A resource usage efficient distributed allocation algorithm for 5G Service Function Chains

Guillaume Fraysse, Jonathan Lejeune, Julien Sopena, Pierre Sens

A resource usage efficient distributed allocation algorithm for 5G Service Function Chains

Guillaume Fraysse[1,2][0000−0001−8269−4701],
Jonathan Lejeune[2][0000−0001−6922−2451], Julien Sopena[2[]], and
Pierre Sens[2][0000−0002−5156−7715]

[1] Orange
guillaume.fraysse@orange.com
[2] Sorbonne Université, CNRS, Inria, LIP6, F-75005, Paris, France
firstname.lastname@lip6.fr

Abstract. Recent evolution of networks introduce new challenges for the allocation of resources. Slicing in 5G networks allows multiple users to share a common infrastructure and the chaining of Network Function (NF)s introduces constraints on the order in which NFs are allocated. We first model the allocation of resources for Chains of NFs in 5G Slices. Then we introduce a distributed mutual exclusion algorithm to address the problem of the allocation of resources. We show with selected metrics that choosing an order of allocation of the resources that differs from the order in which resources are used can give better performances. We then show experimental results where we improve the usage rate of resources by more than 20% compared to the baseline algorithm in some cases. The experiments run on our own simulator based on SimGrid.

Keywords: Computer network management · Distributed algorithms · Network Slicing · Distributed Systems · k-mutex · drinking philosophers · deadlock.

1 Introduction

The flexibility of 5G networks allows the apparition of new services. Complex services rely on Slices split across multiple Network Service Provider (NSP)s. The allocation of a service is now not only the allocation of a single NF but the chaining of multiple NFs. Chains of NFs introduce a constraint on the order of the allocation.

We argue that in such use cases the system can be modeled as a distributed system. The various NFs from the different NSPs can be abstracted as **resources**. One or more **instances** of these resources can be available for each of these resources. We define the allocation of a chain of NFs as a **request** of a **set of resources** with an associated **request order**. The allocation of a request means allocating all the resources in the set while respecting this request order.

In distributed systems, allocation of resources is seen as a **Mutual Exclusion** problem. Several variants of the problem have been defined considering a

single resource [14, 27] or multiple instances of a single type of resource [25, 26]. Very few solutions have been proposed for the problem of Mutual Exclusion for systems with multiple instances of multiple resources [2, 15].

We propose here a distributed algorithm to solve the problem of the allocation of resources to create chains of NFs in 5G Slices for Multi-Domain use cases. We address this as a distributed Mutual Exclusion problem. We show that for systems with multiple instances of resources the selection of the instances to allocate has an influence on performances. Our algorithm extends the LASS algorithm [15] for systems with N instances of M resources. We introduce a subroutine to select the instance of the resource based on the orders of the requests as the LASS algorithm does not address this constraint. The algorithm is based on the transmission of a token that contains the permissions to use the resources. In a Network Functions Virtualization (NFV) network the resources are the nodes themselves, and they can't be transferred from one node to another. In such system each node is the manager of its own resource. We propose an extension to manage systems where the decisions to allocate the resources are made locally by each node.

We introduce a broader description of these use cases as well as some related work and background in Section 2. We describe our problem in Section 3. In Section 4 we introduce our algorithm, and define the allocation order as distinct from the order in the requests. We then introduce the methodology used to evaluate the algorithm with the SimGrid [3] simulator and show the experimental results in Section 5. We finally present our conclusions and future works in Section 6.

2 Related work

The architecture of Telecom Networks is rapidly evolving. Operators have launched the NFV [10] initiative at the European Telecommunications Standards Institute (ETSI) to move the NFs from dedicated hardware to virtualized infrastructures based on Cloud and Software-Defined Networking (SDN) technologies.

Allocation of a single NF is rarely sufficient. More complex services require multiple NFs to inter-operate while respecting an order. To this end ETSI defined the **VNF Forwarding Graph (VNF FG)s** [9] as the analogue to the connection of physical appliances with cables. Following the description of the VNF FG use case in 2013, RFC7665 [22] introduced in 2015 Service Function Chaining (SFC) to allow the allocation of a chain of NFs. In 2017, RFC8402 [23] introduced Segment Routing to allow a node to send a list of instructions to be run by subsequent nodes. problem of resources allocation for multiple NFs in the correct order.

5G is the first generation of mobile networks to include NFV as an enabler for new types of services. 3GPP has introduced **Network Slices** [29, 28] that enables multiple virtual end-to-end networks to share the same NFV infrastructures. A service offered by a Slice can rely on the infrastructures of multiple NSPs, it is then called **Multi-Domain**. This can be the case when a large op-

erator has split its network in multiple subdomains (e.g. datacenters) or when a use case may require the infrastructures of multiple operators.The European 5G Public Private Partnership (5G PPP) initiative launched projects that defined several use cases based on Slices and Multi-Domain. The SLICENET project's eHealth use case [29] requires multiple NSPs to provide Slices that are chained together to provide the service.

Multiple centralized solutions exist for the allocation of resources such as Virtual Network Function (VNF) in networks or the placement of Virtual Machines in Cloud infrastructure [20, 24, 11, 31]. These approaches focus on finding an optimal placement depending on a set of constraints. Some papers focus on finding heuristics to respect Service Level Agreement (SLA)s [17] or security rules [12]. These problems are mostly addressed with Integer Linear Programming (ILP) formulations. This centralized method may not be adequate for Multi-Domain use cases when it is not be possible to have a centralized manager or when the cost of building a global view of the system has a high cost. A centralized method also often requires an a priori knowledge of the requests.

We propose to address such systems as distributed systems, and propose a solution where there is no centralized manager. In such systems resources all execute the algorithms locally and get (resp. send) information from (resp. to) other resources by the passing of messages.

The allocation of resources in distributed systems can be handled as a Mutual Exclusion problem on these resources. The Mutual Exclusion is a fundamental problem in distributed systems and was first described by E. W. Dijkstra in 1965 [7] for systems where multiple processes try to allocate concurrently a single shared resource. Allocating this resource allows them to execute a portion of code known as **Critical Section (CS)** allowing processes to use the resource exclusively. Multiple solutions [14, 27, 30, 21] have been proposed.

The mutual exclusion problem was later generalized in two ways:

- for systems with **one instance of M resources** known as the dining philosophers problem, when the requests are static, and **drinking philosophers problem**, when the requests are dynamic. It was defined by K. M. Chandy and J. Misra in 1984 [4].
- for systems with k **instances of a single resource**, known as the k-mutex problem [25]. A variant of this problem is known as the k-out of-M resources allocation problem [26] when one process tries to allocate multiple instances of a single type of resource.

Algorithms to solve drinking philosophers problems, need to address potential **conflicts** between two requests. A conflict occurs when two requests try to allocate a common resource. If two requests don't conflict, they are allowed to enter their CS simultaneously.

The Dining/Drinking philosophers problem was generalized in 1990 by Awerbuch and Saks [1] as the *Dynamic Job Scheduling problem* where processes require resources that can be used by a single process at a time. A job can only be executed when all its required resources are available for exclusive use by the process. The problem is also related to the *job-shop* scheduling optimiza-

tion problem, in which n jobs have to be scheduled on m machines with the shortest-length schedule. This problem is NP-hard [16].

Algorithms addressing the mutual exclusion for systems with M resources can be divided into two groups: **incremental** algorithms and **simultaneous** algorithms. Algorithms in the first group incrementally allocate resources according to a static total order on the resources. They are using consecutive mutexes on each of the M resources. E. W. Dijkstra's algorithm from this group [6] is the baseline algorithm for our comparison and is detailed in Section 5.2. Algorithms in the second group do not set a predefined total order on the resources but try to simultaneously allocate resources for multiple requests. To achieve this multiple mechanisms have been proposed. Some require a knowledge of the conflict graph [4, 8]. Others rely on a broadcast mechanism with high messages complexity [18] or a global lock that is not efficient when the concurrency of requests is high [2]. All the simultaneous algorithms have in common to build a total order of the requests to schedule them.

Finally, it is possible to extend drinking philosopher and k-mutex problems by considering systems with N instances of M types of resources and requests for k instances of 1 or more types, we call it *the* $k - M - N$ *problem.*

In a system with N instances of M resources, it is necessary to decide which instance to allocate for a given request. Once an instance has been selected, we have a simplification of the $k - M - N$ problem to the drinking philosophers problem, where each instance is uniquely identified by its location. To the best of our knowledge this problem has not been specifically addressed. Some papers address the drinking philosophers problems and mentioned possible extension of their algorithms to systems with N instances [2, 15] but did not consider the selection of the instances as a specific constraint.

Algorithms from the state of the art don't consider the latency of the network links. They also do not address that the nodes selected for a chain of NFs need to respect a specific request order. In our model we add a weight to the edges of the graph to take this latency into consideration and be able to compute a path that respects the order in which resources are used. They also do not take into consideration that network links are not First In First Out (FIFO) channels. Our algorithm makes no assumption on the order in which messages are received.

The LASS algorithm [15] is a simultaneous algorithm that addresses systems with a single instance of M resources. It has been shown that its performance are better than those of incremental algorithms. It builds **allocation vectors** for all requests. These vectors are then used to compute a total order on requests, as detailed in section 3. Our algorithm extends it and includes a preemption mechanism that is used when messages are received in an order that is different from the total order of the requests.

3 Problem Statement

The allocation of resources for VNF Forwarding Graph (VNF FG) in Multi-Domain 5G slices is addressed as a Mutual Exclusion problem for systems with

N instances of M resources. In an example of VNF FG described in [9], packets need to traverse an Intrusion Detection Systems (IDS), a Firewall (FW) and a Load-Balancer (LB). The left part of Figure 1 shows this example with three NSPs.The figure shows the cases where there are three instances of each resource distributed across the three NSPs.

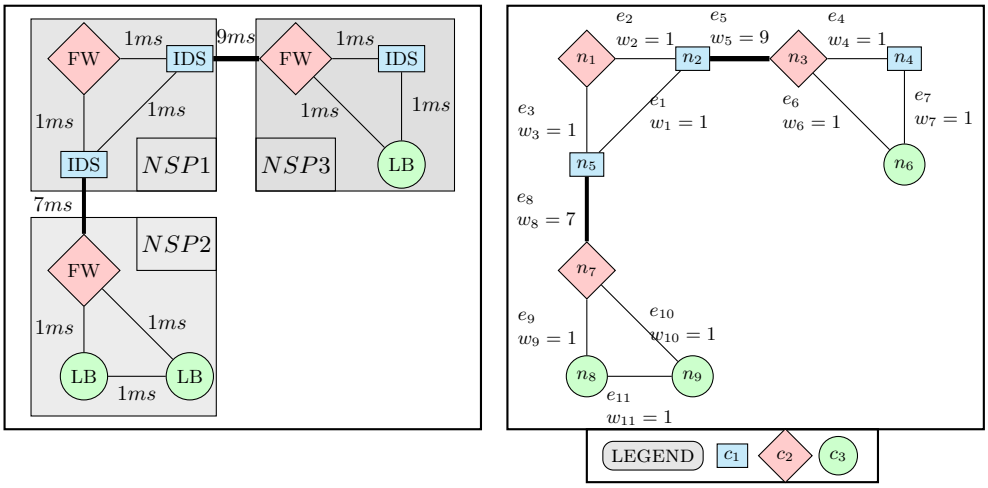


Fig. 1. A system with 3 types of resources: c_1 (IDS), c_2 (FW), and c_3 (LB)

We model our system as a non-directed connected static graph G=($\mathcal{N}, \mathcal{E}$) where $\mathcal{N}$ is the set of **nodes** and $\mathcal{E}$ the set of **edges**. A node contains at most one resource. Edges have positive **weights** that allow to model the latency of the links between nodes. A weight of 0 on an edge allows to model a system with multiple resources on a single node. A node with 2 resources can be modeled in the graph as two nodes holding one resource each and connected by a zero-weight edge. We note $\mathcal{C}$ the set of **types of resources**.

Each node in the graph can issue an allocation request. A request is modeled as a couple $Req(n, \{c_1, \ldots, c_s\})$ where :

- n is the **requesting node**,
- $\{c_1, \ldots, c_s\}$ where $c_r \in \mathcal{C}, \forall r$ is an ordered set of types of resources needed. The **request order** gives the order of the resources in the request. The order of resources can be different across requests.

The right side of Figure 1 shows a model of the use case introduced above. We pose a request $Req_1 = Req(n_1, \{c_3, c_2, c_1\})$ in this system.

n_1 is the requesting node, 3 types of resources c_1, c_2 and c_3 are requested. The request order is $c_3 \leq c_2 \leq c_1$, i.e. first c_3, then c_2 and finally c_1.

Our algorithm does not require a knowledge of the conflict graph like [4, 8]. It requires that each node has knowledge of its neighbors and knows where to find each type of resource so that each node can send messages to others.

The first subroutine of the algorithm presented in Section 4 computes a **path** in the graph. A path contains all the nodes from, and including, the requesting

node, to the last resource requested, respecting the request order. The path contains the nodes holding the requested resources as well as the nodes connecting those, e.g. a valid path for Req_1 is the ordered set of nodes $(n_1, n_5, n_7, n_8, n_7, n_5)$. The originating node is n_1, the first type of resource requested is c_3 and n_8 holds an instance of it. It is necessary to go through n_5 and n_7 to reach n_8 from n_1. Then n_7 and n_5 hold the two other requested resources.

A request is **satisfied** when the path contains nodes that hold all the types of resources in the correct request order and all the requested resources are allocated to it, allowing the requesting node to enter its CS.

The algorithm builds an **allocation vector** for each request. Each entry of an allocation vector is a pair $(n, value)$, called a **counter**, where $n \in \mathcal{N}$ is the node having the requested resource and *value* is a positive integer incremented by the node upon reception of the request, as detailed in Section 4. It is not possible for two requests to get the same counter value for a node n in their allocation vectors. The allocation vector for request Req_r is noted $V_{Req_r} = ((n_1^r, counter_{n_1}^r), \ldots, (n_s^r, counter_{n_s}^r))$ where r is the identifier of the request and s is the size of request Req_r, e.g. a valid allocation vector for Req_1 is $V_{Req_1} = ((n_8, 3), (n_7, 2), (n_5, 4))$. The allocation vectors allow the algorithm to compute a **total order of the requests**.

To sort the requests according to the global order we define the **precedence** of a request Req as its rank in the total order. Req_i precedes Req_j if the average of the counters of the allocation vector V_{Req_i} is less than the average of the counters of the allocation vector V_{Req_j}. For instance, if we consider $Req_2 = Req(n_4, \{c_3, c_2, c_1\})$ and $Req_3 = Req(n_6, \{c_1, c_2, c_3\})$ with $V_{Req_2} = ((n_6, 3), (n_3, 3), (n_4, 1))$ and $V_{Req_3} = ((n_4, 2), (n_3, 2), (n_6, 1))$ their respective allocation vectors, Req_3 will be allocated before Req_2 since $\frac{3+3+1}{3} > \frac{2+2+1}{3}$.

If two allocation vectors have the same average value, it is necessary to add a way to break the tie. No generic method is proposed here since it can be implementation-specific, but the method used in our experimental platform is detailed in section 5.3.

4 Algorithms

Our algorithm consists of two consecutive subroutines:

- the **path computation** subroutine, in which the algorithm selects the resources instances to be allocated and computes a routing path between them that respects the allocation order present in the request,
- the **allocation** subroutine in which the algorithm allocates the resources selected during the path computation subroutine.

4.1 Path computation subroutine

This subroutine assumes that each node of the system has some local knowledge on how to reach each type of resource. Each node keeps an up-to-date local routing table containing the name of the neighbor node that is the closest to

each type of resource as well as the distance to this node. The entry in the routing table can be the node itself for the type of resource it is holding. How these routing tables are built is out of scope of this paper. The solution used in our simulation is described in Section 5. Table 1 gives some routing tables for the system in Figure 1.

Type	Node	D
c_1	n_2	1
c_2	n_1	0
c_3	n_5	9

(a) n_1

Type	Node	D
c_1	n_5	0
c_2	n_1	1
c_3	n_8	8

(b) n_5

Type	Node	D
c_1	n_5	7
c_2	n_7	0
c_3	n_7	1

(c) n_7

Table 1. Routing tables for n_1, n_5 and n_7. D is for Distance.

In this example, node n_1's shortest path to the type of resource c_3 is of length 9 and starts at n_5: (n_1, n_5, n_7, n_8), and node n_5's shortest path to the type of resource c_3 is of length 8 and starts at n_7: (n_5, n_7, n_8).

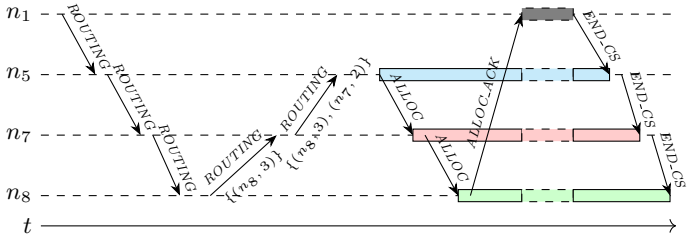


Fig. 2. Algorithm execution for Req_1

Path computation relies on the *ROUTING* message. A first *ROUTING* message is sent by the requesting node to the node holding the first resource requested, according to its routing table. This node then sends a *ROUTING* message to the next node on the path to the node holding the next resource. The operation is repeated on each node until a complete path with all the requested resources in the correct order has been found. Figure 2 shows the messages sent during the execution of the algorithm in the sample system.

This subroutine computes a valid path with the selected instances of all the requested resources in the order given in the request. It does not check that the resources are available nor does it guarantee that the path computed is the shortest path containing all the resources.

Once the algorithm has reached the node holding the last resource requested, the allocation subroutine described below starts from the last node of the path.

Building the allocation vector We compute a total order of the requests to preserve the **liveness** property, i.e. it guarantees that all requests are satisfied in a finite time. We use the method from the LASS algorithm to compute this order based on vectors build for each request.

The first step is to build the allocation vector for the request. The vector is built by each of the node on the computed path. Each node has a local counter that is initialized to 0. This counter acts as a logical clock, but contrary to Lamport's logical clocks [14] it is local and only incremented by the node when it receives a new request. As such it is then not possible for two requests to get the same counter value for a node in their allocation vector. The first request receives the value 1, the second receives the value 2, and so on. Upon reception of a request a node increments its local counter. It then updates the allocation vector with the value of its counter. The updated allocation vector is inserted in the request message when it is forwarded to the next node in the path. As a simplification this has been merged in our implementation with the *ROUTING* messages. Thus, the allocation vector is built during the forwarding of the *ROUTING* messages as shown in Figure 2.

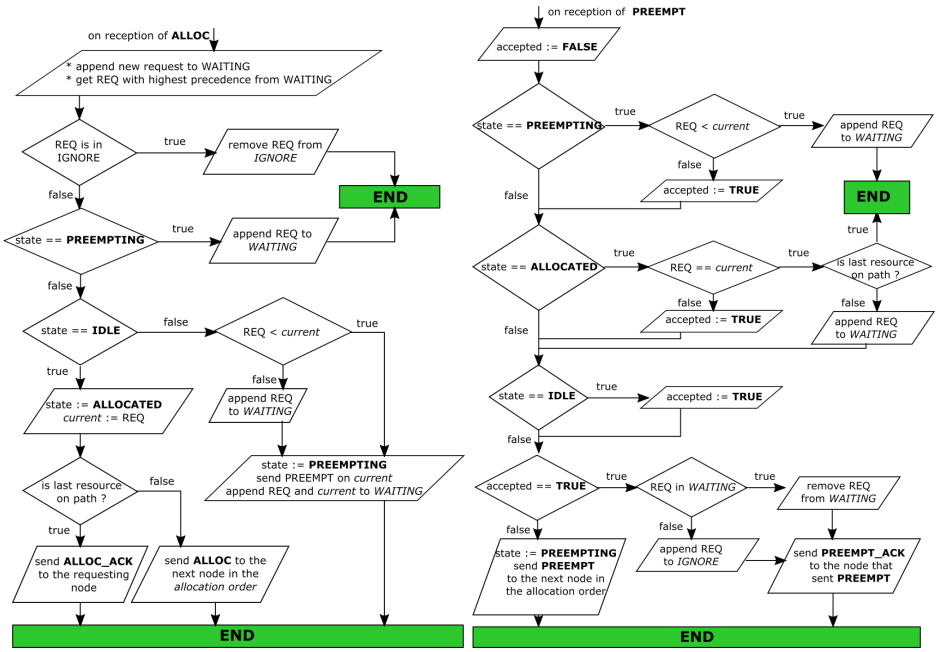
4.2 The allocation subroutine

The allocation subroutine allocates the instances of resources selected during the path computation subroutine. Nodes receive allocations messages in an arbitrary order, and it can be completely different from the order computed on the requests. To deal with this situation, the subroutine has a preemption mechanism to enforce the total order of the requests. Simulations show that the **allocation order** within a request has a strong impact on the performance of the algorithm, we compare here two allocation orders. All this is detailed below.

The allocation The core of the allocation subroutine is based on the *ALLOC* messages. In a system where all the resources are initially in the **IDLE** state, this message is sent to the first node in the *allocation order*, detailed below. This node enters the **ALLOCATED** state and sends an *ALLOC* message to the next node. The operation is then repeated until the last node in the *allocation order* is reached. Then the last node send an *ALLOC_ACK* message to the requesting node to inform it that the allocation of all resources has been made. It then enters its CS and starts using the resources. Upon leaving its CS it sends a *END_CS* message that is forwarded along the path to all the nodes holding the requested resources. The messages sent for the allocation of Req_1 are shown on Figure 2.

The allocation order Each request has an associated partial **request order** for the resources within the request, i.e. the order in which the resources are used, cf. Section 3. We define the **allocation order** as the order used by the algorithm to allocate the resources. There is no relation between the request order and the allocation order and they can be different for a same request. As a mean of comparison, we introduce two different allocation orders that are then evaluated in section 5.

The ***by values*** allocation order sends the *ALLOC* messages according to the values of the counters in the allocation vector. The last node of the path



(a) ***ALLOC* messages** (b) ***PREEMPT* messages**

Fig. 3. State diagrams for *ALLOC* and *PREEMPT* messages

sends a *ROUTING_ACK* message to the requesting node. Upon reception of this message the requesting node sends an *ALLOC* message to the node with the highest counter value in the allocation vector. The allocation then follows the order of the counters in the allocation vector. As an allocation based on this order starts by allocating the node with the highest counter value, it reduces the probability that requests with higher precedence will arrive during the rest of the allocation of the request.

The ***reverse order*** allocation order sends the *ALLOC* messages in the reverse order of the routing path. The first subroutine follows the path as it selects it. With this *reverse order*, the allocation follows the path backwards to go back to the requesting node.

For Req_1 with the computed path (n_1, n_5, n_7, n_8, n_7, n_5) if the allocation vector is $V_{Req_1} = ((n_8, 3), (n_7, 2), (n_5, 4))$, the allocation for *reverse order* follows the order n_5, n_7, n_8. For *by values* the order is n_5, n_8,n_7.

Preemption of resources Since the system is distributed a request can arrive on a node already in the **ALLOCATED** state for a request that has a lower precedence. This can lead to deadlocks. To manage these situations the algorithm preempts the resources to enforce the global order on the requests. This requires an additional state for the nodes, **PREEMPTING**, and two additional messages: *PREEMPT* and *PREEMPT_ACK* .

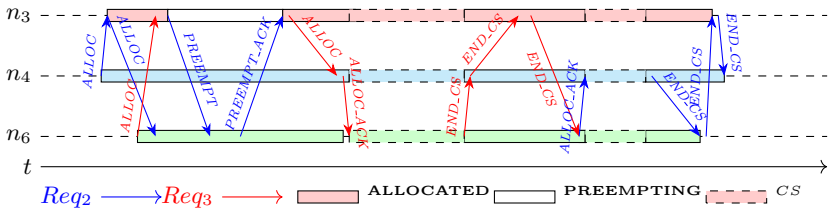


Fig. 4. Allocations for Req_2 and Req_3, preemption of n_6 by Req_2

The state diagram in Figure 3a shows how *ALLOC* messages are handled, Figure 3b shows how *PREEMPT* messages are handled. Apart from the case where the node is **IDLE** described above there are two other cases to consider. If a node is already **PREEMPTING**, or if it is **ALLOCATED** and the new request has a lower precedence than the request that currently holds the resource, then the request is stored in a local *WAITING* queue. Otherwise, i.e., when the node is **ALLOCATED** and the new request has a higher precedence than the request that currently holds the resource, the algorithm performs a preemption of the resource on the request that currently holds it, named *current*. To perform a preemption a node sends a *PREEMPT* message to the node that received its resource, i.e., the node to which it previously sent an *ALLOC* message for *current*. If the node that received the *PREEMPT* is not the last node in the path of *current*, it continues the preemption to put *current* on hold. For this it sends a *PREEMPT* message to the next node in the *allocation order* of *current*. *current* resumes later when it becomes the request with the highest precedence in the *WAITING* queue.

Figure 4 shows how preemption works in the case of Req_2 and Req_3. When n_3 receives the *ALLOC* message for Req_3, it is already **ALLOCATED** because it has previously received the *ALLOC* message for Req_2. It then sends a *PREEMPT* message along the path of Req_2 to n_6. n_6 accepts the preemption, stores Req_2 in its local *WAITING* queue and sends back a *PREEMPT_ACK* message. Req_3 then sends an *ALLOC* message for its last required resource to n_4. Req_3 is now satisfied and enters its CS. When it leaves its CS, n_6 resumes Req_2.

Further considerations on preemptions When a preemption occurs, the algorithm ensures that the resource is always released. Either the node that receives the *PREEMPT* message decides to release it immediately, either it waits for the current request to leave its CS. As CS have finite durations the algorithm ensures that the resource is released in finite time.

If multiple requests have a higher precedence than the one currently holding its resource, a node can receive a *PREEMPT* message when it is already **PREEMPTING**. In this case, priority is given to the request with the highest precedence.

As the communications channels are not FIFO, even if *PREEMPT* messages are sent along the same path than the *ALLOC* messages, a node may receive a *PREEMPT* before it has received the corresponding *ALLOC* . To deal with such situations, each node maintains an *IGNORE* list. When it receives a *PREEMPT*

for a request and it has not yet received the corresponding $ALLOC$, it stores the request in the *IGNORE* list. When it finally receives the $ALLOC$ for a request that is present in the *IGNORE* list, it ignores the $ALLOC$ and removes the request from the list.

5 Experimental results

We present an evaluation of the two allocation orders *reverse path* and *by values* of the algorithm detailed above. We compare their performance through metrics that are introduced below to Dijkstra's *Incremental* algorithm detailed in Section 5.2 as the baseline.

5.1 Metrics and evaluation

Several metrics are used to compare the results of the algorithms:

- the **average usage rate** is the average time during which resources are used. It is the sum of the times during which each resource is used divided by the overall duration of the experiment for all resources. 100% means that all resources are used all the time. 50% means that 50% of the resources are used on average. The objective is to maximize this metric,
- the **average waiting time** is the average time spent by requests between the moment at which they are emitted and the moment they are satisfied. The objective is to minimize this metric,
- The **average number of messages per request** is the ratio between the total number of messages sent in the system for the duration of the test and the number of requests. The objective is to minimize this metric.

5.2 Dijkstra's *Incremental* algorithm

The baseline algorithm for our experiments is Dijkstra's *Incremental* algorithm [6]. It does not require any additional assumption on the system which allows to evaluate it against the same systems as our algorithm described above. This algorithm is selected because it gives the best average usage rate among all the state of the art algorithms evaluated. Other algorithms are not included here for the sake of space. Our implementation of the algorithm relies on the same first subroutine described above for the selection of the path, but does not require the building of the allocation vectors. The second subroutine of the *Incremental* algorithm relies on a static **global order** of the nodes. For the test system in Figure 1, our implementation considers that the global order of nodes is $n_1 < n_2 < \ldots < n_9$ according to the subscript value.

This algorithm has a drawback we call the **domino effect**. It is possible when the number of conflicts is high that nodes wait for each other's requests to be finished. Until these resources become available all the resources from nodes that come after in the order are unavailable. The probability of occurrence of the domino effect increases with the size of the requests.

5.3 Simulating the system with SimGrid

The algorithms have been tested on topologies from the Internet Topology Zoo [13], a dataset of topologies provided by Operators. The topologies are enriched with resources distributed across their nodes. Weights are attributed to the network links between the nodes to simulate the latency. To limit the number of parameters, all experiments in this paper use the same constant weight for all the links. The results presented here are based on the *Cesnet200706* topology which consists of $N = 44$ nodes. This topology was selected because it has a sufficient enough size to avoid the possible bias in the results from topologies that have few nodes. Larger topologies lead to longer simulation times to get similar results. In this topology, the degrees of the nodes vary from 1 to 18, with an average degree of 2.

The simulator is based on SimGrid 3.23 [3]. SimGrid provides multiple Routing Models for the routing of packets between nodes. The experiments use only the *Full* routing model. This model requires all routes to be explicitly described in the configuration file. It ensures that no routing algorithm is used to connect two nodes that are not explicitly connected by a *Link* to each other. It allows the routing of the packets to be made at the application level and thus allows to simulate the first subroutine of the algorithm.

The routing tables necessary for the first subroutine are built statically by the simulator during the initialization using Dijkstra's Shortest Path algorithm [5]. Routing tables are based on the *Link* and *Hosts* of the configuration.

The system is under maximum load during the tests. To achieve this, the simulation starts as many requests as there are nodes in the system, and a new request is sent as soon as a request ends. The contents of the requests are generated by the node following a linear random function. The size of the requests in a single experiment is constant. As shown by other experimental results not included here for sake of space, using a constant size does not affect the results significantly.

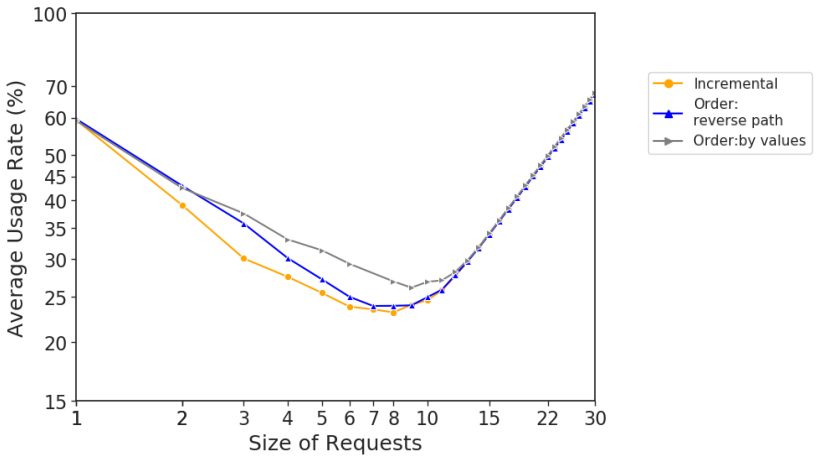
The duration of all experiments is the same. The time spent in CS by the requests is also constant. Empirically we settled for a duration of CS of $300,000$ and a duration of experiment of $500,000,000$ so that time spent in CS is orders of magnitude longer than the time spent to send a message between nodes. These durations are in simulator time unit. This approximately simulates CS of 30s and a total simulation duration of 14 hours. Experiments show that this duration is long enough to make the impact of randomness negligible and the results representative.

In this experimental platform, if two allocation vectors have the same average value, the id of the nodes are used to break the tie. Since requests all have the same size, we first compare the id of the first nodes in each allocation vector. If it is the same node for both requests, we compare the id of second nodes and so on. It this comparison also results in a tie, when both requests are for the same nodes, then the internal identifiers of the requests are compared.

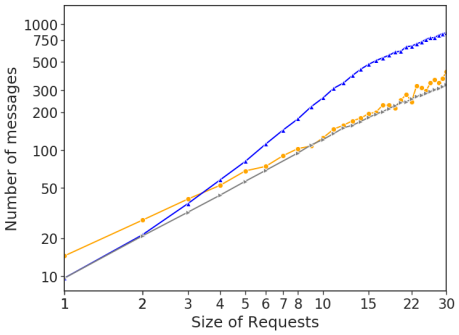
Figure 5 shows the results of the evaluation of the metrics defined in Section 3 for the *Incremental* and the algorithm detailed in Section 4 using two *allocation orders* : *reverse path* and *by values* in two different system configurations detailed

below. The x-axis show the size of requests for a given simulation. Both axes of all the figures use a logarithmic scale.

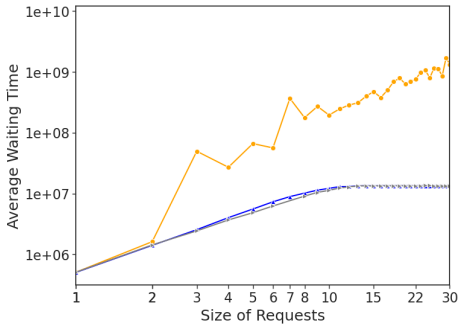
5.4 System with 1 instance of M resources



(a) **average usage rates**



(b) **average nb of messages**



(c) **average waiting times**

Fig. 5. Evaluation for one instance of 44 types of resources

This section shows the evaluation on a system with one instance of $C = 44$ types of resources.

For requests of size 1, Figures 5a and 5c show that the algorithm itself has no influence on the *Average Usage Rate* or the *Average Waiting Time*. In such configuration, no preemption is performed.

For requests of size 2 to 7, we can observe that *Average Usage Rate* decreases for all the algorithms. We reach a minimum between 7 and 9 depending on the algorithms. This can be deduced from Maekawa's proof of his quorum algorithm [19], based on Finite Projective Planes properties, that $\sqrt{N} \approx 6.63$ is the minimum size so that all requests have at least one intersection. Usage rate rises up

again after this lowest point because requests become larger and even if there are more conflicts between requests, each request leads to the allocation of a larger number of resources.

If the size of requests is greater than half the size of the system, then it is not possible to allocate concurrently two requests. In such situations the only possibility is to allocate the requests sequentially. The *Average Usage Rate* of the system then grows linearly with the size of the requests. The figure has been truncated at size 30 for the sake of readability. It is almost 100% for requests of size N as each consecutive request allocates all the resources. It is not exactly 100% due to the cost of the algorithm.

Our algorithms show an improvement on the *Incremental* of the *Average Usage Rate* with both allocation orders. The *by values* allocation order gives the best results for all metrics. For the *Average Usage Rate* we can see improvement of up to 20% from the *Incremental*. As shown in Figure 5b, *by values* does not generate more messages than the *Incremental* whereas *reverse path* shows a larger number of messages for requests of size 4 and more. This is due to the high number of preemptions taking place. The *by values* order limits the number of preemptions by starting with the node that has the highest counter value. This node is the most likely to receive requests with a higher precedence. Once its resource is allocated, the probability that the other nodes are preempted by requests with higher precedence gets lower. Experimental results show in Figure 5c that the *by values* order does not impact negatively the *Average Waiting Time*. The *Average Waiting Time* for the *Incremental* algorithm is significantly worse than for any of the variants. This is due to the domino effect. For the sake of space a full comparison with a near-optimal allocation is not included here, but results show that even the best results included here are 10 to 20 points lower than a near-optimal solution until requests of size 11, after which the difference starts to decrease.

5.5 System with N instances of M resources

Figure 6 shows the *Average Usage Rate* of the same algorithms on the same topology but with a different placement of the resources. Instead of a single instance of each resource, the system holds 4 instances for each of $C = 11$ different types. The figure includes three additional algorithms. Each of this additional algorithm is a variant of one of the three presented above: it uses the same *allocation subroutine* but a different *path computation subroutine*. As the *path computation subroutine* detailed in section 4.1 uses a static routing table, a node always selects the same node for a type of resource. The result is that the load is not well balanced across all the instances, which leads to lower *Average Usage Rate*. For requests of size 4, the lowest *Average Usage Rate* observed, the algorithm with the best result, the *by values*, reaches around 17%. This is lower than the worst result for the configuration with a single instance of M resources in Figure 5a where the *Incremental* reaches 27%.

As shown in the three additional algorithms, the load balancing improves with a simple round-robin on the different instances of each type of resource

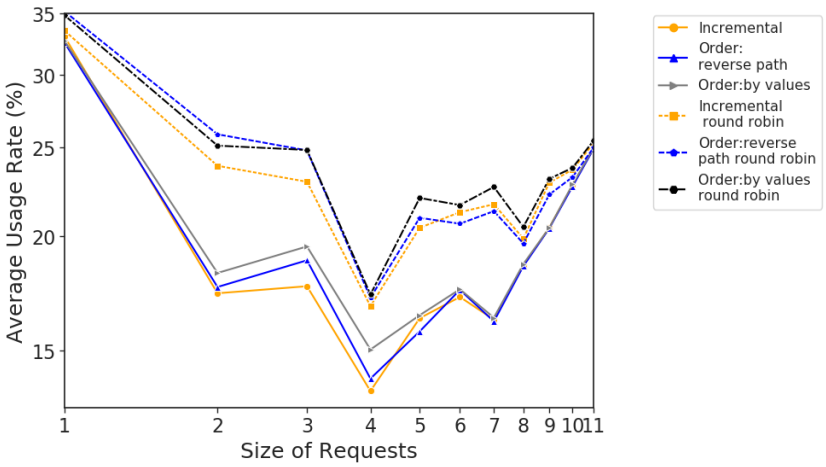


Fig. 6. Evaluation for 4 instances of 11 types of resources

during the *path computation subroutine*. For example with 4 instances of c_1, the first request selects the first instance, the second request the second one, and so on. It starts over with the fifth request that selects the first one. The selection of the route has a significant impact when there are more than one instance of the resources and improves the *Average Usage Rate*.

6 Conclusion and future works

We introduced in this paper a new algorithm for distributed Mutual Exclusion for systems with N instances of M resources, known as the $k-M-N$ problem. This is applicable to the allocation of Multi-Domain Chains of NFs in 5G Slices. We show an improvement of up to 20% of the *Average Usage Rate* of resources from the baseline *Incremental* algorithm, with an *Average Waiting Time* that can be several orders of magnitude lower and no degradation of the number of messages. The results show the impact of the **allocation order** in which allocation is performed.The presented results focus on a few key parameters.

For the *path allocation subroutine* of the algorithm, the results showed the importance of the selection of the instance when $N > 1$. We plan to study how the performance can be further improved.

Our approach to the allocation of resources in the *allocation subroutine* is pessimistic, i.e., it considers that deadlocks are going to happen. But their probability of occurrence can be low in some situations. We plan to study an optimistic approach that let deadlocks occur and fix them a posteriori.

We also plan to implement the algorithm in the scheduler of a NFV platform, if possible in the live network of an Operator.

References

[1] B. Awerbuch and M. Saks. "A Dining Philosophers Algorithm with Polynomial Response Time". In: Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science. 1990, 65–74 vol.1.

[2] A. Bouabdallah and C. Laforest. "A Distributed Token-Based Algorithm for the Dynamic Resource Allocation Problem". In: *SIGOPS Oper. Syst. Rev.* 34.3 (2000), 60–68.

[3] Henri Casanova et al. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (2014), 2899–2917.

[4] K. M. Chandy and J. Misra. "The Drinking Philosophers Problem". In: *ACM Trans. Program. Lang. Syst.* 6.4 (1984), 632–646.

[5] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), 269–271.

[6] E. W. Dijkstra. "Hierarchical Ordering of Sequential Processes". In: *Acta Informatica* 1.2 (1971), 115–138.

[7] E. W. Dijkstra. "Solution of a Problem in Concurrent Programming Control". In: *Commun. ACM* 8.9 (1965), 569–.

[8] David Ginat et al. "An Efficient Solution to the Drinking Philosophers Problem and Its Extensions". In: *Distributed Algorithms*. International Workshop on Distributed Algorithms. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1989, pp. 83–93.

[9] ETSI NFV ISG. *ETSI GS NFV 001: Network Functions Virtualisation (NFV) Use Cases*. 2013. URL: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`.

[10] ETSI NFV ISG. *ETSI GS NFV-MAN 001 V1.1.1 Network Functions Virtualisation (NFV); Management and Orchestration*. 2014.

[11] Brendan Jennings and Rolf Stadler. "Resource Management in Clouds: Survey and Research Challenges". In: *J. Netw. Syst. Manage.* 23.3 (2015), 567–619.

[12] R. Jhawar et al. "Supporting Security Requirements for Resource Management in Cloud Computing". In: 2012 IEEE 15th International Conference on Computational Science and Engineering. 2012, pp. 170–177.

[13] S. Knight et al. "The Internet Topology Zoo". In: *IEEE Journal on Selected Areas in Communications* 29.9 (2011), 1765–1775.

[14] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565.

[15] J. Lejeune et al. "Reducing Synchronization Cost in Distributed Multi-Resource Allocation Problem". In: 2015 44th International Conference on Parallel Processing. 2015, pp. 540–549.

[16] M. R. Garey et al. *The Complexity of Flowshop and Jobshop Scheduling — Mathematics of Operations Research*. 1976. URL: `https://pubsonline.informs.org/doi/abs/10.1287/moor.1.2.117`.

[17] F. Machida et al. “Redundant Virtual Machine Placement for Fault-Tolerant Consolidated Server Clusters”. In: 2010 IEEE Network Operations and Management Symposium - NOMS 2010. 2010, pp. 32–39.

[18] Aomar Maddi. “Token Based Solutions to M Resources Allocation Problem”. In: *Proceedings of the 1997 ACM Symposium on Applied Computing*. SAC ’97. New York, NY, USA: ACM, 1997, pp. 340–344.

[19] Mamoru Maekawa. “A N Algorithm for Mutual Exclusion in Decentralized Systems”. In: *ACM Trans. Comput. Syst.* 3.2 (1985), 145–159.

[20] K. Mills et al. “Comparing VM-Placement Algorithms for On-Demand Clouds”. In: 2011 IEEE Third International Conference on Cloud Computing Technology and Science. 2011, pp. 91–98.

[21] Mohamed Naimi et al. “A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal”. In: *Journal of Parallel and Distributed Computing* 34.1 (1996), 1–13.

[22] Carlos Pignataro and Joel Halpern. *Service Function Chaining (SFC) Architecture*. 2015. URL: `https://tools.ietf.org/html/rfc7665`.

[23] Stefano Previdi et al. *Segment Routing Architecture*. 2018. URL: `https://tools.ietf.org/html/rfc8402`.

[24] Anshul Rai et al. “Generalized Resource Allocation for the Cloud”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. New York, NY, USA: ACM, 2012, 15:1–15:12.

[25] Kerry Raymond. “A Tree-Based Algorithm for Distributed Mutual Exclusion”. In: *ACM Trans. Comput. Syst.* 7.1 (1989), 61–77.

[26] Michel Raynal. “A Distributed Solution to the K-out of-M Resources Allocation Problem”. In: *Advances in Computing and Information - ICCI ’91*. International Conference on Computing and Information. Springer, Berlin, Heidelberg, 1991, pp. 599–609.

[27] Glenn Ricart and Ashok K. Agrawala. “An Optimal Algorithm for Mutual Exclusion in Computer Networks”. In: *Commun. ACM* 24.1 (1981), 9–17.

[28] Peter Rost et al. “Network Slicing to Enable Scalability and Flexibility in 5G Mobile Networks”. In: *IEEE Communications Magazine* 55.5 (2017), 72–79.

[29] SLICENET. *Deliverables - SLICENET*. 2018. URL: `https://slicenet.eu/deliverables/`.

[30] Ichiro Suzuki and Tadao Kasami. “A Distributed Mutual Exclusion Algorithm”. In: *ACM Trans. Comput. Syst.* 3.4 (1985), 344–349.

[31] A. Widjajarto et al. “Cloud Computing Reference Model: The Modelling of Service Availability Based on Application Profile and Resource Allocation”. In: 2012 International Conference on Cloud Computing and Social Networking (ICCCSN). 2012, pp. 1–4.