HDFS. Figure 9 shows the processes the form Hadoop infrastructure as well as the type of messages that those processes send to each other. The processes that compose a Hadoop cluster are the following:

- JobTracker (JT): this is the master process that controls the execution of data processing. There is one JT per cluster and it is responsible for receiving new processes submission and for assigning work to data processing slaves.
- TaskTrackers (TT): these are slave processes that perform computation on each piece of data. They can perform both map tasks and reduce tasks, and it is possible to control the number of execution threads per TT.
- NameNode (NN): this is the master process that manages HDFS. There exists one NN per cluster, however, in order to provide HDDFS with fault tolerance, there is another process, the Secondary Name Node, which is able of substituting the NameNode in case of lack of responsiveness.
- Secondary NameNode (SNN): this is an alternative master process for HDFS management that starts working in case of a NN failure. For this purpose, it should be launched on a different cluster node than the one hosting the NameNode.
- DataNodes (DN): these are slave processes that receives requests for data blocks and perform disk accesses to retrieve those blocks.

Figure 9 shows a typical configuration for a Hadoop cluster. Master processes (JobTracker and Namenode) can execute on the same node or on separate nodes. But the Secondary NameNode should be launched on a different node in order to meet its fault tolerance goal. The rest of the nodes usually host both one TaskTracker and one DataNode.

Following we describe the communication involved in a Hadoop cluster when launching a job (see Figure 9). We can consider two types of communications between Hadoop processes: control messages and data messages. Control messages usually involve small messages that configure the execution of the application code. Data messages can involve big messages containing input or output data from the job tasks.

Launching a new job into a Hadoop cluster involves sending the job configuration data to the JobTracker. This is a control message that contains information as, for example, which is the input data for the application

(pathname of the input file in the HDFS file system) and where to store the final result of the application. Notice that MapReduce is a programming model driven by data. Thus, the way to decide how to divide the job into parallel tasks is based on how the whole input data are divided into pieces of data. As part of the initial job configuration, users can also provide a method to perform this data division (Hadoop also provide default methods to use for this purpose). Once the JobTracker receives the new job, it computes the amount of tasks to perform based on this input data splitting. And starts assigning tasks to the processing slaves by sending control messages. Each control message contains the identification of each data block involved in the piece of input data assigned to the task. When a TaskTracker receives the assignment, it has to retrieve the input data of that task. In order to do that, it performs the following steps: first, it sends a control message to the NameNode with each data block identifier, which will respond with the address of the DataNode that hosts it; second, it sends a control message to each target DataNode with the data block identifier, which will respond with a data message containing the requested data block. The output data of the task are intermediate results that will be processed as part of a reducer task. Reducer tasks are also executed in the slave nodes and are the responsible for combining those partial results from all map tasks in order to get the final result of the job. Thus, mapper tasks send data messages to reducer tasks containing their partial results. When a TaskTracker ends processing a task, it sends a control message to the JobTracker requesting a new task to perform. As long as reducer tasks generate the final output data, it is necessary to write them on the output file. This step involves again control messages between TaskTrackers and the NameNode, to allocate free data blocks for the output file, and data messages between TaskTrackers and DataNodes, to send the output data. In addition, as HDFS implements block replication in order to provide system reliability, those DataNodes that receive new blocks of data to store need to send a replica to other DataNodes by sending data messages.

Hadoop tries to enhance data locality when assigning pieces of data to TaskTrackers. Thus, when a TaskTracker requests a new task to process, the JobTracker selects, if possible, a piece of data stored in the local disk of the node that hosts that TaskTracker. However, this decision is only possible if application programmers provide a method to get the information about which DataNode manages each data block.

Hadoop can also implement rack awareness decisions if the system configurator provides the information about nodes organization in racks. In that case, Hadoop tries to keep different replicas of the same data in nodes that belong to different racks.

Another relevant characteristic that affects the network tracing for Hadoop is that it implements asynchronous communications. That means that each process has some threads dedicated only to receive and send messages, and some other threads to process the receiving messages and to generate messages to send. In order to analyse the impact that network improvements may have on the final performance of the applications it is necessary to get, not only the information about when a message is received but also the information about when the process starts processing that message.

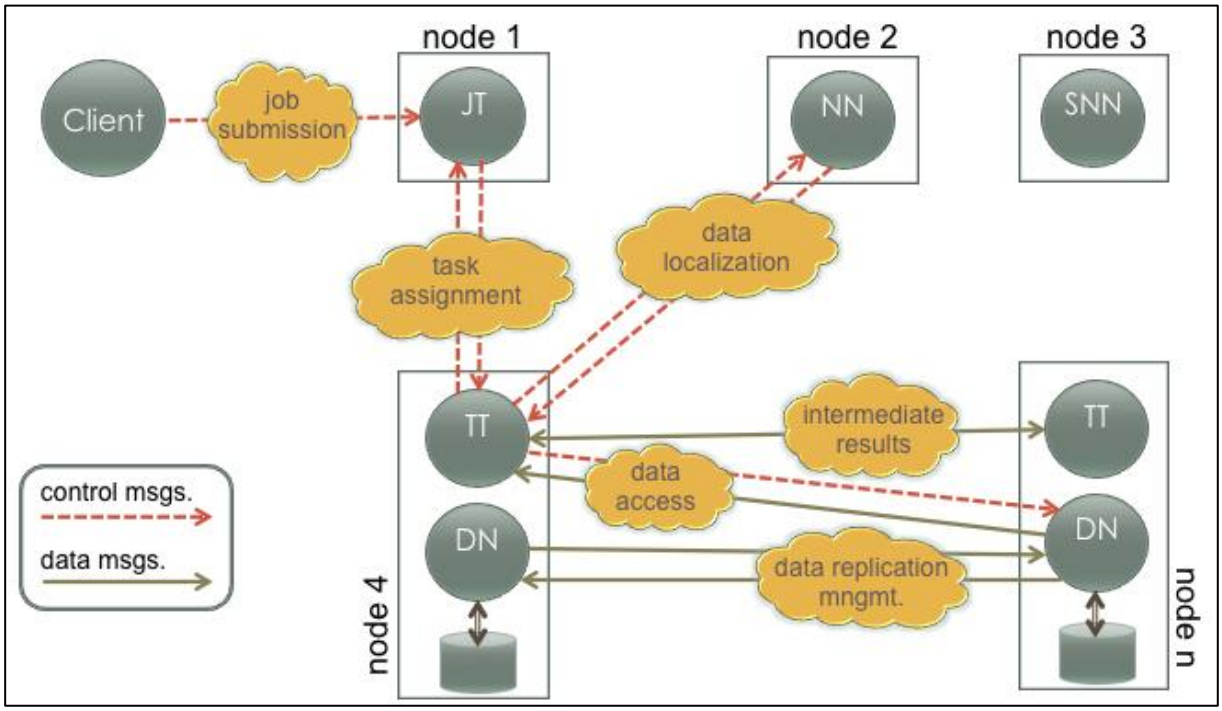

Figure 9: Hadoop processes and types of communication

3.2. Tracing data

In order to get the characterization of the communication involved when executing a Hadoop application, we need to get tracing data about three types of events.

- Characterization of messages sent/received across the network. This characterization can be obtained by the execution of a network sniffer (see section 3.3). In particular, this type of events contain the following package information: LOCAL IP, LOCAL PORT, REMOTE IP, REMOTE PORT, RAW PCKT LEN, APP PAYLOAD LEN, NUM SEQ, NUM ACK, FLAGS
- Information about processes involved in the communication. This information can be generated by Hadoop code at initialization time together with operating system tools that show information about ports in use (see section 3.3). We will use this type of event to relate messages with processes involved in each communication.
- Information about when processes start processing data. This information can only be generated by Hadoop code.

3.3. Monitoring infrastructure for Big Data applications

We have designed and implemented a monitoring infrastructure to get traces of the execution of Hadoop applications. Our main guidelines for this design were the following. First, we aimed to obtain the information about the network usage of a Hadoop job in a format that the simulator is able to read. Second, we wanted to use if possible existing tools that save implementation time. And third, we wanted to minimize the amount of modifications implemented inside Hadoop code.

A network sniffer based on libpcap library can extract all the information except which processes sends and receives each packet. The process associated with the connection of the packet sent/received will be extracted with information generated by operating system tools and by Hadoop code. And the information about when a message starts being processed is extracted also by Hadoop code. All events are generated using Extrae [Ext], a tool developed at BSC that manages the reception and record of events in a suitable format to feed the simulator that we will use in this project. However, in order to generate a complete event describing a process communication, we need to post process all the collected information to merge both the information provided by the sniffer and the information provided by Hadoop and the operating system tools. In Figure 10 we picture the software involved in our monitoring infrastructure as well as the stages involved in the generation of the final execution trace of the application.

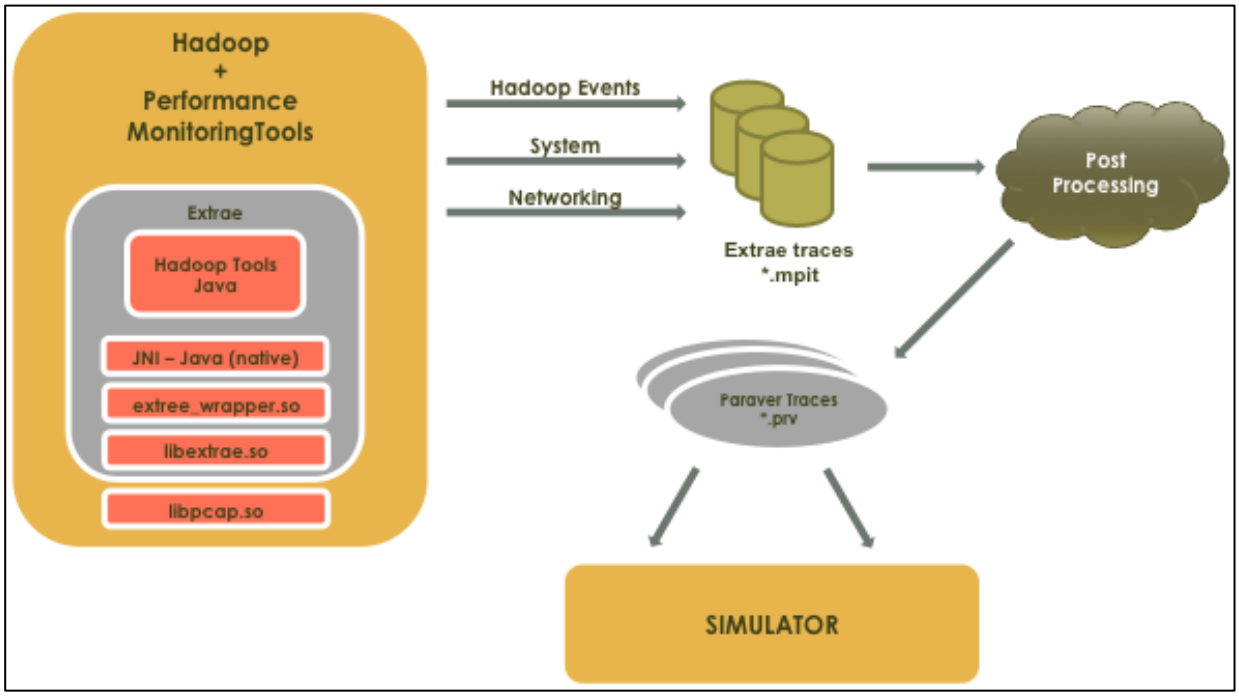

Figure 10: Monitoring infrastructure: software stack and stages

Figure 11 shows the architecture of our monitoring infrastructure for Hadoop application. Following we describe in more detail the tasks performed by each component.

- Hadoop code: we have added to Hadoop code several modifications required by our monitoring infrastructure. First of all, at starting time Hadoop initializes the entire monitoring infrastructure by launching the rest of processes involved in the monitoring tasks. Then, when Hadoop processes are created they record in a log file their process identifier and their role (JobTracker, TaskTracker, etc) as well as the node IP where they are executing and their opened connections. Finally, we have identify in the Hadoop code those functions executed for processing received messages and have modified them with the generation of a Extrae record to mark that moment in time.
- The *sniffer* implemented uses the libpcap 1.4.0 library. The objective of the sniffer is to extract the necessary information from all the network packets leaving or arriving at the node. Extrae will store all this information. When a network packet arrives or leaves the nodes, the kernel verifies if the

packet follow the filter specified by the sniffer. The filter expression accepts all the packets whose source port or destination port is used by the Hadoop processes associated.

- *Dumpingd* is a shell script that launches at configurable intervals the operating system tool *lsof*. This tool records all opened sockets with its associated processes on each side of the socket. During the post-processing phase, the shell script dumping-host-port-pid, will combine this information together with the information dumped by Hadoop in order to associate connections and processes.

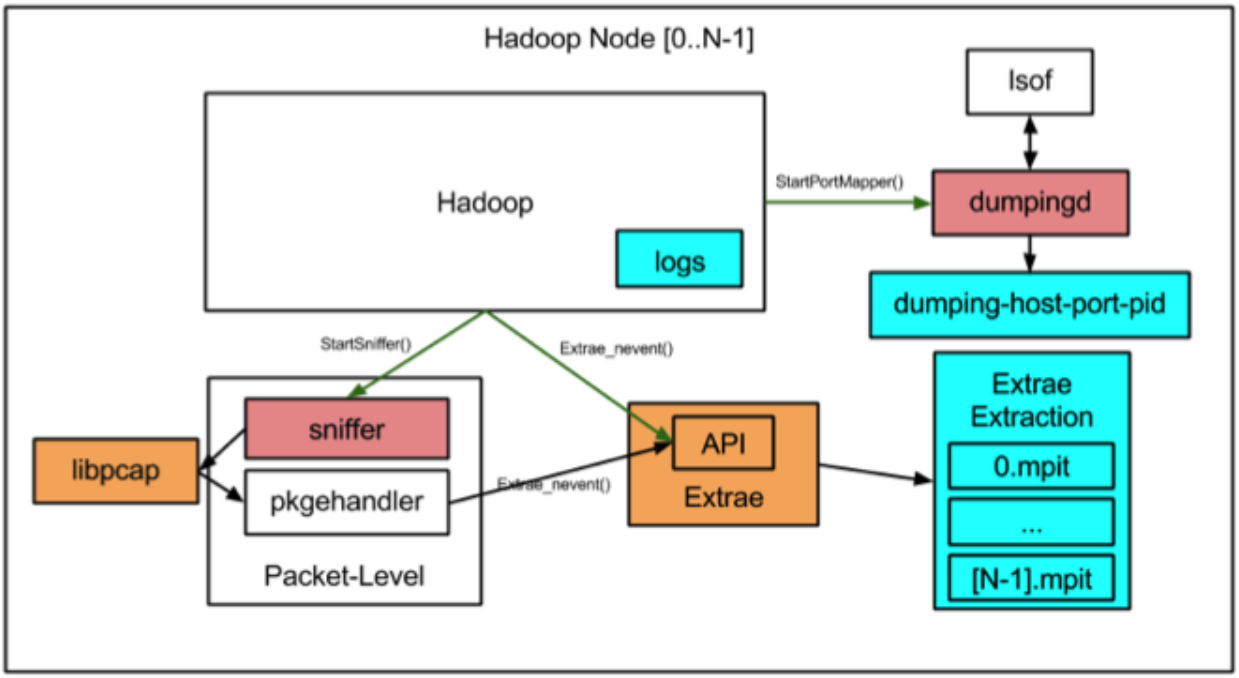

Figure 11: Monitoring infrastructure: architecture

4. Simulation Results

This section shows the simulation experiments of the lightness network carried out through our simulator framework developed during this year. The experiments have been organized into three main sections: validation of the simulator, impact of lightness to scientific applications, comparison with electronic networks. These results represent a preliminary evaluation of the impact of the lightness network. A more detailed evaluation will be performed in the third year of the project taking into account a variety of scientific applications and also BigData applications.

For these experiments it was not assumed the new Architecture on Demand (AoD) in the lightness network. As you know, this AoD architecture is based on introducing a back plane based on an OCS between the TOR and the OPS. This new architecture will add some significant delays in order to configure the OCS at the beginning for the case of the OPS. In order to illustrate the performance of the OPS without the interference of the AoD, it is assumed that the connections in the AoD has been already been configured. The associated cost for configuring the connections in the AoD will be shown in the OCS validation section.

4.1. Validation

A validation of the simulation framework has been performed in order to tune properly the parameters of the simulator to the ones obtained in real experiments. These parameters correspond to the delays for processing and transferring packets in the lightness network for each of the components TOR, OPS, and OCS. In order to gather timing information for the different components a series of experiments were run for this purpose. Also, timing information was also collected from vendors in the case of the production 192x192 Polatis OCS switch [Polatis]. For this validation it was used a series of microbenchmarks that run on servers in order to gather information for various packet sizes ranging from 80Bytes to 32KB which is the minimum and maximum packet sizes in Lightness network. In addition, for these validations it was assumed no cable propagation delay as the cable was very short making this time negligible. In addition, in

these validation experiments it was also checked the behaviour of the different it was also checking the correct functionality of the different components. The following sections show the validation for TOR, OPS, OCS, respectively.

In order to run these microbenchmarks in our simulator framework it was obtained traces from real executions of these microbenchmarks in the Marenostrum supercomputing at BSC. These microbenchmarks were written in C using the MPI on InfiniBand instead of the Ethernet interconnection network. However, we have adapted our simulator to convert the InfiniBand packets to Ethernet packet in order to obtain meaningful results.

4.1.1. TOR

The validation of the TOR was performed by comparing the results obtained by other partners from an experimental evaluation of the TOR. For illustration purposes, it is briefly described the test bed used for the TOR evaluation. Two TORs are connected each other directly via an optical cable as depicted in the following Figure 12. There is one server connected in each of the TORs in order to inject and receive the data. TORs are connected through an optical 10Gbps fibre. The Server 0 sends two packets of various sizes to the Server 1; packet sizes considered are 80 bytes and 32KB. The times to receive the packets are measured on the Server 1. Table 1 shows the measured times for each packet transfer obtained by the UNIVBRIS partner.

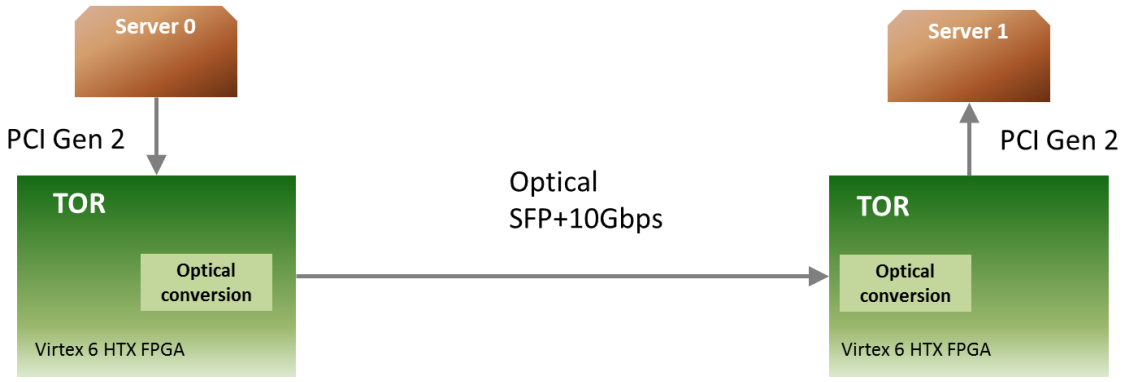

Figure 12: TOR experimental testbed

Packet size	Time (ns)
80B	1742
32KB	107426

Table 1: Measured times for TOR-TOR packet transfers

In order to calculate the overhead associated to the TORs, we have subtracted from the measured times the corresponding time for the optical transmission (assuming 10Gbps bandwidth). This delay is assumed that in reality it is added for any packet. Note that for large messages, the Ethernet packetization mechanism performed in the Servers will generate maximum packet size of 1,500 bytes.

The model for the TOR delay is based on two factors: a fixed cost plus a cost that is proportional to the packet size. The fixed cost is due to the cost (*startup*) associated to all the different components involved in this transfer; whereas the data cost it might correspond to the optical-electrical conversion loss in bandwidth (*optdelay)*. Therefore, the delay in the TOR is calculated for a particular packet size as following,

$$\text{Delay (size)} = \text{startup} + \text{size} \times \text{optdelay}$$

Parameters	Value
startup	1678 ns
optdelay	1,35 ns/byte

Table 2: Values for TOR delay model

Figure 13 shows the comparison between the results obtained from simulator and the experimental setup for the evaluated packet sizes. As you can see, there simulation results match the ones obtained from the experimentation. There is a slight difference on the small packet size due to the fact that we are considering actually two flits for an 80-bytes packet considering a 64-byte flit size.

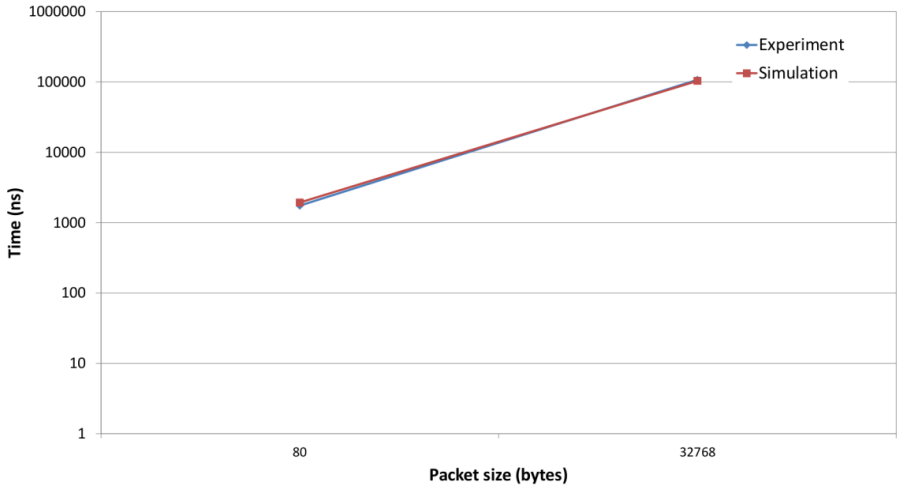

Figure 13: Comparison of the simulated with the experimental results for the TOR-TOR communication

4.1.2. OPS

The validation of the OPS was conducted taking into account the real delays for the main functions in the OPS that are the Routing and ACK processing. These timings were collected experimentally by the TUE partner. Table 3 shows these timings. The ACK processing time in the OPS is already included into the Routing processing time. However, there is a time for ACK processing involved in the TOR which is the one shown.

Parameters	Time (ns)
ACK processing in TOR	20
Routing	30

Table 3: Times for OPS functions

Figure 14 shows the test bed used for the simulations. There are two servers that connect to each of the TORs and these TORs are connected through the OPS with a 10Gpbs optical fibre. The same packet sizes as before were used in this case.

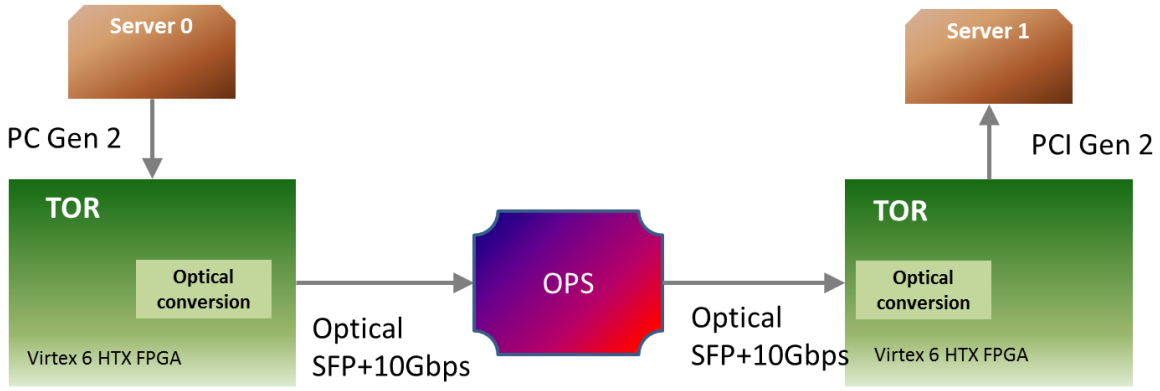

Figure 14: OPS simulation test bed

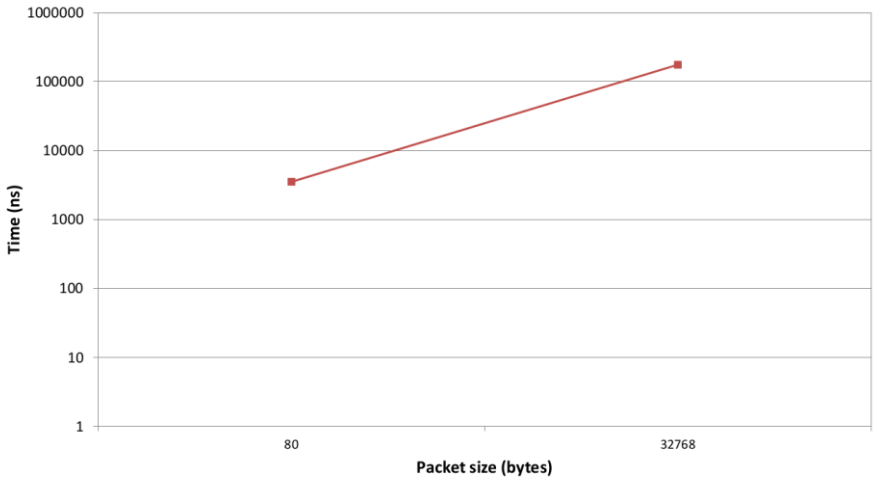

Figure 15: Simulated results for the packet transfer through the OPS

Figure 15 shows the results obtained through our simulator for the communication through the OPS. In this experiment was taken into account the delay of the cable assuming 50 meters between the TORs and OPS.

Additionally, it was conducted an evaluation of the impact of the collisions that may be occurring in the OPS. Figure 16 shows the impact of one packet collision during a ten packet transmission. In this case, it is communicating ten packets of 1500 bytes from one server to another. The test bed consists of two servers. In order to model the behaviour of a suffering a collision in the OPS, the first packet was dropped and then retransmitted from the source TOR. It was observed an additional delay of 3.276ns with respect to no collisions. Packet retransmission was not resulting in any packet out of order. However, further analysis will focus on the impact on packet ordering in the TOR and how this may affect applications.

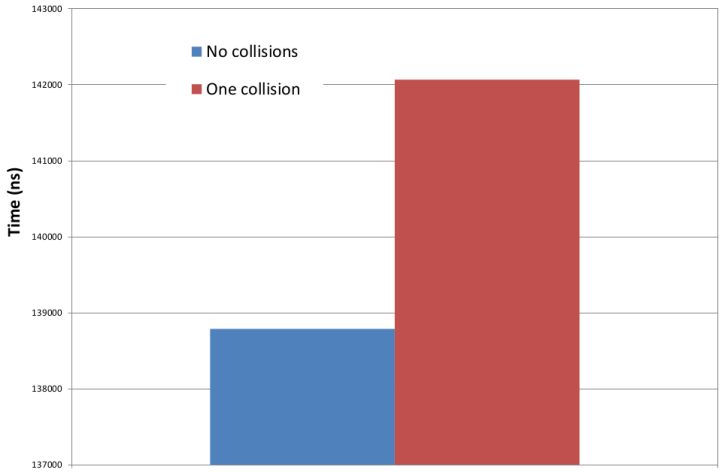

Figure 16: OPS collision results for ten packets of 1500 bytes

4.1.3. OCS

In order to configure properly our simulator to model the behaviour of the OCS it was taken timing results from a UNIVBRIS prototype that is depicted in Figure 17. As it is shown, the TOR has to first request to the control plane to configure the connection in the OCS prior of sending data. Then the control plane configures the OCS and notifies the TOR that the connection is ready to use. According to the experiments the configuration delay is 753ms which include all these communications with the control plane. Also, note that it also includes the 25ms time to configure the Polatis 192x192 port OCS.

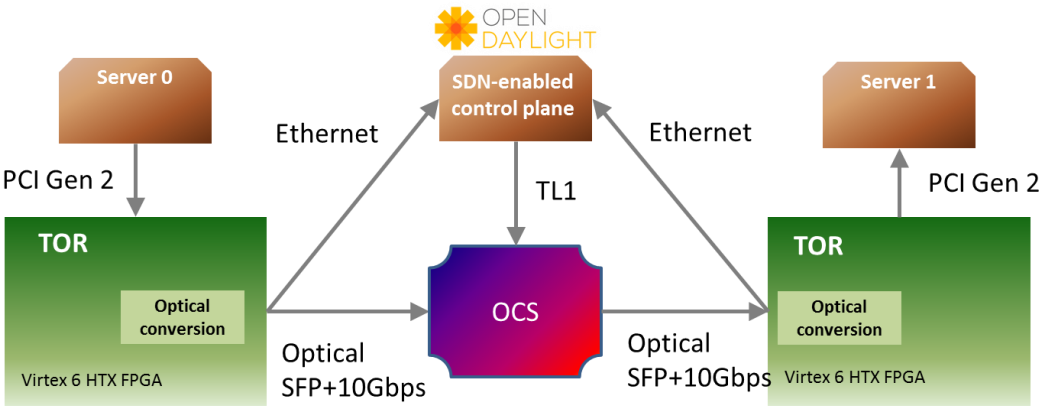

Figure 17: Experimental test bed used for the OCS

Figure 18 shows the simulated results for a series of two packet transfers through the OCS. There is only two packets transferring from one server to the other. These packets have been sent with an inter-message gap of one second. It was evaluated the same packet sizes as in the previous simulations. As you can see, there is huge difference between the first packet and the second packet transfer, especially at 80-bytes packets. The reason for this difference is because the first packet is suffering the full overhead of setting up the connection in the OCS whereas in the second packet there connection is already established. It has more impact on smaller packets because they are more sensitive to latency. For large packet sizes the difference is smaller than for small packets, but still it is quite significant.

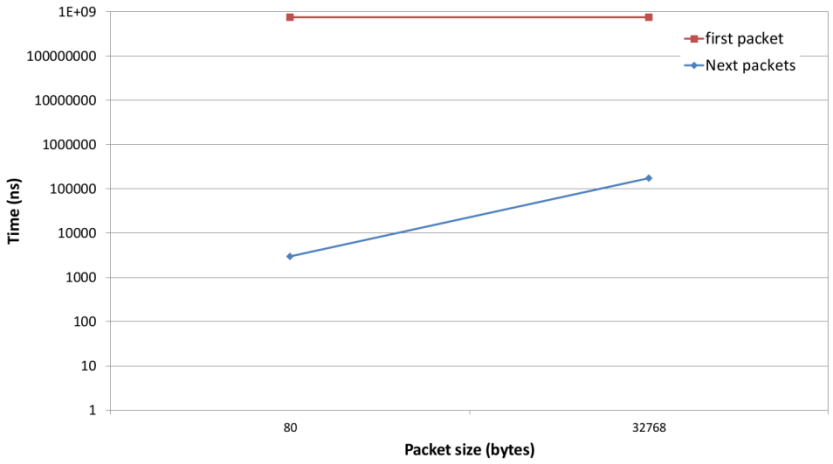

Figure 18: Simulated results for a series of 2-packet transfers through the OCS

4.1.4. Comparison between OPS and OCS

Figure 19 shows the comparison between the OPS and OCS for first and next packets for various packet sizes. As you can see, the main differences are only observed for small packet sizes where the OCS is achieving better performance than the OPS. In particular, the performance improvement is 15%. In addition, it also highlights the huge penalty of setting up the connections in the OCS which is several orders of magnitude higher than the OPS. However, this is only suffered once at the first packet.

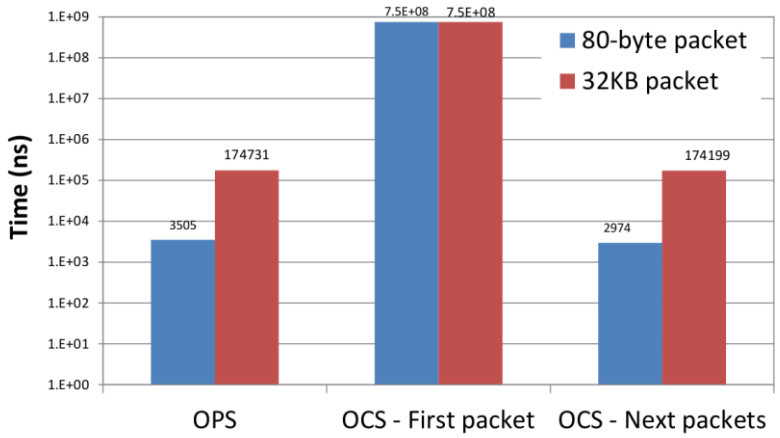

Figure 19: Time comparison between the OPS and OCS based networks

4.2. Comparison with InfiniBand switches

In this section, we have compared the performance of the Lightness network with the performance of the InfiniBand network. For this evaluation an InfiniBand test bed depicted in Figure 20 that replaces the two TORs and the OPS with the corresponding InfiniBand switches. It was compared with the data obtained previously in the Lightness network when we have only one OCS and OPS. Note that the InfiniBand is based on electronic switches for the TOR as well. It was assumed a delay of 200ns for the electronic InfiniBand switches.

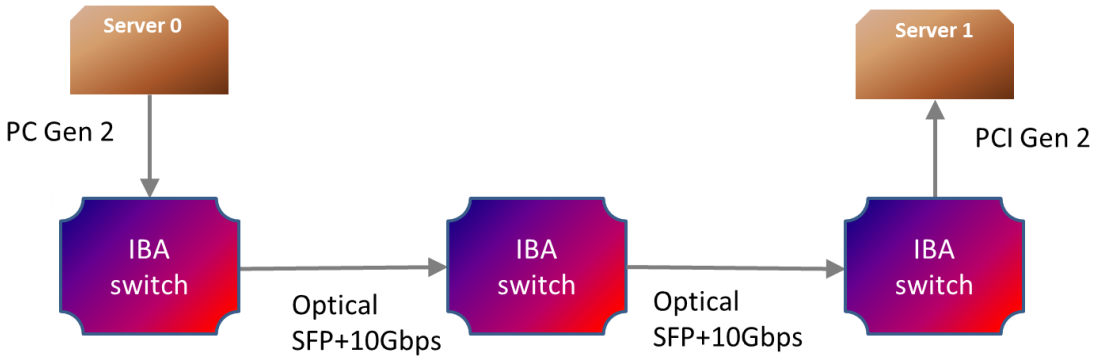

Figure 20: Test bed used to evaluate the InfiniBand network

Figure 21 shows the simulation results for different packet sizes for the InfiniBand and their equivalent OPS lightness network. It can be seen, there is a significant difference on time at small packet sizes. In particular, at 80-byte packet the InfiniBand network achieves 45% and 35% lower time than the OPS and OCS lightness network. The main reason is the high delays suffered in the TORs where there are not suffered on the InfiniBand. However, at large packet sizes this difference is very small, only 2%. Therefore, this results shows that there is still room to improve the performance of TORs by reducing its internal delays.

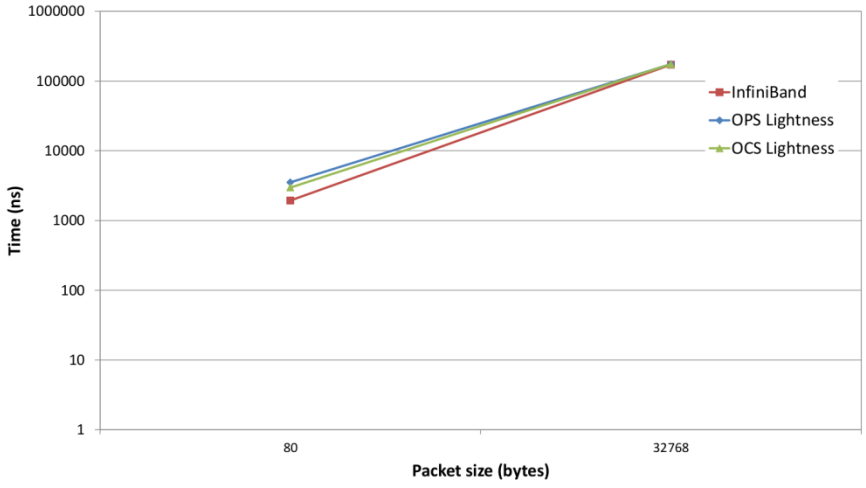

Figure 21: Simulation results for packet transfer through the Lightness and InfiniBand networks.

5. Conclusions

The second year in WP2 was devoted to build the simulator infrastructure that models the Lightness network and allows us to evaluate its impact to HPC and BigData applications. In addition, it was also focused this year on building the tracing tool to extract relevant characteristic of the communication of BigData applications. This is a powerful tool that it enables as to quantify for the first time the performance of Lightness in a wide variety of workloads commonly running in HPC and Data centre systems. Also, this tool is quite flexible to be extended to model other optical components that could be envisioned during this project such as the optical network interface. Also, it will allow us to identify potential bottlenecks in the Lightness network that could be optimized and improved for the benefit of the targeted workloads. In this regard, it was shown that the TOR may be introducing high delays for small packets that could negatively impact the performance of workloads. A further investigation of the causes of these delays will be interested to be conducted and propose solutions to overcome these issues.

Moreover, simulation results shown, as it was expected, the that the OCS provides a very low switching time with respect to the OPS once the connection is already established resulting in a significant performance advantage. This result also paves the way to develop decision algorithms to select which of the optical switches is more efficient to use at runtime. Also, these results were obtained for single-flow microbechmarks. However, it would be interesting to see if that trend is also observed in multi-flow applications.

6. References

[Chandy]	M. Chandy and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". IEEE Transactions on Software Engineering, (5):440–452, 1979.
[del-d22]	"Design document for the proposed network architecture", LIGHTNESS Deliverable 2.2, 2013
[Dim]	Dimemas: http://www.bsc.es/computer-sciences/performance-tools/dimemas
[Ext]	Extrae: http://www.bsc.es/computer-sciences/performance-tools/trace-generation
[IBSim]	Infiniband simulation model http://www.omnetpp.org/component/content/article/9-software/3707-infiniband-v2-released
[IBTop500]	Top500 Statistics, November 2013: http://www.top500.org/statistics/list/#.U5q8iHWSzMU
[Omnet]	OMNeT++: http://www.omnetpp.org/
[Omnest]	OMNEST: http://www.omnest.com/
[Prv]	Paraver: http://www.bsc.es/computer-sciences/performance-tools/paraver
[Rpkg]	OMNEST R package: https://github.com/omnetpp/omnetpp-resultfiles/wiki
[Polatis]	Polatis switches http://ww.polatis.com