Event-Oriented Simulation Module for Dynamic Elastic Optical Networks with Space Division Multiplexing

Mirko Zitkovich[1][a], Gabriel Saavedra[2][b] and Danilo Bórquez-Paredes[1][c]
[1] *Faculty of Engineering and Sciences, Universidad Adolfo Ibáñez, Av. Padre Hurtado 750, Viña del Mar 2520001, Chile*
[2] *Electrical Engineering Department, Universidad de Concepción, Víctor Lamas 1290, Concepción 4070409, Chile*

Keywords: C++ library, EON, Performance Evaluation, SDM, Event-Oriented Simulation, Traffic Engineering.

Abstract: It is well-known that creating Space Division Multiplexing-Elastic Optical Networks (SDM-EON) allocation algorithms can be challenging, especially without the right tools. This paper presents the development of a module of an event-oriented simulator designed to code C++ allocation algorithms in the context of Space Division Multiplexing-Elastic Optical Networks. The built module was tested and validated using an allocation algorithm previously published in the literature. The results were consistent with those in the original article, indicating that the module developed is effective and reliable. The use of specialized tools, such as the module being shown, can significantly increase the effectiveness and precision of this process and can stimulate additional developments in the telecommunications industry.

1 INTRODUCTION

The demand for bandwidth worldwide has been growing over time (Roser et al., 2020), and this trend is unlikely to change. One of the enabling technologies supporting this growth is elastic optical networks (EON) (Gerstel et al., 2012).

EONs have introduced to the fiber the ability to divide the spectrum into small portions called Frequency Slots Units (FSU) (Gerstel et al., 2012), and therefore, each connection uses the spectrum more efficiently. However, the set of FSUs must be contiguous (one slot next to the other) and continuous (the same set of slots) on all links along the route. In addition, each connection can use a specific modulation format, allowing the transfer of more information per FSU at the cost of reduced optical reach (Jinno, 2017).

Space division multiplexing (SDM) is among the potential candidates to extend even more the capacity of optical networks (Puttnam et al., 2021), currently limited by the nonlinear nature of silica. SDM transmits information multiplexed in different spatial channels over a given transmission medium. The main candidates for implementing SDM in optical networks are multi-core (MCF), few-mode, multi-mode fiber, and combinations thereof (Winzer et al., 2018). Each transmission medium proposed for SDM presents physical impairments that need to be considered when allocating resources in an optical network, as they might detrimentally affect other connections (Klinkowski et al., 2018). In particular, for MCF, inter-core cross-talk (XT) transfers energy between adjacent cores at the same wavelength as the propagating signal (Gené and Winzer, 2019).

A commonly explored area is dynamic elastic optical networks. Here, connections are allocated and released from the network following random distributions. Within this context, a typical problem is Routing, Modulation Format, and Spectrum Assignment (RMSA), which consists of finding the route and modulation format and employing some spectrum allocation policy to assign the connection to the specific spectrum of the network. The problem is extended for the new SDM technology by adding the choice of the core to be used for the connection. Thus the problem changes to Routing, Modulation Format, Core selection, and Spectrum Assignment (RMCSA), adding the C for the core choice. Ad-hoc simulators are typically used to measure some metric related to these resource allocation policies, given the complexity of implementing a hardware testbed.

The current availability of simulators is not sufficient, and there is still a demand for more specialized tools that cater specifically to the development and

[a] https://orcid.org/0009-0003-2454-4264
[b] https://orcid.org/0000-0002-5450-3661
[c] https://orcid.org/0000-0001-6590-2329

Zitkovich, M., Saavedra, G. and BÃşrquez-Paredes, D.
Event-Oriented Simulation Module for Dynamic Elastic Optical Networks with Space Division Multiplexing.
DOI: 10.5220/0012084500003546
In *Proceedings of the 13th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2023)*, pages 295-302
ISBN: 978-989-758-668-2; ISSN: 2184-2841

testing of allocation algorithms for SDM-EONs. Addressing this need, the article presents an SDM extension for a flexible optical networks simulator library called Flex Net Sim (Falcón et al., 2021). The extension aims to provide a robust and effective means for coding and validating C++ allocation algorithms for SDM-EONs, allowing the researcher to code RM-CSA algorithms quickly, taking care of the allocation policies, and leaving the simulation details to the library.

The new module introduces significant changes to two of the main components of the library, along with minor modifications to the remaining components. These changes focus on storing link information by adding a new dimension for cores, enhancing how the simulator manages certain network information, and allocating connection requests. Additionally, user-friendly auxiliary macros have been incorporated to facilitate straightforward programming.

This article is organized as follows: First, the "State of the Art" section reviews existing simulation tools. Next, the "Architecture" section outlines our proposed solution and discusses the models and heuristics in the "Model and Heuristics Used" section. The "Specific Problem Implementation" section details practical aspects, while the "Simulation Results" section presents the tool's performance. Finally, the "Conclusions" section summarizes our contributions and findings.

2 STATE OF THE ART

In the field of EONs, there is a limited selection of simulation tools available. Through our research, we could identify only a few, such as EONS (Delvalle et al., 2016), ElasticO++ (Tessinari et al., 2016), FlexGridSim (Moura and Drummond, 2020), CEONS (Aibin and Blazejewski, 2015), and SimEON (Cavalcante et al., 2017).

Regarding SDM specifically, even fewer tools have been found. OMNeT++ is quite versatile and supports SDM; however, it has a steep learning curve due to its utilization of advanced programming concepts, which may be unfamiliar to some researchers. Other tools like VPItransmissionMaker and NS-3 offer SDM support but do not focus on allocation algorithms.

Comparatively to other simulators in the literature, Flex Net Sim is focused on resource allocation techniques in dynamic optical networks and offers tremendous flexibility when coding heuristics, modifying the topology, and adding or removing network resources. Incorporating a multicore optical network handling extension into the simulator is essential to fully utilize its adaptability to the new technology.

3 ARCHITECTURE

The simulation library Flex Net Sim consists of 4 main components that work together to model the system and generate output data: Network, Simulator, Controller, and Allocator, shown in Fig. 1.

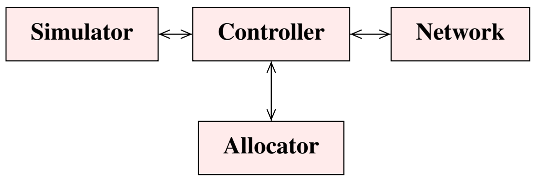


Figure 1: Representation of the architecture.

The library's Network component stores information about the network's nodes, links, slots, and methods for modifying the network state. These methods are atomic and only allow one link or slot modification at a time. This component does not work with lightpaths since it works at the level of links and nodes.

The controller component handles the network's global information, such as established network connections and routes between nodes. It provides methods for assigning or unassigning connections, modifying slot states, and setting connection lightpaths.

The simulator component creates connection requests, which are parameterized by the user, and sends them to the controller for allocation using the allocator component.

Finally, the user can actively contribute to the allocator component by creating the algorithm used for allocation, which is the only part of the simulator where user input is required.

There are two types of events: arrival and departure. Arrival occurs when a new connection attempt is made, while departure takes place when a connection releases previously allocated resources. During an arrival event, the simulator randomly generates a triad (source node, destination node, bitrate) and passes it to the Controller component. This component features a virtual function that invokes the researcher's resource allocation algorithm and returns a mapping of the resources to be assigned by the controller. The Controller then attempts to allocate resources within the network component. If successful, the simulator proceeds to the next event. Conversely, during a departure event, the connection ID to be released is identified, and the controller is responsible for instructing the network component to free up these resources.

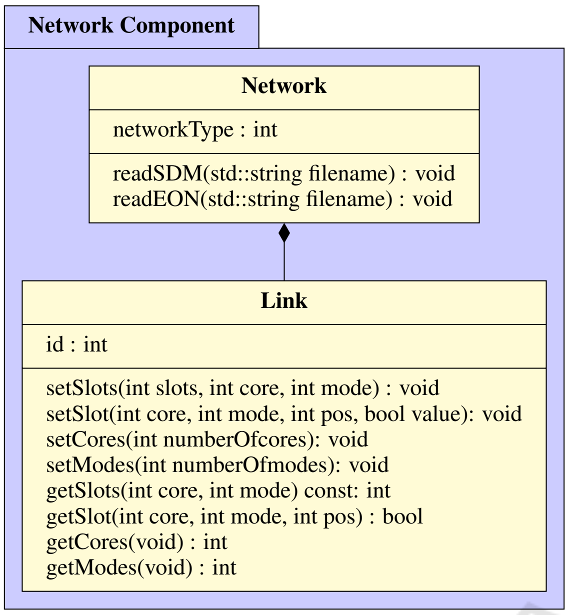

Network Component
Network
networkType : int
readSDM(std::string filename) : void
readEON(std::string filename) : void
Link
id : int
setSlots(int slots, int core, int mode) : void
setSlot(int core, int mode, int pos, bool value): void
setCores(int numberOfcores): void
setModes(int numberOfmodes): void
getSlots(int core, int mode) const: int
getSlot(int core, int mode, int pos) : bool
getCores(void) : int
getModes(void) : int

Figure 2: UML diagram of the network component.

To support SDM multi-core and/or multi-mode fiber, the main changes have been made in the network component, specifically in the link class. These changes allowed every link between each pair of nodes to have a variable number of modes and/or cores defined by the user through the methods shown in Fig. 2. Furthermore, adding cores/modes to the fiber introduces an extra layer of complexity to the problem, as connection establishment now varies depending on the network type. A `networkType` flag was incorporated into the network component to distinguish between different network types. The reading of user-provided files that indicate the topology will expect a certain format depending on this variable, using the methods `readEON` and `readSDM` shown in Fig. 2.

While the network component underwent the most significant changes, other components, such as the controller, were also modified. Depending on the declared network type at the beginning of the simulation, different methods are used in the controller to assign or unassign connections. An important addition was the inclusion of new `MACROS`, as `NUMBER_OF_CORES`, `NUMBER_OF_MODES`, and `ALLOC_SLOTS_SDM`. The former receives the route ID and the link ID as arguments and returns the number of cores in that link. The second `MACRO` receives the same arguments but returns the number of modes of the identified link. Finally, the latter receives as arguments the link ID, the core, the mode, and the starting and ending slot (assuming contiguity). This macro is in charge of changing the state of the slots in the identified link.

4 SPECIFIC PROBLEM IMPLEMENTATION

We use the two heuristics presented in (Tan et al., 2016) to test the module. The first is a Distance Adaptive RCSA (DA-RCSA), and the second is a First Fit RCSA (FF-RCSA). The main difference between the two is that the DA-RCSA uses different modulation formats, while the FF-RCSA only uses BPSK.

The network is represented by a directed graph $\mathcal{G}(\mathcal{N},\mathcal{L})$, where $\mathcal{N}$ is the set of nodes, and $\mathcal{L}$ is the set of optical links. Each link l_i is composed of a set C of cores, where each core c_j corresponds to the $j-th$ core. Each core c_j has a vector $\vec{S}$, representing the set of slots composing the usable spectrum. Each slot can be in either of two states, used or free. Also, each connection can be allocated using a modulation format $m \in M$.

All routes are pre-calculated using the K-shortest path algorithm for both algorithms. At startup, the DA-RCSA calculates the route length and, according to this value, selects the highest modulation format that meets the required optical range, i.e., the one that uses fewer slots. Then, the algorithm performs a FF on the core c to find candidate slots. Once found, calculate the XT of the current core as the candidate slots were used; if the threshold is met, the slots are set as used, and the connection is allocated. In the FF-RCSA algorithm , only BPSK is used. Therefore, the modulation format loop is not used in this pseudocode.

The fiber is composed of 7 cores: six around a center core. Both algorithms are implemented in three parts: crosstalk calculation, threshold verification, and main allocation algorithm. The codes can be found in the GitLab repository[1].

The library calculates by default the blocking probability of the allocation algorithms. We also added the maximum spectrum utilization of the fiber as an additional metric.

4.1 Crosstalk Calculation

We use the statistical mean XT of a homogeneous trench-assisted multi-core fiber that can be formulated with Equations 1, 2, and 3 (Muhammad et al., 2015):

[1]https://gitlab.com/DaniloBorquez/flex-net-sim/-/tree/master/examples/SDM

$$h = \frac{2k^2 r}{\beta w_{tr}} \quad (1)$$

$$XT = \frac{n - n \cdot exp[-(n+1) \cdot 2 \cdot h \cdot L]}{1 + n \cdot exp[-(n+1) \cdot 2 \cdot h \cdot L]} \quad (2)$$

$$XT_{dB} = 10 \cdot \log_{10} XT \quad (3)$$

Where h is the mean crosstalk increase by unit length. k, r, β, and w_{tr} are the coupling coefficient, bend radius, propagation constant, and core pitch, respectively. XT corresponds to the mean crosstalk, using h previously calculated, and n and L, where n is the number of adjacent active cores (which are the cores using slots on the same position), and L is the path length, measured in *meters*. Finally, XT_{dB} corresponds to the threshold in decibels.

The allocation process must satisfy a crosstalk threshold to complete the resource assignment. This means the XT cannot surpass a specific value to establish a successful connection.

4.2 Main Allocation Algorithms

Codes 1 and 2 implement DA-RCSA and FF-RCSA. The code must begin with `BEGIN_ALLOC_FUNCTION` reserved word with the name of the allocation algorithm as a parameter, and it must finish with the reserved word `END_ALLOC_FUNCTION`.

Code 1: DA-RCSA.

```
1   BEGIN_ALLOC_FUNCTION(DA) {
2     int currentNumberSlots;
3     int currentSlotIndex;
4     int bitrateInt = bitRates_map[REQ_BITRATE];
5     int numberOfSlots;
6     std::vector<bool> totalSlots;
7     for (int r = 0; r < NUMBER_OF_ROUTES; r++){
8       for (int c = 0; c < NUMBER_OF_CORES(r, 0); c++)
          ↪ {
9         double routeLength = 0;
10        totalSlots = std::vector<bool>(LINK_IN_ROUTE
            ↪ (r, 0)−>getSlots(), false);
11        for (int l = 0; l < NUMBER_OF_LINKS(r); l++){
12          routeLength += LINK_IN_ROUTE(r,l)−>
              ↪ getLength();
13          for (int s = 0; s < LINK_IN_ROUTE(r, l)−>
              ↪ getSlots(); s++){
14            totalSlots[s] = totalSlots[s] |
                ↪ LINK_IN_ROUTE(r, l)−>getSlot(c,
                ↪ 0, s);
15          }
16        }
17        for (int m = 0; m <
            ↪ NUMBER_OF_MODULATIONS; m++)
            ↪ {
18          numberOfSlots = REQ_SLOTS(m);
19          if (routeLength > REQ_REACH(m)) continue;
20          currentNumberSlots = 0;
21          currentSlotIndex = 0;
22          for (int s = 0; s < totalSlots.size(); s++) {
23            if (totalSlots[s] == false) {
24              currentNumberSlots++;
25            } else {
26              currentNumberSlots = 0;
27              currentSlotIndex = s + 1;
28            }
29            if (currentNumberSlots >= numberOfSlots) {
30              double XT_treshold =
                  ↪ XT_treshold_by_bitrate[m];
31              if (!isOverTreshold(c, sim.getPaths()−>at(
                  ↪ SRC)[DST][r], numberOfSlots,
                  ↪ currentSlotIndex, routeLength,
                  ↪ XT_treshold)){
32                for (int l = 0; l < NUMBER_OF_LINKS(
                    ↪ r); l++) {
33                  ALLOC_SLOTS_SDM(
                      ↪ LINK_IN_ROUTE_ID(r, l), c,
                      ↪ 0, currentSlotIndex,
                      ↪ numberOfSlots)
34                }
35                currentUtilization = currentUtilization + (
                    ↪ numberOfSlots ∗
                    ↪ NUMBER_OF_LINKS(r));
36                if (currentUtilization/totalCapacity >
                    ↪ maxUtilization) maxUtilization
                    ↪ = currentUtilization/
                    ↪ totalCapacity;
37                return ALLOCATED;
38              } else {
39                currentSlotIndex++;
40              }
41            }
42          }
43        }
44      }
45    }
46    return NOT_ALLOCATED;
47  }
48  END_ALLOC_FUNCTION
```

Lines 2-6 in Code 1 initialize auxiliary variables. Loops in lines 8-9 are used to go through the routes and cores, respectively. Lines 12-17 create an auxiliary vector representing the utilization of all links in the current route in terms of slots, representing a transparent connection. Lines 18-29 execute the FF algorithm, searching for contiguous slots in the auxiliary vector previously defined. Line 30-32 check that the requested number of slots is satisfied and then check the threshold. If the threshold is satisfied, Lines 33-35 allocate the resources to the current core. Line 37 updated the extra metric that we calculated. Line 38 returns that the allocation process was successful (`ALLOCATED`). Finally, if the algorithm tries all routes/cores and does not find enough space to allocate, it returns `NOT_ALLOCATED` in line 47.

Code 2: FF-RCSA.

```
1  BEGIN_ALLOC_FUNCTION(FF) {
2    int currentNumberSlots;
3    int currentSlotIndex;
4    int bitrateInt = bitRates_map[
         ↪ REQ_BITRATE];
5    int numberOfSlots;
6    std::vector<bool> totalSlots;
7    int BPSK = 2;
8    for (int r = 0; r < NUMBER_OF_ROUTES; r
         ↪ ++){
9      for (int c = 0; c < NUMBER_OF_CORES(r,
           ↪ 0); c++){
10       double routeLength = 0;
11       totalSlots = std::vector<bool>(
             ↪ LINK_IN_ROUTE(r, 0)->getSlots
             ↪ (), false);
12       for (int l = 0; l < NUMBER_OF_LINKS(r
             ↪ ); l++){
13         routeLength += LINK_IN_ROUTE(r,l)->
               ↪ getLength();
14         for (int s = 0; s < LINK_IN_ROUTE(r
               ↪ , l)->getSlots(); s++){
15           totalSlots[s] = totalSlots[s] |
                 ↪ LINK_IN_ROUTE(r, l)->
                 ↪ getSlot(c, 0, s);
16         }
17       }
18       numberOfSlots = REQ_SLOTS(BPSK);
19       if (routeLength > REQ_REACH(BPSK))
             ↪ continue;
20       currentNumberSlots = 0;
21       currentSlotIndex = 0;
22       for (int s = 0; s < totalSlots.size()
             ↪ ; s++) {
23         if (totalSlots[s] == false) {
24           currentNumberSlots++;
25         } else {
26           currentNumberSlots = 0;
27           currentSlotIndex = s + 1;
28         }
29         if (currentNumberSlots >=
               ↪ numberOfSlots) {
30           double XT_treshold =
                 ↪ XT_treshold_by_bitrate[
                 ↪ BPSK];
31           if (!isOverTreshold(c, sim.
                 ↪ getPaths()->at(SRC)[DST][r
                 ↪ ], numberOfSlots,
                 ↪ currentSlotIndex,
                 ↪ routeLength, XT_treshold))
                 ↪ {
32             for (int l = 0; l <
                   ↪ NUMBER_OF_LINKS(r); l++)
                   ↪  {
33               ALLOC_SLOTS_SDM(
                     ↪ LINK_IN_ROUTE_ID(r, l)
                     ↪ , c, 0,
                     ↪ currentSlotIndex,
                     ↪ numberOfSlots)
34             }
35             currentUtilization =
                   ↪ currentUtilization + (
                   ↪ numberOfSlots *
                   ↪ NUMBER_OF_LINKS(r));
36             if (currentUtilization/
                   ↪ totalCapacity >
                   ↪ maxUtilization)
                   ↪ maxUtilization =
                   ↪ currentUtilization/
                   ↪ totalCapacity;
37             return ALLOCATED;
38           } else {
39             currentSlotIndex++;
40           }
41         }
42       }
43     }
44   }
45   return NOT_ALLOCATED;
46 }
47 END_ALLOC_FUNCTION
```

The Code 2 is similar to Code 1, and shows the implementation of FF-RCSA. The main difference is presented in the modulation part. The code shown for the DA algorithm tries different modulation formats to allocate a connection, while FF algorithms only try Binary Phase-Shift Keying (BPSK). So, lines 18-20 of Code 1, where the modulation format is variable, are replaced by lines 19-20 in Code 2, where this variable takes a fixed value of BPSK. The rest of the code is the same as Code 1.

4.3 Main Program Code

The main program is shown in Code 3. Line 1 is the Flex Net Sim library included. Lines 3-9 are auxiliary variables. The function shown in line 11 is triggered after each connection departure. We use this function to update the auxiliary variables concerning maximum utilization.

Code 3: Main simulation code.

```
1  #include "./simulator.hpp"
2  #define DA_RCSA 0
3  #define FF_RCSA 1
4  double lambdas[10] = {100, 200, 300, 400,
       ↪ 500, 600, 700, 800, 900, 1000};
5  int active_algorithm = DA_RCSA;
6  int numberConnections = 1e5;
7  double currentUtilization = 0;
8  double maxUtilization = 0;
9  BEGIN_UNALLOC_CALLBACK_FUNCTION {
10   currentUtilization = currentUtilization
         ↪ - (c.getSlots()[0].size() * c.
         ↪ getLinks().size());
11   if (currentUtilization/totalCapacity >
         ↪ maxUtilization) maxUtilization =
              ↪ currentUtilization/totalCapacity;
12   }
13   int main(int argc, char* argv[]) {
14     for (int lambda = 0; lambda < sizeof(
              ↪ lambdas)/sizeof(double); lambda
              ↪ ++) {
15       sim = Simulator(std::string("./networks
                ↪ /NSFNet.json"),
16                       std::string("./networks/
                              ↪ NSFNet_routes.
                              ↪ json"),
17                       std::string("./networks/
                              ↪ bitrates.json"),
18                       SDM);
19       if (active_algorithm == DA_RCSA)
                ↪ USE_ALLOC_FUNCTION(DA, sim)
20       else USE_ALLOC_FUNCTION(FF, sim)
21
22       USE_UNALLOC_FUNCTION_SDM(sim);
23       sim.setGoalConnections(
                ↪ numberConnections);
24       sim.setLambda(lambdas[lambda]);
25       sim.setMu(1);
26       sim.init();
27       sim.run();
28
29       currentUtilization = 0;
30       maxUtilization = 0;
31     }
32     return 0;
33   }
```

Line 16 starts the program, composed of a loop, to get the blocking probability and maximum utilization of each traffic load. Lines 18-21 create a simulator object with a specific network, routes for that network, bitrates to be used, and the SDM flag. Lines 23-24 determine which algorithm will be used, which in this case is DA_RCSA. Line 26 sets the algorithm for each connection departure for the Simulator object. Lines 27-30 sets the principal parameters of the simulation and initialize it. Finally, Line 32 executes the simulation, and lines 34-35 reset auxiliary variables to be recalculated on the next loop.

4.4 Network Files

The simulator receives three files for the construction of the network, and it requests. To explain its structure and for the sake of space, we present a 3-node network. An example of the first file can be seen in Code 4, and it corresponds to the structure of the network, specifies the number of nodes and their id, as well as the links that connect them and their characteristics: length, id, number of cores, number of modes and their respective slots. The slots are specified in a list of lists where the first dimension corresponds to the core and the second dimension to the mode, i.e., if you want to specify that the number of slots of the first core with three modes is 30, it would be $slots = [[30, 30, 30], \ldots]$. The example shows a 3-node 7-core single-mode network.

Code 4: Network JSON file.

```
{
    "alias": "NetExample",
    "name": "Network_Example",
    "nodes": [
        {"id": 0}, {"id": 1}, {"id": 2}
    ],
    "links": [
        {"id": 0, "src": 1, "dst": 0,
            "length": 500,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]]
        },{ "id": 1, "src": 0, "dst": 1,
            "length": 500,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]]
        },{ "id": 2, "src": 2, "dst": 1,
            "length": 200,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]]
        },{ "id": 3, "src": 1, "dst": 2,
            "length": 200,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]]
        },{ "id": 4, "src": 0, "dst": 2,
            "length": 600,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]]
        },{ "id": 5, "src": 2, "dst": 0,
            "length": 600,
            "number_of_cores": 7,
            "number_of_modes": 1,
            "slots": [[80], [80], [80], [80
                ↪ ], [80], [80], [80]],

        }
    ]
}
```

The second file corresponds to the pre-calculated paths between each pair of nodes. These routes are unidirectional, so setting all the routes between nodes 0 and 1 will not establish the routes between nodes 1 and 0. It is essential to mention that the order of the paths in the file defines the order in which they will be iterated in the allocating algorithm. Code 5 shows an example of the paths between the 3-node network.

Code 5: Routes JSON file.

```
{
    "name": "Network␣Example",
    "alias": "NetExample",
    "routes": [
        {"src": 0, "dst": 1,
            "paths": [[0, 1], [0, 2, 1]]
        },{ "src": 0, "dst": 2,
            "paths": [[0, 2], [0, 1, 2]]
        },{ "src": 1, "dst": 0,
            "paths": [[1, 0], [1, 2, 0]]
        },{ "src": 1, "dst": 2,
            "paths": [[1, 2], [1, 0, 2]]
        },{ "src": 2, "dst": 0,
            "paths": [[2, 0], [2, 1, 0]]
        }
    ]
}
```

The third file corresponds to the bitrates and modulation format requests. Each modulation has its name, reach, and the required number of slots. Like the routes, the order in which the modulation formats are placed in the JSON file determines the order in which they will be iterated in the allocating algorithm. Code 6 shows an example of two bitrates $\{40, 100\}$ each with two modulation formats $\{$16-QAM, QPSK$\}$

Code 6: Bitrates JSON file.

```
{
    "40": [
        {"16QAM": {"slots":1, "reach":1500},
         "QPSK": {"slots":2, "reach":4000}
        }
    ], "100": [
        {"16QAM": {"slots":2, "reach":1500},
         "QPSK": {"slots":4, "reach":4000}
        }
    ]
}
```

5 SIMULATION RESULTS

The simulation was conducted in a Windows Subsystem for Linux Version 2 (Windows 10) installed on an HDD using an Intel Core i5 9400F, 3.90 GHz, and 16 GB of RAM, obtaining a total CPU execution time equal to 20 seconds for the FF-RCSA algorithm and 22 seconds for the DA-RCSA algorithm. We simulate 10^5 connections in the NSFNet topology (14 nodes, 44 unidirectional links) shown in Fig. 3. Each fiber has seven cores and 80 spectrum slots.Fiber parameters used by (Tan et al., 2016) are $0.05m$,$4 \times 10^6 1/m$,4×10^{-4}, and 4×10^{-5}, for bending radius, propagation constant, coupling coefficient, and core pitch, respectively. Cross-talk thresholds used are $-14dB$, $-18.5dB$, and $-25dB$ for BPSK, QPSK, and 16-QAM modulation formats, respectively.

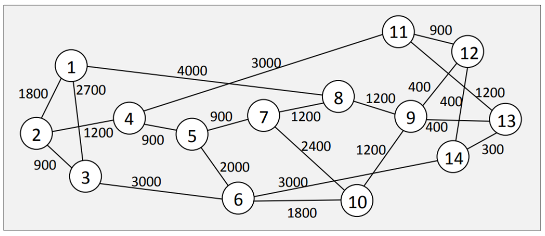


Figure 3: NSFNet topology with distance in km (Tan et al., 2016).

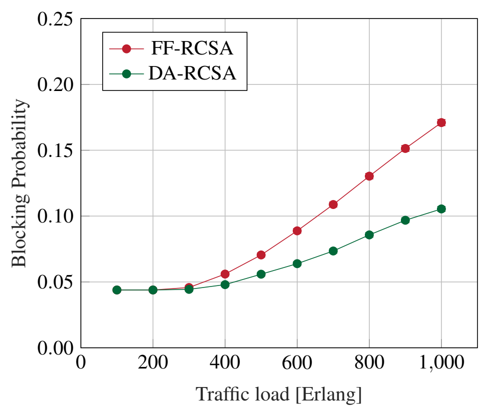
The bitrates were uniformly distributed between $B = \{40, 100, 200\}$ Gbps, and a total of three modulation formats were used (BPSK, QPSK, 16-QAM), with their respective optical reach and frequency slot capacity (Table 1 at (Tan et al., 2016)).

Fig. 4a shows the blocking probability of both algorithms. The results show that the use of the DA-RCSA algorithm performs better compared to the FF-RCSA. Fig. 4b displays the maximum resource utilization of the network. These results also show that the DA-RCSA algorithm alleviates network usage using different modulation formats compared to FF-RCSA. Although all the results obtained are consistent with those at (Tan et al., 2016), they are not exactly the same, and this may be due to several reasons. Firstly, as there is a generation of random variables involved (to simulate the arrival of the connections), it is improbable to replicate the exact same results, either because it was programmed differently or because there is no access to the seed used.

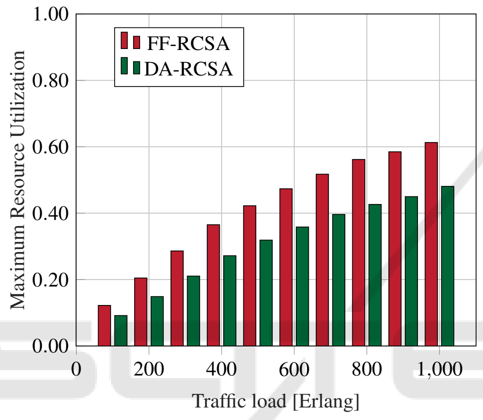
On the other hand, the number of connections used by the researchers was 5,000 compared to the 100,000 used during our simulations. Given that the higher the number of simulated connections, the better the confidence interval of the metrics obtained, we can state that our values are more reliable. Despite these differences, it could be said that the proportions between the two algorithms are quite similar, which would lead us to conclude the same as the researchers.

6 CONCLUSION

In this paper, we presented a module for the FlexNet-Sim C++ library that extends its support for using SDM. The results obtained are consistent with the reference example presented. With the addition of this module, researchers and developers can now investigate and expand their knowledge of SDM-enabled elastic optical networks using the FlexNet-Sim C++ library.



(a) Blocking probability



(b) Resource utilization

Figure 4: Numerical results.

Future work is expected to implement multiband in multi-core/mode modules to enhance the simulator's capabilities further. There is also the possibility of implementing other types of events used in optical network research, such as fragmentation and fault tolerance.

ACKNOWLEDGEMENTS

This work received financial support from ANID FONDECYT Iniciación 11220650 and Fondecyt 1231826.

REFERENCES

Aibin, M. and Blazejewski, M. (2015). Complex elastic optical network simulator (ceons). In *2015 17th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4.

Cavalcante, M. A., Pereira, H. A., and Almeida, R. C. (2017). SimEON: an open-source elastic optical network simulator for academic and industrial purposes. *Photonic Network Communications*, 34(2):193–201.

Delvalle, L., Alfonzo, E., and Roa, D. P. P. (2016). Eons: An online rsa simulator for elastic optical networks. In *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–12.

Falcón, F., España, G., and Bórquez-Paredes, D. (2021). Flex net sim: A lightly manual.

Gené, J. M. and Winzer, P. J. (2019). A universal specification for multicore fiber crosstalk. *IEEE Photonics Technology Letters*, 31(9):673–676.

Gerstel, O., Jinno, M., Lord, A., and Yoo, S. B. (2012). Elastic optical networking: a new dawn for the optical layer? *IEEE Communications Magazine*, 50(2):s12–s20.

Jinno, M. (2017). Elastic optical networking: Roles and benefits in beyond 100-gb/s era. *Journal of Lightwave Technology*, 35(5):1116–1124.

Klinkowski, M., Lechowicz, P., and Walkowiak, K. (2018). A study on the impact of inter-core crosstalk on sdm network performance. In *2018 International Conference on Computing, Networking and Communications (ICNC)*, pages 404–408.

Moura, P. M. and Drummond, A. C. (2020). FlexGridSim: Flexible Grid Optical Network Simulator. http://www.lrc.ic.unicamp.br/FlexGridSim/.

Muhammad, A., Zervas, G., and Forchheimer, R. (2015). Resource allocation for space-division multiplexing: Optical white box versus optical black box networking. *Journal of Lightwave Technology*, 33(23):4928–4941.

Puttnam, B. J., Rademacher, G., and Luís, R. S. (2021). Space-division multiplexing for optical fiber communications. *Optica*, 8(9):1186–1203.

Roser, M., Ritchie, H., and Ortiz-Ospina, E. (2020). Internet. *Our World in Data*. https://ourworldindata.org/internet.

Tan, Y., Yang, H., Zhu, R., Zhao, Y., Zhang, J., Liu, Z., Ou, Q., and Zhou, Z. (2016). Distance adaptive routing, core and spectrum allocation in space division multiplexing optical networks with multi-core fibers. In *2016 Asia Communications and Photonics Conference (ACP)*, pages 1–3.

Tessinari, R. S., Puype, B., Colle, D., and Garcia, A. S. (2016). Elastico++: An elastic optical network simulation framework for omnet++. *Optical Switching and Networking*, 22:95 – 104.

Winzer, P. J., Neilson, D. T., and Chraplyvy, A. R. (2018). Fiber-optic transmission and networking: the previous 20 and the next 20 years (invited). *Opt. Express*, 26(18):24190–24239.