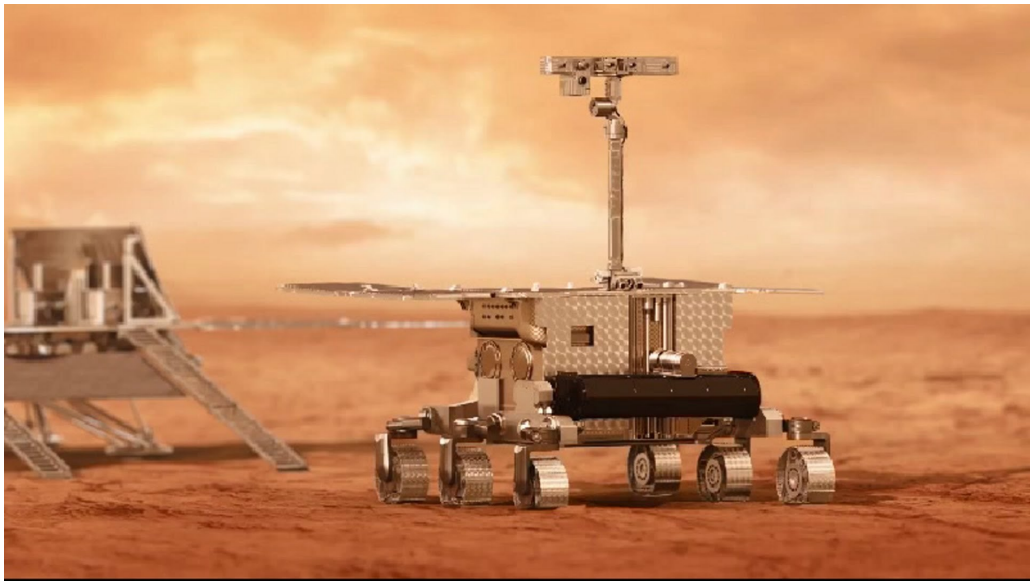
R&D project

Modelisation of ExoMars 2020's network using SystemC

Anas Naïri – Julien Plante

Saint Petersburg State University of Aerospace Instrumentation.



Contents

1 Task formulation

- Read documentation about the ExoMars2020 rover to establish the rover's network structure
- Implement the rover's network on SystemC, only the SpaceWire's network layer. Channels must handle the speed of transmission and error bit frequency, and devices must generate data into the network and receive data from it
- Create a base log-system to record results of the simulation through SQlite
- Split the network into two different parts to parallelise, and compare the modelling time between a single model and a parallel model

2 Introduction

Our second project is a direct continuation with the previous one. To summarize briefly, our first R&D project was about implementing a SpaceWire network using SystemC, to simulate data sent from Earth and received on a spacecraft.

In fact, this R&D project, which is to simulate the data-handling network architecture of the ExoMars2020 rover, involves to use a SpaceWire router with a view to interconnecting the multiple scientific instruments. Since the rover uses a lot of cameras to observe, analyse and interpret its environment, the processing of images taken by these instruments is well needed. In that case, as we can see in the figure below, the rover got a dedicated chip for this kind of process. Data will be sent from equipment, then to the memory, next to the processor which will give instructions to the image processor.

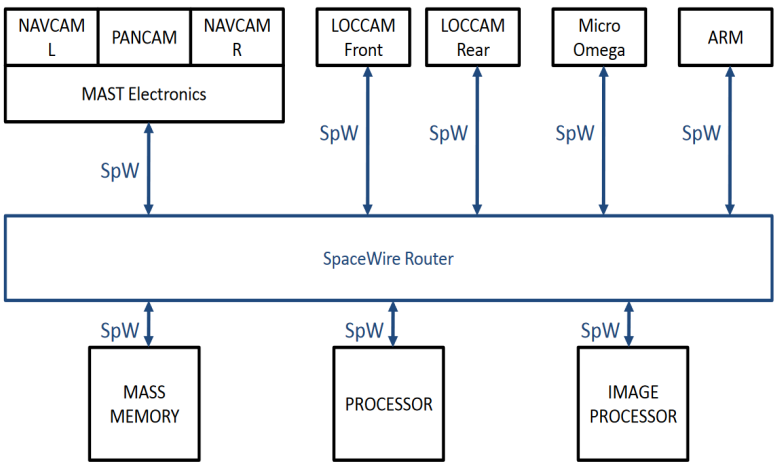


Figure 1: Data-handling structure of ExoMars' rover

Consequently, such amount of image transmission can be well handled by the use of a SpaceWire router, because, as we already know, SpaceWire is a good data-handling network which allies high speed of communication, low power consumption, and low cost implementation. It is ideal in the case of ExoMars2020, because it needs to connect all equipment to memory and processors. We also read that the ExoMars2020 rover uses the RMAP protocol to communicate and exchange data between different nodes, and it is an interesting task to implement this protocol, since it gives us a real packet structure, and a more accurate representation of what happens in SpaceWire networks than during our first task.

3 Description of ExoMars2020 rover

The ExoMars2020 mission has for goal to search evidence of past or present life on the surface and/or subsurface of Mars. Thanks to its rover, it will, in addition, understand more the planet's surface and subsurface, like its origin of formation and the composition of it, by the use of multiple scientific instrumentation as cameras, radars and spectrometers.

3.1 PanCam

The PanCam is used to record 2D & 3D images of the Mars' surface, in order to understand the geological and morphological situation on the planet. In fact, the PanCam is composed of two WAC (Wide Angle Camera), one on the left, the other on the right to have a stereoscopic view of the ground, and a HRC (High Resolution Camera). Also, it operates to produce images in the near infrared and visible wavelength. This device, in synergy with the others, allows the rover to chose the best potential locations site to drill, for example shallow ground that may contains gases or water. The HRC and WAC, both operates with an image resolution of 1024 x 1024 pixels, the first one in RGB, the second with a multispectral filter wheel. The PanCam communicates with packets of 10-bit data.

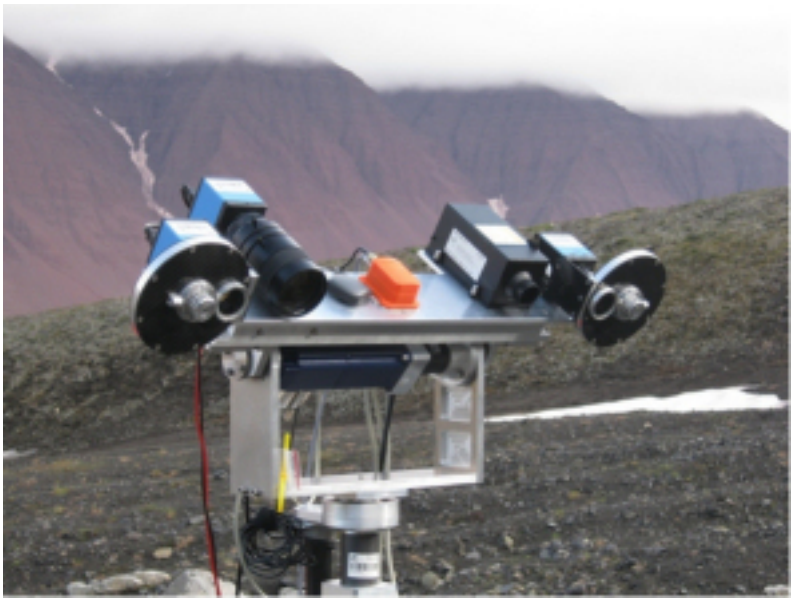


Figure 2: The PanCam prototype during tests in Mars-like conditions
Credit: AMASE

3.2 ISEM

The ISEM (or Infrared Spectrometer of ExoMars), as its name means, is an infrared beam that senses the infrared part of the sun's light reflection onto Mars' surface. With this method used on the rocks near of the rover, it obtains the spectrometer of these rocks and then get information about the geological composition of the planet's surface. Indeed, this device helps the rover to chose a potential location to drill in, because the presence of some minerals may indicate good places where to find past life on Mars. It communicates using packets of 16-bit data.

3.3 CLUPI

The CLUPI, or Close-Up Imager, is a high-resolution camera which goal is to take close picture of rocks to be able to detect each details of these ones. Thus, we can know to what type of rock we are seeing or surrounded with, by analysing the texture, the color, the morphology, etc. Also, it will give information about the original context of possible formation of these rocks, like, geological major events that they experienced throughout their existence. Plus, with this kind of close up pictures, we should be able to detect bio-signatures at the surface of rocks. It is permitted because the CLUPI has an image resolution of 2652 x 1768 pixels in RGB. It can take picture in the visible wavelength, with an exposure time of 1024 seconds. It communicate with packets of 14 bits.

Figure 3: Close up picture of a rock taken from CLUPI

3.4 WISDOM

To find if there are some evidence of the past and present life on Mars, we can search in the subsurface, where organics molecules could be shielded from destructive events. In fact, the WISDOM, or Water Ice Subsurface Deposit Observation on Mars, is a ground penetrating radar which will help to notify and describe the type of the shallow surface that is pointed by this one. If an information about a place in subsurface has been processed as a potentially location of organics molecules, the ExoMars' drill will then dig in it.

In addition, WISDOM is working with others instruments to precise if a place in the subsurface is relevant or not by gathering information from Adron, PanCam and Ma_Miss. From the Adron, we can obtain information about the composition of a source of water in liquid or ice form. From the PanCam, we get 3D information of the rover's environment, that can help it to better filter locations to dig in. And, because the Ma_Miss is in the drill, it is in direct contact with the subsurface, thus data from this one are compared with those from WISDOM that will permit to set a 3D map of the subsurface. By the way, WISDOM communicate with other equipment by packets of 16-bit data.

3.5 Adron

The Adron is a detector of radiations of neutrons that are sensed from the subsurface of the planet. With the data from WISDOM, it permits to detect the water distribution in the Mars' subsurface and the presence of certain elements in this water. Then, the combination of the both instruments' data would help to localize the best places to drill in order to find evidences of potential past or present life on or below the surface of Mars. It communicate with packets of 9-bit data.

3.6 Ma_MISS

Ma_MISS, for Mars Multispectral Imager for Subsurface Studies, is a spectrometer located in the drill of the rover, used to determine horizontal and vertical composition of the Martian soil. By illuminating the borehole and analysing the reflected light and its spectrum, it will be able to gather information about the distribution of minerals, especially of water-related ones, to search potential indicators of life.

It will work alongside the three other spectrometers (RLS, MicrOmega, MOMA), being specialised in studying unexposed material, and in collaboration with WISDOM and ADRON to choose interesting drilling location

Figure 4: The instrument, located on the drill.
Credit: SELEX Galileo

3.7 MicrOmega

MicrOmega is an infra-red spectrometer made to identify composition of Martian soil samples at a grain scale, after their gathering by the drilling system. It is similar as RLS and MOMA in this way, since the three spectrometers will study samples collected by the drill. The infra-red study of the samples is adapted to find evidences of past or present carbon and water presence. It uses an infra-red hyper-spectral microscopic imager to acquire the spectrum of a 250×256 pixels square (~5×5 mm^2) for 320 wave length, between 0.95 and 3.65 µm. Thus having a maximum of 20 480 000 bytes to transmit.

3.8 RLS

RLS uses the Raman effect to find life signatures in Martian soil samples in a non-destructive way. The measurements carried out by the RLS will be performed as described within the ExoMars

Rover Reference Surface Mission, which includes six experiment cycles (with two samples each, one extracted from a surface target and the other at depth) and two vertical surveys (with five samples each extracted at different depths). It generates information about a 2048×512 pixels of 15 µm, totalling a surface of approximately 30.7×7.7 µm^2, and 1 048 576 bytes.

3.9 MOMA

MOMA (Mars Organics Molecule Analyser) is an instrument designed to detect organic molecules in spots of interest detected by the collaboration of RLS and MicrOmega, thus providing extremely precise analysis of Martian environment, and great information about potential origin, evolution and distribution of life on the planet. To do so, it will study samples gathered by the drill, as well as analysing the gases of the Martian atmosphere. It features to modes of operation : Gas Chromatograph-Mass Spectrometry (MOMA GC-MS) and Laser Desorption-Mass Spectrometry (MOMA LD-MS), the first one being used to analyse atmosphere gases, and the second soil samples. No information about size of data produced was found.

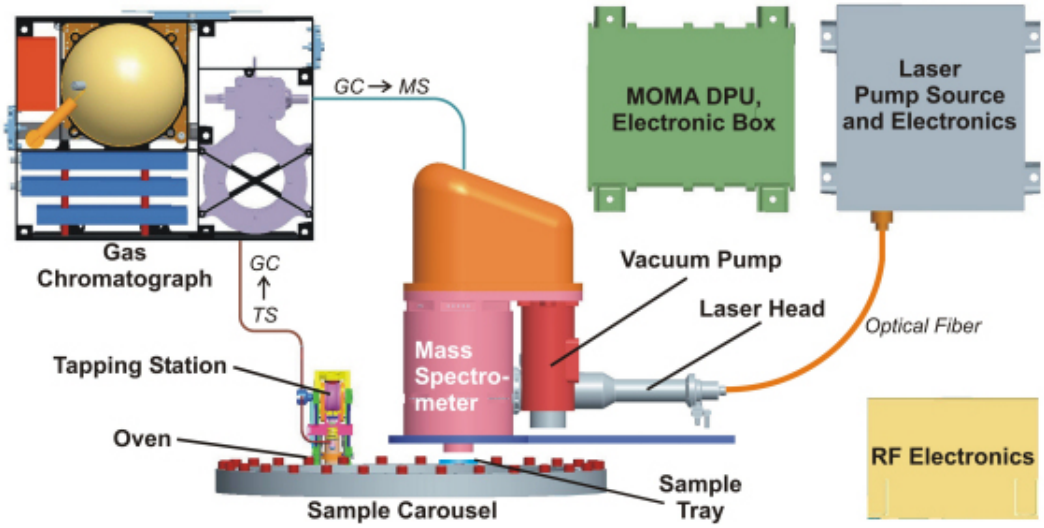


Figure 5: The MOMA instrument and its modules.
Credit: Max Planck Institute for Solar System Research

4 Model structure and functioning description

The final structure of our project looks like this:

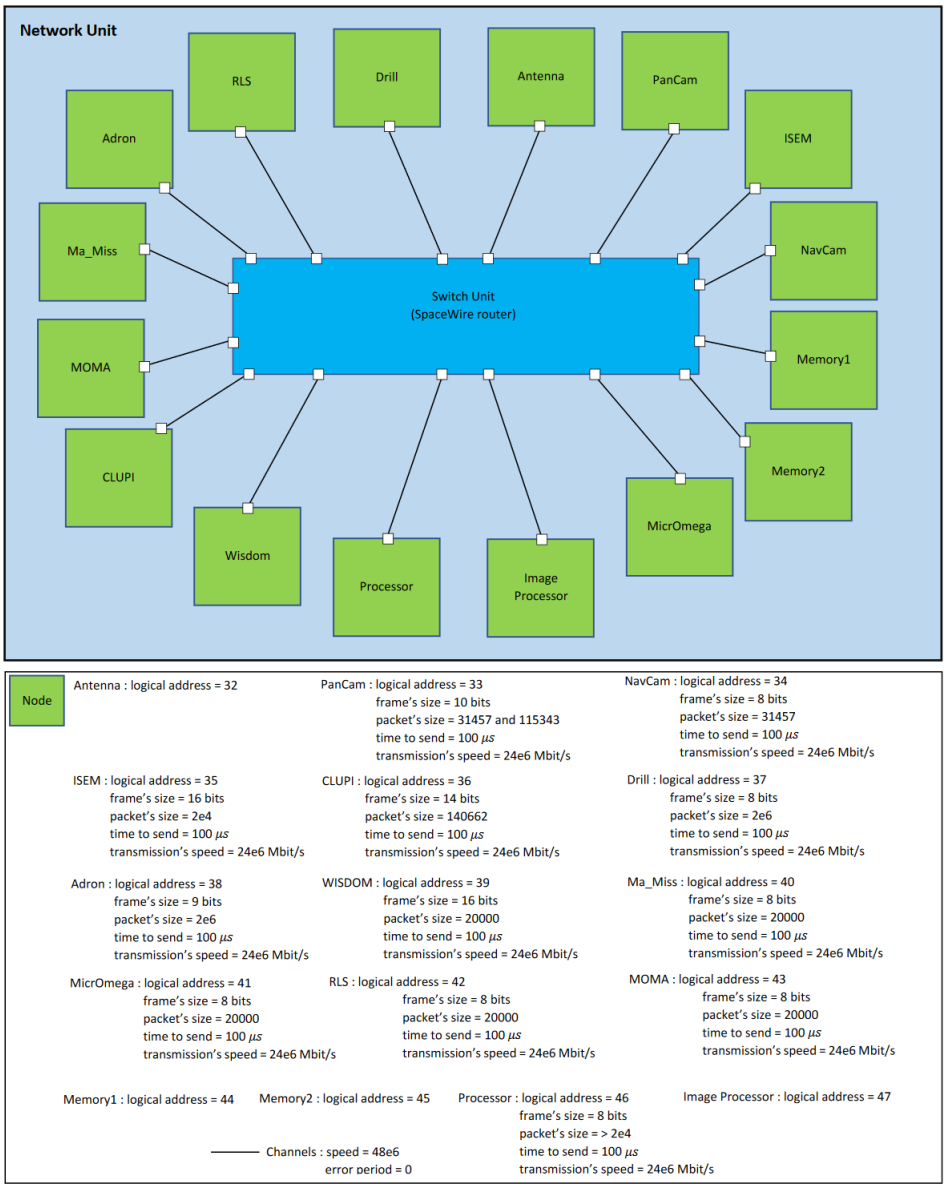


Figure 6: Final structure

As in the first task, the SpaceWire Router is modelled by the C++class `SwitchUnit` and the top module containing all the others by `NetworkUnit`. But one major difference is that each node is modelled by the same C++class named `Node`, specialised thanks to a configuration file. This was made in order to avoid boilerplate code, since all nodes have a similar behaviour, errors due to wrong copy of new functionalities in each class, and to prepare the work for a more complex simulator, using a GUI to configure any SpaceWire network.

All details about these improvements will be explained in the following subsections.

4.1 Features

SpaceWire

Our project features the modelling of the the third OSI layer of the SpaceWire specification, as requested, but also of a part of the second layer, since it was already done during the first task.

The third OSI layer covers the use of packets to transmit data from one node to another, with possible routing on the way. About the second layer, we modelled only a part of it, since it was kept from the first task and was not mandatory for this one. We only have the transmission by frames of packets, allowing for transmission delay simulations and transmission error modelling (due to some possible incident radiation). It also made us think on the problem of data collision, since we used bidirectional channels, and forced us to create some security mechanism. But we did not implement the Error End of Packet (EEP) and error link recovery at this level.

The only communication protocol implemented is RMAP, since we found out that ExoMars-2020 uses this protocol. More details about its implementation can be found in the description of Node

Log system

We added a logging system, to increase debugging possibilities. It was at first under the form of a simple text logging.The information printed to the standard output was also written to files, with one log for each node, resulting in a less mixed output. But this was not very practical, since analysis on these files was tedious, because of no formal format used. This is why we quickly decided to investigate database logging, which was one optional task.

We chose to use SQLite as database, since its use was much simpler and more adapted (at least at the moment) than MongoDB, the other database client investigated. Indeed, we wanted to use database as a handy file format to store our information, which is easily allowed by SQLite, which stores everything in one file. MongoDB on its side uses the client/server paradigm, and would have forced us to go through the localhost, bringing complications. We successfully logged packet transmission using SQLite, and the ability to perform SQL requests proved its interest, allowing us to find bugs, and permitting custom analysis of what happened during the simulation after its end.

Parallelisation

We were asked to separate our network into two different parts, and paralellising the simulation. Indeed, we transmit in the network big packets, and the simulation began longer and longer as we added features. An idea to increase simulation speed was to run the two independent parts of the network in parallel, hoping for a performance increase. To do so, we described the two distinct parts using a configuration file (see below), and used preprocessor directives alongside preprocessor definitions to compile two different executables, each one of them being able to simulate one part of the network. We then used the Windows API, and the function `CreateProcessA` to run the two parts concurrently.

To launch the two processes, build and run the Visual Studio Project “Create_two_Processes”. It should use `msbuild` to compile each part of the network and then the Windows API function `CreateProcessA` to execute them. But a problem may occur, if the path to msbuild is not the same on our computers, or if the SystemC library was not compiled with the same options (Debug or Release, x86 or x64). To solve this try tweaking the Network project options.

Configuration file

As already explained, we used a configuration file to describe nodes behaviour and different parts of the network. It is also specifies parameters of the channels, which are the same in the whole network, which could be a limitation, but could also be tweaked.

At first, we used a custom configuration file and wrote the parser ourselves. It was doing the job, but adding functionalities was complicated and error prone, and the parser itself not very fault tolerant and strong when facing syntax mistakes. Moreover, we had the idea of a GUI to describe all the parameters, and judged that it would be harsh to interact with our custom configuration file.

So we moved to the JSON format, using the header-only C++JSON parser RapidJSON. It proved to be much stronger, and allowed us to easily add new features, without needing to check for bugs. It currently supports channels specification, network part specification, nodes declaration, description of their interaction with the network (when to send packets, how many, the type *etc*), and the self generation of packets, modelling an experiment where results are stored in a node's memory, with similar arguments. The only limitation of this configuration file is that it doesn't allow to describe routers for now, so one router is created for each network part.

To easily gather information from the DOM, we wrote a loader contained into a class, which can be found in `helperlib.h` and the corresponding implementation file `helperlib.cpp`.

Graphical User Interface

Another optional task we began is the Graphical User Interface (GUI). Sadly, we didn't have time to finish it, but is still an interesting feature of the project.

We chose the Qt framework to develop it, since it's one of the most developed widget toolkit with GTK+, and seemed easier to use than the later, since it is written in C++and thus doesn't need bindings, which is the case for GTK+. Moreover, the existence of QtDesigner allowed us to easily design our UI and quickly obtaining interesting results. At the time of writing, it is able to generate the configuration file, compiling and starting simulation of the different parts of the network, and also analysing the results thanks to a simple SQLite database viewer.

Planned improvements are reading from existing configuration file to load network parameters, network part management when editing nodes, and possibility to choose from which database we want to read, as well as a refresh button for the database viewer. Further possible, but hypothetical improvements could be to add a graphical representation of the network, and a graphical editor, instead of parameters only, and real time animation of the graphical editor during the simulation.

4.2 Network Unit

`NetworkUnit` kept its role since our first task and the corresponding report, namely to instantiate all the others modules and linking them with channels. But its way to do this changed drastically, since everything is allocated dynamically, to allow customisation through the configuration file.

To do so, it firstly recover all the configuration needed using the `JsonConfigLoader`. It then dynamically allocates every node, which pointers are stored into the vector `instruments` to keep a handle on each one of them. The same is done with channels. At the same time, we connect the instrument to the channel and the router, using the function `SwitchUnit::connect` that creates routing tables automatically.

> Remark: One of the goals after the first task was to increase memory safety of our application, since we used `new` and `delete`, which can cause memory leaks if the

deallocation is forgotten or in case of a crash, if stack unwinding doesn't happen. This is why we used `make_unique`, and `std::unique_ptr<T>` declared in the header `memory`. This is a smart pointer, deallocating memory when going out of scope, and thus safer than raw pointers.

After that, `NetworkUnit` adds other parameters of the config file to the nodes created, concerning generation of packets and transmissions they have to do.

This module is also in charge of opening the database and passing its pointer to each node, as well as creating the trace file and make it watch each channel. Its definition can be found in `networkunit.h` and the implementation in `networkunit.cpp`.

4.3 Node

This module describes a SpaceWire node with only one port, that can be configured to our need through the configuration file. It contains multiple methods, corresponding to multiple levels of abstraction in our model of the node, which were useful to structure the code. They contain a memory that can be written or read through the RMAP protocol, or written directly by the node itself, simulating the generation of data. Since RMAP is a well known standard, we decided not to describe it in this report, but more information can be found in [1]

Firstly, there are three type of daemons, which are by the way the three main type of threads of this module (we named them daemons because it usually describe threads that run in the background, handling everything important). The main one is `receiver_daemon`, in charge of handling reception of packet, and choosing what should be done with the packet. It is written to receive RMAP packets, and sending back replies if needed. This daemon is only launched once, and runs during the whole simulation, which is not necessarily the case of the others. `sending_daemon` and `generating_daemon` are very similar, the first one being in charge of sending RMAP commands, and the second one generating data directly to the node memory. Any number of these two can be spawned, according to the fields "Connections" and "Generations" of the config file. They can also terminate before the end of the simulation, if their role is finished.

Secondly, the function `send` allows to send a packet of any type, and waiting for a reply. If the reply has a positive status (1), the function ends, and if the status is 0, the packet is sent again, until a positive status is received, allowing for safe communication and error recovery. There is no `recv` function currently, since we only use the reply as acknowledgement packet, but we could create a `recv` function that sends an acknowledgement back, for both commands and replies.

Finally, the function `send_raw` and `recv_raw`, at the lowest level, each one having as parameter one packet and which are respectively in charge of sending actual data into the port and receiving actual data, determining the type of packet and accordingly create the correct packet before returning. One interesting difficulty for the `recv_raw` function was that RMAP uses different header structures for different actions, and we had to determine the type of the packet one the fly, while receiving, to correctly determine the end of the header, and the beginning of the data.

This module also features regular logging and database logging, which are not its first role, but still key points, especially for debugging and analysis of the model. Tables are firstly created in the database, to be ready to store data in the member function `init_db`. Packets are after that automatically stored whenever one is sent or received.

The definition of this module can be found in `Node.h` and the implementation in `Node.cpp`.

4.4 Packet

This class aims to simplify communication throughout the network, and clearly separating layers 2 and 3 of the OSI model. It contains two vectors, one for the header and one for the data, since each part is of variable size in the RMAP protocol, as well as two checksums (CRC), for the header and the data. Data can be inserted or extracted as it would be done for a stream, using operators << and >>, which allows to compute the CRC on the fly, as it would be done in a real equipment. But this interface is not always handy, and we needed to add functions about the end of the header, because of the variable header size of the RMAP protocol, and maybe the class would need a rewrite to provide simpler interface.

That being said, it is very useful to contain packets simply, more than the FIFOs we used during the first task, and can be printed in a clean way through the overloading of `operator<<`.

The definition of this module can be found in `Packet.h` and the corresponding implementation in `Packet.cpp`.

4.5 SwitchUnit

`SwitchUnit` didn't change a lot. We just increased memory safety by dropping the bidimensional C-style array we had before for a `std::vector`. We also added the simple `connect` function, that allows to connect a node to the router through a channel, and automatically modify the routing table to match the new connection. The only limitation here is that we didn't write any `connect` function to link two routers, but we didn't need this at the time of writing, since ExoMars2020 contains only one router. But if we are to develop a more complete and versatile simulator, this could become a need.

The definition of this module can be found in `switchunit.h` and the corresponding implementation in `switchunit.cpp`.

5 Simulation results

To show the results on the console, we set an example where ISEM sends a write command with data of size 25 to PanCam, wanting to access its 10^{th} memory address, then at 100 µm NavCam reads at the same address of PanCam, thus reading the data written before. Here is what we got :

5.1 Console output

```
0 s nu.ISEM sending packet of size 25 to 33
   33   0bus0000000000100001 <- destination address
    1   0bus0000000000000001 <- protocol identifier
  113   0bus0000000001110001 <- packet information
    0   0bus0000000000000000 <- destination key
   32   0bus0000000000100000 <- source address
    0   0bus0000000000000000 ┐ transaction id
    0   0bus0000000000000000 ┘
    0   0bus0000000000000000 <- extended write address
    0   0bus0000000000000000 ┐
    0   0bus0000000000000000 │ write address
    0   0bus0000000000000000 │
   10   0bus0000000000001010 ┘
    0   0bus0000000000000000 ┐
    0   0bus0000000000000000 │ data length
   25   0bus0000000000011001 ┘
39360   0bus1001100111000000 <- header crc
15808   0bus0011110111000000 ┐
55264   0bus1101011111100000 │
36411   0bus1000111000111011 │
48712   0bus1011111001001000 │
  738   0bus0000001011100010 │
61469   0bus1111000000011101 │
43121   0bus1010100001110001 │
22175   0bus0101011010011111 │
57534   0bus1110000010111110 │
45090   0bus1011000000100010 │
48949   0bus1011111100110101 │
47838   0bus1011101011011110 │
54871   0bus1101011001010111 │ data
47710   0bus1011101001011110 │
11684   0bus0010110110100100 │
28113   0bus0110110111010001 │
15378   0bus0011110000010010 │
49856   0bus1100001011000000 │
45567   0bus1011000111111111 │
 3539   0bus0000110111010011 │
33880   0bus1000010001011000 │
54485   0bus1101010011010101 │
 1274   0bus0000010011111010 │
10656   0bus0010100110100000 │
64904   0bus1111110110001000 ┘
51604   0bus1100100110010100 <- CRC
```

Figure 7: ISEM writes inside the memory of PanCam

Figure 8: PanCam receives the packet

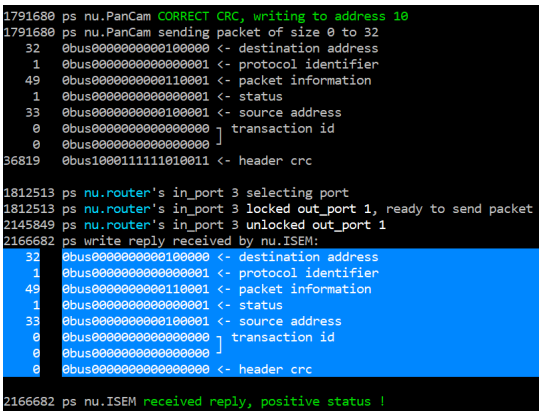


Figure 9: PanCam writes the packet in its 10^{th} address, and sends a reply to ISEM, which receives it

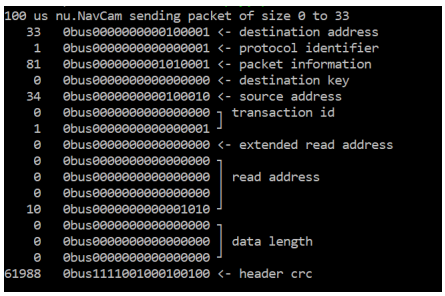


Figure 10: NavCam sends a read packet to PanCam to get the packet of the address 10

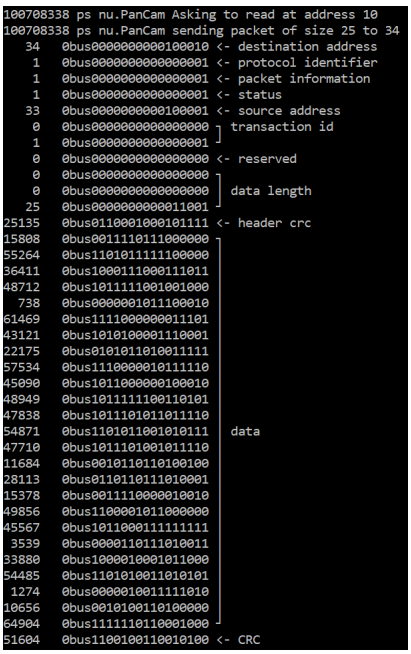


Figure 11: PanCam reads at its 10^{th} address and sends it to NavCam

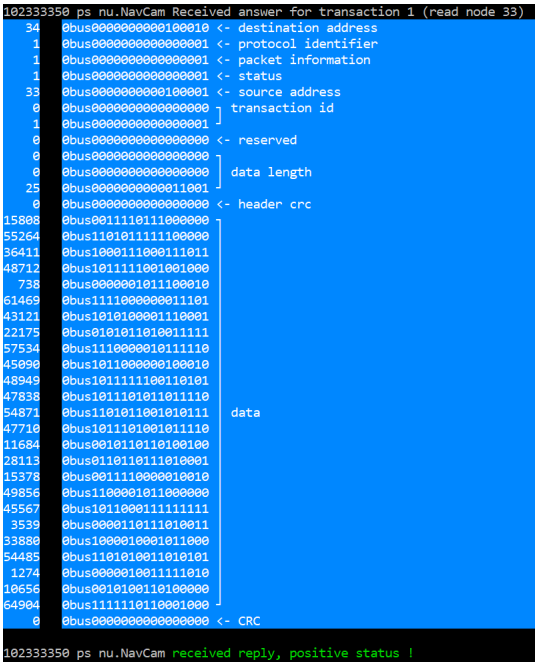


Figure 12: NavCam receives the reply from PanCam to read the packet from it

5.2 Traces

The following trace file can be opened in the file Network\traces\network_0.vcd, but it is modified each time we recompile the code.

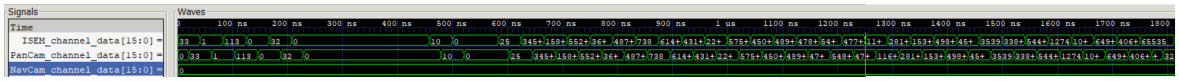


Figure 13: Trace wave - ISEM writes into PanCam

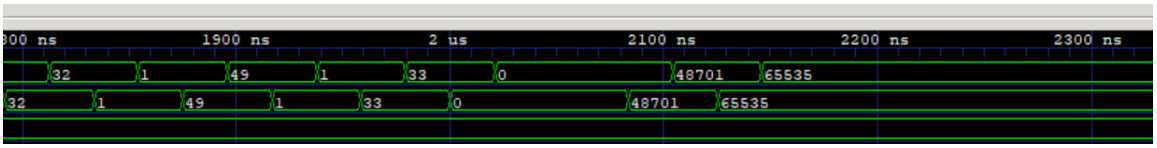


Figure 14: Trace wave - PanCam sends reply to ISEM

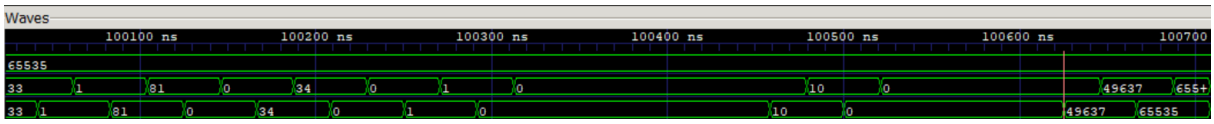


Figure 15: Trace wave - NavCam sends a read packet to PanCam

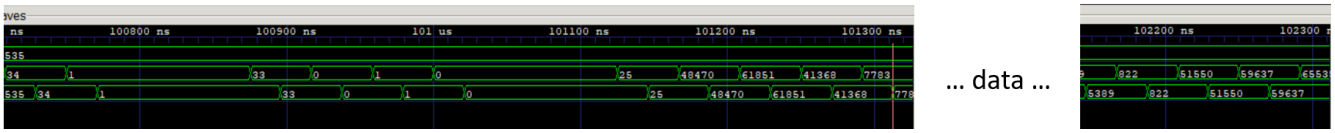


Figure 16: Trace wave - NavCam receives the packet from PanCam

5.3 Database

As we explained in Section 4, we added a database logging system. Here we chose to present a simple example showing the four RMAP packet types. To avoid having too complicated information, as in the previous subsection, we chose to present the database using only three nodes: ISEM (logical addreess 32), PanCam (33) and NavCam (34). ISEM firstly sends a write packet with some data to PanCam, wanting to write a data of size 25 at the memory address 10:

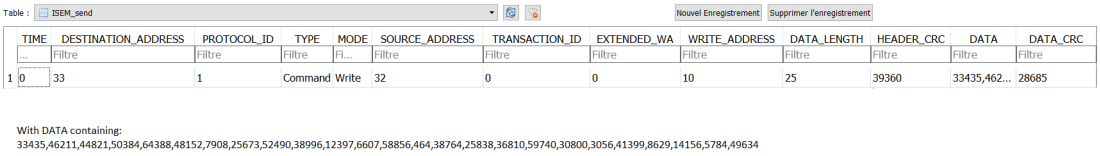


Table : ISEM_send — Nouvel Enregistrement — Supprimer l'enregistrement

	TIME	DESTINATION_ADDRESS	PROTOCOL_ID	TYPE	MODE	SOURCE_ADDRESS	TRANSACTION_ID	EXTENDED_WA	WRITE_ADDRESS	DATA_LENGTH	HEADER_CRC	DATA	DATA_CRC
	...	Filtre	Filtre	Filtre	Fi...	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre
1	0	33	1	Command	Write	32	0	0	10	25	39360	33435,462...	28685

With DATA containing:
33435,46211,44821,50384,64388,48152,7908,25673,52490,38996,12397,6607,58856,464,38764,25838,36810,59740,30800,3056,41399,8629,14156,5784,49634

Figure 17: ISEM sends write command to PanCam

After having the operation happened correctly, here is what ISEM receives from PanCam:

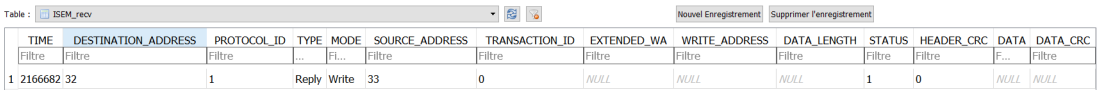


Table : ISEM_recv — Nouvel Enregistrement — Supprimer l'enregistrement

	TIME	DESTINATION_ADDRESS	PROTOCOL_ID	TYPE	MODE	SOURCE_ADDRESS	TRANSACTION_ID	EXTENDED_WA	WRITE_ADDRESS	DATA_LENGTH	STATUS	HEADER_CRC	DATA	DATA_CRC
	Filtre	Filtre	Filtre	...	Fi...	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	F...	Filtre
1	2166682	32	1	Reply	Write	33	0	NULL	NULL	NULL	1	0	NULL	NULL

Figure 18: ISEM receives write reply from PanCam

Notice how `STATUS` takes the value 1. This means that the writing was successful. In the event where writing was not possible, or the CRC was wrong, its value would be 0, and the command packet would be sent again.

Then, PanCam generates data (for example by taking a little photo of only 50 frames of 16 bits) to its address 0:

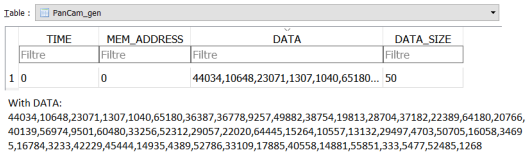


Table : PanCam_gen

	TIME	MEM_ADDRESS	DATA	DATA_SIZE
	Filtre	Filtre	Filtre	Filtre
1	0	0	44034,10648,23071,1307,1040,65180...	50

With DATA:
44034,10648,23071,1307,1040,65180,36387,36778,9257,49882,38754,19813,28704,37182,22389,64180,20766,
40139,56974,9501,60480,33256,52312,29057,22020,64445,15264,10557,13132,29497,4703,50705,16058,3469
5,16784,3233,42229,45444,14935,4389,52786,33109,17885,40558,14881,55851,333,5477,52485,1268

Figure 19: PanCam generates data and stores it at its memory address 0

The last step allows to verify if all was written correctly, and if the read primitive works as expected. NavCam first sends to PanCam a read command to read at address 10, then immediately after to read at address 0:

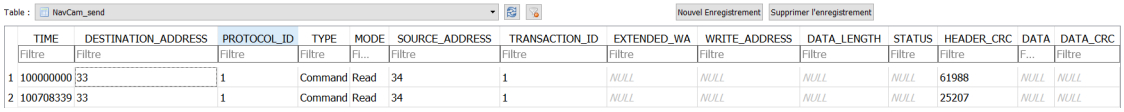


Table : NavCam_send — Nouvel Enregistrement — Supprimer l'enregistrement

	TIME	DESTINATION_ADDRESS	PROTOCOL_ID	TYPE	MODE	SOURCE_ADDRESS	TRANSACTION_ID	EXTENDED_WA	WRITE_ADDRESS	DATA_LENGTH	STATUS	HEADER_CRC	DATA	DATA_CRC
	Filtre	Filtre	Filtre	Filtre	Fi...	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	F...	Filtre
1	100000000	33	1	Command	Read	34	1	NULL	NULL	NULL	NULL	61988	NULL	NULL
2	100708339	33	1	Command	Read	34	1	NULL	NULL	NULL	NULL	25207	NULL	NULL

Figure 20: NavCam sends read commands to PanCam to retrieve data stored in a distant node

And finally, NavCam receives the two read replies to its two read command, recovering data, that is exactly the same as the one that was written by ISEM and generated by PanCam: The database logging can prove useful in the same way with the final network, simulation we can't run properly on our computers with correct parameters because of the amount of RAM it used. Because it exceeds the memory allocated to our program, we obtain a `bad_alloc` exception

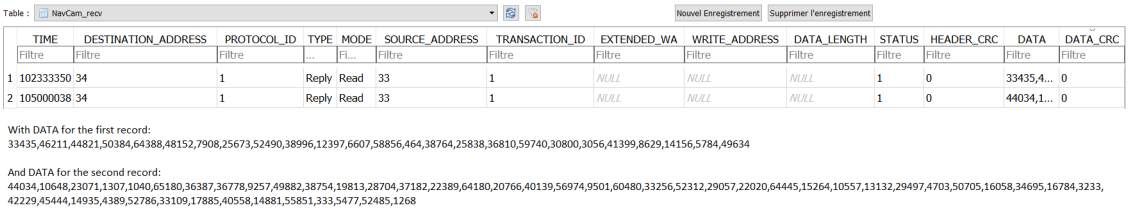


Table : NavCam_recv

Nouvel Enregistrement | Supprimer l'enregistrement

	TIME	DESTINATION_ADDRESS	PROTOCOL_ID	TYPE	MODE	SOURCE_ADDRESS	TRANSACTION_ID	EXTENDED_WA	WRITE_ADDRESS	DATA_LENGTH	STATUS	HEADER_CRC	DATA	DATA_CRC
	Filtre	Filtre	Filtre	...	Fi...	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre
1	102333350	34	1	Reply	Read	33	1	NULL	NULL	NULL	1	0	33435,4...	0
2	105000038	34	1	Reply	Read	33	1	NULL	NULL	NULL	1	0	44034,1...	0

With DATA for the first record:
33435,46211,44821,50384,64388,48152,7908,25673,52490,38996,12397,6607,58856,464,38764,25838,36810,59740,30800,3056,41399,8629,14156,5784,49634

And DATA for the second record:
44034,10648,23071,1307,1040,65180,36387,36778,9257,49882,38754,19813,28704,37182,22389,64180,20766,40139,56974,9501,60480,33256,52312,29057,22020,64445,15264,10557,13132,29497,4703,50705,16058,34695,16784,3233,
42229,45444,14935,4389,52786,33109,17885,40558,14881,55851,333,5477,52485,1268

Figure 21: The two read replies, with same data as written before in each memory address

before the end of the simulation. The software used to visualise the SQLite databases was useful for us, and is an open source project named DB Browser for SQLite. The interface presented in the screenshots come from this software.

5.4 Modelling time comparison

The ExoMars2020 in our project is composed of several equipment, which are divided into two main parts : the mast and the main body of the rover. The mast can be composed of the PanCam, NavCam, and the ISEM. The main body contains the CLUPI, the Drill, Adron, WISDOM, Ma_Miss, MicrOmega, RLS, MOMA. Both parts are connected to an antenna which should send information that were gathered to the probe in orbit around Mars, then to the Earth. Plus, each one got its own memory, and, also there are the two processors which process the images. One little problem we had when launching the simulation with the planned packet size according to publications about ExoMars2020, is that we exceed the memory allocated to our program, resulting in a bad_alloc exception. We had to reduce the packet size to solve this problem, but using a more powerful computer with more RAM to simulate should solve this error.

Thus, one interesting thing to test and to compare is the modelling time between the simulation of a single model and the one of a parallel model. So, we decided to observe the modelling time of a parallel model that involves the mast part and the main body part working at the same time, and this by creating two processes which are launching two executable that were pre-compiled.

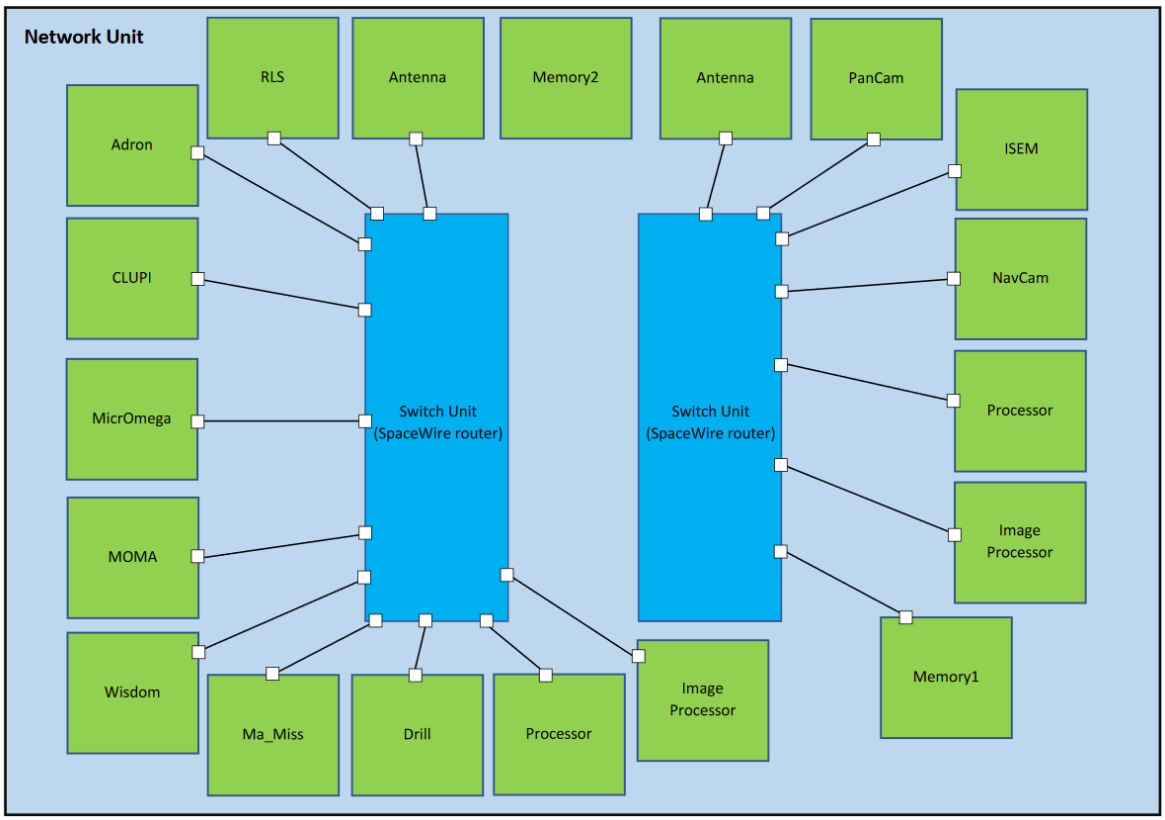


Figure 22: Network Architecture of the parallel model

Then, we decided to do a first try by launching a simple simulation without database handling the base log-system, where, every equipment is writing at least one packet with a size more than 20000 to the memory, then the antenna and the processor read packets from the memory. Ultimately, the processor writes into the image processor. So, as a result of, we got an execution time of $24.07s$ for the single model, and $6.28s$ for the parallel model. We obtain a very promising result. But, in the case of using SQlite, to generate database for the log-system, we got an execution time of $54.19s$, and $50,47s$, this can be explained by the fact that SQLite isn't made for parallel programming, the developers of it advise to avoid the use of parallel programming with database produced by SQLite. Even though, they try to suggest some tools do some within SQLite, but they are not so relevant to use. In fact, when one is opening a database to write in it, it will prevent the other part to open another one. So in future developments of this project, it would be useful to switch to another database system, such as MongoDB for example. This has still to be investigated.

6 Conclusion

We successfully developed and ran a model of the network of ExoMars2020's rover using SystemC. Because little information about the network itself was distributed – most information concerns the scientific research – this model may not be totally relevant, but is a strong basis for future simulations, if more details about the network's structure were to be revealed.

We implemented the network layer of SpaceWire, as well as a part of the data link layer, to continue on the path we started during the first task. We successfully reused code from the first task, allowing a great gain of time, especially for the `SwitchUnit`, ports and channels that stayed nearly untouched, the only modifications being that `SwitchUnit` is now able to generate its routing table automatically and that channels can simulate error transmission simulation. We wrote the `Packet` class, allowing logical separation between data link layer and network layer.

With the `Node` class, we created a way to describe SpaceWire RMAP-using nodes, with the only main limitation being that these nodes only have one port, which could be modified in the future, and describe them using a configuration file written with the JSON format. After separation of our network in two parts, we parallelised its simulation, allowing for a significant increase of performances when simulating without using databases. We implemented text logging and database logging, allowing great possibilities of analysis of a simulation, with the only limitation with database logging being that it removes the interest of running the simulation in parallel. And finally, we began creating a Graphical User Interface, which would allow for simple way to design any SpaceWire network.

This model, while not complete, allowed us to discover things about the key points of the ExoMars2020 rover's network. The order of magnitude of the transmission time for the heaviest packets is of a few seconds, using channels with a speed of $24\,\mathrm{Mb/s}$. The rover being on Mars and carrying scientific experimentation, this result is correct. Indeed, there is no need for extreme performance in this situation, the most important feature being safety of transmissions. When simulating using error simulation in channels, errors were always detected and solved by sending the buffered packet again.

But further ameliorations could still be really interesting, starting with the GUI, that we didn't have time to finish. It could evolve to a generalised SpaceWire network simulator written with SystemC, with the special feature of creating multiple network parts that can run concurrently. In that event, we should switch from SQlite to MongoDB for example, or at least another database that support parallel writes.

We should also explore solutions to the `bad_alloc` error, maybe by using in real time a database to store all packet and node informations during runtime, thus using less memory.

We could also continue our work on the class `Node` to allow it to have multiple ports, which is allowed by the SpaceWire standard. Moreover, we could improve `SwitchUnit` so that it could connect to other routers automatically and modify its routing table automatically, since this feature is only available when working with nodes for the moment. We may also add some algorithm, such as Dijkstra's algorithm to find preferred route and only using best routes in the routing tables.

To finish, the authors would like to acknowledge the support of Nikolai Sinyov, our tutor during the development of this project for his advices and availability.

References

[1] Steve Parkes and Chris McClements. *SpaceWire Remote Memory Access Protocol*. University of Dundee, 2005. link.

[2] PanCam team. The PanCam Instrument for the ExoMars Rover. *Astrobiology*, 17(6 and 7), 2017. link.

[3] ISEM team. Infrared Spectrometer for ExoMars: A Mast-Mounted Instrument for the Rover. *Astrobiology*, 17(6 and 7), 2017. link.

[4] CLUPI team. The Close-Up Imager Onboard the ESA ExoMars Rover: Objectives, Description, Operations, and Science Validation Activities. *Astrobiology*, 17(6 and 7), 2017. link.

[5] WISDOM team. The WISDOM Radar: Unveiling the Subsurface Beneath the ExoMars Rover and Identifying the Best Locations for Drilling. *Astrobiology*, 17(6 and 7), 2017. link.

[6] ADRON team. The ADRON-RM Instrument Onboard the ExoMars Rover. *Astrobiology*, 17(6 and 7), 2017. link.

[7] Ma_MISS team. Ma_MISS on Exomars: Mineralogical Characterization of the Martian Subsurface. *Astrobiology*, 17(6 and 7), 2017. link.

[8] MicrOmega team. The MicrOmega Investigation Onboard ExoMars. *Astrobiology*, 17(6 and 7), 2017. link.

[9] MOMA team. The Mars Organic Molecule Analyzer (MOMA) Instrument: Characterization of Organic Material in Martian Sediments. *Astrobiology*, 17(6 and 7), 2017. link.

[10] RLS team. The Raman Laser Spectrometer for the ExoMars Rover Mission to Mars. *Astrobiology*, 17(6 and 7), 2017. link.

[11] Steve Parkes. *SpaceWire User's Guide*. STAR-Dundee, 2012. link.