Distributed resource allocation for virtual networks

Guillaume Fraysse

THÈSE DE DOCTORAT

pour l'obtention du grade de
DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité : Informatique
École Doctorale Informatique, Télécommunications et Électronique (ED130, Paris)

Distributed resource allocation for virtual networks

AUTEUR: GUILLAUME FRAYSSE

Soutenue le 18 décembre 2020 devant le jury composé de:

Sébastien MONNET	*Rapporteur*	Professeur, Université Savoie Mont Blanc
François TAIANI	*Rapporteur*	Professeur, Université de Rennes 1
Anne FLADENMULLER	*Examinatrice*	Maîtresse de Conférences (HDR), Sorbonne Université
Djamal ZEGHLACHE	*Examinateur*	Professeur, Telecom SudParis
Pierre SENS	*Directeur de thèse*	Professeur, Sorbonne Université
Jonathan LEJEUNE	*Encadrant*	Maître de Conférences, Sorbonne Université
Julien SOPENA	*Encadrant*	Maître de Conférences, Sorbonne Université
Imen GRIDA BEN YAHIA	*Encadrante*	Orange

Résumé

Les dernières évolutions des infrastructures réseaux permettent d'apporter plus d'élasticité et de dynamicité à la gestion des réseaux. La 5^{eme} génération de réseaux (5G) permet la création de chaînes ordonnées de fonctions sur des réseaux virtuels (*"slices"*) qui peuvent être multi-domaines, voire multi-opérateurs. Les solutions d'orchestration centralisée habituellement trouvées dans les réseaux peuvent ne pas répondre à certains des nouveaux cas d'usage. Cette thèse plaide pour l'opportunité d'une solution distribuée au problème d'allocation de ressources pour ces chaînes ordonnées de fonctions.

Un nouvel algorithme distribué, découpé en deux parties distinctes, est proposé. La première partie calcule un chemin en prenant en compte les contraintes sur l'ordre des ressources et leur placement sur la topologie réseau. La seconde alloue les ressources en utilisant des horloges vectorielles pour l'ordonnancement des requêtes et en utilisant un mécanisme de préemption pour le faire respecter. Plusieurs heuristiques sont proposées pour chacune de ces parties. Une méthode numérique est proposée pour comparer les performances de algorithme à l'espérance mathématique. Les performances sont ensuite comparées avec celles de quatre algorithmes de l'état de l'art sur une plateforme d'évaluation basée sur le simulateur SimGrid [Cas+14]. Les résultats montrent jusqu'à 20% d'amélioration du taux d'utilisation moyen des ressources, sans dégrader les autres métriques mesurées.

Abstract

The recent evolution of network infrastructures allows for more elasticity and dynamicity to network management. The 5^{th} generation of networks (5G) allows the creation of Chains of Network Functions on top of virtual networks (*"slices"*) that can be multi-domain, or even multi-operators. Centralised solution usually used for network management might not be adequate for these newer use cases. This thesis makes the case for the opportunity of a distributed solution to the problem of the allocation of resources for these sorted chains of functions.

A new distributed algorithm, split in two distinct part, is introduced. The first part computes a path that takes into account the constraints on the order of the resources and their placement on the network topology. The second part allocates the resources using vectors of counters for the scheduling or requests and a preemption mechanism to enforce it. Several heuristics are proposed for both parts. A numerical method is proposed to compare the performance of the algorithm to the expected value. The performances are then compared with those of four algorithms from the state of the art on an evaluation platform based on the SimGrid simulator [Cas+14]. Results shows an improvement of up to 20% of the *Average Usage Rate* while not degrading the other metrics.

Contents

List of Acronyms

3GPP 3rd Generation Partnership Project. 12, 14, 15

API Application Programming Interface. 5, 13–15, 20, 21

CS Critical Section. 8, 19, 26–29, 31–34, 37, 42, 43, 45, 47, 50, 52, 62, 65, 76, 81–84, 86–95, 97–102

DiPP Dining Philosophers Problem. 30–32, 34, 35, 37, 38, 78, 80, 82, 86, 87, 89, 90, 129

DrPP Drinking Philosophers Problem. 3, 31, 32, 34, 36–39, 78, 79, 86–90, 94, 95, 127, 129

ETSI European Telecommunications Standards Institute. 12, 18, 19

FIFO First In First Out. 40, 50

IaaS Infrastructure as a Service. 13, 19

IoT Internet of Things. 10, 15

NFV Network Functions Virtualisation. 4, 5, 12–19

ONOS Open Network Operating System. 20, 21, 23

SDN Software-Defined Networking. 5, 9, 12–16, 19–23, 126

VM Virtual Machine. 12, 16, 19

VNF Virtual Network Function. 5, 9, 12–19, 39, 106

Chapter 1

Introduction

I am Groot!

Groot, *Guardians of the Galaxy*

This thesis advocates a distributed management of networks in some specific use cases. It addresses the problem of the allocation of resources for network slices. The main contribution is a new distributed algorithm for the allocation of resources in systems with multiple instances of multiple types of resources. Its objective is to maximise the usage rate of the resources. Multiple heuristics are proposed for this algorithm. These heuristics are evaluated and compared to algorithms from the state of the art with experiments run on a simulator based on SimGrid [Cas+14]. A numerical method is proposed to compute the expected value for the usage rate of the resources and is used to evaluate the performance of the algorithm.

1.1 Context and motivation

Telecommunications networks are geographically distributed, access points such as radio antennas, fibre optics terminators or even Low Earth orbiting satellites are located all over the world (or in space) to enable network access to every users. However, networks are typically managed by a logically centralised component. In the standardised architectures used by almost all network operators, each access point sends data to what is called the Core Network that is typically physically distributed to increase the network resiliency. It is logically centralised and includes a centralised database, for example the Home Subscriber Server (HSS) found in 3G and 4G mobile networks or the Unified Data Management (UDM) in 5G. In parallel, virtualisation is offering more and more elasticity to infrastructures, giving room for new paradigms for network services. The telecommunications industry has introduced the Network Functions Virtualisation (NFV) standard to leverage them. NFV enables the management of **chains of network functions** not only of stand-alone functions. This introduces a constraint on the order of the functions in the chains. **Network slicing** is another of these new paradigms that is introduced in 5G. Slices are virtual networks that can be instantiated across multiple network domains or even across multiple network providers. A centralised management might not always be a good fit for all the new multi-domain services. Having multiple providers side by side, each with their own centralised manager, and each managing a subset of the overall resources, might lead to starvation or scalability issues.

1.2 Contributions

This thesis makes the case that the allocation of resources for multi-domain network slices can be likened to a generalisation of the Mutual Exclusion problem to systems with multiple instances of multiple types of resources. First, the allocation of ordered chains of network functions is modelled as an allocation problem in a system with multiple instances of multiple types of resources.

A new distributed modular algorithm for the allocation of resources in systems with multiple instances of multiple types of resources is proposed. Few distributed algorithms from the state of the art target these systems and none of them consider the constraint on the order in which resources are used. The proposed algorithm takes this constraint into account. Its objective is to maximise the usage rate of network resources while trying to minimise the number of messages necessary. The algorithm is composed of two consecutive subroutines. The first is the *path computation* subroutine that select instances in the network that satisfy the constraint on the order and computes the path connecting them. The second is the *allocation* subroutine that allocate the selected resources. The allocation follows a total order of the requests that is computed from the allocation vectors of the request. These allocation vectors are composed of counter values set by each of the selected instance. The algorithm includes a preemption mechanism to enforce this order when multiple requests wants to allocate a same node concurrently.

Heuristics are proposed for each of the two subroutines of the algorithm. Heuristics for the *path computation* offer different balancing of the load across instances to maximise the usage rate. The order followed to allocate the resources during the *allocation* subroutine has a direct influence on the performance of the algorithm. Three heuristics with different allocation orders are proposed.

An experimental evaluation of the performance of the heuristics for both subroutines of the algorithm is presented. The performance of the *allocation* subroutine is compared to the mathematical *expected value* for the usage rate in the experimental settings. The performance is also compared with four distributed algorithms for the allocation of resources from the state of the art. Results show that, using the best heuristic, the algorithm offers up to a 20% increase of the *Average Usage Rate* from the best performing algorithm from the state of the art on this metric. It also shows that the same heuristic does not degrade the performance on the other metrics.

1.3 Structure of this manuscript

Chapter 2 gives some background on the evolution of networking infrastructures and some of the newest use cases that they enable. It introduces three of those evolutions. The first evolution described is the concept of Chains of Virtual Network Functions (VNFs) that allows the management of an ordered set of VNFs instead of a single VNF. It then describes Software-Defined Networking (SDN) a novel approach to network management that introduces Application Programming Interfaces (APIs) to dynamically manage networks. Finally it introduces network slices, a recent architecture proposed to have multiple virtual networks on top of NFV infrastructures. The problem addressed in this work is the allocation of resources in systems with multiple instances of multiple types of resources. This chapter describes the problem and introduces the model that is used throughout this thesis. It also describes a SDN use case that shows that stacking centralised managers might not be a fitting solution to manage distributed resources.

Chapter 3 details the state of the art of distributed solutions to the allocation of resources problem. Distributed architectures have been proposed since at least the 1960s. A seminal paper on this topic has been authored by E. W. Dijkstra in 1965 [Dij65]. He defined and solved the Mutual Exclusion problem in which multiple nodes (computers) need to use a single resource (for example a printer). In these settings a single node can use the resource at any given time and all the nodes need to use it eventually. Later works extended the problem for systems with multiple types of resources or with multiple instances of a single type of resources. A popular teaching tool for this problem is the Dining Philosophers Problem where philosophers sit around a table and have to decide who has the right to eat in a system where there is not enough silverware for everyone. These solutions have been proposed for network problems like the allocation of radio spectrum or Mobile Ad Hoc Networks. The work presented here tries to make the case that this class of solution could be suited for some of the newest use cases in networks.

Capitalising on the previous efforts listed in this state of the art, a new distributed modular algorithm for the allocation of resources is introduced in **Chapter 4**. This algorithm addresses the constraint of the order of the resources. Its objective is to maximise the usage of the resources for network operators. It is composed of two subroutines. A *path computation* subroutine that computes a path with instances of the requests type that respects the order. An *allocation* subroutine that allocates the resources according to a total order of the requests. This total order is computed from the counter values read on the nodes when computing the path, and then set in an allocation vector specific to each request. An algorithm is not a Swiss Army knife that can work for all settings, this is why several heuristics are proposed for both subroutines and their advantages and drawbacks listed. These heuristics and the algorithm are thoroughly evaluated in the last two chapters.

Chapter 5 shows the results of the experimental evaluation of the algorithm using the SimGrid simulator [Cas+14] and Open MPI [Gab+04]. The heuristics of the algorithm are compared with each other on three metrics: the *Average Usage Rate*, the *Average Waiting Time* and the *Average Number of Messages*. Maximising the usage of the resources raises the question of how high it could theoretically go, so a numerical solution is proposed to compute the *expected value* in the experimental settings, and compare it to the performance of the algorithm. Comparing the heuristics of the *path computation* subroutine show that load-balancing across the different instances of a type of resources improves the *Average Usage Rate* while degrading the length of the path. The heuristics of the *allocation* subroutine show the importance of the order followed to allocate the resource and that the *Average Usage Rate* can be improved when the probability that another request tries to allocate one of the instance is taken into account.

Then, **Chapter 6** details four algorithms from the state of the art with different solutions to the problem. Dijkstra's Incremental algorithm [Dij71], while being the first algorithm proposed to solve a neighbouring problem with static requests, can be easily adapted to the model and offers the best performance of the four. Chandy-Misra algorithm [CM84] was the first to formulate and solve the allocation of multiple types of resources, but is not adequate for systems where the possible requests are not known a priori. Rhee's modular algorithm [Rhe95] is based on a model close to the one modelled in this work. Bouabdallah-Laforest algorithm [BL00] is the most recent algorithm to improve the performance in systems with one instance of multiple types of resources. Their performances are compared with that of the proposed algorithm. Results show that the best heuristic for the *allocation* subroutine offers an improvement from these algorithms

in the experimental settings. The best heuristic allows an increase of up to 20% of the *Average Usage Rate* and does not degrade the two other metrics.

Finally, the general conclusion enumerates the contributions of this work and lists the current limitations, as well as possible future work.

1.4 Publications

[Fra+18b] G. Fraysse et al. "Towards Multi-SDN Services: Dangers of Concurrent Resource Allocation from Multiple Providers". In: *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), short paper*. 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN). Feb. 2018, pp. 1–5

[Fra+18a] G. Fraysse, J. Lejeune, J. Sopena, and P. Sens. "Mapping the Allocation of Resources for 5G Slices to the K-MUTEX with n Instances of m Resources Problem". In: *2018 14th International Conference on Network and Service Management (CNSM), short paper*. Nov. 2018, pp. 318–322

[Fra+20] Guillaume Fraysse, Jonathan Lejeune, Julien Sopena, and Pierre Sens. "A Resource Usage Efficient Distributed Allocation Algorithm for 5G Service Function Chains". In: *Distributed Applications and Interoperable Systems*. Ed. by Anne Remke and Valerio Schiavoni. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 169–185

Chapter 2

Background and problem statement

Contents

> If you're havin' Perl problems,
> I feel bad for you, son
> I got 99 problems,
> So I used regular expressions
> Now I have 100 problems
>
> Randall Munroe,
> *XKCD #1171, "Perl Problems"*

This first chapter explains the background on the evolution of network infrastructures that led to the formulation of the problem statement addressed in this work. The first section is an overview of the evolution of networks since the first telecom networks to the fifth, and currently last, generation of networks (5G). Then the next section has a closer look on some of the latest evolution of network infrastructures: Software-Defined Networking (SDN), network slicing and chains of Virtual Network Functions (VNFs). Additionally it details an end-to-end use case that leverages all these evolutions. The last

section details the resource allocations problem statement and the model used throughout this thesis. It also showcases centralised solutions are not always relevant and why a distributed solution, such as the one detailed in the next chapters, can be an option.

2.1 The convergence of telecommunications and computer networks

The first networks, retroactively called Plain Old Telephony System (POTS), introduced at the end of the nineteenth century (1876) relied on *analogue* transmission. They were initially used for interpersonal voice calls. The most famous image of this is the very first call made by Alexander Graham Bell to his assistant Thomas Watson. Almost a century later, the first generation of cellular networks (1G) appeared in the late 1970s in Japan, it was also based on analogue transmission between the handsets and the radio towers. Figure 2.1 gives an overview of the evolution of networks.

Interpersonal voice calls remained the main service offered until these networks started to be replaced by networks based on *digital* transmission even if other services, such as fax, started to appear. Fixed telecom networks moved to the Integrated Services Digital Network (ISDN) standard, first deployed in 1988 in the United States. ISDN made it easier to offer additional services like voicemail, conference calls, telephone exchanges for home and enterprise customers.

Mass adoption of cellular networks started in the 1990s when the Global System for Mobile Communications (GSM) standardised the 2nd Generation (2G) of mobile networks that relies on digital transmission. 2G at launch allowed interpersonal voice calls using a *circuit-switched* network similar to fixed networks. In circuit-switched networks, a circuit is dedicated to the communication between the two endpoints.

The idea to interconnect computers arose in the late 1950s and was enabled by the conception of new *packet-switched* networks. In packed-switched networks the data is grouped in packets and each packet has a header that allows the identification of the recipient and of the packet route to it. In the late 1960s the Defense Advanced Research Projects Agency (DARPA) began working on what would eventually become the Internet to interconnect computers networks. The IP and TCP protocols were both described in 1974 and are still the backbone of the World Wide Web. In 2020 most of the internet still relies on the version 4 of the IP protocol (IPv4) that has been around since 1981 [Pos81].

At this point telecom networks, mainly used for voice calls, and computer networks relied on very different technologies. Even when telecom networks moved to digital transmission they still used a whole set of network protocols different from the TCP/IP stack of computer networks. However these networks could use common physical layers, and it became more and more common for these networks to share copper or fibre optic lines. The introduction of the Digital Subscriber Line (DSL) technology allowed to use the same copper pair to transmit both ISDN and DSL data, by using different frequencies than ISDN. DSL was introduced once the cost allowed it in the late 1990s to transmit data between subscribers most notably to provide Internet interconnection. Similarly wireless and fixed telecom networks could share parts of a common backbone, the wireless components of the Radio Access Network are only some of the parts of a cellular networks and can be connected to other nodes of the networks with landlines.

Efforts to standardise how to make voice calls using computers and computer networks started in the late 1990s and resulted in the ITU Telecommunication Standardisation Sec-

tor (ITU-T) H.323 [ITU96] and Internet Engineering Task Force (IETF) Session Initiation Protocol (SIP) [HSS99] protocols as the first standards for Voice over IP (VoIP), respectively in 1996 and 1999.

Access to the internet from Cellular Networks was introduced with General Packet Radio Service (GPRS) in 2000 on 2G networks. This introduced an additional packet-switched network in mobile networks that were only circuit-switched before that. The next major evolution appeared with 4G networks. 4G was the first cellular networks technology to introduce VoIP on the packet-switched network, with Voice over LTE (VoLTE). This allowed future telecommunications networks to rely only on packet-switching both for voice and data services. The first commercial VoLTE service was launched in 2012 in Dallas (Texas, United States) [Inc12].

Slowly computer and telecommunications networks have converged to use common network technologies. All communications have moved, or are moving, to IP networks and circuit-switched networks are becoming an artefact of the past. The latest generation of Telecommunication Networks, 5G networks, now tries to consolidate all of these evolutions to enable VoIP communications, access to IP networks from mobile phones as well as other endpoints such as computers, Internet of Things (IoT) sensors ...

This evolution also can be seen on the network infrastructures. Even if telecom networks have always included computing elements (such as processors, memory, ...) they used to be based on dedicated hardware due to the specificity of the protocols and services that were used only in the telecom world. The main service node of the 2G/3G voice networks is the Mobile Switching Center (MSC). A typical MSC from the 1990s or 2000s was composed of a large number of specialised processors such as Digital Signal Processing (DSP) for the treatment of the signal. New network functions can for the most part be run on what is referred to as Customer of the Shelf (COTS) hardware, the same type of servers that are used by other industries requiring computing power. COTS hardware is mostly based on x86 processors.

2.2 Evolution of the architecture of services

Since the time when telecommunications and computer networks started to rely on the same IP technologies and COTS hardware, their infrastructure has evolved and started to converge too. In 2012 the European Telecommunications Standards Institute (ETSI), an industry forum composed of Network Operators, has standardised **NFV** [ISG14] to leverage the recent evolution in computing and IP networking infrastructures: the **virtualisation** of computing resources and the programmability and **management functions** offered by the Cloud Computing paradigm, and **Software-Defined Networking (SDN)**. This evolution was introduced during the lifespan of 4G but 5G, the fifth generation of mobile networks, is the first to include these evolutions from the start.

Telecommunication operators, or network service providers, host NFV infrastructures that are composed of:

- **Chains of Virtual Network Functions (VNFs)**: a Network Function is the unitary component of a NFV network. Common VNFs can be a switch, a router, a firewall or other security equipment. In the case of network service providers these services are standardised by the 3rd Generation Partnership Project (3GPP), for example in 3G and 4G networks the Home Subscriber Server (HSS) is the main user database. These functions are called VNF in the NFV context because they are virtualised.

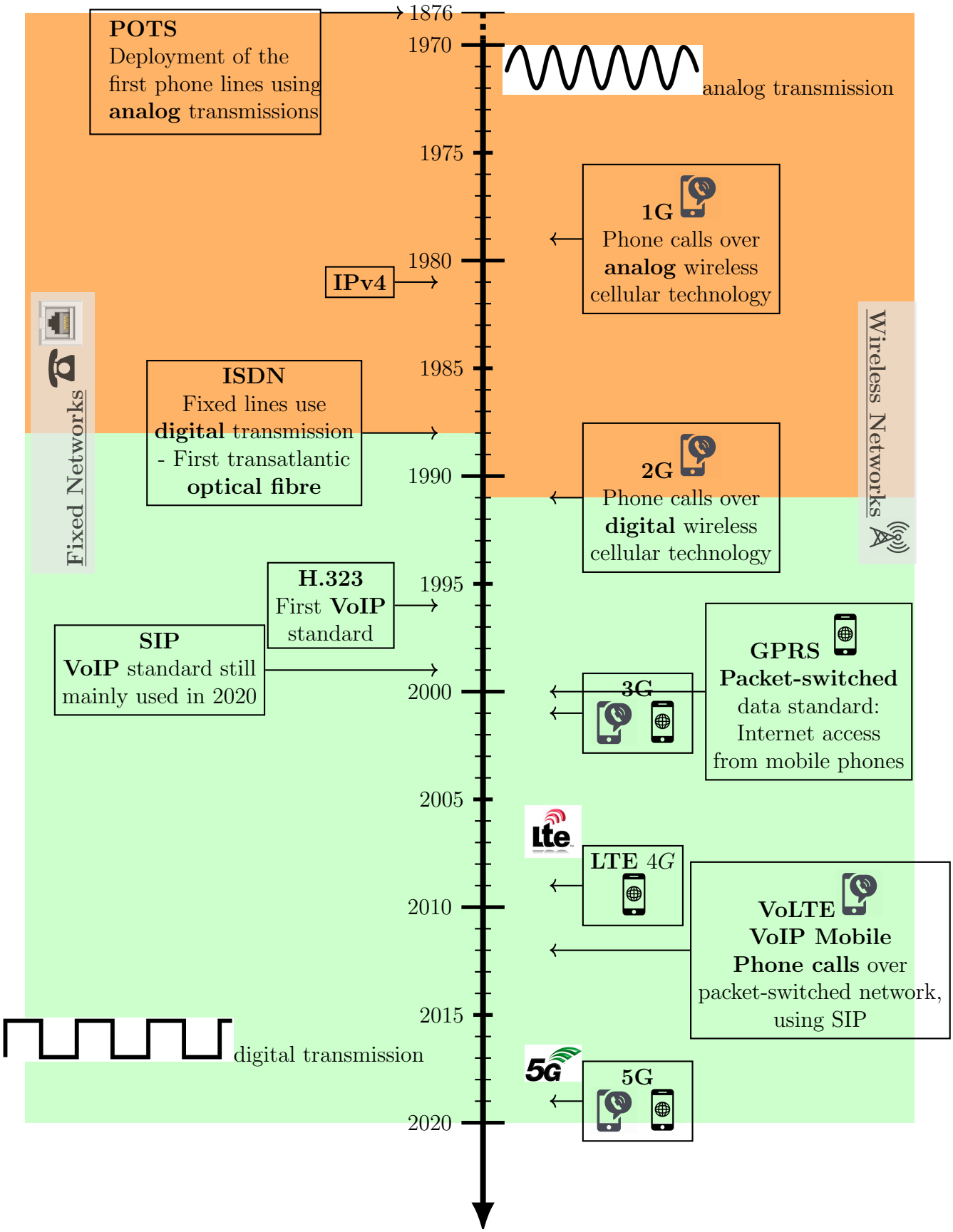


Figure 2.1: Evolution of Telecom networks

Some use cases consider chains of VNFs, as detailed below.

- A virtualised infrastructure that leverages Cloud and **SDN** technologies. This allows to run the Virtual Machines or containers that host the VNF. SDN is detailed in more details in Section 2.2.2.
- Multiple **slices** or parts of slices. Slices are virtual networks running on top of one, or more, NFV infrastructure(s), they are introduced in Section 2.2.3.

2.2.1 Chains of VNFs

NFV infrastructures are composed of multiple VNFs. Allocation of a single VNF is not always sufficient, many use cases require multiple VNFs to inter-operate. To this end, in 2013 the ETSI NFV Industry Specification Group (ISG) identified VNF Forwarding Graphs [ISG13] as a use case for NFV. A VNF Forwarding Graph defines a chain of VNFs, i.e., the sequence of VNFs that packets traverse and are analogous to the connection with cables of physical appliances.

Figure 2.2 shows two chains of VNF, in blue and green, in a system where three VNFs are available: a firewall (FW), an Intrusion Detection System (IDS) and a load-balancer (LB). The red chain uses successively the FW, IDS and LB. The blue chain only uses the FW and LB.

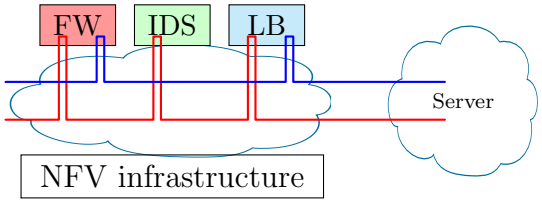


Figure 2.2: Example of Service Functions Chaining: two chains in red and blue

The technology's capabilities mean that a large number of virtual VNFs can be connected together in a NFV environment. Because it's done in software using virtual circuits, these connections can be set up and torn down as needed with service chain provisioning through the NFV orchestration layer.

2.2.2 SDN

Cloud Computing offers abstractions of computing infrastructures through the exposition of APIs by the Infrastructure as a Service (IaaS) orchestrator. Likewise SDN offers abstraction of network resources and network services through a **NorthBound API** exposed by a **SDN controller**. Origins of SDN lies in the observation [Kre+15; Ope16] that IP networks are vertically integrated by vendors in their hardware middle-boxes (switches, routers, ...). Both **control plane** and **forwarding plane** (also called data plane, or user plane) are bundled. The control plane is the part of the network component in charge of the service logic. The forwarding plane is the part in charge of forwarding the network packets. The acronym SDN first appeared in a 2009 article [Gre09] to define the works done at Stanford University that led to the **OpenFlow** protocol as described by McKeown et al. in their seminal white-paper in 2008 [McK+08]. They consider that this strong integration has hindered innovation and evolution of networking architecture and made

managing IP networks increasingly complex. Kreutz et al. [Kre+15] cite the example of the very slow transition of IPv4 to IPv6 as an example of these limitations.

The SDN controller is the main component of the control plane. It relies on the **SouthBound API** to discuss with the forwarding plane that is composed of physical nodes such as switches. SDN applications can interact with the SDN controller using the NorthBound API. A sample SDN architecture, taken from [Wic+15], is shown in Figure 2.3.

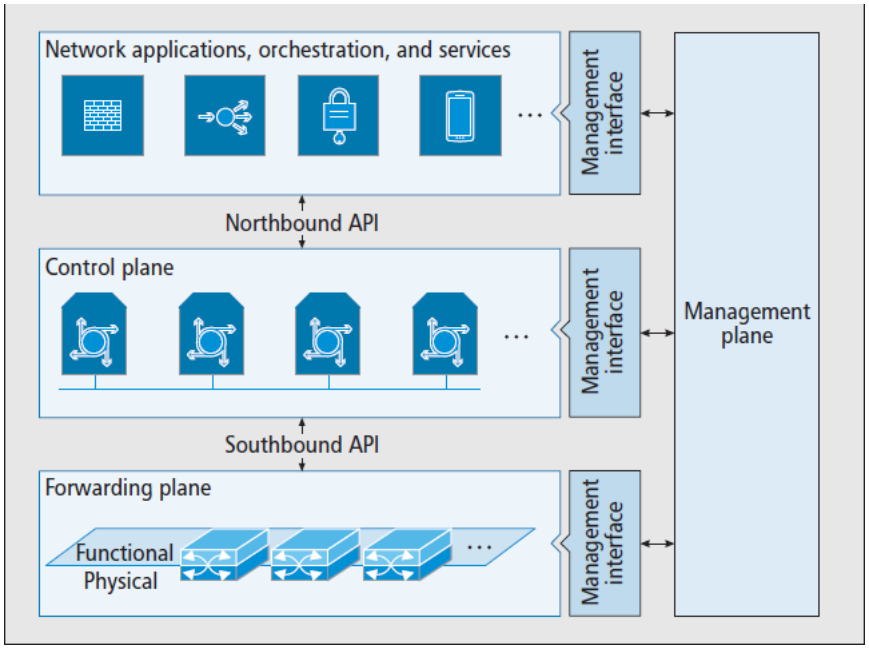


Figure 2.3: High-level conceptual architecture of SDN from [Wic+15]

In their 2013 paper Jain et al. [Jai+13] present B4, a Wide Area Network (WAN) that connects Google's data centres across the globe with its own characteristics:

- it is a distinct WAN from their user facing WAN
- massive bandwidth requirements deployed on a limited number of sites (scalability is not expected to go beyond a few dozens sites)
- elastic traffic that enables to optimise the bandwidth used
- full control over the edge servers and the network

This WAN is deployed using SDN, based on OpenFlow as the SouthBound API and a modified version of Onix [Kop+10] as the SDN controller. Their drivers for such solution was that typical WAN are over-provisioned and have $30 - 40\%$ average utilisation. This over-provisioning allow an higher resiliency to network failure but at the cost of two to three times the necessary bandwidth and equipment. Considering the massive traffic Google has to deal with, and its increase that is faster that the increase of traffic on the Internet, they chose this solution to be cost-efficient. Many of the links of this WAN run at near 100% utilisation and all links average 70% utilisation over long time periods, which is an improvement in efficiency of two to three times.

2.2.3 Network slicing

The 5G standard introduced the concept of network slicing [3GP16]. It is a concept independent of the notion of chains of VNFs introduced above. A slice is a virtual network composed of a set of VNFs built on top of one or several NFV infrastructures. The slicing concept was first introduced by the Next-Generation Mobile Networks (NGMN) in January 2016 [All16]. Since then several Standards Developing Organisation (SDO) and Industry Fora have launched works to analyse the use cases and impact of network slicing. The concept is not completely new, 3GPP first introduced a concept of end-to-end slicing of 4G mobile networks in their release 13 as DEdicated CORe Network (DECOR).

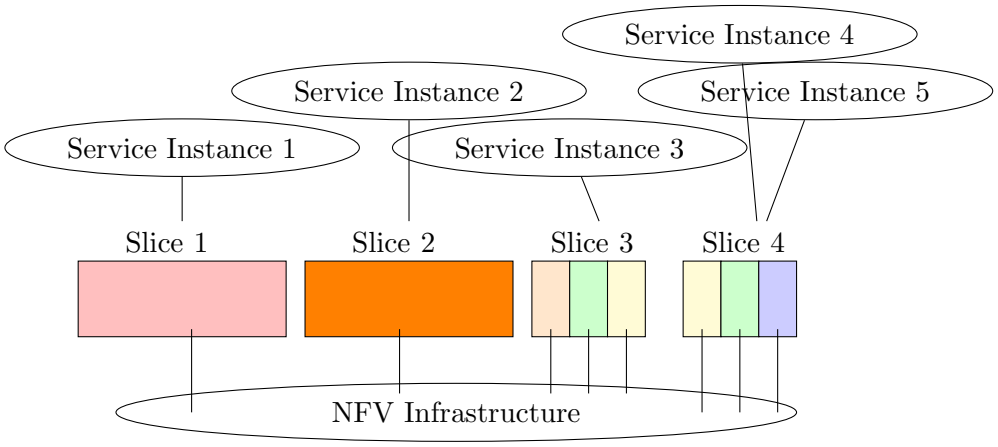


Figure 2.4: Network Slicing architecture, adapted from [All16]

Figure 2.4 is adapted from the architecture introduced by the NGMN for network slicing. On top of a single NFV infrastructure 4 slices use different subsets of the VNFs available. Slice 1 uses one VNF in pale red, slice 2 uses also one VNF in orange. Slices 3 and 4 both use chains of 3 VNFs. In turn these slices are used by Service Instances, i.e., instances of services offered to customers. There is 1 Service Instance running on slice 1. Two different Service Instances run on slices 2 and 3. Two different Service Instances, or two instances of the same service, run on slice 4. This allows 5 different Service Instances to run on a single NFV infrastructure.

The resources are exposed by different APIs (Cloud, SDN, NFV MANO, ...) that can be used simultaneously by different services or users. This is a shift from traditional telecom networks where resources are often used by a single service. NFV introduced the possibility for multiple functions to share the same infrastructure but slicing also allows multiple end-to-end services to share infrastructures.

Following the introduction of networks slices in 5G, 3GPP has standardised 3 types of slices with different objectives:

- enhanced Mobile BroadBand (**eMBB**): bandwidth of 10Gbps
- massive Machine Type Communications (**mMTC**): density of up to 1 million devices/km^2
- ultra-Reliable Low Latency Communication (**uRLLC**): as low as 1ms latency

A service can not be optimised for all three, a trade-off has to be found. Some applications might require to be optimised for a single objective: Vehicle-To-Vehicle communications require uRLLC, ultra high-definition video streaming requires eMBB while massive IoT deployments may require mMTC. A use case is detailed below in Section 2.3.2.

2.3 Multi-domain services

This section first defines the different possible architectures for services that are split across multiple domains and then introduces a recent use case.

2.3.1 Multi-domain architectures

The standard architectures for Cloud, SDN or network slicing focused on infrastructures within a single *domain*. There are multiple reasons for which infrastructures might require to be deployed in multiple domains. For scalability: when a single domain reached its maximum capacity and the only way to add more capacity is to add another domain. Infrastructures can be split in multiple domains for resilience or security, for example to remove or isolate a compromised domain in case of an incident or an attack and still be able to provide the service to the user. A service can also require to leverage services from multiple providers, each in their own domain.

A **domain** is a set of resources with its dedicated manager. A provider can have one or more domains. For example a domain in the context of SDN is the topology and set of resources managed by one SDN controller. In the case of a Cloud it is the set of resources managed by an orchestrator, e.g., an instance of OpenStack. For a network it can be a Point of Presence (PoP), a set of VNFs in a small data centre on the border of the network.

An architecture is **multi-domain** when it involves multiple domains. The domains can depend from a provider, or be split between different providers. Sung et al. [Sun+16] gives some insight on networks consisting of multiple domains stating that large internet-facing services such as Facebook are often hosted on a *network of networks*, where each sub-network has unique characteristics. Facebook's network consists of edge PoPs, a backbone, and data centres. The devices, topology and management tasks vary per sub-network. Yet, all of them must be configured correctly in order for the entire network to work.

Figure 2.5 show an example of multi-domain end-to-end architecture for network slices across three network service providers (NSP) A, B and C. Provider A is composed of three domains A1, A2 and A3. Provider B and C are both composed of a single domain. A domain here is a NFV infrastructure either with virtualised computing, storage and network resources orchestrated by OpenStack, a Cloud Orchestrator and a SDN Controller as A1, A2 and C1 or with only network resources orchestrated by a SDN controller as A3 and B1. There are two 5G slices. Slice 1 runs 4 VNFs that are hosted on 4 VMs in domains A1, A2 and C1. Slice 2 runs 2 VNFs that are hosted on 2 VMs in domains A2 and C1. Orthogonal to the slices is a management plane with specialised orchestrators that communicates with the orchestrators of the various domains to manage these VNFs.

2.3.2 A 5G multi-domain use case: eHealth telemedecine

The European Commission launched the 5G Infrastructure Public Private Partnership (5G PPP) [PPP] joint initiative with the European industry. 5G PPP identified several vertical industry sectors which provided their needs and potential barriers for adoption: automotive, manufacturing, media, energy, health, public safety and smart cities. As part of the projects funded by this initiative, several use cases for these verticals have been detailed. This section focuses on one in particular : eHealth in-Ambulance telemedecine [Ale17] from the SliceNet 5G PPP project.

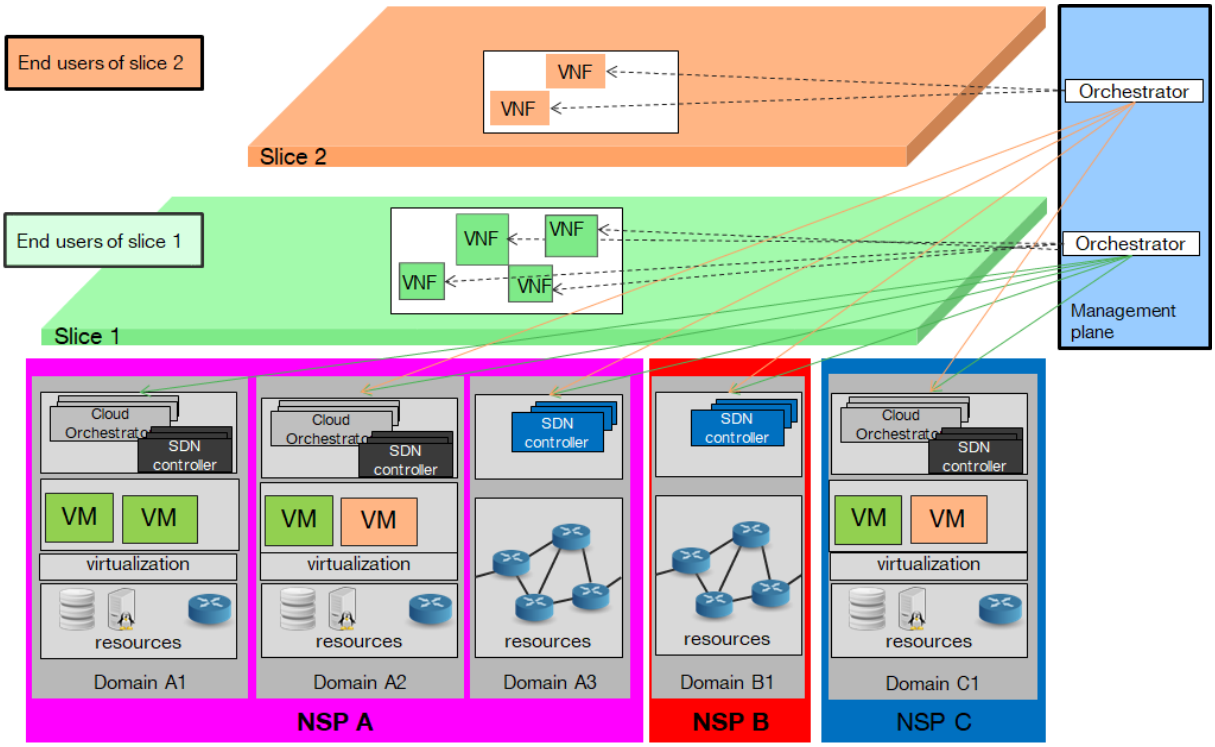


Figure 2.5: Architecture of multi-domain network slices

In this use case an ambulance serves as a connection hub for the devices inside. A connected device, glasses in this specific use case, is present in the ambulance and connects to the emergency department team at the destination hospital. The glasses are used by the paramedics present in the ambulance to assist a patient and send real-time live video stream of what they see to the emergency team at the hospital. The requirements expressed for this use case include enhanced Mobile BroadBand (eMBB), to support the bandwidth required (around 10Mbps) by the high-definition video stream, as well as ultra-Reliable and Low-Latency communications (uRLLC) due to patient safety factors (10ms peak to peak jitter, end-to-end latency inferior to 100ms, less than 0.05% packet loss and 99,999% reliability).

Figure 2.6 shows that this use case involves 4 network service providers (NSP) [Eur18]. Providers NSP1 and NSP3 provide connectivity with their Radio Access Networks (RAN) and the network services for the ambulance in their Core and Mobile Edge Computing (MEC) platforms, for the ambulance. Two providers are necessary to have sufficient coverage of the RANs. Provider NSP2 may be an ISP that provides the Wan-Area Network that connects the others providers and can also host other VNFs or management functions for the end-to-end use case, e.g., it can configure QoS parameters between the domains. Provider NSP4 is the enterprise domain for the hospital and its computing infrastructure as well as the devices present in the ambulance.

2.4 Resource allocation problems in networks

This thesis focuses on the problem of allocating resources in networks for multi-domain use cases. Networks rely on multiple kinds of resources and there are multiple known problems for allocating these resources. The first section introduces the problem statement

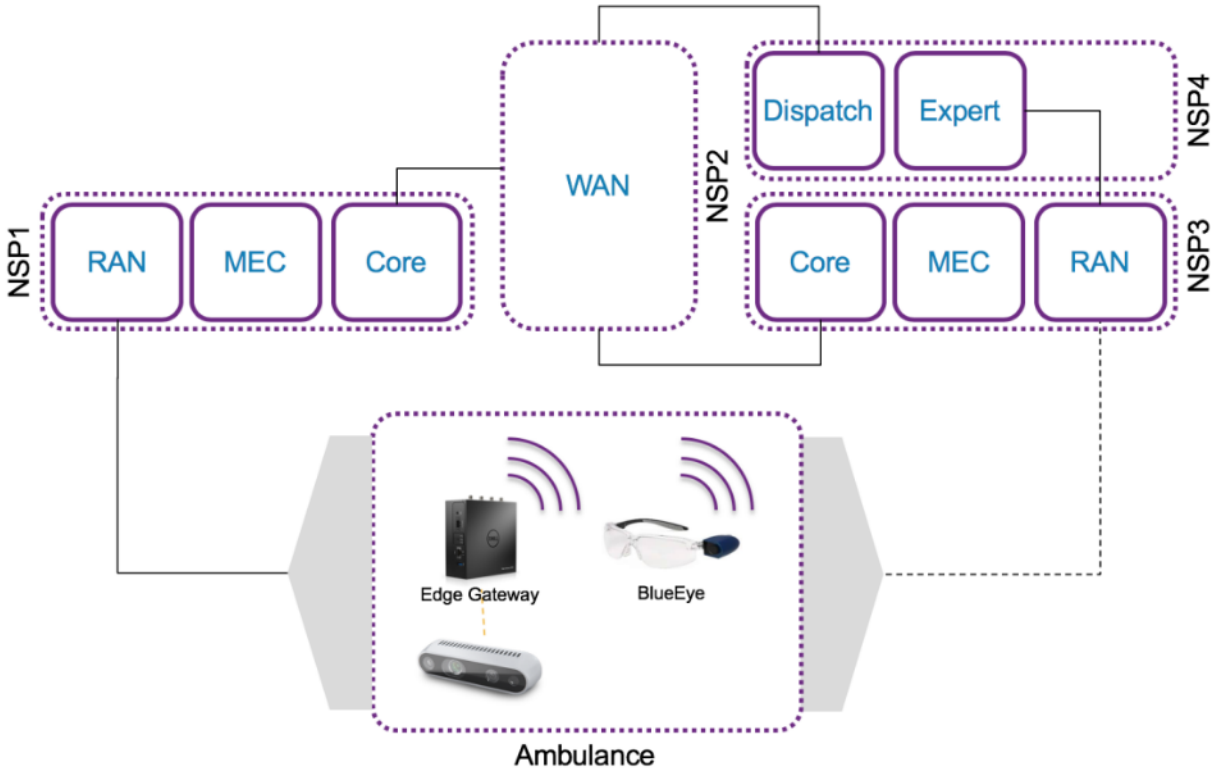


Figure 2.6: Overview of the eHealth integrated multi-domain slicing-friendly infrastructure

and some formalism. Then Section 2.4.2 presents centralised solutions for the allocation of VNF in NFV. Section 2.4.3 shows with an experiment why centralised solutions are not always suited for multi-domain use cases. The last section introduces the distributed approach that is developed in the next chapters.

2.4.1 Problem statement and model

Resource model The left part of Figure 2.7 shows a system with three network service providers (NSP). Three types of resources are available: an Intrusion Detection Systems (IDS), a Firewall (FW) and a Load-Balancer (LB). There are three instances of each type of resources distributed across the three providers.

The set of resources is modelled as a **non-directed connected static communication graph** G=$(\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is the set of **nodes** and $\mathcal{E}$ the set of **edges**. A node holds exactly an instance of one type of resources. $\mathcal{C} = \{c_1, c_2, ..., c_C\}$ is the set of **types of resources** where C is the total number of types. An edge represents a network link.

Edges have positive **weights** to model the latency of the links between nodes noted $\mathcal{W} = \{w_1, w_2, ..., w_\mathcal{E}\}$, $\forall w_x \in \mathcal{W}, w_x \in \mathbb{N}$ and $w_x \geq 0$. A weight of 0 on an edge models a system with multiple resources on a single node. A node with 2 resources can be modelled in the graph as two nodes holding one resource each and connected by a zero-weight edge.

The right part of Figure 2.7 shows the same system in the proposed model.

Request model Each node in the graph can issue **allocation requests**. An allocation request is modelled as a couple $Req(n, [c_1,\ldots,c_s])$ where :

- n is the **requesting node**,

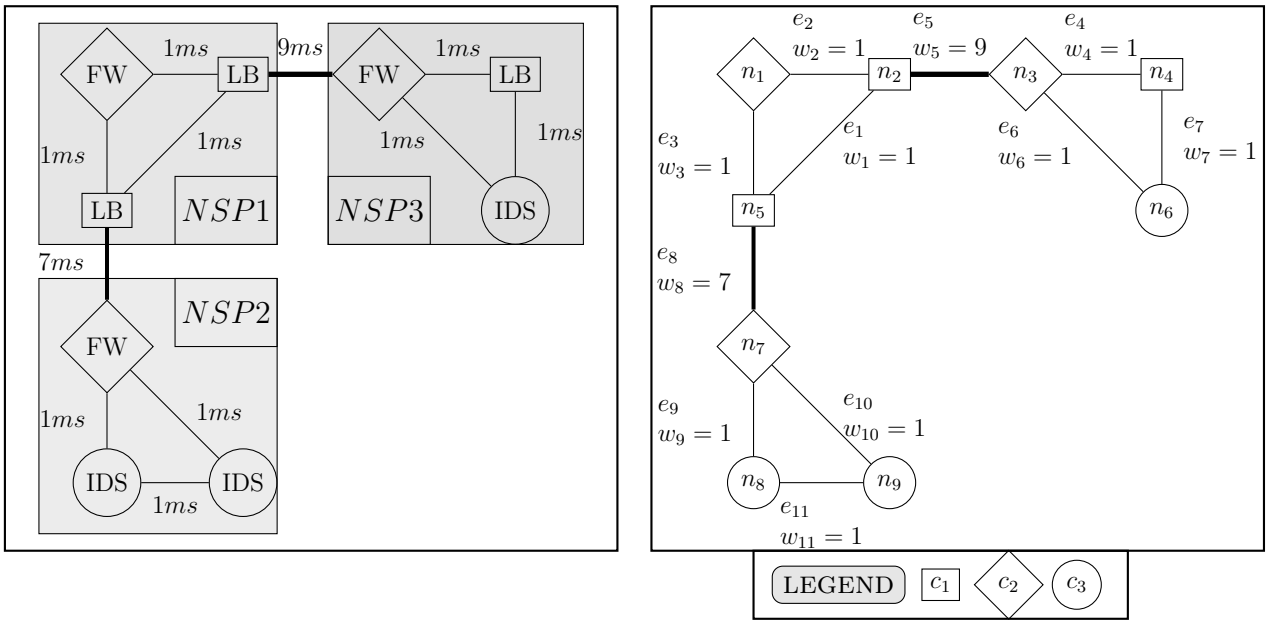


Figure 2.7: A system with 3 types of resources: c_1 (LB), c_2 (FW), and c_3 (IDS)

- $[c_1, \ldots, c_s]$ 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.

In an example of chain of VNFs described by the ETSI NFV Working Group [ISG13], packets need to traverse an IDS, a FW and a LB.

A request Req_1 can be noted as $Req_1 = Req(n_1, [c_3, c_2, c_1])$ in the system introduced above. n_1 is the requesting node, 3 types of resources c_1, c_2 and c_3 are requested. The request order is $c_3 < c_2 < c_1$, i.e., first c_3, then c_2 and finally c_1.

Allocation of a request The problem addressed is to allocate the requests, i.e., select instances of types of resources in the order requested and allocate them to the requesting node. When all the instances selected are allocated, the requesting node can enter its Critical Section (CS) and start using the resource. Figure 2.8 shows a selection of instances for Req_1: the instances in nodes n_8, n_7 and n_5 are selected and the requesting node n_1 can use these resources in that order.

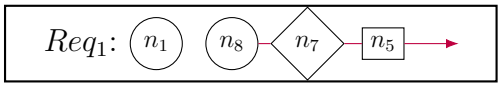


Figure 2.8: Selection of instances for Req_1

2.4.2 Solutions for the resource allocation problem

Telecommunications networks and computing infrastructures often have a management function. For NFV architectures ETSI introduced the MANagement and Orchestration (MANO) architectural framework [ISG14]. A SDN network relies on a SDN controller. Cloud Computing IaaS relies on an orchestrator. This function is often centralised, i.e., a single instance manages all the resources of the domain.

Among the roles of the MANO is the placement of VNF in the NFV infrastructure. When managed by a centralised orchestrator, most of these problems are NP-hard variants of the **bin packing** problem. Bari et al. [Bar+15] proposed an Integer Linear Programming (ilp) formulation to optimise the operational costs and usage while respecting the Service Level Agreements (SLAs). Jia et al. [Jia+16] also try to minimise operational costs and propose an online algorithm, i.e., an algorithm that handles requests as they arrive, to address the problem that they classify as the multiple-knapsack problem, a problem known to be NP-hard. Carpio et al. [CDJ17] address the problem of placing VNF with replicas to load-balance the network functions and try three optimisation methods (Linear Programming, Genetic Algorithm and a Random Fit Placement algorithm).

One sub-problem for the placement of VNF on a NFV infrastructure is the problem of the placement of VMs on Cloud infrastructures. Some surveys ([JS15; Wid+12]) analysed VM placement techniques aiming at improving the way VMs are placed on the baremetal machines that compose Cloud infrastructures. Mills et al. [MFD11] compared 18 VM placement algorithms inspired by bin-packing literature. They conclude that the choice of the algorithm for VM placement had no significant impact, but that the algorithm for the selection of the cluster, i.e., a subset of servers in the infrastructure with their own properties, leads to the most significant impact on resource optimisation.The objective of VM placement is usually to reduce cost for a customer. However Feng et al. [Fen+12] addressed it from the point of view of the provider and considered optimising the revenue as their objective. Mijumbi et al. analyse in [Mij+16] some of the existing projects related to NFV MANO. They concluded that current projects focused on centralised solutions which pose scalability imitations. Centralised solutions are usually *offline*, i.e., they do not handle requests as they arrive but compute a solution once a set of requests has been received. This is because either the method used, or the cost and duration of the computation, does not allow the online handling of requests.

All of the above, and many more, techniques are fitting centralised architectures. Most of them handle requests offline and are not scalable to large number of resources due to their computation costs. They do not consider the multi-domain architectures described in Section 2.3.

2.4.3 Problems of concurrent resource allocation from multiple providers

Multi-domain use cases require an orchestration of the resources of all domains. In some cases it can be possible to have a centralised orchestrator that manages the resources of all the domains. However, this type of solution is not always available or possible.

The rest of this section details an example of one of the problems that can arise when multiple centralised orchestrators are involved and each of them only manages the resources of its domain. The example shows the problem for a multi-domain SDN service.

Allocation use case This section introduces a problem faced by clients to allocate resources distributed across multiple SDN providers. The content is adapted from a paper originally published at the 21^{st} Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN) in 2018 [Fra+18b].

One of the SDN promises is the programmability of networks through APIs. Those APIs allow different users to access the network concurrently. Thus leading to the allocation of

dedicated *resources* by a given domain. These APIs allow *users* to integrate the *services* exposed by *domains* in their *applications*.

A *user* is an application or a person (e.g., system administrator) that uses NorthBound APIs to interact with the SDN domains.

Consider two providers α and β with no prior knowledge of each others and that cannot share information. No assumption is made on the architecture of the SDN controllers themselves: they can be either centralised (like NOX [Gud+08]) or distributed (like Onix [Kop+10] or Open Network Operating System (ONOS) [Ber+14]).

Figure 2.9 illustrates a use case with two users, Alice and Bob. They both want to allocate one resource from each of the two providers α and β. Depending on how their requests are processed, four results are possible.

1. Alice can allocate resource from both providers α and β. Bob cannot allocate any.

2. Symmetrically to the previous case, Bob can allocate resource from both providers α and β. Alice cannot allocate any.

3. Alice can allocate one resource from providers β and Bob one from α.

4. Symmetrically to the previous case, Alice can allocate one resource from provider α and Bob one from provider β.

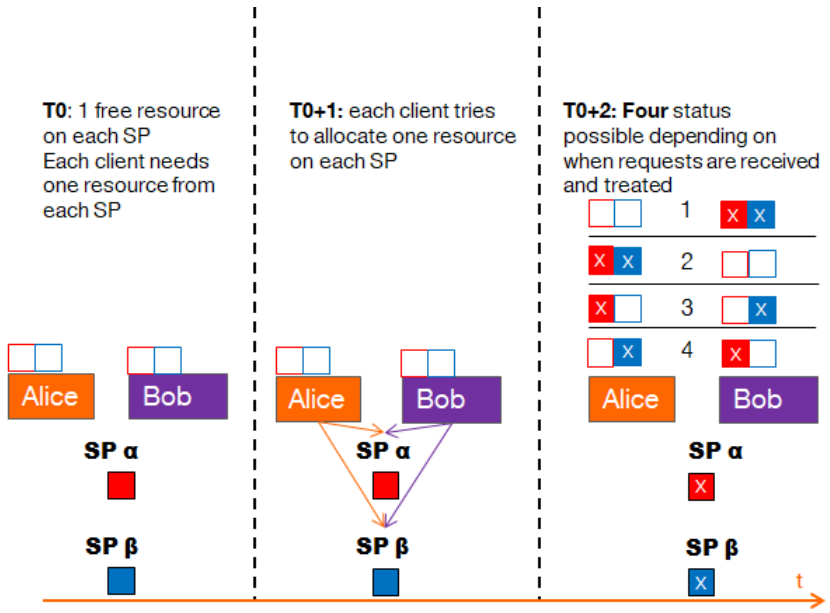


Figure 2.9: Illustration of the concurrent allocation of two critical resources from two SDN providers: 4 results are possible

Implementation A basic experiment to illustrate these four outcomes considers that the two users, Alice and Bob, both want to request the creation of two flows. One flow in each of the two distinct SDN networks of providers α and β. Each of the user expects that his applications hosted by these two providers can communicate once the network is configured. The architecture of the test-case is represented in Figure 2.10.

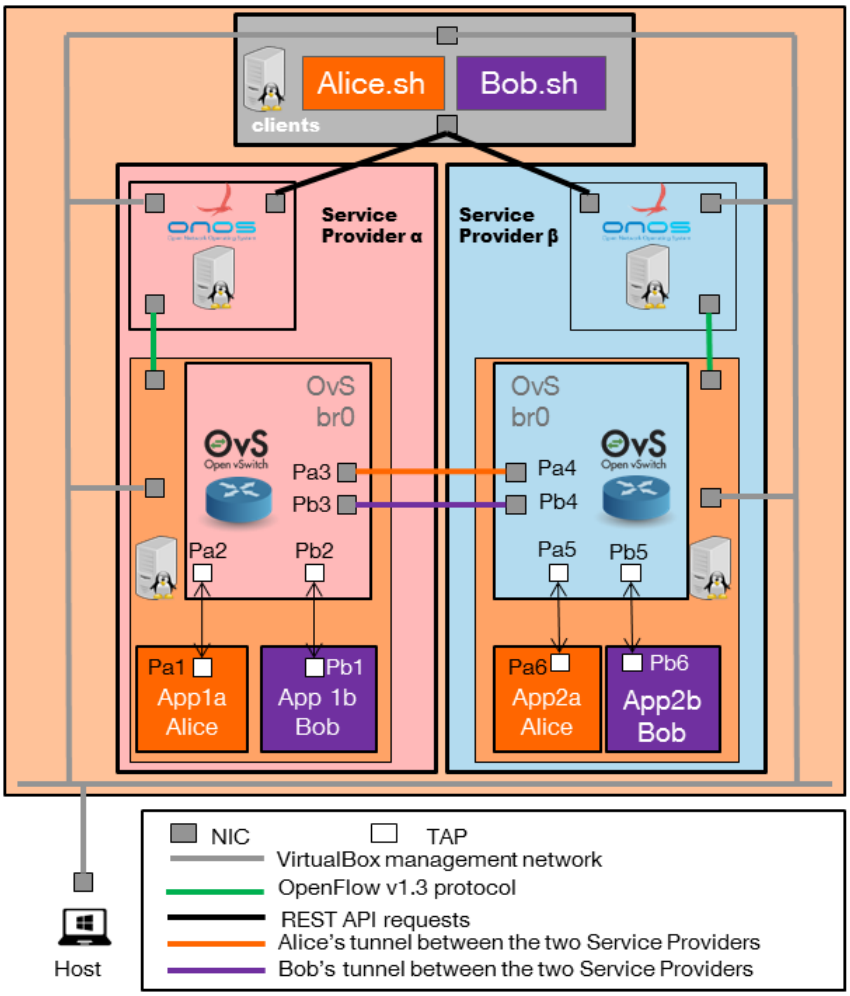


Figure 2.10: Sample test case: 2 authors Alice and Bob and 2 SDN providers

These two providers are each simulated by one instance of the open-source ONOS SDN controller [Ber+14]) and one virtual switch. Each ONOS controller is connected to its virtual switch using the OpenFlow 1.3 protocol as the SouthBound API. The expected result is that once the flows in each provider are set-up, App1a (resp. App1b) the application of Alice (resp. Bob) which runs in Domain α is able to send messages to App2a (resp. App2b) the application of Alice (resp. Bob) which runs in Domain β. They are able to communicate because flows will be created between the appropriate ports of the switches. Those applications App1a,b are Python applications that send ICMP ECHO messages displayed on the standard output by App2a,b. This is represented in Figure 2.11. For this Alice (resp. Bob) requests the creation of a first flow between Pa1 and Pa2 (resp. Pb1 and Pb2) to SP α. It then requests a second flow between Pa5 and Pa6 (resp. Pb5 and Pb6) to SP β. Tunnels are pre-configured for each user to allow direct communication between the two providers: between Pa3 and Pa4 for Alice, between Pb3 and Pb4 for Bob.

On a fifth VM, Alice and Bob are simulated by Bash and Python scripts that use the NorthBound REST API of ONOS to request the creation of flows on the controllers. Each script requests the creation of a single flow by each provider. An umbrella Bash script runs the Alice and Bob scripts in a uniform random order. Random timers are also introduced in the scripts to ensure that requests are not send in the same order every

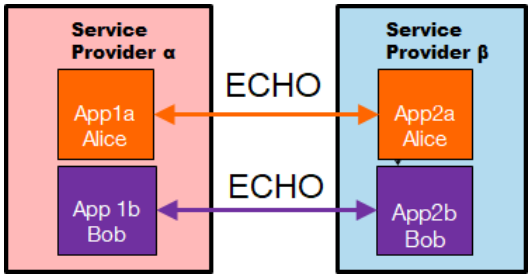


Figure 2.11: Use case implementation: each user runs a SDN application in each of the two domains α and β. Their applications send and receive ECHO messages

time. This allows the simulation of the concurrent allocation of resources by multiple users in varying conditions.

The resource considered in this example is the number of flows on a SDN switch. Memory limits the maximum number of flows that can be created on each switch. In the test environment it has been set very low to allow the creation of a single flow on each switch, using the *flow_limit* parameter in Open vSwitch.

One simulation consists on running the umbrella script, which in turn runs the two Alice and Bob scripts. After each simulation the configurations of the switches are reset. All the VMs are hosted by the same computer running the VirtualBox hypervisor.

When the Alice and Bob scripts run concurrently the four outcomes introduced above in Figure 2.9 are expected. At best only one user gets the two resources and the other is left waiting. Worst cases are the cases labelled 2 and 3, when none of the user gets all the resources required and both wait indefinitely.

Table 2.1 shows the results when the simulation is run 1000 times.

Result	Occurrence
Alice gets none, Bob gets two resources	33.8%
Alice and Bob gets one resource each (either from α or β)	26.8%
Alice gets two resources, Bob gets none	39.4%

Table 2.1: Sample result observed when simulation is run 1000 times

In this test 26.8% of the time each script/user only gets one of the two resources it is expecting. Alice got both resources 39.4% of the time and Bob 33.8% of the time. As there are four possible outcomes and requests are uniformly random, one could expect that each user gets both resources 25% of the time, and in the remaining 50% they each get only one resource. It is possible to obtain a result closer to these theoretical values but this would require to coordinate the users. This shows that the results are very difficult to predict. They depend on when each user sends his requests, when these requests are received and how they are managed by the providers.

As expected due to the random factors, running this simulation multiple times leads to slightly different numbers. However, this result is sufficient to show that a significant number of times, in this run 26.8%, none of the user can move forward as it only managed to allocate one of the two resources he required. In this case both users are said to be

experiencing **starvation**, they need to synchronize their allocations so they can each get the resources in turn.

Starvation occurs because the resource satisfies some properties:

- **unshareable**: the resource can be used by only one user at a given time,
- **not preemptable**: only the user who has allocated the resource can release it,
- **not interchangeable**: no other resource from another component of the system can be allocated to obtain the same result.

The resource considered here, i.e., the *flow_limit* parameter in ONOS, satisfies these properties. Additionally the problem occurs only because in the experiment this parameter has been configured at a very low value. In a live production environment the problem would only occur if the two providers had reached their full capacity. This is unlikely to happen because engineering teams of production platform usually plan capacity to avoid this situation. For this reason this use case was not considered further after this first experiment and the next chapters focus on a different use case.

2.5 Conclusion

Network infrastructures leverage new paradigms to enable new use cases. Some of these new use cases can be multi-domain which pose the question of the adequacy of the centralised managers usually found in networks. With the apparition of network slices, the resources of network providers can be shared not only by multiple services, but also with services that use resources from other providers. Centralised solution might not be able to scale across all resources of all the domains, or a network service provider might not want to allow an external manager to access their resources. Allocating resources for network slices across multiple domains could even lead to starvation if the centralised managers in each domain do not share information as showcased in a SDN example.

This might be an opportunity to consider a distributed solution for the allocation of resources in networks. The requirement to allocate chains of network functions inside these slices create an additional constraint on the order in which the resources are allocated. The next chapter provides a state of the art of distributed solutions for the allocation of resources. Then, a new algorithm for the distributed allocation of resources is introduced in Chapter 4. Multiple heuristics for this algorithm are then evaluated in Chapter 5. The performance of the algorithm is compared to that of algorithms from the state of the art in the last chapter.

Chapter 3

State of the art

Contents

> He's clever, [...], I must admit there are some smart people even among the intelligentsia.
>
> Mikhail Bulgakov, *The Master and Margarita*

The previous chapter introduced the problem of the allocation of resources for networks and the motivation for a distributed solution. The distributed allocation of resources is typically addressed as a variant of the Mutual Exclusion problem first described by E.W.Dijkstra in 1965 [Dij65]. Distributed resource allocation is not a new approach for resource allocation in networks. Several papers [BHJ02; RWX04] have considered mutual exclusion algorithms for the dynamic assignment of channels in radio spectrum. Badrinath et al. proposed in [BAI94] to use two classic mutual exclusion algorithms [Le 77; Lam78] for Mobile Ad hoc wireless NETworks (MANET). They propose modifications of these algorithms to consider the mobility of nodes to reduce the search cost and the number

of messages. Following this initial work, other algorithms have been proposed to address this specific problem [WK97; WCM01; WWV01; BVP02; Ben+04].

This chapter first describes in more details the model for distributed systems used throughout this work in Section 3.1. Next, section 3.2 details a state of the art of the variants of the Mutual Exclusion problem and the multiple solutions that have been proposed to solve them. The last section 3.3 explains how the performance of these algorithms is evaluated and shows how they compare to each other.

3.1 Definition and model for distributed resource allocation

3.1.1 Definition of distributed mutual exclusion

The Mutual Exclusion problem was first described by Dijkstra in 1965 [Dij65] for systems with N **nodes** that share one resource. In such systems each node, sometimes also called **site**, is an autonomous computing unit that can run **processes**. Each node runs a Mutual Exclusion algorithm locally that allows it to enter the **Critical Section (CS)** and to use the resource exclusively. When a node needs to use the resource it execute **requestCS** function. When it is done using the resource it runs a **releaseCS** function. The Mutual Exclusion problem is the coordination of all the nodes such that a single node can enter its CS at a given time, i.e., when the node is the sole user of the resource while the others are waiting. It is assumed that the CS has a finite duration, i.e., the resource will eventually be released. A **conflict** occurs when multiple nodes are requesting the same resource at a given time. An extensive formal description of distributed systems has been made by Lynch et Fischer in 1981 [LF81].

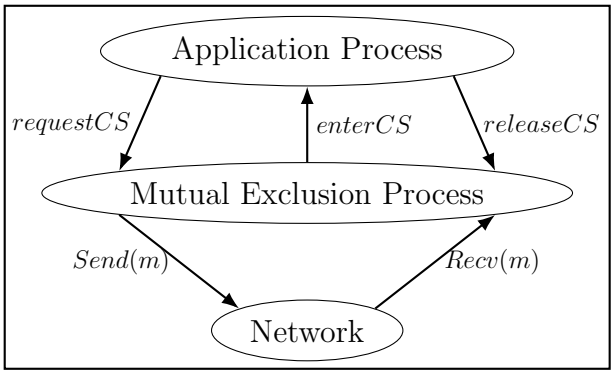


Figure 3.1: Distributed mutual exclusion system architecture (from [WW11])

A representation of the components of the system appears in Figure 3.1, courtesy of Welch and Walter [WW11]. Each node executes two processes :

- the mutual exclusion process which manages the access to the CS
- the application process, that requires to enter the CS, to use the resources and release the CS

The application process can request the Mutual Exclusion process to enter the CS by calling the *requestCS* function. The Mutual Exclusion process communicates with the

other nodes: it can send and receive messages using *Send(m)* and *Recv(m)* functions. Once it becomes possible, depending on the algorithm used, the Mutual Exclusion process notifies the Application Process that it can enter the CS using the *enterCS* function. Once the application process is done using the resource, it notifies the Mutual Exclusion process that it wants to leave the CS and releases the resource by calling the *releaseCS* function

In this specific system it is assumed that there is a single shared resource, but later works proposed extensions detailed below. Some of these works address systems with multiple resources, cf. Section 3.2.2. In this document, the terminology used is **types of resources**. Other works address systems with multiple **instances** of resources, cf. Section 3.2.3.

3.1.2 Properties of the Mutual Exclusion

A solution to the Mutual Exclusion problem needs to satisfy two properties :

- ***safety***: there is at most one process in the CS
- ***liveness***: each request has to be satisfied within a finite time (under the hypothesis that the allocated resources are eventually released).

If the *liveness* property is not respected, i.e., a node waits indefinitely to get its requests satisfied, it is said to be experiencing **starvation**.

All the solutions proposed to guarantee the liveness rely on a **total order of the requests**.

3.1.3 Description of the system

For the reminder of this document, the system follows an *asynchronous timing model* as defined in [Lyn96], i.e., the separate components take steps in arbitrary order. The system is an *Asynchronous Message-Passing System* and is assumed to have the following properties:

1. There are N nodes in the system.
2. The system is modelled as a non-directed connected communication graph.
3. Communication links are **reliable**. All messages are eventually delivered, they are never duplicated.
4. Nodes are **reliable**. In particular the failure of nodes or the apparition of new nodes is not taken into account.
5. Nodes communicate with each other only by **message-passing**.
6. Communication times between nodes is **bounded** and can be unknown.
7. **No common clock** is shared by the nodes
8. The time spent in its CS by a node is **finite**, i.e., a node use the resources that has been allocated to him for a finite duration.

3.2 Distributed algorithms for the allocation of resources: state of the art and taxonomy

3.2.1 One instance of one type of resources: the Mutex problem

Several taxonomies of the original Mutual Exclusion problem have been proposed: [Ray91b; Sin93; Vel93]. The taxonomy proposed below builds on these and extends them with newer variants of the problems and newer algorithms.

Figure 3.2 shows a sample system for this problem. This system is a 5-node network and there is a single instance of one type of resources held by one of the node, in green in the figure. The edges are the edges of the communication graph.

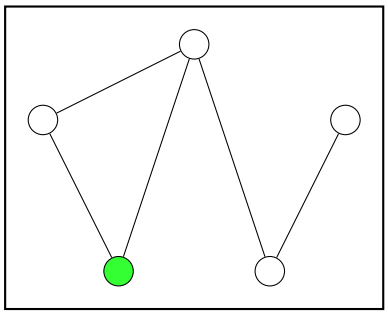


Figure 3.2: Sample system with one instance of one resource, held by the green filled node.

Early solutions: shared memory

The problem of Mutual Exclusion was initially introduced for systems with one instance on one type of resources. Initial solutions by Theodorus J. Dekker (referenced in [Dij65]), for systems with only two processes, and Dijkstra [Dij65], for systems with n processes, were designed for systems relying on **shared memory**. In such systems all the nodes access a shared memory concurrently using an atomic operation that guarantees that only one of them accesses them at a given time, i.e., it relies on a lower level (hardware, Operating system) mechanism like *compare-and-swap*. Dijkstra's algorithm was improved first by Donald Knuth in 1966 [Knu66]. Then de Bruijn improved Knuth's solution in 1967 [dBru67] to reduce the maximum number of turns, i.e., the number of loop iterations, before a process enters its CS to $(1/2) * N * (N - 1)$. Eisenberg and McGuire proposed a further improvement [EM72] so that no node has to wait more than $(N - 1)$ turns to enter its CS.

Leslie Lamport proposed a new solution in 1974 [Lam74], known as the Bakery algorithm as it is inspired by the method used by some bakeries (or other shops for that matter) to give tickets with a unique number to customers. This solution [Lam19] is the first that does not require a lower-level Mutual Exclusion.

Mutexes are today available in most Operating Systems for the synchronisation of the threads of a program, e.g., in the Linux kernel [1], in the FreeBSD kernel [2] or the Windows Win32 API [3]. Most programming languages offer abstractions (lipthread for C, std::mutex

[1] `https://www.kernel.org/doc/html/latest/locking/mutex-design.html`
[2] `https://www.freebsd.org/cgi/man.cgi?query=mutex&sektion=9`
[3] `https://docs.microsoft.com/en-us/windows/win32/sync/mutex-objects`

in C++[4], or have built-in mechanisms for concurrency that avoid the programmer to address the problem himself. Erlang[5] and Go [6] are two examples of languages based on the Communicating Sequential Processes (CSP) model [Hoa78] which relieves the developer from having to manages the concurrency himself. Java's API proposes *synchronized* blocks and *ReentrantLock* Objects.

Message passing solutions and first permission-based algorithms

Distributed systems evolved and it became more common to have independent nodes with no shared memory but that were able to communicate by the passing of messages. All the other algorithms presented below rely on the passing of messages. It is possible to use message passing to emulate shared memory [Cad+17], so algorithms using shared memory can be adapted to message passing.

New Mutual Exclusion algorithms based on message passing appeared in the late 1970s. Lamport's 1978 landmark algorithm [Lam78] introduced the notion of **logical clocks** to compute a total order of the CS requests based on logical timestamps. This algorithm requires $3 * (N - 1)$ messages in a complete communication graph. This algorithm was improved by Ricart and Agrawala in 1981 [RA81] by removing the need for acknowledgement messages, thus lowering the number of required messages to $2 * (N - 1)$. O. Carvalho and G. Roucairol [CR83] in turn proposed a variation of the algorithm from Ricart and Agrawala that can require less messages by considering asymmetric nodes (i.e., all nodes do not have the same exact configuration at the time of their initialisation). Sanders [San87] introduced in 1987 a terminology for this first type of algorithms: **permission-based**.

Permission-based algorithms enforce the *safety* property by the reception of a sufficient number of permissions from other nodes.

Quorum-based permission-based solutions

Another approach proposed was to divide the topology in smaller subsets where each node is part of one and only one of these subsets. Maekawa's 1985 algorithm [Mae85] relies on properties of **finite projective planes** to show that each subset needs to be only of size $\sqrt{N}$. However it is prone to deadlocks [Sin93]. Other algorithms based on quorums where proposed in [San87; AE91; MNR91].

Token-based algorithms

Gérard Le Lann proposed a solution in 1977 [Le 77] where the nodes are connected in a ring topology and send **tokens**. This solution is similar to how token ring networks (IEEE standard 802.5 [IEE04]) work: a control token guarantees that a single node is transmitting at a given time. Alain K. Martin's algorithm [Mar85] relies on a similar ring topology.

After Carvalho and Roucairol proposed a modification of their algorithm, Ricart and Agrawala replied with a modification [RA83] of their message-passing algorithm to use a token that allows the node that holds it to enter its CS, in this algorithm the token is transmitted along a virtual topology that connects the nodes. Suzuki and Kazami [SK85]

[4] `https://en.cppreference.com/w/cpp/thread/mutex`

[5] `https://cacm.acm.org/magazines/2010/9/98014-erlang/fulltext`

[6] `https://golang.org/doc/faq#csp`

proposed in 1985 an extension of Ricart-Agrawala to transform it to a token-based algorithm using an additional PRIVILEGE message. Neilsen and Mizuno's algorithm [NM91] builds a Directed Acyclic Graph (DAG) virtual topology with the nodes to circulate a token. Mukesh Singhal proposed in [Sin89] an algorithm aided by heuristic to deduce the current location of the token. Several algorithms have been proposed relying on a similar static-tree structure to send a token [vdSne87; TN87].

Some algorithms relaxed some of the usual assumptions found in other works. In Raymond [Ray89b] and Helary-Plouzeau-Raynal [HPR88] algorithms nodes only communicate with their neighbours, they do not require a knowledge of the whole topology.

Naimi-Tréhel [NT87] improved the complexity of existing token-based algorithms to logarithmic by using a dynamic tree structure. This algorithm is further described in Section 6.5.1.

Mishra and Srimani [MS90] proposed two fault-tolerant Mutual Exclusion algorithms. The first one is an extension of [SK85], but ensure that the system can recover when a node failure is detected. The second one assumes that at any moment one of the nodes acts as a centralised controller. Nishio et al. [NLM90] also proposed an extension of [SK85] to make it resilient to failures. Sopena et al. [Sop+05] proposed a fault tolerant extension of Naimi-Tréhel [NT87].

Ye-In Chang [Cha94] builds on Andrzej Goscinski work by [Gos89; Gos90; Gos91] to propose extensions of several algorithms, [SK85; Sin89; Ray89b; Lam78; RA81], to address real-time systems. In those variants requests are ordered based on priorities instead of the time when they happened. Frank Mueller also proposed algorithms based on priorities [Mue98; Mue99].

At first permission-based algorithms required more messages to gather permissions from the nodes than what was needed by token-based algorithms to circulate a token. However, Naimi-Tréhel [NT87] (token-based) and Agrawal-El Abbadi [AE91] (permission-based) both have complexities that are logarithmic with the number of messages, cf. the comparison the complexities of algorithms in next section 3.3. Token-based algorithms are considered more prone to failure, as on top of the possible node failures it is possible that the token gets lost or duplicated.

3.2.2 One instance of n types of resources: dining/drinking philosophers

The Mutual Exclusion problem was first generalised for multiple types of resources in 1971 by E.W. Dijkstra [Dij71] as the **Dining Philosophers Problem (DiPP)**. In this formulation of the problem, 5 philosophers are sitting around a table, and a table is laid for them to dine. On the table is served a spaghetti dish that can only be eaten with two forks. Each philosophers is sitting in front of one plate and there are two forks on each side of each plate. This forbids two neighbours to eat simultaneously. Such system is represented in Figure 3.3.

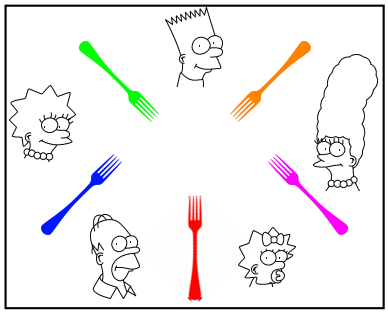


Figure 3.3: Sample system with one instance of n types of resources: the dining philosophers problem

The algorithm proposed by Dijkstra [Dij71] is an **incremental** algorithm : the resources are allocated incrementally according to a total order of the forks. It has a side effect called the *domino effect* [Ran75]. This algorithm is detailed in Section 6.2. Nancy A. Lynch [Lyn80; Lyn81] generalised the formulation of the problem to a graph with any number of nodes representing the resources connected via edges that represent the resource sharing constraints, and she proposed a graph-colouring approach to improve the performance of Dijkstra's algorithm. Francez-Rodeh [FR80] proposed a symmetric distributed solutions, i.e., a system where all nodes have the same exact configuration at the time of their initialisation. Lehmann-Rabin [LR81] proposed a probabilistic solution for symmetric systems and proved that there is no perfectly symmetric non-probabilistic solution to the DiPP. Chandy-Misra [CM84] relaxed this symmetry constraint and proposed a DiPP algorithm, detailed in Section 6.3, based on the acyclicity of the graph where forks are transmitted along the edges. The directions of the edges at any time guarantee that there is no cycle in the graph. Page et al.[PJC93] proposed an algorithm that improved the performances. Styer and Peterson[SP88] improved the worst case time and failure locality (cf. definition in Section 3.3) of Lynch [Lyn80; Lyn81].

In the same 1984 paper Chandy-Misra [CM84] also propose a further generalisation, the **DrPP**. In the DiPP **requests** are static: each node (philosopher) always requires the same subset of resources, i.e., his left and right forks. In the DrPP, the requests are dynamic, each node can request a different subset of the resources (bottles) at each request. An algorithm that solves the DrPP can also solve the DiPP. Even if the opposite is not true, some DiPP algorithms can be extended to solve the DrPP problem.

Several algorithms that target the DrPP problem, like Chandy-Misra, require that the **conflict graph** is known a priori. The nodes of the *conflict graph* are the requests. If two requests need a common resource there is an edge between the two corresponding nodes. By definition for the DiPP problem the *conflict graph* is always known a priori because requests are static.

Chandy-Misra [CM84], in the same paper, propose a DrPP algorithm that uses their DiPP algorithm as a first subroutine. The forks are not the resources but are *auxiliary resources* that allow the requesting node to allocate the bottles, which are the actual resources. Welch-Lynch [WL93] generalised the idea of *modular* algorithms and their solution can use a DiPP algorithm to select a node (resp. philosopher) which enters a dining CS, then once the node is in a dining CS it can request the resources (resp. bottles) it needs.

Ginat-Shankar-Agrawala proposed an algorithm [GSA89] that does not use a DiPP subroutine. It is based on a time-stamping approach similar to Lamport's logical clocks.

Figure 3.4 shows a system with one instance of n types of resources. In this example $n = 5$ and each type of resources is represented on the nodes by a different shape and colour. Each edge represents a communication link between two nodes. Contrary to the settings of the DiPP problem, in this system a resource can be shared by more than 2 nodes of the system in Figure 3.3 and any node can request any resource. Algorithms that address the DiPP problem can not easily be adapted to such a system.

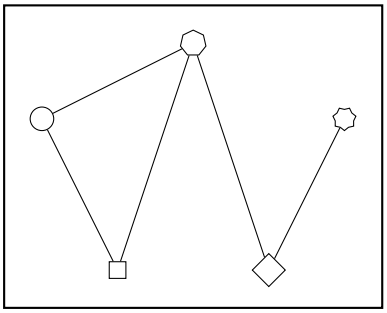


Figure 3.4: Sample system with one instance of n types of resources, edges represent the communication graph

Algorithms have been proposed to address such systems. Bouabdallah-Laforest [BL00] relies on Naimi-Tréhel's mutex algorithm [NT87] to circulate a *control token* along the virtual tree. A node needs to own the Control Token to request its resources. This algorithm is detailed in Section 6.5.

Aomar Maddi [Mad97] calls this problem the *AND-Synchronisation problem* where each process can get an exclusive access to a set of resources rather than to a single one. The paper distinguishes between *implicit* requests (when processes wait for a token to arrive) and *explicit* requests (when requests are sent to the processes). He introduces two algorithms, one for each case. In the algorithm for *explicit* requests each process holds a token for its resource, the tokens are sent to the requesting process based on a virtual clock. The algorithm for *implicit* requests sends these tokens along a virtual ring topology.

Awerbuch-Saks [AS90] proposed a further generalisation called the **dynamic job scheduling** problem. In this description of the problem, **jobs** request any number of resources and need to be scheduled. In this model, the jobs are the nodes and they are *dynamic*, i.e., jobs come and go. The objective of the algorithm is to minimise the waiting time (or delay) between the moment a request is initiated and the moment when the resources are allocated, cf. Section 5.1.1 for a formal definition. It uses a **distributed queue** to achieve a polynomial-bounded waiting time that is function of the number of conflicts. Bar-Ilan and Peleg [BP92] proposed a further improvement on performances for synchronous complete communication systems. Another extension was proposed by Choy-Singh [CS95] to include mechanisms for fault-tolerance using the doorway mechanism in Lamport's Bakery algorithm [Lam74]. Rhee's moduler algorithm [Rhe95; Rhe98] builds a distributed queue but like Welch-Lynch [WL93] starts by using a DiPP or DrPP as a subroutine to get a lock on the set of resources. Instead of keeping the lock for the duration of the CS it uses it only to adjust the total order of the requests in the global scheduler, and contrary to Welch-Lynch [WL93] it improves the performances of the algorithm used as a subroutine. Rhee's algorithm is detailed in Section 6.4. Weidman et al.[WPP91] also proposed to reuse Chandy-Misra as a subroutine for their dynamic allocation problem, with the same response time and message complexity as Chandy-Misra's. Sivilotti [SPS00]

proposes an algorithm that reuses Chandy-Misra's mechanism of using forks as a mean to provide dynamic priorities, and Choy-Singh's [CS95] mechanism of double doorway of thresholds to allow low-priority nodes to overtake high-priority nodes.

The Lejeune-Arantes-Sopena-Sens (LASS) algorithm [Lej+15] argue that the algorithms mentioned above all require a **global lock** to serialise the requests. Their algorithm does not require such global lock, and does not use a distributed queue either. Instead it relies on *allocation vectors* of counters to compute a total order of the requests. It does however make the assumption that resources are not managed locally by the nodes holding them and moves the resources along the graph during its execution.

3.2.3 m instances of one type of resources: k-mutex, k-out of-M

Two sub-problems have been defined for systems with m instances of one type of resources. Figure 3.5 shows a system where there are three instances, in green, of a single type of resources. Edges represent the communication links.

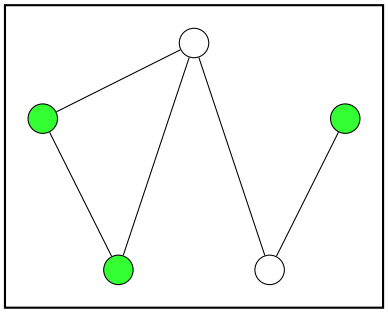


Figure 3.5: Sample system with m green filled instances of 1 type of resources

The first sub-problem is known as the *k-mutex*, or as the l-exclusion problem. The system allows up to k entries to a CS. This problem is solved by algorithms such as the permission-based ones proposed by Raymond [Ray89a](which extends Ricart-Agrawala), and Srimani-Reddy [SR92] (which extends Suzuki-Kasami). Token-based solutions have been proposed by Makki et al. [Mak+92], Bulgannawar-Vaidya [BV95] (an extension of Tréhel [TN87]) and Chaudhuri-Edward [CE06; CE08]. Kakugawa et al. [Kak+94] proposed a solution based on an extension of coteries, called k-coterie, and using logical clocks. A common data type to solve this problem in programming language is known as the **counting semaphore**.

The second sub-problem is when one process tries to allocate k instances of the same type ($k < M$) called the k-out of-M resources allocation problem. It has been defined by Michel Raynal [Ray91a] who proposed an extension of Ricart-Agrawala for the problem as the allocation of k instances sequentially using a Mutual Exclusion algorithm could lead to a deadlock. This requires that the algorithms satisfies the *safety* and *liveness* properties. The definition of the *safety* property is different than for the classical Mutual Exclusion problem: the number of resources which are allocated to the processes at any time is always less than or equal to M (each resource being allocated to only one process at a time). The definition for *liveness* remains the same. Roberto Baldoni proposed a permission-based quorum-based solution as an extension of Maekawa [Bal94]. Manabe and Tajima [MT99] proposed a permission-based algorithm based on logical clocks.

3.2.4 m instances of n types of resources

A more general problem is shown in Figure 3.6 shows for a system with $n = 3$ types of resources, and 1 (for the green resource) or 2 (for the red and blue resources) instances of each of them. Edges represent the communication links.

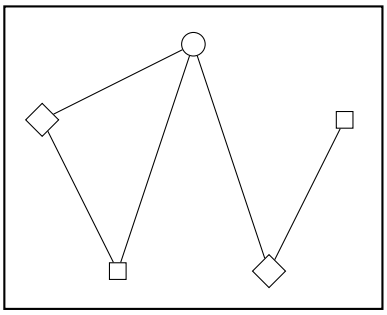


Figure 3.6: Sample system with m instances of n types of resources

No algorithm has been proposed specifically for systems with multiple instances of multiple types of resources. In this section are listed algorithms already mentioned in the sections above and that proposed extensions to this more general problem. Multiple algorithms addressing the DrPP have proposed extensions for cases where there are multiple **instances** of each resource. They all consider that instances are all equivalent and can be used indistinctly.

Token-based algorithms consider one token per instance. Bouabdallah-Laforest [BL00] mentions a more general problem than the DrPP where several instances of each type of resources are available. The extension proposed is to use a token per instance instead of a token per type of resources. Upon reception of the token a node can choose among the available instances. Maddi [Mad97] algorithm also proposes to circulate a token per instance for systems with multiple instances.

Ginat-Shankar-Agrawala [GSA89] proposed two extensions of their DiPP algorithm to address systems with more than one instance of each resource. In their baseline model each node holds one instance of one type of resources. In the first extension, a node can hold *multiple instances* of one type of resources. In the second extension a node can hold *multiple types* of resources and multiple instances of each type. They propose to add a notion of *quantity* to their algorithm.

Other DrPP algorithms do not address explicitly extensions for multiple instances but can be adapted. Chandy-Misra [CM84] DrPP algorithm allows only a single node to enter its CS and use resources at a given time. Adapting it for systems with multiple instances only requires to consider that a fork can stand for any number of instances. Rhee [Rhe95] is dependent on the algorithm used as a subroutine to lock the resources. If the algorithm chosen can handle systems with multiple instances of each type of resources then a solution would be to consider that each resource manager needs to handle a queue per instance.

It should be noted that it can be possible to adapt the problem so that a DiPP or DrPP algorithm can be used. None of the algorithms considers the placement of the instances. Selecting any instance arbitrarily can result in a higher latency due to the distance between nodes. In this case it is possible to consider each instance as a new type of resources by itself. The problem then becomes a DiPP or DrPP. The algorithm presented in the next chapter does just that. It tries to minimise the overall length of the path connecting the nodes holding the types of resources requested and to achieve that

consider that each instance is different depending on its location in the communication graph.

3.2.5 Classification of algorithms

A classification of the distributed algorithms for the allocation of resources has been proposed by Jonathan Lejeune in its PhD thesis [Lej15], it is represented in Figure 3.7. It sums up what has been described above. The least general problem is the original mutex problem for systems with 1 instance of 1 resource, and the most general is for systems with m instances of n types of resources. A solution that addresses a problem can also be used to solve a less general problem. For example a solution to the DiPP problem can also solve the mutex problem.

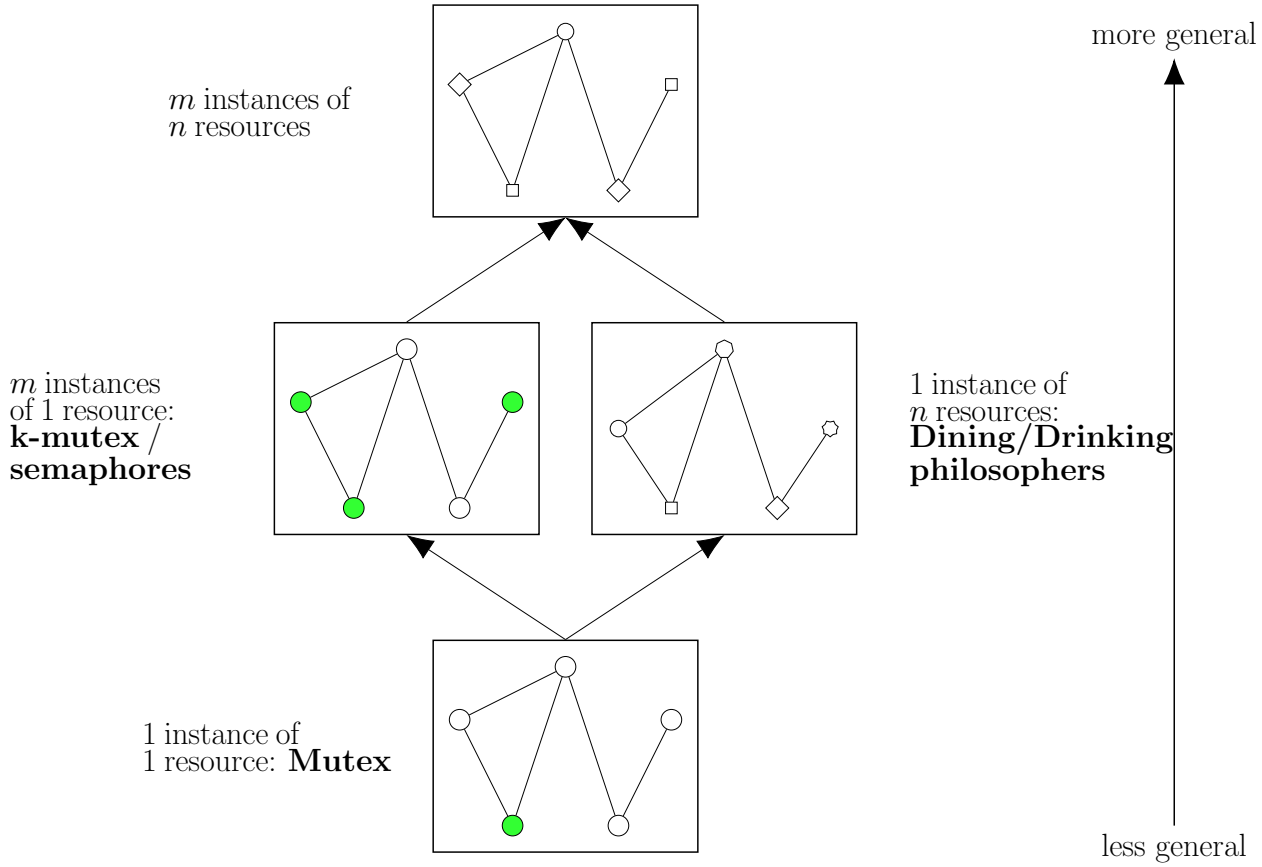


Figure 3.7: Classification of problems for the distributed allocation of resources

Figure 3.8 shows a timeline of the different problems and algorithms mentioned above, as well as the relationships between the various solutions.

3.3 Performance evaluation and comparison

In his taxonomy [Sin93], Mukesh Singhal lists three main metrics to compare the performances of Mutual Exclusion algorithms :

- the **message complexity**: the total number of messages sent and received by a node for each request to access the shared resources.

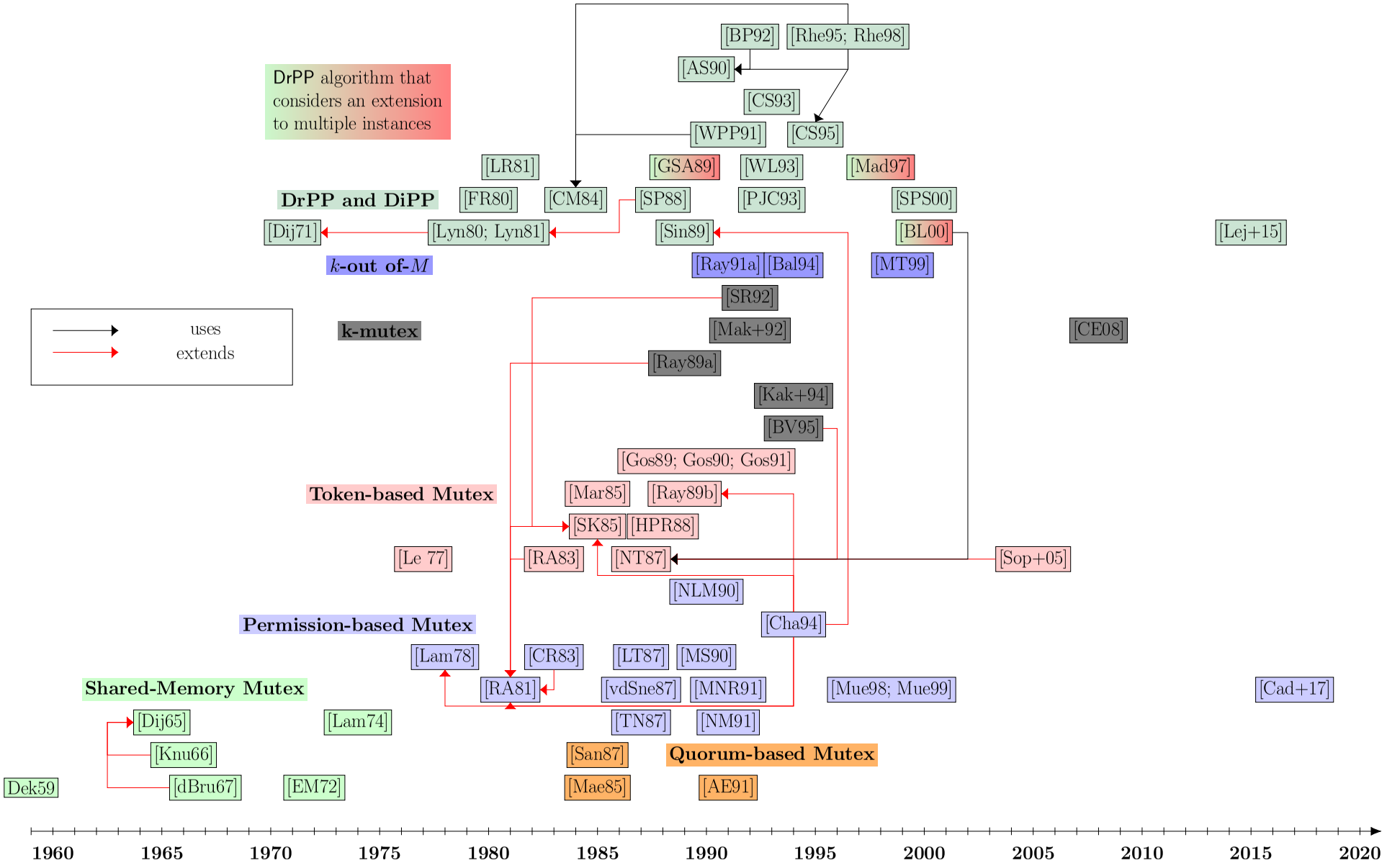


Figure 3.8: Timeline

- the **response time**: the time interval a request waits to enter its CS after its request message has been sent out
- the **synchronisation delay**: the number of sequential messages required after a node leaves its CS for a new node to enter its CS

Nancy A. Lynch's 1980 paper [Lyn80] is perhaps the first to consider the response time. Before that the most common metric considered was the number of messages.

Choy and Singh [CS95] proposed an additional metric **failure locality**. They define the *m-neighbourhood* of a node as the set of nodes at a distance of a most m from the node in the *conflict graph*. They define the failure locality of an algorithm to be m if a process is free from starvation even if processes outside of its m-neighbourhood have failed.

Table 3.1, taken from [Vel93] shows the message complexity of some permission-based algorithms. Table 3.2, also from [Vel93], shows the performance of some token-based algorithms. In these tables N stands for the number of nodes in the system.

Authors	Problem	Message complexity
Dijkstra[Dij65]	shared-memory	-
Eisenberg and McGuire[EM72]	shared memory	-
Lamport's bakery[Lam74]	shared memory	-
Lamport's Logical clock[Lam78]	message passing	$3 * (N-1)$
Ricart and Agrawala[RA81]	message passing	$2 * (N-1)$
Carvalho-Roucairol[CR83]	message passing	0 to $2 * (N-1)$
Maekawa[Mae85]	message passing/quorum	$i * \sqrt{N}, 3 \leq i \leq 5$
Sanders[San87]	message passing	$\lvert I_i - \{i\}\rvert + 2(\lvert R_i - \{i\}\rvert)$
I is the inform set and R the request set, I is a subset of R and $\lvert I\rvert < \lvert R\rvert$		
Raynal[Ray89c]	message passing	$2 * (N-1)^2$
Agrawal-El Abbadi[AE91]	message passing	$O(log(N))$
Singhal[Sin92]	dynamic structure	$(N-1)$ to $3 * (N-1)/2$

Table 3.1: Performance of permission-based Mutex algorithms, after [Vel93]

Authors	Problem	Message complexity
Le Lann[Le 77]	message passing	
Ricart and Agrawala[RA83]	message passing	N
Suzuki and Kazami[SK85]	token	$L * N + (N-1)$
L is the number of Mutual Exclusion entries		
Naimi-Tréhel[NT87]	token /dynamic tree	$log(N)$
van de Snepscheut[vdSne87]	token /static tree	$log(N)$
Raymond[Ray89b]	token /static tree	max:$2 * D$, average: $log(N)$
D is the diameter (longest path length) of the tree		

Table 3.2: Performance of token-based Mutex algorithms, after [Vel93]

Rhee [Rhe95], extending results from [AS90] and [CS95], describes the complexity of several DiPP and DrPP algorithms. This is shown in Table 3.4. The variables used in the formulas are described in Table 3.3.

Variable	Value represented
δ	maximum number of conflicting processes at any time
U	number of nodes IDs
r	maximum number of resources in a request
k	maximum number of resources in the system

Table 3.3: Variables used for performance evaluation

Authors	Pb	Message complexity
Dijkstra[Dij71]	DrPP	$O(N)$
Lynch[Lyn81]	DiPP	$O(\delta)$
Styer and Peterson[SP88]	DiPP	$O(\delta^{log(\delta+1)})$
Choy and Singh[CS92]	DiPP	$O(\delta)$
Page et al.[PJC93]	DiPP	$O(\delta^2)$
Chandy and Misra[CM84]	DrPP	$O(N)$
Rhee[Rhe95]	DrPP	$O(max\{\delta^2, r\delta\})$
Awerbuch and Saks[AS90]	dyn	$O(\delta^2 log(U))$
Weidman et al.[WPP91]	dyn	$O(\delta)$
Bar-Ilan and Peleg[BP92]	dyn	$O(\delta log(U))$
Choy and Singh 1[CS93]	dyn	$O(\delta^2 + \delta log * (U))$
Choy and Singh 2[CS93]	dyn	$O(\delta log * (U))$
Choy and Singh 3[CS93]	dyn	$O(\delta^2 + \delta log * (U))$
Rhee[Rhe95]	dyn	$O(max\{\delta^2 + \delta log * (U), r\delta\})$
Ginat et al.[GSA89]	dyn	> 0 and $< 2^k$
Bouabdallah-Laforest[BL00]	dyn	$O(log(N)$

Table 3.4: Performance of DiPP and DrPP algorithms, from [Rhe95]

3.4 Conclusion

Chapter 2 introduced the problem of the allocation of resources in networks addressed in this work. None of the distributed algorithms presented in the above state of the art address all the constraints. There are few algorithms for systems with multiples instances of multiples types of resources and none of them address the placement of the resources in the topology. None of them address the order in which the resources are used, required for chains of functions, either. Additionally some have prerequisites, like the a priori knowledge of the *conflict graph*.

The next chapter introduces an algorithm that address all the requirements of the problem statement with no new prerequisite. The following chapters show the evaluation of the heuristics proposed for the algorithm, as well as the comparison with some of the algorithms introduced above: Dijkstra's incremental algorithm [Dij71] as well as Chandy-Misra [CM84], Rhee [Rhe95] and Bouabdallah-Laforest [BL00] DrPP algorithms.

Chapter 4

A distributed algorithm for the allocation of resources

Contents

It's educational

Pixies, *U-Mass*

This chapter details a distributed algorithm for the allocation of resources in networks, addressing the problem described in Chapter 2 to allocate ordered chains of VNFs. The

algorithm was first described in the paper *"A Resource Usage Efficient Distributed Allocation Algorithm for 5G Service Function Chains"* published at the 2020 Distributed Applications and Interoperable Systems (DAIS) conference [Fra+20].

The algorithm is modular and consists of two consecutive subroutines:

- the **path computation** subroutine, detailed in Section 4.2, in which the algorithm selects the instances of types of resources to be allocated and computes a routing path between them. This path respects the order present in the request.
- the **allocation** subroutine, detailed in Section 4.3, in which the algorithm allocates the resources selected during the path computation subroutine.

Each of these subroutines can choose among multiple heuristics depending on the objectives. Possible heuristics for both subroutines are introduced in Section 4.5, along with their pros and cons.

The algorithm provides a solution to the problem statement described in Section 2.4 using the system described in Section 3.1.3. It allocates requests for an ordered set of resources in a system with multiple instances of multiple types of resources. It does not assume that the communication links are First In First Out (FIFO), i.e., messages can arrive in any order. It also does not assume that the nodes have a knowledge of the full communication graph. No a priori knowledge of the *conflict graph* is required either.

4.1 Variables of nodes and messages

Each subroutine relies on multiple types of messages. Table 4.1 shows which message uses which variable. The variables sent with the messages are described in Table 4.2. The messages and how they use these variables are described in the next sections.

The algorithm requires nodes to keep track of several *local variables*. They are listed in Table 4.3. These *local variables* are used by both subroutines and are prefixed by **self.** in the pseudo-code in the next sections. The types defined for the variables are the types used in this chapter, but an implementation of the algorithm could choose other types for some of them. For example, the identifier of nodes (*id* variable) could be integers instead of strings.

In the pseudo-code in the next sections, it is assumed that the index for vectors starts at 0.

Messages	**Variable**			
	request	***path***	***allocVector***	***preempter***
ROUTING	X	X	X	
ROUTING_ACK	X	X	X	
ALLOC	X	X	X	
ALLOC_ACK	X	X	X	
PREEMPT	X	X	X	X
PREEMPT_ACK	X	X	X	X
END_CS	X	X	X	

Table 4.1: Variables used by messages

Variable	Type	Description
request	couple (**requestId**: integer, **resourceTypes**: Vector of strings)	A couple with a request identifier and the vector of all the requested types of resources. The order in the vector is the order in which the resources will be used, i.e., resourceTypes[i] is the i^{th} resources that will be used.
path	Vector of couples (**id**: string, **length**: integer)	Path from the requesting node to the current node. Each couple in the vector contains a node identifier and its distance to the previous node
allocVector	Vector of couples (**id**: string, **counter**: integer)	The allocation vector for the request. Each couple in the vector contains the identifier of a node holding one of the type of resources requested and the value of the counter for that node applicable to the request
preempter	**id**: string	The identifier of the node that initiated the preemption

Table 4.2: Variables used in messages

Local variable	Type	Role
id	string	The identifier of the node
resourceType	string	The type of resources held by the node
counter	integer	The counter of the node. This counter is used to compute the total order of the requests, cf. Section 4.3.2.
waiting	vector of t-uple(request, path, allocVector)	Set of identifiers of requests waiting for their turn to be allocated.
ignore	vector of integer	Set of request identifiers to be ignored upon arrival, needed to deal with the distributed nature of the system.
status	one of {**IDLE**, **ALLOCATED**, **PREEMPTING**}	Current status of the node
allocatedRequest ***allocatedPath*** ***allocatedVector***	same as request, path and allocVector variables of messages, cf. Table 4.2	Variables of the request currently allocated on the node
currentHandler	string	Identifier of the next node in the *allocation order* of the request currently allocated on the node (allocatedRequest), i.e., the node currently handling this request

Table 4.3: Local variables of nodes

4.2 Path computation

The first subroutine computes a path with nodes that hold instances of the types of resources requested and respecting the order in the request. It relies on two messages: *ROUTING* and *ROUTING_ACK*.

The requesting node starts by sending an initial *ROUTING* message towards the node holding the first type of resources in the request. Subsequent *ROUTING* messages are sent by each node on the path until all the types of resources have been selected.

The last node on the path then sends a *ROUTING_ACK* message to the requesting node.

Simultaneously this subroutine gathers the required information to build an *allocation vector* for the request. The *allocation vectors* are used to compute a total order of the requests. This is described below and then illustrated by an example in Section 4.4. When this subroutine ends, the second one described in Section 4.3 starts to allocate the resources.

This subroutine computes a valid path with the selected instances of all the requested types of resources in the order given in the request. It does not check if the instances are available.

4.2.1 Requesting to enter the CS

When a node wants to request resources it does so by building a first *ROUTING* message in the *requestCS* function, the pseudo-code of which is shown in Algorithm 4.1. The ordered vector of requested types of resources is given as a parameter to the function in the variable ***request***, described in Table 4.2. For instance, *request.resourceTypes[i]* is the i^{th} type of resources requested according to the order in the request.

This function computes two local variables: the ***path*** (line 3), and the ***allocVector*** (line 4). The *path* is initialised (line 7), with a vector containing a couple: the identifier of the requesting node and 0, which is the distance between the requesting node and itself. As the request has just been initiated and no resource has been found yet, the *allocVector* is initialised with an empty vector (line 8).

Then, using the *getPathToResourceType* function, the node identifies the node to which the next message must be sent (line 11). The function updates the ***path*** variable to append the information gathered. ***path*** now contains the first two nodes of the path and the distance between them (line 12). The requesting node then sends a *ROUTING* message to this node (line 13) with the variables ***request***, ***path*** and ***allocVector***.

Finally the function *requestCS* synchronously waits for a *ROUTING_ACK* message (line 16). This message will contain new values for the variables *request*, *path* and *allocVector*, noted respective r, p and a. These new values are sent as parameters to the *allocate* function, detailed in the dedicated section 4.3, that will start the allocation subroutine.

```
1  Function requestCS(request)
2  # Local variables
3  path: vector of couple (id: string, length: integer)
4  allocVector: vector of couple (id: string, counter: integer)
5
6  # Initialisation of variables
7  path ⟵ [(self.id, 0)]
8  allocVector ⟵ []
9
10 # Send request to first node on path
11 pathToNextResource ⟵ getPathToResourceType(request[0])
12 path.append(pathToNextResource)
13 send ROUTING(request, path, allocVector) to pathToNextResource.id
14
15 # Wait for the ack then start the allocation subroutine
16 waitfor ROUTING_ACK(r, p, a)
17 allocate(r, p, a)
18 EndFunction
```

Algorithm 4.1: *requesting the CS*

4.2.2 Forwarding the messages

The algorithm assumes that each node in the system has some local knowledge on how to reach all the types of resources. Each node keeps an up-to-date local routing table containing the names of its neighbours, i.e., the nodes connected to it by an edge, that are closer to each type of resources as well as the distances to these nodes. Table 4.4 shows an example of a routing table for a node n_1. The table shows that there are two instances of types of resources c_1, each at a distance of 1. One can be reached through n_2, the other through n_5. The entry in the routing table is the node itself for the type of resources it holds, in the example it is assumed n_1 holds an instance of c_2. How these routing tables are built and updated is out of scope here. The assumption is that the topology is static, cf. Section 2.4.1. The solution used in the experiments is described in Chapter 5.

Type of resources	**Next node on the path**	**Total distance D to this type of resources**
c_1	n_2	1
	n_5	1
c_2	n_1	0
	n_5	8
	n_2	10
c_3	n_5	9
	n_2	11

Table 4.4: Sample routing table a node.

The ***getPathToResourceType*** function selects the next node on the path to a given type of resources, it checks its routing table and selects one of the node according to the **routing heuristic**. This function takes one parameter: the type of resources. It returns a couple with the identifier of one node and its distance to the current node. *Routing heuristics* are detailed in Section 4.5.2 and evaluated in the next chapter.

4.2.3 Computing the path

The pseudo-code for the handling of *ROUTING* messages is shown in Algorithm 4.2. Upon reception of a message of this type a node checks if the type of resources it holds (variable *self.currentType*) is of the next requested type (lines 20 to 23). If this is the case it appends itself to the *allocVector* along with the current value of its *self.counter* variable (line 22), and then increment *self.counter* (line 21). A distinct value of *self.counter* is attributed to each request allowing the computation of the total order of the requests, cf. Section 4.3.2.

If at least one type of resources has still not been selected (line 25), the algorithm uses the *getPathToResourceType* function to identify the node where to send a *ROUTING* message (line 29). It then sends the *ROUTING* message (line 32) with the *request* variable and the updated values of *path* and *allocVector* as parameters.

Otherwise, if all the types of resources have been selected, the node sends a *ROUTING_ACK* message to the requesting node that will start the allocation subroutine (line 38). This message is the last sent by this subroutine and is handled in the allocation subroutine described in Section 4.3 because the allocation of resources starts when this message is received (Algorithm 4.1, line 17).

```
19 Function handleRouting(request, path, allocVector)
20 if self.resourceType == request.resourceTypes[size(allocVector)]
21   self.counter ⟵ self.counter + 1
22   allocVector.append(self.id, self.counter)
23 endif
24
25 if size(allocvector) != size(request.resourceTypes)
26   # The end of the path has not been reached yet
27   # send updated allocVector to next node
28   nextResourceType ⟵ request.resourceTypes[size(allocVector)]
29   pathToNextResource ⟵ getPathToResourceType(nextResourceType)
30   path.append (pathToNextResource)
31
32   send ROUTING(request, path, allocVector) to pathToNextResource.id
33   return
34 endif
35
36 # This is the case when the path computation finishes
37 # All resources have been found, sends ROUTING_ACK to requesting node
38 send ROUTING_ACK(request, path, allocVector) to path[0].id
39 return
40 EndFunction
```

Algorithm 4.2: Reception of *ROUTING* messages

4.2.4 Total ordering of requests

The algorithm computes a total order of the requests to preserve the **liveness** property, i.e., it guarantees that all requests are satisfied in a finite time. The method from the LASS algorithm [Lej+15] based on allocation vectors built for each request is used to compute this order.

The first step is to build the allocation vector for the request. This vector is built by each of the node on the *path* that holds one of the requested instances. Each node has a local counter, the *local variable* **self.counter**, that is initialised to 0. This counter acts

as a logical clock, but contrary to Lamport's logical clocks [Lam78] 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. This can be seen in line 21 for the *handleRouting* function.

Once the node has updated its local counter, it then updates the allocation vector with the value of its counter (line 22). The updated allocation vector is inserted in the *ROUTING* message when it is forwarded to the next node in the path (line 32). All the allocation vectors are different so as to enforce the total order of the requests, cf. Section 4.3.2.

4.3 Allocation

The second subroutine allocates the instances that were selected during the path computation subroutine. This allocation follows the total order defined by the **allocation order** heuristic and described above in Section 4.2.4. When multiple concurrent requests need to allocate the same instance, the total order of the requests is followed. If needed a preemption mechanism is present to enforce this order.

Initially all the nodes start in the **IDLE** state. When their instance is allocated for a request they enter the **ALLOCATED** state. When a node is already **ALLOCATED** and receives a new request for its instance, this new request might preempt the instance if it comes before the request that had initially allocated the instance in the total order. In this case the node enters the **PREEMPTING** state. The transitions from one state to another are shown on the state diagram of figure 4.1.

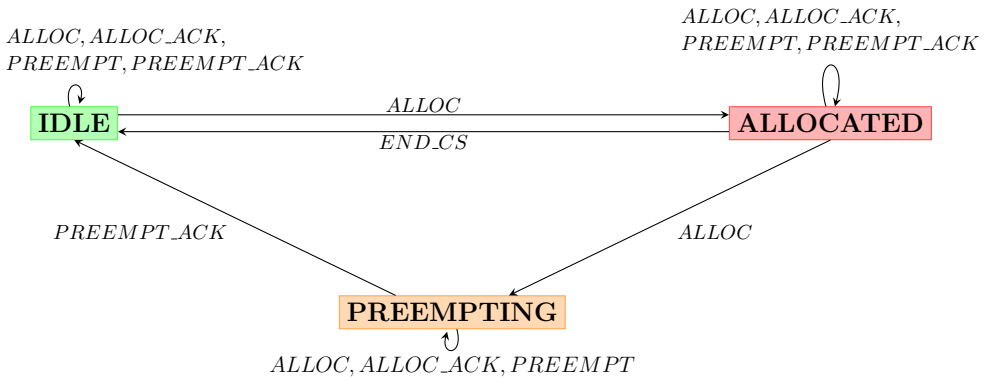


Figure 4.1: State diagram of nodes

4.3.1 Allocating the resources

The core of the allocation subroutine is based on *ALLOC* messages. In a system where all the nodes are initially in the **IDLE** state, a node that receives an *ALLOC* message enters the **ALLOCATED** state and sends an *ALLOC* message to the next node according to the *allocation order* heuristic. The operation is then repeated until the last node in the *allocation order* is reached. Then, the last node sends an *ALLOC_ACK* message to the requesting node to inform it that the allocation is done. It then enters its CS and starts using the instances. Upon leaving its CS it runs the function *leaveCS* shown in Algorithm

4.10 and sends a *END_CS* message that is forwarded along the path to all the nodes holding the requested types of resources.

ALLOC messages are sent to nodes according to the *allocation order* heuristic. Some possible heuristics are detailed in the next section. To follow this *allocation order*, five functions are assumed to be available and their outputs depend on the selected heuristic:

- *firstNodes*: a function that takes an allocation vector as parameter and returns a vector with the identifiers of the nodes to allocate first,
- *nextNode*: a function that takes a path and a node identifier as parameters and returns the node identifier of the node in the path that comes after the node given as an input,
- *previousNode*: a function that takes a path and a node identifier as parameters and returns the node identifier of the node in the path that comes before the node given as an input,
- a *isLast* function that takes two parameters, a path and a node identifier, and returns *true* if the node identifier passed as a parameter is the last node on the path and *false* otherwise,

Start of the subroutine The first step of this subroutine is to handle the *ROUTING_ACK* message with the function *allocate* which pseudo-code is shown in Algorithm 4.3. This function is called at the end of the path computation subroutine. It uses the ***firstNodes*** function (line 42) to identify the nodes where to start the allocation depending on the **allocation order** heuristic. For most heuristics the vector returned by *firstNodes* contains a single element because they allocate resources sequentially. Once the node(s) to reach is(are) identified, the algorithm sends an *ALLOC* message to it(them).

To make the pseudo-code more readable, as the *path* has already been computed during the first subroutine, it is assumed that a ***sendFwd*** function is available. *sendFwd* sends a message from one node to another according to the *path* previously computed. When a message reaches a node on the path that does not hold any of the requested type of resources, it forwards the message to the next node in the *path*. This function *sendFwd* is used in the pseudo-code of this section instead of the *send* function used in the first subroutine.

```
41 Function allocate(request, path, allocVector)
42 nodesToAllocate ⟵ firstNodes(allocVector)
43 for node in nodesToAllocate
44   sendFwd ALLOC(request, path, allocVector) to node
45 endfor
46 EndFunction
```

Algorithm 4.3: Allocation of requests

Handling the *ALLOC* message The pseudo-code for the handling of *ALLOC* messages is shown in Algorithm 4.4. The function relies on several of the *local variables* described above in Section 4.1 to keep tracks of all the allocations received. If the node is in the **ALLOCATED** state, the *local variables* ***allocatedRequest, allocatedPath, allocatedVector*** hold respectively the identifier of the request that allocated the instance, its path, and its allocation vector. The variable ***currentHandler*** contains the identifier

of the node that was next on this request, i.e., the node that is currently handling the request.

The function maintains two *local variables*: *self.waiting* and *self.ignore*. The *self.waiting* vector contains t-uples of all the requests that have tried to allocate the instance held by the current node and have not been able to yet, along with their *paths* and *allocation vectors*. The *self.ignore* vector contains the identifiers of requests that will be ignored when they arrive, it is necessary due to the distributed nature of the system where messages can arrive in any order. The use of these two vectors is detailed below.

At first the algorithm checks (lines 49 to 52) if the request is in the *self.ignore* vector. If this is the case it removes it from the vector and stops there.

If the node is currently in the **IDLE** state, it enters the **ALLOCATED** state and stores the current request, its path, and its allocation vector respectively in the *allocatedRequest*, *allocatedPath* and *allocatedVector* variables (lines 57 to 61). Then the algorithm checks (line 62) if the current node is the last node that needs to be allocated according to the *allocation order* using the ***isLast*** function. If this is the case, it means that all the types of resources have been allocated for the current request and the allocation is finished so the algorithm sends an *ALLOC_ACK* message to the requesting node, which is also *path[0]* (line 63). If this is not the case it sends an *ALLOC* message along the path of the request to the next node according to the *allocation order*. The algorithm uses the *currentHandler* variable to store the identifier of the next node, this operation allows it to know which node to send preemptions to as detailed below (lines 66-67).

If the node is currently in the **ALLOCATED** state when receiving an *ALLOC* message, the algorithm checks (line 74) the *precedence* of the current request with the *precedence* of the request that had allocated the instance before, according to the total order of requests (cf. 4.3.2). If the current request has a lower precedence, the algorithm attempts a preemption and the node enters the **PREEMPTING** state (line 75). It also appends the current request, as well as the request that had previously allocated the instance, to the *self.waiting* vector because the two requests are now on hold. Finally it sends a *PREEMPT* message to the node stored in *currentHandler* with the values stored in *allocatedRequest, allocatedPath, allocatedVector* variables as parameters (line 79).

Finally if the node is in the **PREEMPTING** state when receiving an *ALLOC* message, the current request is stored in the *self.waiting* vector (lines 85 to 88). This request will be handled in a later call to the *handleAlloc* function, once the node has left this state.

Upon reception of an *ALLOC_ACK* message a node simply enters its CS as shown in the pseudo-code in Algorithm 4.5.

```
47 Function handleAlloc(request, path, allocVector)
48 # Check if request is in the self.ignore list
49 if request.requestId in self.ignore
50   self.ignore ⟵ self.ignore − request.requestId
51   return
52 endif
53
54 # If IDLE send ALLOC to next node or
55 # ALLOC_ACK if the node is the node holding the last resource
56 switch self.status
57   case IDLE:
58     status ⟵ ALLOCATED
59     allocatedRequest ⟵ request
60     allocatedVector ⟵ allocVector
61     allocatedPath ⟵ path
62     if isLast(self.id, allocVector)
63       sendFwd ALLOC_ACK(request, path, allocVector) to path[0].id
64       return
65     else
66       currentHandler ⟵ nextNode(self.id, allocatedPath)
67       sendFwd ALLOC(request, path, allocVector) to currentHandler
68       return
69     endif
70
71 # If ALLOCATED, compare current and new requests
72   case ALLOCATED:
73     self.waiting ⟵ self.waiting + (request, path, allocVector)
74     if comparePrecedence(allocVector, allocatedVector) < 0
75       self.status ⟵ PREEMPTING
76       self.waiting ⟵ self.waiting + (allocatedRequest, allocatedPath,
77                                      allocatedVector)
78       sendFwd PREEMPT(allocatedRequest, allocatedPath, allocatedVector,
79                       self.id) to currentHandler
80       return
81     endif
82   endcase
83
84 # If PREEMPTING, append to waiting
85   case PREEMPTING:
86     self.waiting ⟵ self.waiting + (request, path, allocVector)
87     return
88   endcase
89 endswitch
90 EndFunction
```

Algorithm 4.4: Handling of *ALLOC* messages

```
91 Function handleAllocAck(request, path, allocVector)
92 enterCS()
93 EndFunction
```

Algorithm 4.5: Handling of *ALLOC_ACK* messages

4.3.2 Computing the total order of requests

To sort the requests according to the total order, the algorithm computes the **precedence** of requests. A request Req_i will be allocated before a request Req_j if

$precedence(Req_i) < precedence(Req_j)$. This allows the definition of the rank of a request in the total order.

```
94 Function precedence(allocVector)
95 # local variables
96 sum: integer
97 precedence: float
98
99 sum ⟵ 0
100
101 for element in allocVector
102   sum ⟵ sum + element
103 endfor
104
105 precedence ⟵ sum / size(allocVector)
106 return precedence
107 EndFunction
```

Algorithm 4.6: Computing the precedence of two allocation vectors

Req_i is said to have a lower precedence than Req_j if the average of the counters of the allocation vector V_{Req_i} of Req_i is lesser than the average value of the components of the allocation vector V_{Req_j} of Req_j. For instance, consider two requests Req_A= $Req(n_4, [c_3,c_2,c_1])$ and $Req_B = Req(n_6, [c_1,c_2,c_3])$. If their allocation vectors are $V_{Req_A} = ((n_6, 3), (n_3, 3), (n_4, 1))$ and $V_{Req_B} = ((n_4, 2), (n_3, 2), (n_6, 1))$ Req_B has a lower precedence than Req_A, i.e., Req_B will be allocated **before** Req_A. The pseudo-code for the functions ***precedence*** and ***compare*Precedence** is shown in Algorithm 4.6 and Algorithm 4.7.

```
108 Function comparePrecedence(allocVector1, allocVector2)
109 # local variables
110 precedence1: float
111 precedence2: float
112
113 precedence1 ⟵ precedence(allocVector1)
114 precedence2 ⟵ precedence(allocVector2)
115
116 if precedence1 < precedence2
117    return −1
118 endif
119
120 if precedence1 > precedence2
121   return 1
122 endif
123
124 # precedence1 == precedence2
125 return tiebreak(precedence1, precedence2)
126 EndFunction
```

Algorithm 4.7: Comparing the precedence of two allocation vectors

It is possible for the average values of two vectors to be equal. In this case a function *tiebreak* is assumed to be available (line 125) to break the ties between requests, for example the identifier of nodes can be used.

4.3.3 Preempting an instance

Since the system is distributed a request can arrive on a node already in the **ALLOCATED** state for a request that has a higher precedence. This can lead to starvation.

To manage these situations the algorithm preempts the instances to enforce the total order of the requests. This relies on two messages: *PREEMPT* and *PREEMPT_ACK*. These messages take an additional parameter: *preempter*. This variable stores the identifier of the node that initiated the preemption.

First the function *handlePreempt* decides whether it needs to forward the preemption, to accept it and send back a *PREEMPT_ACK*, add the request to the *self.waiting* vector or to do nothing. The pseudo-code is shown in algorithm 4.8.

As it is not assumed that communication links are FIFO, it is possible that a node receives a *PREEMPT* message for a request that it is not aware of yet, i.e., it has not received the corresponding *ALLOC* message. In this case, the node is either **IDLE** or **ALLOCATED** for another request and acknowledges the preemption, cf. the paragraph "Acknowledging the preemption" below. Otherwise, the decision depends on the current state of the node:

- If the node is **ALLOCATED** for the request that initiated the preemption (lines 138-152), there are two sub-cases. If the node is the node holding the last requested instance, then the request has already entered its CS and it is too late to preempt the instance, the algorithm waits for the request to leave its CS and does nothing (line 140). If it is not the node holding the last requested instance then the request is appended to the *self.waiting* vector (line 142), the state of the node changes to **PREEMPTING** and a *PREEMPT* message is sent along the path of the request (lines 143 and 145).

- If the node is **PREEMPTING** (lines 129-135) and the new request has a higher precedence than the request that currently holds the instance, then the request is stored in the local *self.waiting* vector, otherwise the node acknowledges the preemption, cf. the dedicated paragraph below.

Acknowledging the preemption If the node has accepted the preemption, it is either because it did not know the request yet or because it was already preempting the instance for a request with a lower precedence. It starts by appending the request to the *self.ignore* vector (line 155). After that, it sends back a *PREEMPT_ACK* message to the node that initiated the preemption (line 156). When a node receives a *PREEMPT_ACK* message, it executes the *handlePreemptionAck* function. The pseudo-code is shown in Algorithm 4.9. In any case, the node changes back its status to **IDLE**.

If the node receiving the message is the node that initiated the preemption then it finds the request with the lowest precedence stored in the *self.waiting* variable (line 163) and resumes its allocation (line 164).

Otherwise, it removes the request from the *self.waiting* vector (line 166) because the request is not being allocated anymore for now. Then, it sends a *PREEMPT_ACK* message to the previous node in the path (line 167). This message is propagated until the node that initiated the preemption has been reached (line 169).

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 duration the algorithm ensures that resources are released in finite time.

Because of the distributed nature of the system, even if *PREEMPT* messages are sent along the same path as the *ALLOC* messages, a node may receive a *PREEMPT* before it has received the corresponding *ALLOC*. To deal with such situations, each node maintains

a *self.ignore* vector. When it receives a *PREEMPT* for a request and it has not yet received the corresponding *ALLOC*, it stores the request in the *self.ignore* vector (line 155 of Algorithm 4.8). When it finally receives the *ALLOC* for a request that is present in the *self.ignore* vector, it ignores the *ALLOC* and removes the request from the vector (lines 49-52 of Algorithm 4.4).

```
127 Function handlePreempt(request, path, allocVector, preempter)
128 # If already in PREEMPTING~state, append to waiting
129 if self.status == PREEMPTING
130   if precedence(allocVector) > precedence(allocatedVector)
131     self.waiting ⟵ self.waiting + (request, allocVector, path)
132     return
133   else
134     continue
135   endif
136 else
137   # If allocated by the preempting request, check if the node is the last
138   if allocatedRequest.requestId == request.requestId
139     if isLast(self.id path)
140       return
141     else
142       self.waiting ⟵ self.waiting + (request, allocVector, path)
143       status ⟵ PREEMPTING
144       sendFwd PREEMPT(request, path, allocVector, preempter)
145               to nextNode(self.id, allocVector)
146       return
147     endif
148   else
149     # Only possible if IDLE or ALLOCATED by another request
150     continue
151   endif
152 endif
153
154 # The node accepts the preemption
155 ignore ⟵ ignore + (request, allocVector, path)
156 sendFwd PREEMPT_ACK(request, path, allocVector, preempter) to preempter
157 endif
158 EndFunction
```

Algorithm 4.8: Handling of *PREEMPT* messages

```
159 Function handlePreemptionAck(request, path, allocVector, preempter)
160 self.status ⟵ IDLE
161
162 if(self.id == preempter)
163     request, path, allocVector ⟵ requestWithLowestPrecedence(self.waiting)
164     handleAlloc (request, path, allocVector)
165   else
166   self.waiting ⟵ self.waiting − (request, allocVector, path)
167   pathToPreviousResource ⟵ previousNode(self.id, allocVector)
168   sendFwd PREEMPT_ACK(request, path, allocVector, preempter)
169           to pathToNextResource.id
170 endif
171 EndFunction
```

Algorithm 4.9: Handling of *PREEMPT_ACK* messages

4.3.4 Leaving the CS

When a node is done using the allocated instances, it leaves its CS by running the function *leaveCS* (Algorithm 4.10). This function is run by the requesting node and sends a *END_CS* message to the first node in the *allocation order* (line 174).

```
172 Function leaveCS(request, path, allocVector)
173 pathToNextResource ⟵ nextNode(self.id, allocVector)
174 sendFwd END_CS(request, path, allocVector) to pathToNextResource.id
175 EndFunction
```

Algorithm 4.10: Leaving the CS

When a node receives a *END_CS* message it runs the function *handleEndCS* shown in Algorithm 4.11. It changes its state back to **IDLE** (line 178). It then identifies the next node according to the *allocation order* (line 183) and sends it a *END_CS* message (line 184). When the node that receives the message is the last according to the *allocation order* it just does not send any more message (line 182).

```
176 Function handleEndCS(request, path, allocVector)
177 if self.id in allocVector
178   self.status ⟵ IDLE
179   self.waiting ⟵ self.waiting − (request, path, allocVector)
180 endif
181
182 if not isLast(self.id, allocVector)
183   pathToNextResource ⟵ nextNode(self.id, allocVector)
184   sendFwd END_CS(request, path, allocVector) to pathToNextResource.id
185 endif
186 EndFunction
```

Algorithm 4.11: Handling the *END_CS* messages

4.4 Examples

The examples in this chapter are based on the system shown in Figure 4.2, this is the same system that was introduced in Section 2.4.1.

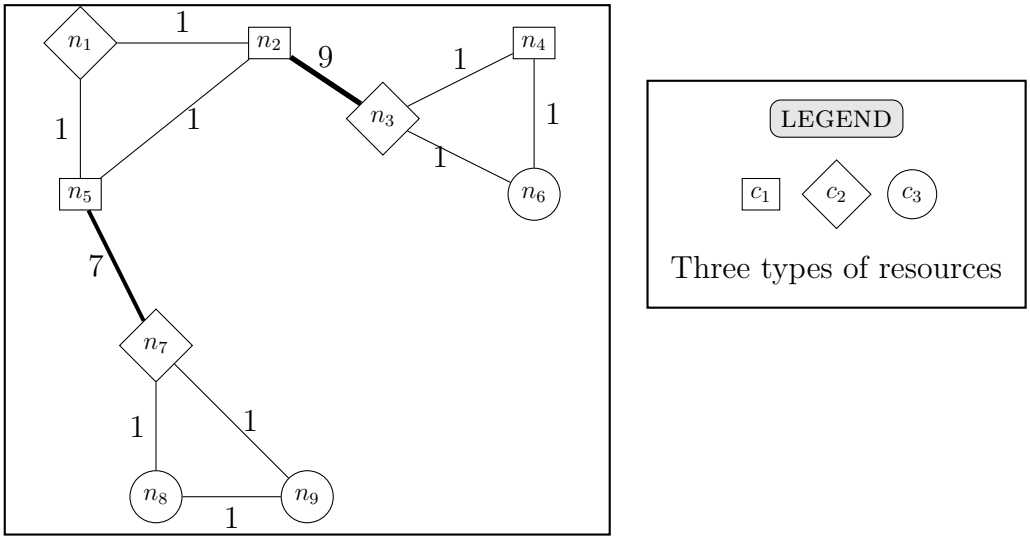


Figure 4.2: The sample system

Table 4.5 gives the routing tables for this system. In this example, node n_1's shortest path to the type of resources 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 resources c_3 is of length 8 and starts at n_7: (n_5, n_7, n_8).

Type	Node	D
c_1	n_2	1
	n_5	1
c_2	n_1	0
	n_5	8
	n_2	10
c_3	n_5	9
	n_2	11

(a) n_1

Type	Node	D
c_1	n_2	0
	n_5	1
	n_3	10
c_2	n_1	1
	n_5	8
	n_3	9
c_3	n_5	9
	n_3	10

(b) n_2

Type	Node	D
c_1	n_4	1
	n_2	9
c_2	n_1	0
	n_2	10
c_3	n_6	1
	n_2	18

(c) n_3

Type	Node	D
c_1	n_4	0
	n_3	10
c_2	n_3	1
c_3	n_6	1
	n_3	19

(d) n_4

Type	Node	D
c_1	n_5	0
	n_2	1
c_2	n_1	1
	n_7	7
c_3	n_7	8
	n_2	11

(e) n_5

Type	Node	D
c_1	n_4	1
	n_3	10
c_2	n_3	1
c_3	n_6	0
	n_3	19

(f) n_6

Type	Node	D
c_1	n_5	7
c_2	n_7	0
	n_5	8
c_3	n_8	1
	n_9	1
	n_5	18

(g) n_7

Type	Node	D
c_1	n_7	8
c_2	n_7	1
c_3	n_8	0
	n_9	1
	n_7	19

(h) n_8

Type	Node	D
c_1	n_7	8
c_2	n_7	1
c_3	n_9	0
	n_8	1
	n_7	19

(i) n_9

Table 4.5: Routing tables for nodes of system from figure 4.2. D is for Distance.

4.4.1 Example of running the algorithm with one request

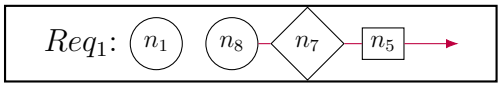


Figure 4.3: Sample request Req_1

The request $Req_1 = Req(n_1, [c_3,c_2,c_1])$, previously used in Section 2.4.1, is used here as the example to describe the behaviour of the algorithm, assuming this is the only request in the system. The selection of instances found for Req_1 is shown in Figure 4.3. Figure 4.4 shows the messages sent during the execution of the algorithm for Req_1 in the sample system to

select these instances. For this example, it is assumed that the *getPathToResourceType* function always returns the node that is the closest for a given type of resources.

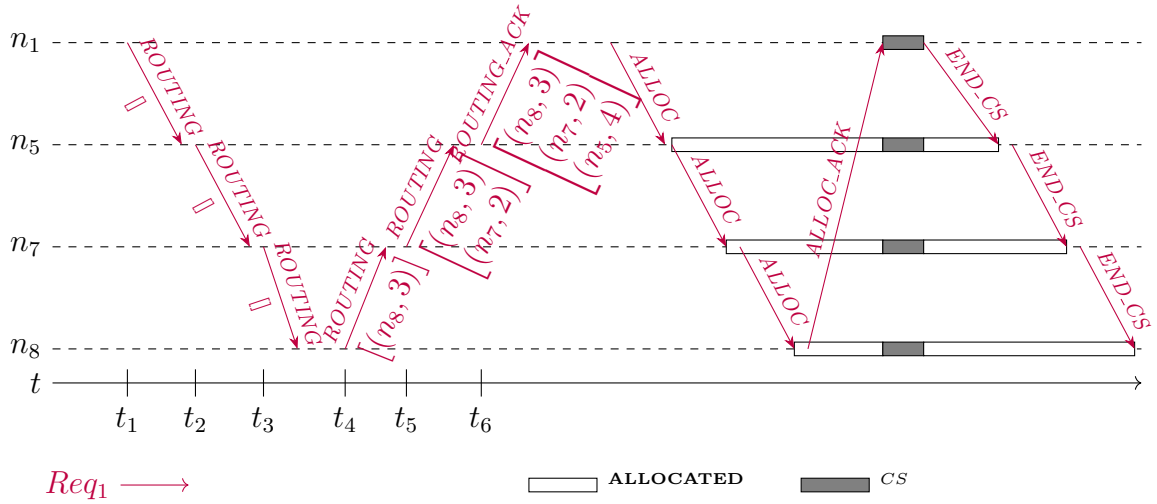
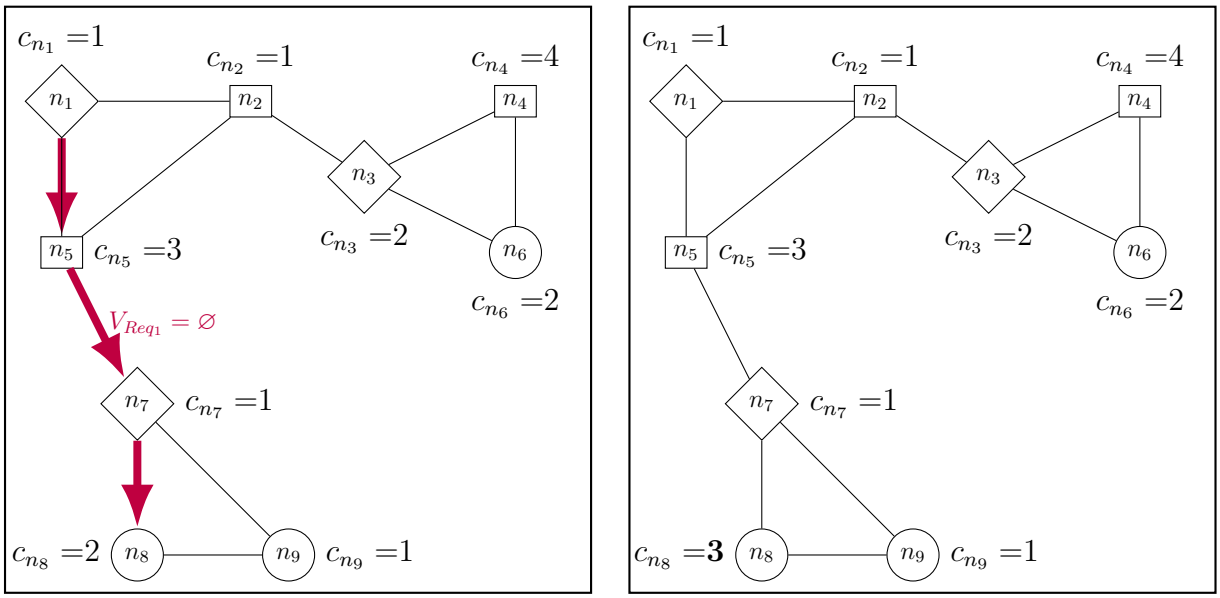


Figure 4.4: Algorithm execution for Req_1, showing only the allocVector variable of *ROUTING* and *ROUTING_ACK* messages



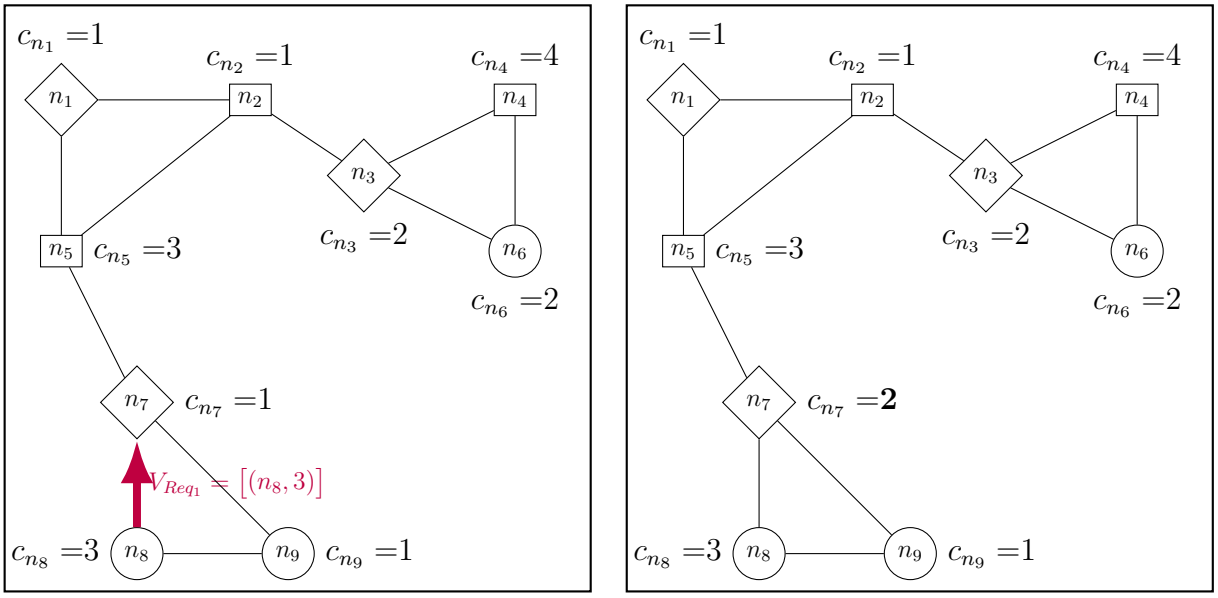
(a) t_1: **requestCS()**, Req_1 is sent towards n_8

(b) t_2: **handleRouting()**, n_8's counter is incremented

Figure 4.5: Running the path computation subroutine Req_1 1/3

Figure 4.5 shows n_1 initiating the request by running *requestCS*. The *getPathToResourceType* function returns n_5 for the type of resources c_3. n_1 sends an initial *ROUTING* message to n_5 with $path = [(n_1, 0), (n_5, 1)]$ and an empty *allocVector*. When it receives this message, n_5 runs the *handleRouting* function and, as it does not hold the first type of resources, sends a *ROUTING* message to the node n_7 returned by *getPathToResourceType* for c_3 with the variable $path = [(n_1, 0), (n_5, 1), (n_7, 7)]$ and an empty *allocVector*. When n_7 receives this message it runs the *handleRouting* function,

and as it does not hold an instance of type c_3 either, it sends a *ROUTING* message to n_8, according to the output of the *getPathToResourceType* function, with $path = [(n_1, 0), (n_5, 1), (n_7, 7), (n_8, 1)]$ and an empty *allocVector*. When n_8 receives the *ROUTING* message, its instance is of the first requested type of resources c_3. Assuming that its current counter value is 2, it increments its counter to 3, updates *allocVector* and sends a *ROUTING* message in the direction of a node holding the second type of resources requested c_2, n_7 according to the output of the *getPathToResourceType* function.



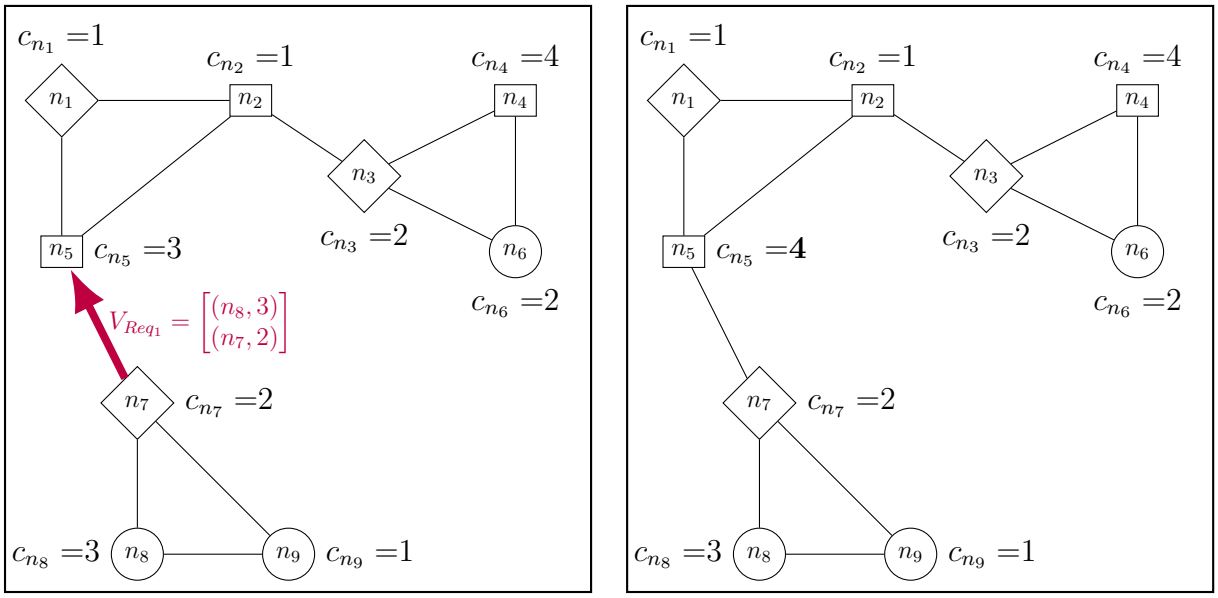
(a) t_3: **handleRouting()**, a *ROUTING* message is sent to n_7

(b) t_4: **handleRouting()**, n_7's counter is incremented

Figure 4.6: Running the path computation subroutine Req_1 2/3

The variables are now $path = [(n_1, 0), (n_5, 1), (n_7, 7), (n_8, 1), (n_7, 1)]$ and $allocVector = [(n_8, 3)]$. When n_7 receives this message, as it holds the second type of resources requested it increments its local counter from 1 to 2 and updates *allocVector*. This is shown in Figure 4.6.

n_7 then sends a message to n_5, the node in the direction of the last requested type of resources. The variables are now $path = [(n_1, 0), (n_5, 1), (n_7, 7), (n_8, 1), (n_7, 1), (n_5, 7)]$ and $allocVector = [(n_8, 3); (n_7, 2)]$, cf. figure 4.7. When the message reaches n_5, it holds the last requested type of resources. It updates *path* and *allocVector* and sends a *ROUTING_ACK* message to the requesting node n_1.



(a) t_5: **handleRouting()**, a *ROUTING* message is sent to n_5

(b) t_6: **handleRouting()**, n_5's counter is incremented

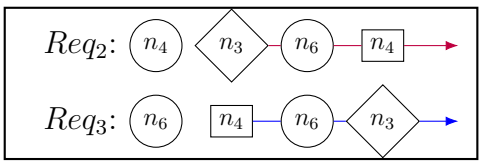
Figure 4.7: Running the path computation subroutine for Req_1 3/3

The allocation subroutine then starts. Assuming that the *allocation order* heuristic goes through the resources in the order in which they appear in the *allocation vector*, n_1 sends an *ALLOC* message to n_5. Then further *ALLOC* messages are sent by n_5 to n_7 and n_7 to n_8. Once the allocation is finished n_8 sends an *ALLOC_ACK* back to the requesting node n_1.

When n_1 is done using the resources it sends an *END_CS* message to n_5. Then subsequent *END_CS* messages are sent by n_5 and finally n_7.

4.4.2 Example of two concurrent requests with preemption of an instance

Figure 4.9 shows how preemption works in the case of the requests Req_2 and Req_3 described in Figure 4.8a. Initially all nodes are in the **IDLE** state and no other request is in the system. At the beginning of this example, the path computation subroutines of these two requests are finished and their *allocation vectors* have been computed, they are shown in Figure 4.8b.



(a) Selection of instances for Req_2 and Req_3

$$V_{Req_2} = \begin{bmatrix}4\\3\\6\end{bmatrix} \quad V_{Req_3} = \begin{bmatrix}5\\4\\3\end{bmatrix}$$

(b) Allocation vectors

Figure 4.8: Sample requests Req_2 and Req_3

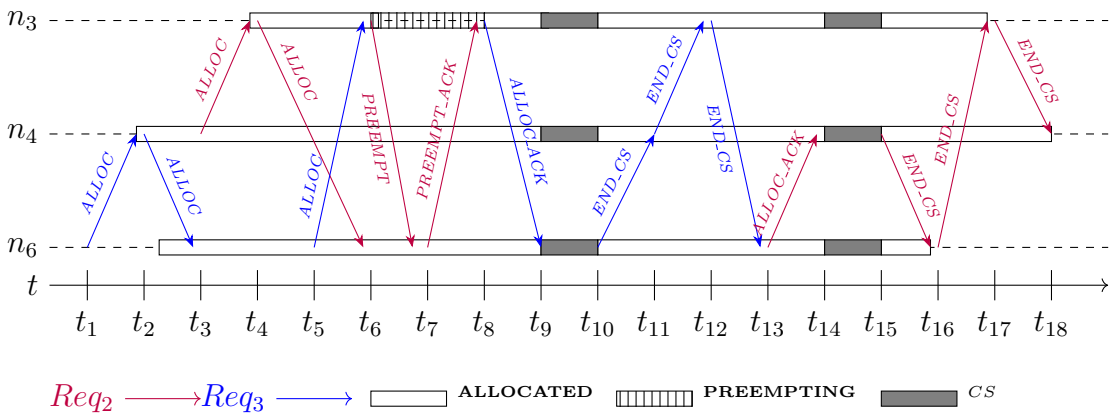
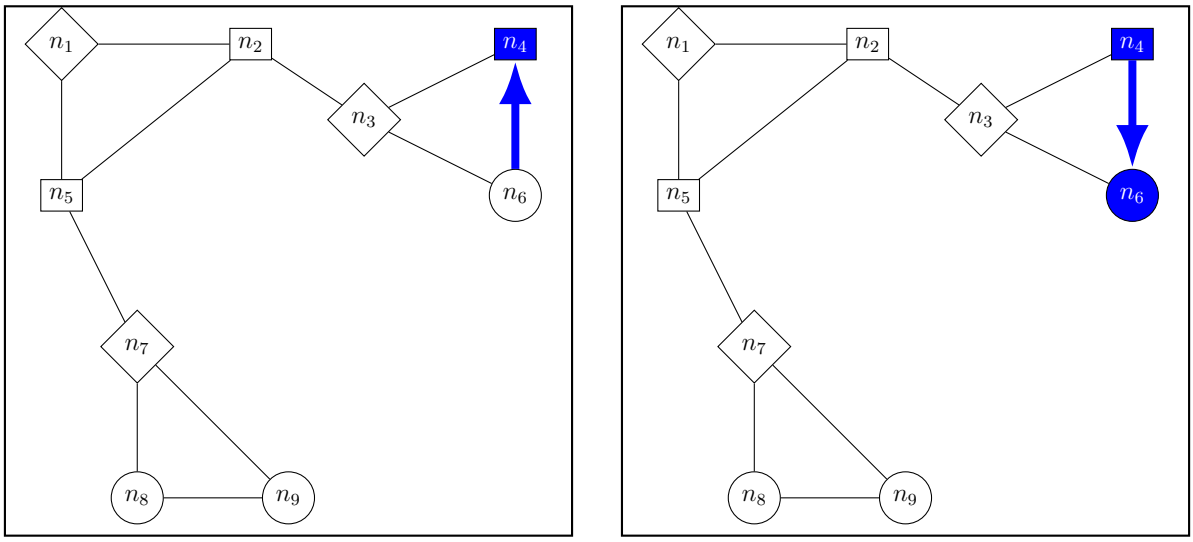


Figure 4.9: Allocations for Req_4 and Req_5, preemption of n_6 by Req_4

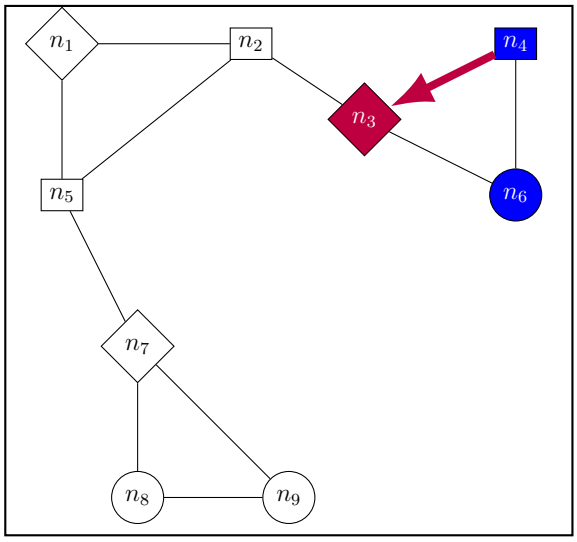


(a) t_1: **allocate()**, the allocation of Req_3 starts. n_3 sends an $ALLOC$ message to n_4.

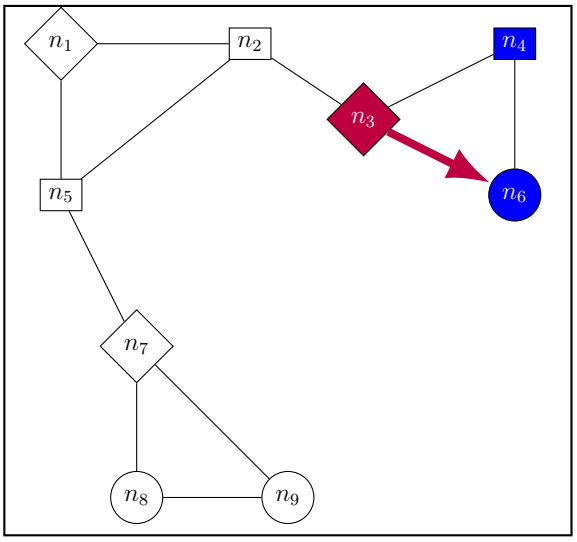
(b) t_2: **handleAlloc()**, allocation of Req_2 continues. n_4 sends an $ALLOC$ message to n_6.

Figure 4.10: Allocation of Req_2 and Req_3 1/4

In this scenario, n_6 start the allocation of Req_3, at t_1 shown in Figure 4.10a, by sending an $ALLOC$ message to n_4, the first node to hold a type of resources according to the order of the request. Upon reception of this message, n_4 enters the **ALLOCATED** state. Then, at t_2 (Figure 4.10b), it continues the allocation of Req_3 and sends an $ALLOC$ message to n_6, the node that holds the second requested type of resources.



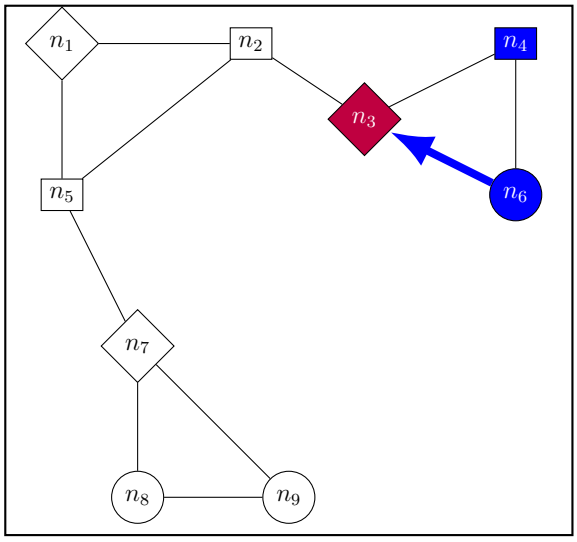
(a) t_3: **allocate()**, allocation of Req_2 starts. n_4 sends an $ALLOC$ message to n_3.



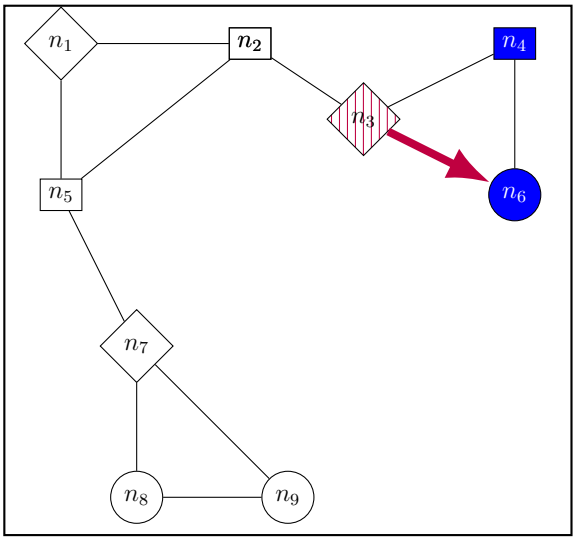
(b) t_4: allocation of Req_2 continues. n_3 sends an $ALLOC$ message to n_6.

Figure 4.11: Allocation of Req_2 and Req_3 2/4

At t_3, as shown in Figure 4.11a, n_4 starts the allocation of Req_2 and sends an $ALLOC$ message to n_3 which enters the **ALLOCATED** state. Then, at t_4 (Figure 4.11b), n_3 continues the allocation of Req_2 with an $ALLOC$ message to n_6.



(a) t_5: **handleAlloc()**, allocation of Req_3 continues. n_6 sends an $ALLOC$ message to n_3.

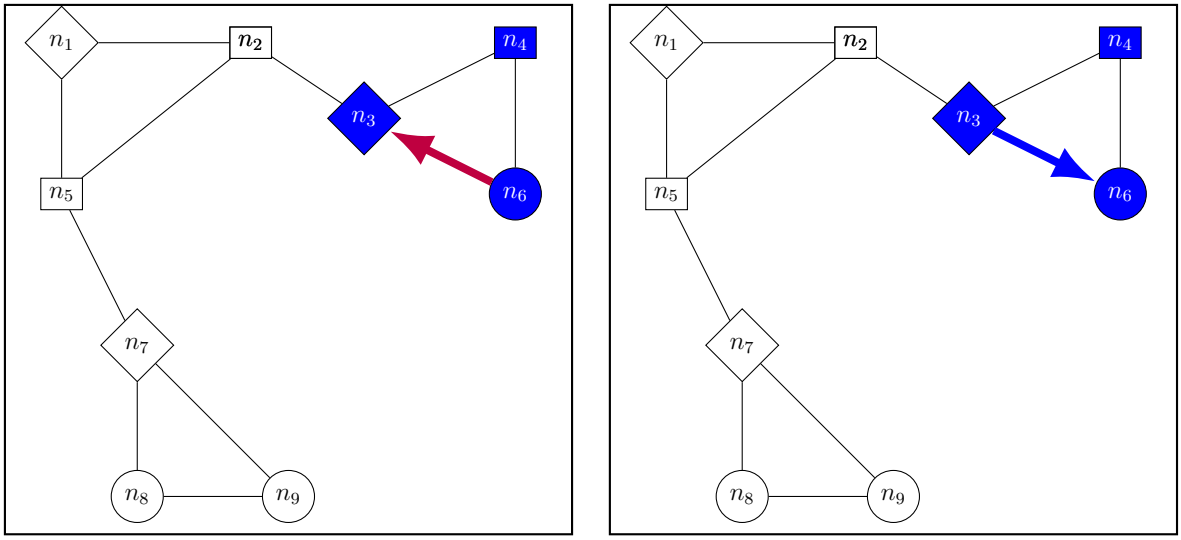


(b) t_6: **handleAlloc()**, n_3 receives Req_3 after it has sent an $ALLOC$ message for Req_2. It enters the **PREEMPTING** state and sends a $PREEMPT$ message to n_6.

Figure 4.12: Allocation of Req_2 and Req_3 3/4

At t_5 (Figure 4.12a), n_6 continues the allocation of Req_3 with an $ALLOC$ message to n_3. Due to the distributed nature of the system, the last two messages cross each other. When it receives the $ALLOC$ message for Req_3, n_3 is already in the **ALLOCATED** state for Req_2. It checks the precedence of the two requests using their allocation vectors, and computes that Req_3 has the priority over Req_2. It then decides to start a preemption and

enters the **PREEMPTING** state. Because it has already sent an allocation message to n_6 for Req_2 it sends a *PREEMPT* message to n_6 at t_6.



(a) t_7: **handleAlloc()**, n_6 sends a *PREEMPT_ACK* message. Upon reception n_6 becomes **ALLOCATED** for Req_3.

(b) t_8: **handleAlloc()**, Req_3 ends. n_3 sends an *ALLOC_ACK* message to the requesting node n_6.

Figure 4.13: Allocation of Req_2 and Req_3 4/4

At t_7, as shown in Figure 4.13a, n_6 receives the *PREEMPT* message. Because the preemption is for Req_2 and it is currently in the **ALLOCATED** state for Req_2, it acknowledges the preemptions by sending back a *PREEMPT_ACK* message to n_3. Now that the preemption is done, n_3 can now enter the **ALLOCATED** state for Req_3. At t_8, the allocation of Req_3 is finished so n_3 sends a final *ALLOC_ACK* to the requesting node n_6 (Figure 4.13b).

4.5 Heuristics

Each of the two subroutines of the algorithm can rely on several heuristics, each with its own objective. This section details these heuristics and the pros and cons of each of them. This is then summed up in Table 4.6. All the heuristics are evaluated experimentally in the next chapter.

4.5.1 Routing heuristics

The path computation subroutine allows the use of various heuristics for the *getPathToResourceType* function described in Section 4.2.2. This function takes a type of resources as an input and returns a couple with the identifier of a node and its distance to the current node.

The first heuristic, called ***shortest***, selects the node holding one instance of the requested type of resources that is the closest from the current node, according to the routing table. This heuristic allows the minimising of the length of the path. However, it does not guarantee that the path computed for a whole request is the shortest path containing all the resources as the sum of the shortest path can be different from the shortest end-to-end

path. The drawback of this heuristic is that for a given type of resources it always returns the same node, resulting in poor load-balancing across the various instances. It is mostly appropriate for systems with a single instance of each type of resources.

The second heuristic, called ***round-robin***, performs a round-robin across the different instances of the input type of resources. For example if there are three instances of the request type of resources, the first time the *getPathToResourceType* is called it returns the first instance. The second time it is called, it returns the second instance then the third instance the third time. After that it starts over. It allows the balancing the load across all the instances but does not consider the distance between the nodes. The resulting path has, most of the times, a longer length than the path computed with the ***shortest*** heuristic.

4.5.2 Allocation order heuristics

The allocation subroutine relies on an **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 2.4.1. The **allocation order** is 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.

Three heuristics are proposed here to select this allocation order. For each of these heuristics, it is necessary to implement the four functions introduced in Section 4.3.1: *firstNodes*, *nextNode*, *previousNode* and *isLast*.

The ***reverse*** allocation order heuristic sends the *ALLOC* messages in the reverse order of the routing path. With this heuristic the path computation subroutine follows the path in one direction, and the allocation subroutine follows the path in the opposite direction returning to the requesting node. This reduces the number of required messages. This heuristic requires some minor changes to the pseudo-code in Section 4.2: lines 16-17 of function *requestCS* (Algorithm 4.1) and line 38 of *handleRouting* function (Algorithm 4.2) need to be adapted so that it is not the requesting node that receives the *ROUTING_ACK* message.

The ***byvalues*** allocation order sends the *ALLOC* messages according to the values of the counters in the allocation vector. Starting by allocating the node with the highest counter value reduces the probability that requests with higher precedence arrive during the rest of the allocation of the request. Upon reception of the *ROUTING_ACK* 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 values of the counters in the allocation vector.

The ***parallel*** allocation order sends *ALLOC* messages to all the nodes present in the allocation vector in parallel. This allows the heuristic to respect the total order of the requests and does not require any preemption to enforce it. However, the experimental results detailed in Section 5.3 show that this heuristic does not give the best usage rate of resources. Other heuristics achieve a better usage rate because not enforcing the total order of requests increase concurrency. The total order is not optimal, having an optimistic strategy and not enforce it systematically improves the usage rate. This allows requests to be allocated when no conflict is detected.

4.5.3 Recap and example

If the allocation vector for Req_1 is $V_{Req_1} = ((n_8, 3), (n_7, 2), (n_5, 4))$, the allocation for the *reverse* heuristic follows the order n_5, n_7, n_8. For the *byvalues* heuristic the order is n_5, n_8,n_7 because $4 > 3 > 2$. In the case of the *parallel* heuristic, when the path computation subroutine finishes, the requesting node n_1 sends $ALLOC$ messages to the three nodes n_8, n_7,n_5 in parallel.

Table 4.6 sums up all the heuristics detailed in this section as well as their pros and cons.

Subroutine	Name of the heuristic	Pros	Cons
Path computation	***shortest***	Minimise the path length	In systems with multiple instances of each type of resources: load is not shared across instances
	round-robin	In systems with multiple instances of each type of resources: maximise the usage rate of resources by load-balancing requests across instances	Longer path
Allocation	***reverse***	Lower number of $ALLOC$ messages	Highest number of preemptions, lower usage rate of resources
	byvalues	Lower number of preemptions, higher usage rate of resources	High number of preemptions
	parallel	Enforce the order of requests, no preemption and lower number of overall messages	Lower usage rate of resources

Table 4.6: Heuristics

4.6 Algorithm Complexity

To compare the complexity of the algorithm with other algorithms from the state of the art, the variables introduced in Table 3.3 are used.

The path computation subroutine requires $ROUTING$ messages to go through at most every node of the system as many times as there are nodes in the requests. If the requests

are of size r, then the maximum number of *ROUTING* messages emitted is r. The algorithm requires an additional *ROUTING_ACK* message resulting in at most $(r + 1)$ messages for the first subroutine.

Then, the allocation subroutine requires a similar r number of *ALLOC* messages with an additional *ALLOC_ACK* at the end. There are $(r + 1)$ of these two messages. The number of preemptions is variable but at most each of the δ conflicting requests could require a preemption for each of the r resources in a request. Therefore the maximum number of *PREEMPT* is $\delta * r$. The *PREEMPT* are followed by *PREEMPT_ACK*, so there are also at the most $\delta * r$ *PREEMPT_ACK* messages. The maximum number of messages for the allocation subroutine is $(r + 1) + 2 * (\delta * r)$.

Once requests are over, the algorithm informs resources to leave their CS. This requires r *END_CS* messages.

The maximum number of messages emitted for both subroutines is then $(r+1)+(r+1)+2*(\delta*r)+r = (2*\delta+3)*r+2$, making the complexity $O(r*\delta)$.

If the system has N nodes, then it also has N resources. Each message could possibly be forwarded across all N nodes. This results in a maximum number of messages in the system to be $[(2*\delta+3)*r+2]*N$.

4.7 Conclusion

This chapter introduced a new distributed algorithm composed of two subroutines for the allocation of resources in networks. This algorithm allocates requests for an ordered set of resources in a system with multiple instances of multiple types of resources. It does not assume that nodes have a full knowledge of the communication graph. It also does not require any knowledge of the *conflict graph* either.

The instances of types of resources are first selected when the path is computed according to a *routing heuristic*. They are then allocated following an *allocation order heuristic*. When conflicting requests arrive on a node, a preemption occurs to enforce a total order of the requests. This total order is defined by computing the *precedence* of the allocation vector. The elements of this vector are the values of the local counters of each node that holds one of the selected instance.

The performance of the algorithm is impacted by the heuristics selected. The next chapter shows an experimental evaluation of the performance of all the heuristics of the algorithm. Chapter 6 then compares it to the performance of algorithms from the state of the art in the same experimental settings.

Chapter 5

Performance analysis

Contents

I'm worse at what I do best
And for this gift, I feel blessed

Nirvana, *Smells like teen spirit*

This chapter gives an extensive view of the performance of the algorithm detailed in the previous chapter. First the metrics are defined in Section 5.1. Then, the experimental environment is described in Section 5.2. The performance of the heuristics presented in Section 4.5 is first evaluated for systems with 1 instance of n resources in Section 5.3, then for systems with m instances in Section 5.4. Last a method to compute the *expected value* is introduced in Section 5.5 and its results are compared to the performance of the heuristics of the algorithm.

The comparison of performance with algorithms from the state of the art is done in the next chapter.

5.1 Metrics and reference configuration

This section describes how the performance of algorithms is evaluated. First the metrics are defined in Section 5.1.1. Then, the next subsection defines the reference system configuration used for the tests.

5.1.1 Metrics definition

Three metrics are used to compare performance:

- the **Average Usage Rate**: the average percentage of used instances for the duration of the experiment. 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 maximise this metric,
- the **Average Waiting Time**: the average time, given in units of simulator discretized time, spent by requests between the moment at which they are emitted and the moment they are satisfied, i.e., the time between the sending of the first *ROUTING* message and the reception of the *ALLOC_ACK* message by the requesting node. The objective is to minimise this metric,
- the **Average Number of Messages per request**: 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 minimise this metric.

5.1.2 System configuration

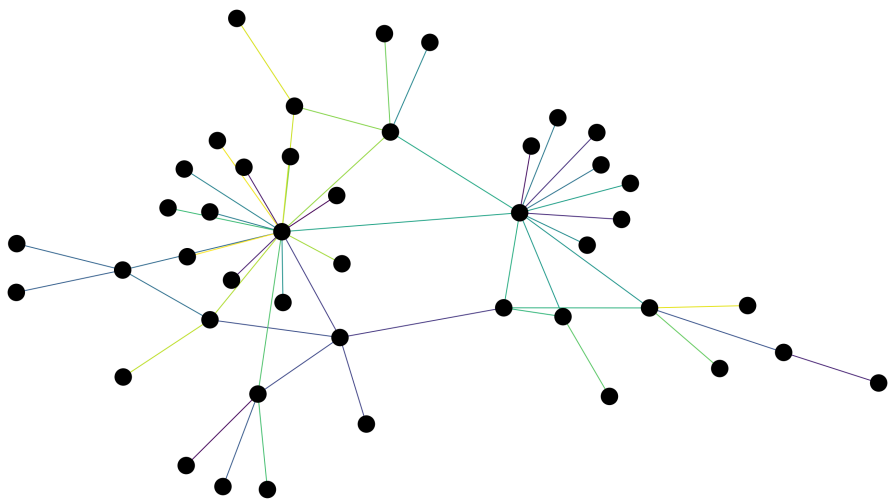


Figure 5.1: The *Cesnet200706* 44-node topology used for the experiments

A test configuration consists of a large number of parameters and each of them potentially influences the results. The results are presented for a chosen **reference system configuration** that was selected because it is representative of the main results. An extensive evaluation of all the parameters and their impacts on performance was done.

This evaluation led to the selection of the values presented here. The main parameters for this reference configuration are:

- A **topology** called *Cesnet200706* with 44 nodes, shown in Figure 5.1. It is extracted from the Internet Topology Zoo [Kni+11]. This topology was selected because its size is large enough to avoid bias in the results from topologies that have too few nodes. Larger topologies lead to longer simulation times to get similar results while providing no added information. In this topology, the degrees of the nodes vary from 1 to 18, with an average degree of 2.

- the **size of requests** is constant, i.e., all requests for a given test have the same size and are for the same number of resources. This size takes multiple values in a set of tests and is represented on the X-axis of the charts.

- the **load** is set so that each node starts a new request as soon as its previous requests has ended.

For a given test, the workload contains requests that all have the same size, defined by the **size of requests** constant listed above. In a given request, the types of resources are all different. The types of resources in a request are selected randomly, using a uniform distribution. When a request is generated each type of resources is selected randomly across all types of resources, then (if the request is of size 2 or more) the second type of resources in the request is selected among the remaining types of resources. This is repeated until all types of resources have been selected.

For each test configuration multiple algorithms, or multiple heuristics of a single algorithm, are compared in the next sections and chapter.

5.2 Experimental environment

A simulator was developped to run the experiments. First the two implementations of the infrastructure based on the SimGrid simulator and the Open MPI library are described below. Then the Grid'5000 platform used to run the experiments is introduced.

5.2.1 Implementations of the infrastructure

The first infrastructure implemented, that is used for most of the experiments detailed here, is based on SimGrid [Cas+14] (version 3.23 from June 2019). SimGrid is an Open-Source (LGPL licence) framework for the simulation of distributed computer systems.

The second implementation available in the simulator uses the Open MPI library (version 3.1.3). Open MPI is an Open-Source implementation of the Message Passing Interface [Gab+04]. MPI is a message-passing interface standardised by the MPI Forum.

The routing tables necessary for the first subroutine of the algorithm are built statically by the simulator during the initialisation. The routing tables for all the nodes are built based on the *Link* and *Hosts* of the SimGrid configuration, using Dijkstra's Shortest Path algorithm [Dij59].

For the tests performed here, the system is put under a heavy load. Each node starts a first request after some random time, to avoid the special case where all the requests arrive at the same time. Then, when the request is finished, i.e., all resources have been

allocated and the requesting node has released its CS, it sends a new request. This is repeated until the end of the test.

The duration of all experiments is the same. The time spent in CS by the requests is also constant. Empirically the value for the duration of CS was set to $300,000$ 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 the unit of discretised time used by the simulator. This approximately simulates CS of 30 seconds and for a total duration of around 14 hours for a whole simulation. Experiments show that this duration is long enough to make the impact of randomness negligible and observe similar results on each run.

5.2.2 The Grid'5000/SILECS platform

Simulations are run on the Grid'5000 [Bal+13], a French research platform. This platform allow the booking of full servers and the deployment of custom images of Operating Systems on it. Booking a full server ensures that no other experiment that could impact the performance is running on it. Since 2018, Grid'5000 is merging into the Super Infrastructure for Large-Scale Experimental Computer Science (SILECS) platform. A custom server image based on Debian 10 is used. This allow a precise control of what the server is running. This image includes the simulator and its dependencies such as SimGrid. A SimGrid program is mono-threaded allowing the execution of multiple simulations in parallel on a server with multiple cores/threads.

5.3 Systems with one instance of n types of resources

In this section the performances of the three heuristics for the *allocation order* described in Section 4.5.2 are evaluated experimentally and compared. The system considered is a system with 1 instance of n types of resources. Here $n = 44$ as the topology used for the tests has 44 nodes and each node holds one type of resources.

In systems with one instance of each type of resources the routing heuristic has no influence, because there is no choice to be made among instances. In this case the two heuristics introduced in Section 4.5.1 return this single instance.

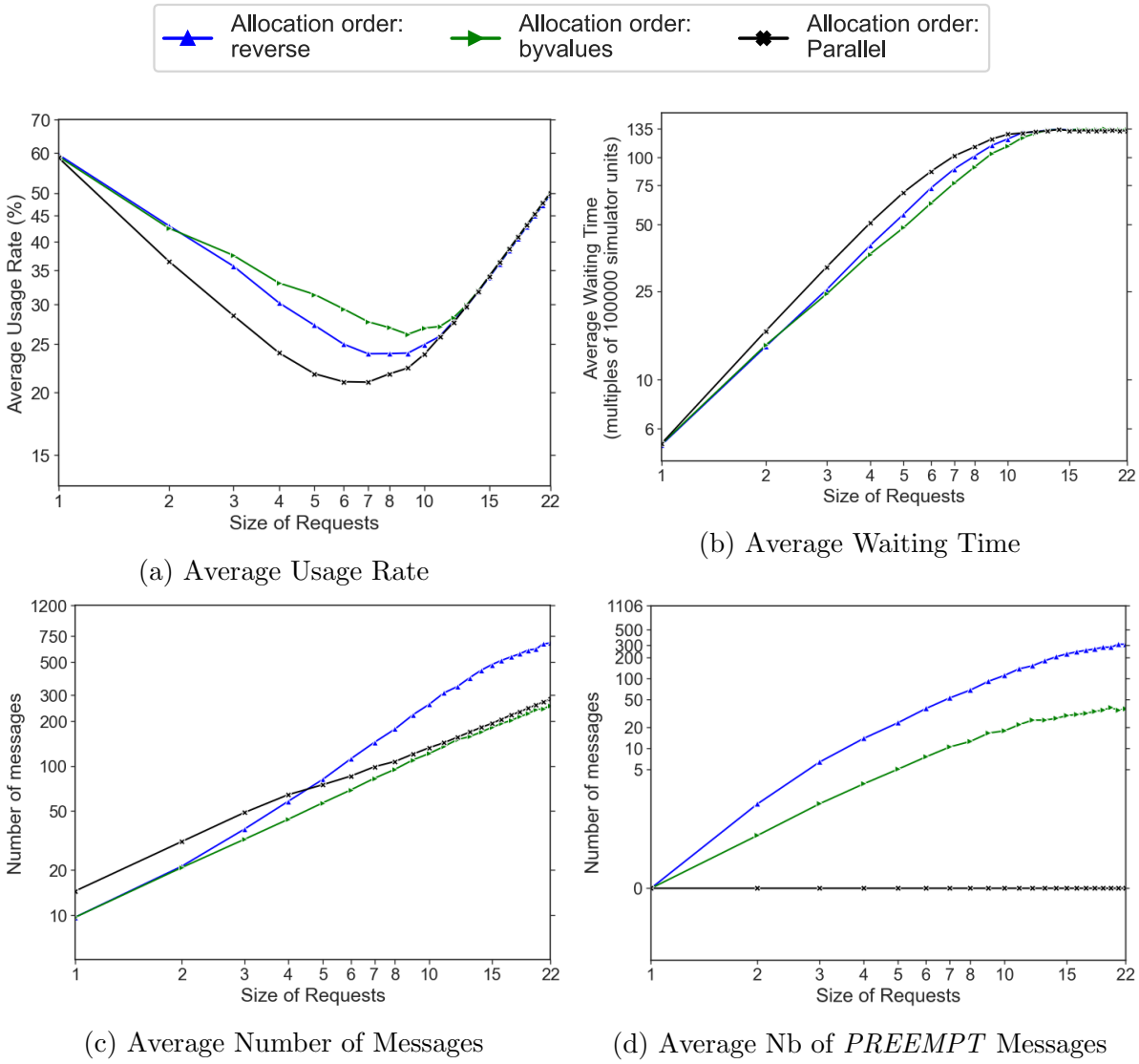
Figure 5.2 shows the comparison of the performance of the three heuristics for the allocation order presented in Chapter 4: *byvalues* in green, *reverse* in blue and *parallel* in black.

As for all performance graphs, the three metrics presented are the three introduced in Section 5.1.1. The X-axis shows the size of the requests. The Y-axis shows the considered metric. Both axis have a logarithmic scale.

For requests of size 1, the heuristic itself has no influence on the *Average Usage Rate*, as can be seen in Figure 5.2a, or on the *Average Waiting Time*, (figure 5.2b). In such configuration, the algorithm never performs any preemption.

For requests of size 2 to 7, the *Average Usage Rate* decreases for all the algorithms. A minimum is reached around 8 or 9, depending on the algorithm because the probability that requests conflict increases with the size of the requests. Two requests conflict if they have at least one instance in common.

Figure 5.3 shows the probability that two requests do not conflict according to the size of the requests. This probability is the ratio of the number of possible requests in a subsystem S to the total number of possible requests. The number of resources in



(a) Average Usage Rate

(b) Average Waiting Time

(c) Average Number of Messages

(d) Average Nb of *PREEMPT* Messages

Figure 5.2: Comparison of *allocation order* heuristics in a system with one instance of 44 types of resources

subsytem S is the total number of resources in the original system minus the number of resources in one request. For requests of size s, it is computed by dividing the number of permutations of size s in a system of size $(N - s)$ by the total number of permutations of size s in the system of size N. For example, for requests of size $s = 10$ in the system of size $N = 44$ this probability is $\frac{^{10}P_{44-10}}{^{10}P_{44}} \approx 5.28\%$. It can also be seen that the *Average Usage Rate* becomes linear when the size of requests is around 15 or more. This is because the probability that requests conflict is almost 100% for requests of size 15.

When the size of requests is superior to half the size of the system, here $\frac{44}{2} = 22$, then it is not possible to allocate concurrently two requests. In such situations the only possibility is to allocate the requests sequentially. The usage rate of the system keeps growing linearly with the size of the requests. It is almost 100% for requests of size N as

each consecutive request will allocate all the resources. It is not exactly 100% due to the cost of the algorithm. This is why the X-axis of the charts presented here stop at 22.

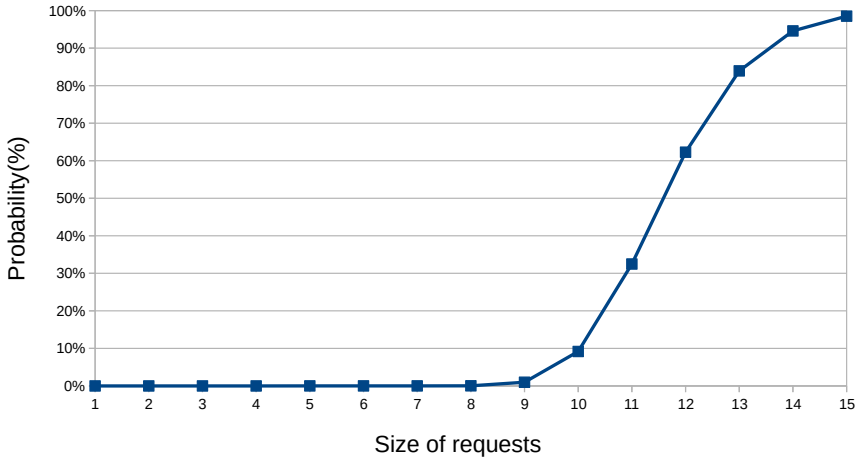


Figure 5.3: Probability that two requests are not totally different, i.e., that two requests have at least one conflicting resource

Figure 5.2a shows a pretty poor performance of the *parallel* heuristic on the *Average Usage Rate* compared to the other heuristics. For requests of size 7, it reaches 21.28% when the *reverse* reaches 24.5% and the *byvalues* 27.27%. The *byvalues* heuristic achieves a significantly better result on this metric with respectively a 28% and 11% increase on this example.

Figure 5.2b shows no notable difference in the *Average Waiting Time* between the three heuristics. The *byvalues* heuristics performs very slightly better than the others. The *parallel* heuristics introduces a waiting time a little longer in average than the two others

Figure 5.2c shows that the *Average Number of Messages* is lower for the *byvalues* allocation order for requests of size 3 and more. The *parallel* heuristic requires slightly more messages but there is no significant difference. This is because the *parallel* heuristic requires two messages, an *ALLOC* and an *ALLOC_ACK*, to allocate an instance where the other two heuristics only require one *ALLOC* message per resource plus an additional *ALLOC_ACK* at the end of the allocation subroutine. The difference shrinks when the size of requests increase because the two other heuristics lead to preemptions which generate more messages. The *reverse* can require twice as many messages.

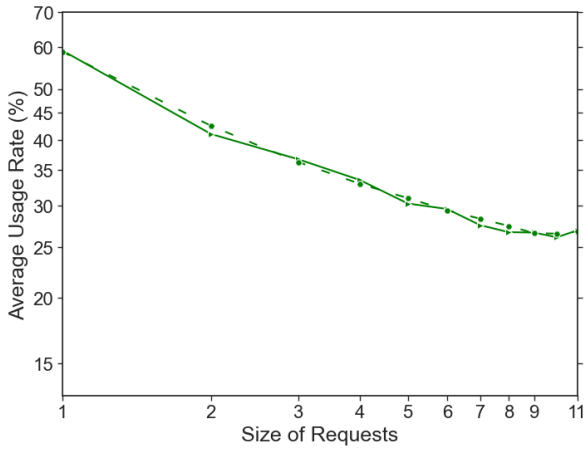
Figure 5.2d shows the *Average Number of PREEMPT Messages*, i.e., a subset of the messages shown in Figure 5.2c. It shows that the difference in performance is due to the high number of preemptions taking place when using the *reverse* allocation order. The probability of preemptions taking place is lower when using the *byvalues* and *parallel* allocation orders. The *byvalues* allocates the resources in the decreasing order of the elements in the allocation vector. A higher value for an element means that the resource it represents has been allocated more time than the others. As the requests in the simulations are generated using a linear random generator, the probability that the next requests try to allocate it again is lower. The *reverse* does not take into account the resource occupation at all and only allocates the resources according to the path computed for the request.

Results show that the *parallel* heuristic never preempts resources as it results in all the nodes knowing about the ongoing requests earlier than with other heuristics. This leads to the nodes to respect the computed total order of the requests than with other heuristics.

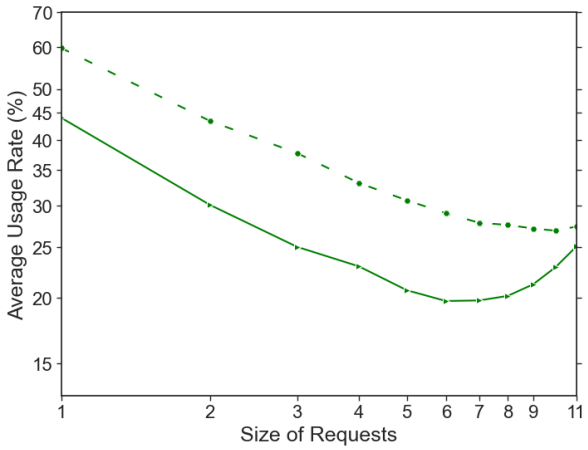
This shows that even if the total order of the requests is necessary to decide which request should have the priority on a node, it is not the reason behind the performance. If it is enforced every time, it gives the results observed here. This shows that the computed order is far from optimal. The algorithm actually improves the performance on the *Average Usage Rate* by not enforcing this order when it is not needed to break the order between competing requests.

5.4 System with m instances of n types of resources

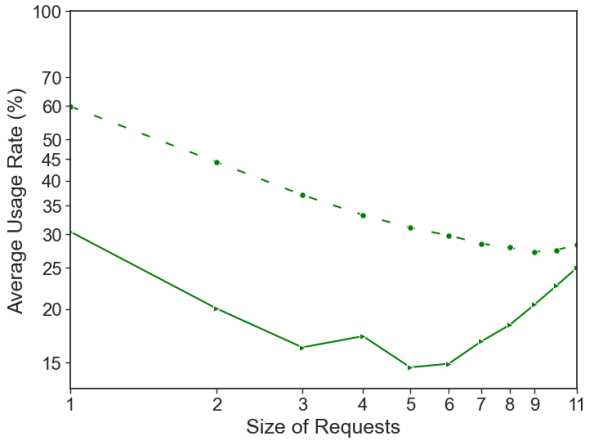
This section evaluates the performance of the different heuristics of the algorithm in systems with m instances of n types of resources. In such a system the *routing heuristic* has an impact on the performance. The *allocation order* heuristic has the same impact that was observed for systems with a single instance evaluated in the previous section, therefore only the *byvalues* heuristic, that gives the best performance, is evaluated.



(a) Evaluation for 1 instance of 44 types of resources



(b) Evaluation for 2 instances of 22 types of resources



(c) Evaluation for 4 instances of 11 types of resources

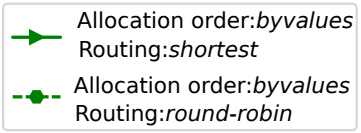


Figure 5.4: Evaluation of *Average Usage Rate* for various numbers of instances and types of resources

Figures 5.4a, 5.4b and 5.4c show the *Average Usage Rate* of the allocation order

byvalues heuristic for two different *routing heuristics*. All experiments are on the same topology with a same overall number of instances (44), but with a different placement of the instances. Each of the figures shows the results for a different number of instances for each type of resources:

- Figure 5.4a shows the results for 1 instance of 44 types of resources,
- Figure 5.4b shows the results for 2 instances of 22 types of resources,
- Figure 5.4c shows the results for 4 instances of 11 types of resources.

With the *shortest* heuristic, a node always selects the same node for a given type of resources. The result is that the load is not well balanced across all the instances, which leads to a lower *Average Usage Rate* when the number of instances in the system increases. For requests of size 5, the lowest *Average Usage Rate* observed is around 15% when there are 4 instances in Figure 5.4c. This is lower than the worst result for the configuration with a single instance of n resources in Figure 5.4a where the *parallel* reaches 21.28%.

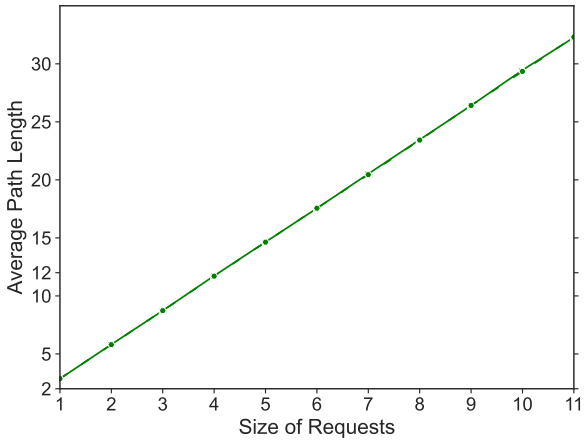
As shown by the *round-robin* heuristic, the load balancing improves with a round-robin on the different instances of each type of resources during the *path computation subroutine*. For example with 4 instances of c_1, the first request selects the first instance, the second request the second one, 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 types of resources and improves the *Average Usage Rate*. In a systems with multiple instances, a round-robin on all the instances allows an *Average Usage Rate* that is similar to that of systems with 1 instance, as can be seen when comparing the values reached by the *round-robin* heuristic in Figure 5.4c and the values reached by both heuristics in Figure 5.4a.

The trade-off is that the length of the path with the *round-robin* heuristic can be longer than with the *shortest* heuristic, because the different instances can be spread across the graph. Figures 5.5 show the *Average Path Length* computed for the requests. When there is a single instance, Figure 5.5a shows that there is no difference in the *Average Path Length* as the two heuristics select the single instance each time. When the number of instances increases the *Average Path Length* is higher when using the *round-robin* heuristic. Figure 5.5b shows that the difference increases with the size of the requests to reach a maximum of around 23%, the *Average Path Length* is a little below 26 for requests of size 11 using the *shortest* heuristic when it is a little above 32 with the *round-robin* one. In a systems where there are 4 instances as in Figure 5.5c, the difference between the two heuristics increases. For requests of size 6, the *Average Path Length* with the *shortest* heuristic is around 12, when it is 17 with *round-robin*, an increase of around 41%. This difference increases to reach almost 60% for requests of size 11.

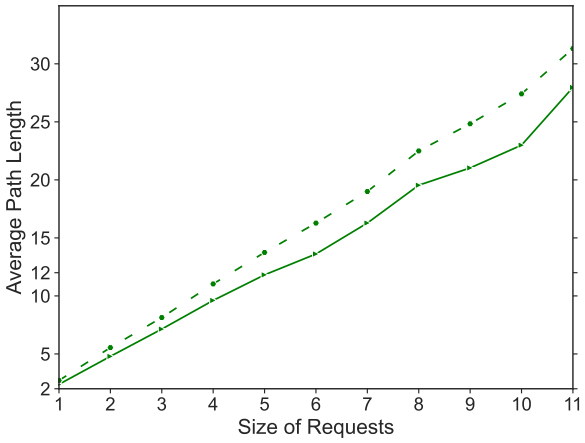
5.5 Computing the *expected value* for the *Average Usage Rate*

The next chapter 6 shows experimental comparative results of the algorithm introduced in the previous chapter against several state-of-the-art algorithms. However, this evaluation raises the question of whether it is possible to get a better performance.

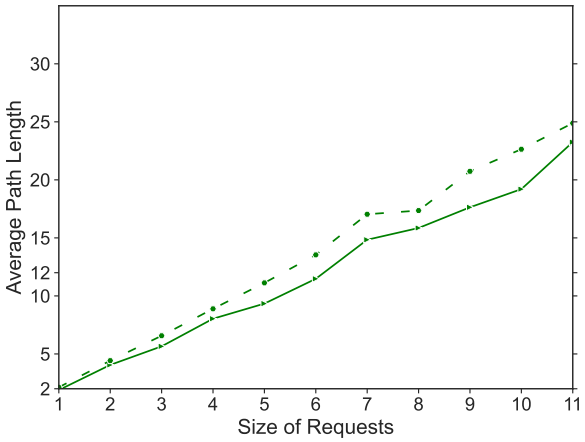
Computing the optimal scheduling for a set of requests is a complex numerical optimisation problem. This section describes a numerical method to compute the *expected*

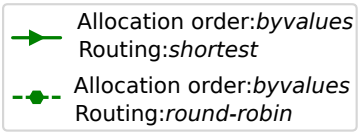


(a) Evaluation for 1 instance of 44 types of resources



(b) Evaluation for 2 instances of 22 types of resources





(c) Evaluation for 4 instances of 11 types of resources

Figure 5.5: Evaluation of *Average Path Length* for various numbers of instances and types of resources

value for the *Average Usage Rate* in the experimental settings. This is different from computing the optimal solution but the *expected value* gives a good indication on the overall performance of these algorithms. As can be seen in the experimental results, the distributed mutual exclusion algorithms evaluated have significantly lower performance than the *expected value* on this metric. The terminology *expected value* refers here to what is commonly noted $E(X)$ for a random variable X in probability theory and defined as the arithmetic mean of a large number of independent realizations of X. The result is an ideal value that could be obtained in a system with a full knowledge of the requests and the states of the nodes considering that communications have no cost (i.e. communications are instantaneous).

5.5.1 Description of the method

In the experimental protocol described in Section 5.1 each of the N nodes sends a request when the test begins. Each node sends a new request after its previous one was completed so that there is always N requests in the system and the system is under a constant load. This method computes the *Average Usage Rate* after a given number of requests. The prerequisite is that all the requests are equiprobable. This is consistent with the experimental settings described in Section 5.1.2 where the generation of requests use a uniform distribution.

It is not needed to consider the end of requests, so this system is a pure-birth process, similar to a Yule process [Kar14]. It computes the n-step transition probability of this process.

In this Markov chain, each state s_i represents the total number of resources used in the system. When the system receives a new request it enters a new state if and only if all the resources it requires are available. Otherwise the state of the system does not change.

In the rest of this description, simple example systems are used to explain the states of the Markov chain and the computation of the n-state probability.

5.5.2 Examples

Before diving in the general method, this section provides two examples in a system with 1 instance of 4 types of resources for requests of size 1 and 2. Requests of size 1 are a special case that allows the introduction of the general idea. With requests of size 2 or more, computing the probability of occurrence of a request is more complex.

Example: system with 1 instance of 4 types of resources, requests of size $s = 1$

Let's first consider the case where the size s of requests is 1 in a system with 1 instance of 4 types of resources. The size of requests is constant. The value associated with each state of the chain is the number of resources used in this state. For a system of size 4 there are 5 possible states, noted $\{s_0, s_1, s_2, s_3, s_4\}$, describing the number of resources used: $\{0, 1, 2, 3, 4\}$.

The vector of possible states is then $V_{states} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$.

There are 4 possible requests, one for each of the 4 resources.

Initially no resource is used (the system is in state s_0). Upon reception of a request of size 1 the system reaches state s_1 with a probability of 1, i.e., when a request of size 1 arrives on a system where no resources are currently allocated then one resource becomes allocated in 100% the cases.

When in state s_1, requests for the currently used resource do not impact the occupation of the system and it remains in state s_1, with a probability of $\frac{1}{4}$. Requests for other resources will use another resource and the systems reaches state s_2 (with a probability of $\frac{3}{4}$).

This gives the following Markov chain in Figure 5.6.

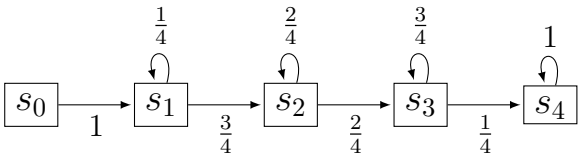


Figure 5.6: Markov chain for the example with requests of size $s = 1$

The transition matrix of this Markov chain is:

$$M_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{3}{4} & 0 & 0 \\ 0 & 0 & \frac{2}{4} & \frac{2}{4} & 0 \\ 0 & 0 & 0 & \frac{3}{4} & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$p0$ is the row vector containing the initial probabilities of being in each state at time 0. Its value is:

$$p_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Computing the average usage rate $Usage_4$ after 4 requests can be obtained with:

$$Usage_4 = \frac{p_4 * V_{states}}{4} = \frac{(p_0 * M_1^4) * V_{states}}{4} = \frac{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{3}{4} & 0 & 0 \\ 0 & 0 & \frac{2}{4} & \frac{2}{4} & 0 \\ 0 & 0 & 0 & \frac{3}{4} & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^4 * \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}}{4}$$

$$Usage_4 \approx \frac{2.7344}{4} \approx 68.36\%$$

Example: system with 1 instance of 4 types of resources, requests of size $s = 2$

In statistics k-permutations of n can be noted using various symbols. For readability reasons, hereafter the notation ${}^nP_k$ is used to denote the number of k-permutations of a k-element subset of an n-set.

This section introduces another example in the same system with 1 instance of 4 types of resources as the section above. For requests of size 2, the set of all possible requests is $\{(1,2),(2,1),(1,3),(3,1),(1,4),(4,1),(2,3),(3,2),(2,4),(4,2),(3,4),(4,3)\}$, i.e., all the

possible k-permutations. The total number of k-permutations is ${}^4P_2 = 12$. Requests are of size 2 and all resources are allocated at once. The set of possible states of the system is then $\{0, 2, 4\}$, i.e., either no, two or four resources are used at a given time. This gives the vector of possible states $V_{states} = \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix}$.

For this example the value of $p0$, the row vector containing the initial probabilities of being in each state at time 0 is:

$$p_0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

This gives the Markov chain of figure 5.7.

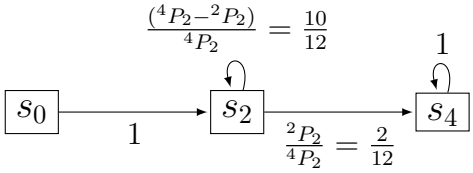


Figure 5.7: Markov chain for the example with requests of size $s = 2$

At the beginning there are no resources used in the system. Whatever the first request is the system enters state s_2, as the request is of size 2 . If the first request is $\{(1, 2)\}$, then the system may be fully occupied and enter s_4 if the second request is not in conflict with the first one. This happens only if the second request is $\{(3, 4)\}$ or $\{(4, 3)\}$. This gives a probability of $\frac{2}{12}$. In all other cases it remains in s_2.

The *Average Usage Rate* $Usage_4$ after 4 requests of size 2 can be computed:

$$Usage_4 = \frac{p_0 * M_2^4 * V_{states}}{4} = \frac{\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 0 & \frac{({}^4P_2 - {}^2P_2)}{{}^4P_2} & \frac{{}^2P_2}{{}^4P_2} \\ 0 & 0 & 1 \end{bmatrix}^4 * \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix}}{4}$$

$$Usage_4 = \frac{\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 0 & \frac{10}{12} & \frac{2}{12} \\ 0 & 0 & 1 \end{bmatrix}^4 * \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix}}{4} \approx \frac{2.8426}{4} \approx 71.06\%$$

5.5.3 Generalisation to requests of size s on any system with 1 instance

The examples in the sections above can be generalised to a system with 1 instance of n types of resources with requests of size s. The number of resources used in the last state of the system might not be n if s is not a divisor of n. Let's consider n' the integer immediately inferior to $\frac{n}{s} * s$, $n' \approx \frac{n}{s} * s$. $n = n'$ when s is a divisor of n. In this case, the states of the system are s_i where the number of resources used for s_i is $i * s$ and $i \in [0, n']$. For example in a systems with $N = 44$ nodes and for requests of size $s = 3$, $n' = 14 * 3 = 42$, because it only possible to allocate 14 requests of size 3 in a system with 44 resources.

The vector of possible states of the system is : $V_{states} = \begin{bmatrix} 0 \\ s \\ 2*s \\ 3*s \\ \dots \\ i*s \\ \dots \\ n' \end{bmatrix}$

The total number of possible requests is the total number of k-permutations of requests of size s in a system of size n, ${}^nP_s$. As in the above examples, the system leaves a state to enter the next one only if the new request requires only resources that are still available. In state s_i, $i*s$ resources are currently used. There are $\frac{{}^nP_s - {}^{n-(i*s)}P_s}{{}^nP_s}$ possible requests for the remaining $n-(i*s)$ resources.

This gives the Markov chain shown in Figure 5.8.

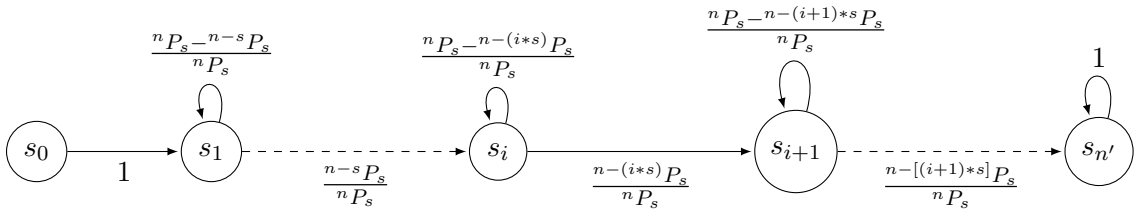


Figure 5.8: Markov chain for the generalised case

From this Markov chain, the transition matrix M_n given below can be deduced.

$$M_n = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \frac{{}^nP_s - {}^{n-1}P_s}{{}^nP_s} & \frac{{}^{n-(1*s)}P_s}{{}^nP_s} & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \frac{{}^nP_s - {}^{n-(2*s)}P_s}{{}^nP_s} & \frac{{}^{n-(2*s)}P_s}{{}^nP_s} & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & \frac{{}^nP_s - {}^{n-(i*s)}P_s}{{}^nP_s} & \frac{{}^{n-(i*s)}P_s}{{}^nP_s} & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & \frac{{}^nP_s - {}^{n-(i+1)*s}P_s}{{}^nP_s} & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 1 \end{bmatrix}$$

To compute the usage rate $Usage_n$ after the n^{th} request:

$$Usage_n = \frac{p_n * V_{states}}{n} = \frac{p_0 * M_n^n * V_{states}}{n} = \frac{\begin{bmatrix} 1 & 0 \dots & 0 \end{bmatrix} * M_n^n * \begin{bmatrix} 0 \\ s \\ 2*s \\ 3*s \\ \dots \\ i*s \\ \dots \\ n' \end{bmatrix}}{n}$$

Even if computing the optimal scheduling of a specific set of requests is a NP-complete problem, this method computes the *expected value* of the *Average Usage Rate*. For an optimal scheduling of a given set of a requests, it could be higher or lower than what

is computed because the optimal scheduling depends on how the set of requests can be arranged.

This method is used in the next section to compute the *expected value* of the *Average Usage Rate* for a given size of requests and compare it to the performance of the allocation orders subroutine of the algorithm detailed in Section 4.3.

5.5.4 Comparison with the *expected value*

In this section the performance of the algorithm described in Chapter 4 is compared with the mathematical expectation *Average Usage Rate* computed with the Markov chain detailed above in the previous section. Only the *byvalues* heuristic is considered in the comparison as it was the one giving the best result.

Figure 5.9 shows, in grey and noted *expected value*, the values of the *expected value* of the *Average Usage Rate* computed using the method presented in the above section. The figures computed using this method assume that each node emits a new request at each step, whereas in the experimental setup a new request is sent only when a previous request has released its CS.

The *expected value* for the *Average Usage Rate* is significantly higher than the results of the best heuristic of the algorithm in the previous chapter. This shows that there is still room for improvement. This *expected value* of the *Average Usage Rate* can be approached more easily with a centralised algorithm.

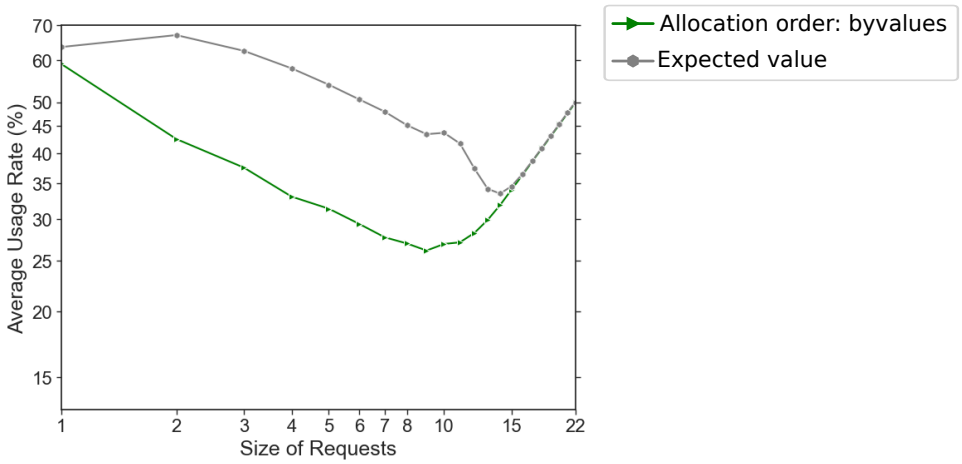


Figure 5.9: Comparison with *expected value* of the *Average Usage Rate* in a system with one instance of 44 types of resources

5.6 Conclusion

This chapter compared the performance of the various heuristics proposed for the algorithm presented in Chapter 4. It shows that for systems with a single instance, there is no difference in the path computed by the two routing heuristics *shortest* and *round-robin*. In system with multiple instances, load-balancing the requests, for example using the *round-robin* heuristic improves the *Average Usage Rate*. Of the three heuristics proposed for the allocation order, the *byvalues* offers the best performance on all the three considered metrics because it limits the number of preemptions. It also does not enforce the order of the requests when it is not required.

Comparing the performance of these heuristics with the expected value of the *Average Usage Rate* shows that the performance of the algorithm is well below. This is because a distributed algorithm, as explained in the previous chapter, only has a partial view of the system.

It should theoretically be possible to build a consistent distributed global view of the requests that would allow a solution to get closer to the expected value. However, that could require a large number of messages (see Section 5.1.1 for the definition of metrics) that make such a solution non scalable.

The next chapter compares the algorithm with other algorithms from the state of the art, and shows that even if the performance is well below the expected value, it is better than the state of the art.

Chapter 6

Experimental comparison with state of the art algorithms

Contents

> That is a fact. And fact is the most stubborn thing in the world.
>
> Mikhail Bulgakov, *The Master and Margarita*

This chapter compares the performance of the *byvalues allocation order* heuristic in-

troduced in Section 4.5.2 to the performance of algorithms from the state of the art. The selected algorithms tackle the problem of the distributed allocation of resources in systems with one instance of multiple types of resources. The simulation environment and methodologies for the tests are the same that are used in Chapter 5 to compare the performance of the heuristics.

Four algorithms are studied, following the chronological order of their publications:

- Dijkstra's incremental algorithm [Dij71], the first solution proposed to the Dining Philosophers Problem (DiPP) to allocate a static set of resources. It is also applicable to the Drinking Philosophers Problem (DrPP) and still widely used. It is described in Section 6.2

- Chandy-Misra algorithm [CM84], the initial solution to the DrPP, described in Section 6.3

- Rhee's algorithm [Rhe95] is modular and builds a distributed queue of requests. It is based on a model close to the model described in Section 2.4.1. The implementation used for the evaluation uses Chandy-Misra DrPP algorithm as a subroutine. It is described in Section 6.4

- Bouabdallah-Laforest algorithm [BL00], a simple and scalable token-based algorithm that builds distributed queues for the resources to solve the DrPP, described in Section 6.5

Some of these algorithms are rather old but, as shown in the state of the art in Chapter 3, there is a limited number of recent algorithms. The most recent published algorithm seems to be the LASS algorithm [Lej+15] that addressed a model different than the one used here. The algorithm described in Chapter 4 reuses its scheduling based on vectors of counters to target the model described in Section 2.4.1. Before that Bouabdallah-Laforest seems to be the latest algorithm that improved significantly the performance of DrPP algorithms. Rhee's algorithm target a model that is similar to the one used here. The other selected algorithms are older but Dijkstra's still offers the best *Average Usage Rate* of these four algorithms. Chandy-Misra is a classic algorithm in the field. It is not expected to perform well in the model because it requires an a priori knowledge of the *conflict graph*. The analysis shows why it is not an adequate solution for the system. Chandy-Misra algorithm is also used as a subroutine by Rhee's algorithm.

Code for the algorithms is provided in a public Git repository: `https://gitlab.com/gfraysse/algorithms/tree/master/Distributed%C2%A0Mutual%20Exclusion/`. The code is in the Go language [1] because Go supports concurrency using the Communicating Sequential Processes (CSP) model [Hoa78] which, along with the syntax of the language, makes the code very similar to pseudo-code while being runnable.

These algorithms are compared to the *allocation subroutine* of the algorithm detailed in Chapter 4. The experimental setup remains unchanged from the previous chapter and the same *reference system configuration* is used. Only the *byvalues allocation order heuristic* of the algorithm is considered for the comparison because the results in Chapter 5 show that this heuristic gives the best performance for all metrics in the experimental settings. In Section 6.6, the performances of the four algorithms of the state of the art as well as those of the three heuristics are summed up and a global comparison is made.

[1] `https://golang.org/`

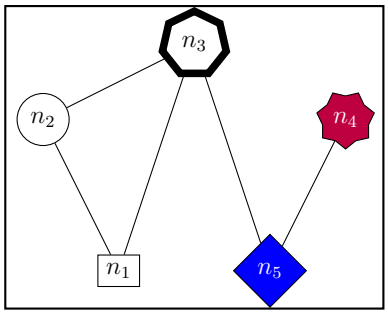
The experiments show that the *byvalues* heuristic performs better than these selected algorithms from the state of the art on the three metrics that are measured.

6.1 System setup

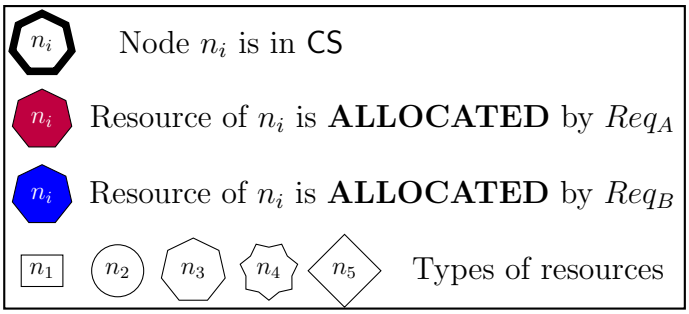
In the tests presented in this chapter, the system considered is the *reference system configuration* described in Section 5.1.2. The system holds a single instance of 44 types of resources.

The implementations of all the algorithms rely on the same first subroutine described in Chapter 4 for the selection of the instance of types of resources and the computation of the path. In a system with multiple instances, this allows the use of any DrPP algorithm because each instance of a same type of resources becomes a specific instance once it has been selected. The DrPP algorithm has to allocated the instances selected during the *path computation* subroutine. This allows the comparison of the performance of the *allocation* subroutine heuristics to any algorithm that addresses the DrPP.

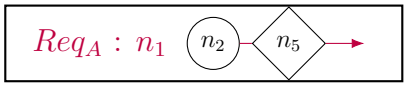
All the examples given in this chapter are based on the same scenario. The scenario considers a system with 1 instance of 5 types of resources, it is shown in Figure 6.1a. At the beginning, two sample requests Req_A and Req_B shown in 6.1c and 6.1d are emitted in the system. Given a global clock, Req_A is emitted by n_1 before Req_B is emitted by n_4. In this configuration the path computation subroutine always returns the path to the single instance so its results are identical for all algorithms and this subroutine has no influence at all in the overall comparison.



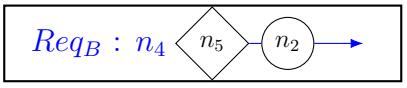
(a) Sample system with 1 instance of 5 types of resources



(b) Legend



(c) Selection of instances for sample request Req_A emitted by n_1



(d) Selection of instances for sample request Req_B emitted by n_4

Figure 6.1: Scenario used in this chapter

6.2 Dijkstra's *Incremental* algorithm

Dijkstra's *Incremental* algorithm [Dij71] relies on a **static total order** of the set of resources. Despite being the oldest algorithm detailed in this chapter, it is the algorithm from the state of the art that achieves the best performance for the *Average Usage Rate*.

However it has the worst performance for the *Average Waiting Time* due to the domino effect explained below.

6.2.1 Description of the algorithm

The algorithm, as initially described by E. W. Dijkstra, is formulated for use in a system where processes use a *shared memory*. The problem addressed is the DiPP, as described in Chapter 3.

The algorithm maintains a variable *mutex* and 2 arrays with an entry per process: *C*, which holds the state of the process (among *thinking*, *hungry* or *eating*), and *prisem*, which has a value of 1 when the process can start eating and 0 otherwise. The algorithm relies on the basic operations on semaphores first described by E. W. Dijkstra in [Dijte]: *P* (also called *wait* or *acquire*) and *V* (also called *signal* or *release*). It also relies on a *test(w)* function that calls the *V* function on *prisem[w]* if philosopher *w* is allowed to eat. The pseudo-code is shown in algorithm 6.1.

```
1  cycle begin think;
2    # Process waits for global mutex
3    P(mutex);
4         # Once process w gets the mutex, it becomes hungry
5         C[w] := 1;
6         # Process w requests to eat
7         test (w);
8    # Release the mutex
9    V(mutex);
10
11   # Wait for the mutex of process w.
12   P(prisem [w]);
13   # Once mutex is acquired, the process can eat
14    eat;
15
16   # Once the process is done eating, inform others
17   # First lock the global mutex
18   P(mutex);
19       # Process is done eating, enters the thinking state
20       C[w] := 0;
21       # Inform both neighbours that process w is done eating
22       test((w + 1) mod 5);
23       test((w - 1) mod 5);
24   # Release the global mutex
25   V(mutex);
26 end.
```

Algorithm 6.1: Dijkstra's *Incremental* algorithm, as originally formulated for 5 philosophers, from [Dij71].

The algorithm ensures that each process (or philosopher) will eventually start eating. It allocates the resources (forks) by following the numbering of the philosophers (lines 22 and 23).

Adaptation to the model

E. W. Dijkstra's model is a shared-memory system, it requires the algorithm to use a *mutex* variable to modify the state of each node. In the model used here, each node manages its instance and its own state. The *P* and *V* operations on the *prisem[w]*

variable are implemented by the messages *ALLOC* and *ALLOC_ACK*, respectively, sent to (respectively by) node *w*. A local queue of pending requests on each node plays the role of the semaphore.

Also, in E. W. Dijkstra's model a resource (fork) can only be allocated by the two neighbouring nodes (philosophers). Another adaptation is required to allow for any number of resources, and not only two, to be requested according to the defined order.

Requesting the CS

When a node wants to enter its CS and allocate resources for a request, it selects, in the request, the node holding a requested resource that comes first according to the order of resources, i.e., nodes. Then, it sends an *ALLOC* message to this node, the *P* operation. If the resource is not currently allocated, the node allocates it and replies with an *ALLOC_ACK* message, the *V* operation. Otherwise, the request is appended to its local queue to be handled once the request currently holding the resource has released its CS. Upon reception of an *ALLOC_ACK* message, the requesting node sends an *ALLOC* message to the next node holding one of the requested resources according to the total order of the set of resources.

Releasing the CS

When a node releases its CS, it sends *END_CS* messages to the nodes managing the allocated resources so they can release them. Then, each of these nodes looks into its local queue for pending requests that can now continue.

Pros and cons

This algorithm is easy to understand and implement. It delegates the locking of resources to a basic mutex operation. On the evaluation platform, this operation is implemented as a centralised lock on the requesting node with the *ALLOC* and *ALLOC_ACK* messages playing the role of the mutex operations. It offers a good performance on the *Average Usage Rate* metric.

This algorithm has a drawback called the **domino effect** [Ran75]. As the total order is based on the identifiers of the nodes, requests can pile up in the queue for some nodes waiting for their resources to be available. Until these resources become available all the locked resources from nodes which come before in the order of the nodes are unavailable.

Nancy A. Lynch [Lyn80; Lyn81] proposed an improvement of the algorithm based on a graph where the nodes represent the resources and are connected via edges that represent the resource sharing constraints. Her improvement is based on a colouring of this graph in the DiPP model described by E. W. Dijkstra where requests emitted by a node are always for the same set of resources. In the model used here, requests are not known a priori and any node can request any resource making the graph complete with one colour per node. Therefore it is impossible to optimise the colouring.

6.2.2 Example of execution in sample system

Initial state For the example the total order of resources is $n_1 < n_2 < \ldots < n_5$ according to their subscript values. The execution of the sample requests Req_A and Req_B

is shown in two figures. Figure 6.2 shows the timeline of the messages while figures 6.3 to 6.6 show how the messages are sent on the topology.

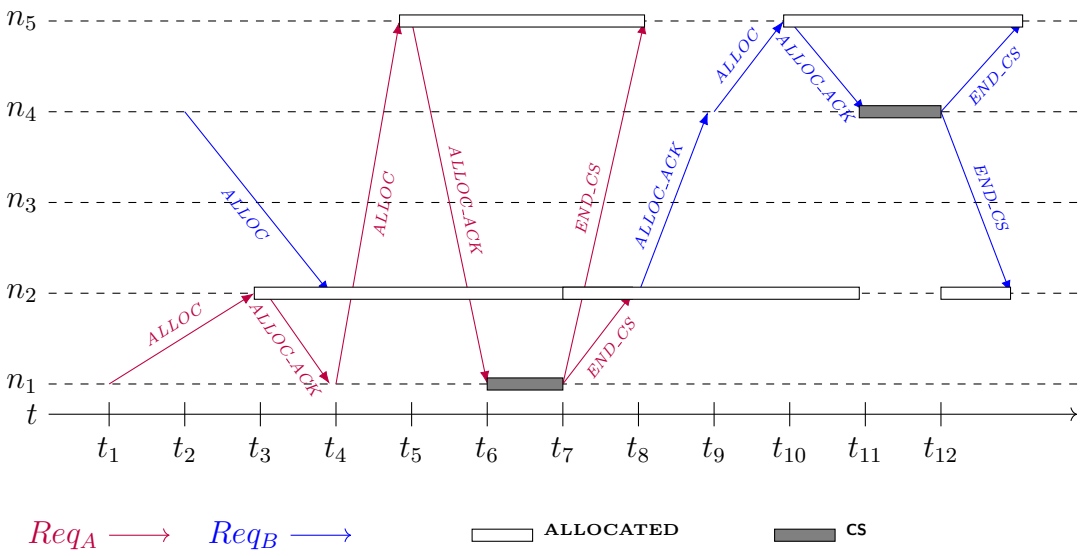
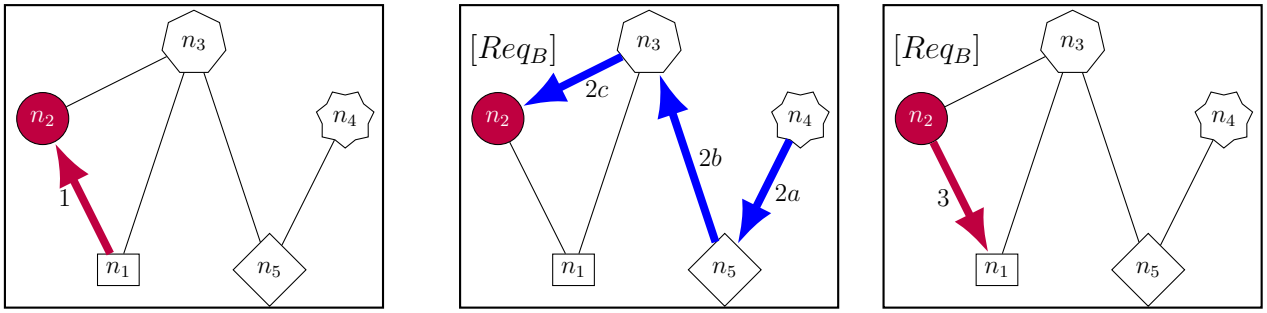


Figure 6.2: Timeline of sample execution of Dijkstra's Incremental algorithm

Execution At t_1, n_1 requests to enter its CS for Req_A. It sends an $ALLOC$ message to n_2, the node in the request that comes first according to the order (Figure 6.3a).



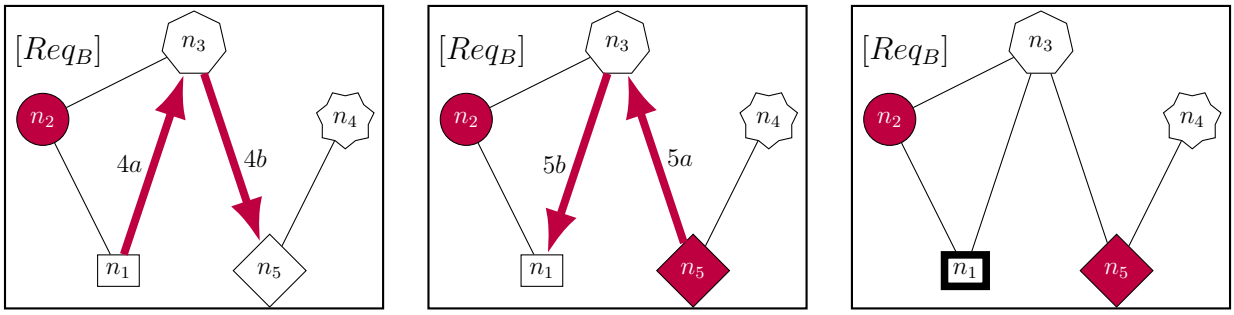
(a) t_1: Req_A is emitted from n_1 and tries to allocate n_2 first

(b) t_2: Req_B is emitted from n_4. It is stored in the local queue of n_2.

(c) t_3: Req_A continues, n_2 acknowledges

Figure 6.3: Sample execution of Dijkstra's Incremental algorithm 1/4

At t_2, n_4 requests to enter its CS for Req_B. Similarly to n_1, it first sends an $ALLOC$ message to n_2. n_2 is already $ALLOC$ for Req_A so it stores Req_B in a local queue of pending requests (Figure 6.3b). At t_3, n_2 replies with an $ALLOC\_ACK$ to n_1 because it has received Req_A first (Figure 6.3c), as shown by the timeline in Figure 6.2.



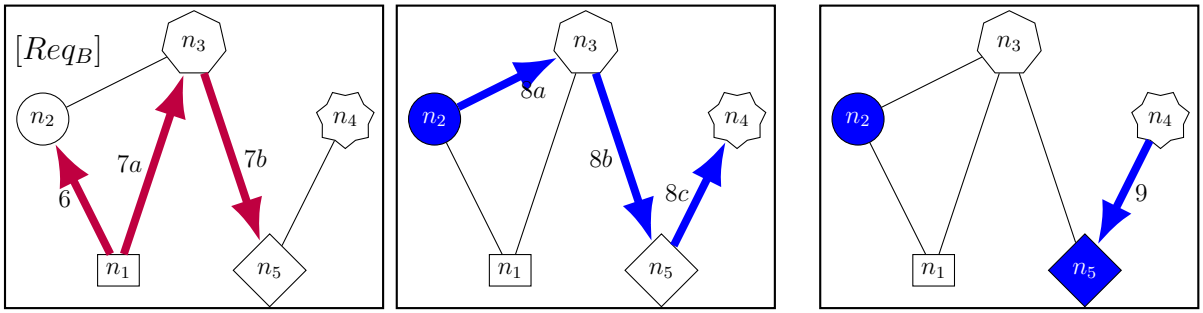
(a) t_4: Req_A continues

(b) t_5: Req_A allocation finishes

(c) t_6: Req_A is allocated

Figure 6.4: Sample execution of Dijkstra's Incremental algorithm 2/4

At t_4, n_1 then sends an *ALLOC* to n_5 (Figure 6.4a), that replies with an *ALLOC_ACK* at t_5 (Figure 6.4b). At t_6, Req_A is allocated and n_1 can enter its CS and use the resources of n_2 and n_5 (Figure 6.4c).



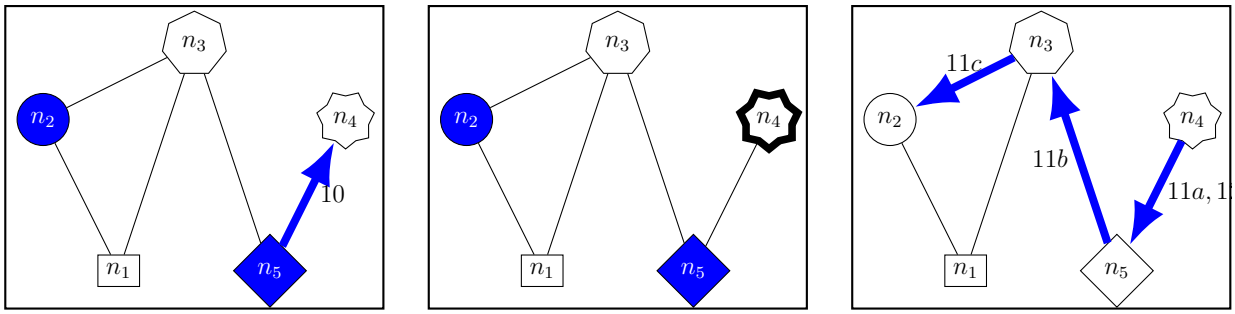
(a) t_7: Req_A is done using resources, n_1 releases them

(b) t_8: n_2 resumes and acknowledges Req_B

(c) t_9: n_4 allocates n_5

Figure 6.5: Sample execution of Dijkstra's Incremental algorithm 3/4

When n_1 releases its CS at t_7 it sends *END_CS* messages to n_2 and n_5 (Figure 6.5a). At t_8, n_2 and n_5 check their local queues (Figure 6.5b). The queue of n_5 is empty for now, and n_2 read the pending request Req_B in its local queue, removes it from its queue, and decides to resume it by sending an *ALLOC_ACK* message to n_4.



(a) t_{10}: Req_B allocation finishes

(b) t_{11}: Req_B is allocated, n_4 uses resources

(c) t_{12}: n_4 releases resources

Figure 6.6: Sample execution of Dijkstra's Incremental algorithm 4/4

Then, at t_9 n_4 sends an *ALLOC* message to n_5 (Figure 6.5c), which acknowledges with

an *ALLOC_ACK* at t_{10} (Figure 6.6a), allowing n_2 to enter its CS and use the resources at t_{11} (Figure 6.6b). Once n_4 releases its CS, at t_{12}, it sends *END_CS* messages to n_2 and n_5 (Figure 6.6c).

Message Complexity

For each request, the algorithm requires to send an *ALLOC* message to each node holding one of the requested resources to allocate them, which in turn sends an acknowledgement *ALLOC_ACK* message to the requesting node. It also requires one *END_CS* message per resource in the request to release it. There are 3 messages per resource requested, hence the message complexity of Dijkstra's incremental algorithm is $O(N)$, where N stands for the number of resources. This is summed up in Table 3.4 in Section 3.3.

In the example here:

- Req_A requires the messages 1, 3, 4a, 4b, 5a and 5b to allocate the resources as well as the messages 6, 7a and 7b to release them. A total of six meaningful messages, plus three additional forwarding messages.

- Req_B requires the messages 2a, 2b, 2c, 8a, 8b, 8c, 9 and 10 to allocate the resources as well as the messages 11a, 11b 11c and 12 to release them. A total of six meaningful messages, plus six additional forwarding messages.

6.2.3 Performance evaluation

Simulation results The comparison of the results of Dijkstra's *Incremental* algorithm and of the *byvalues* allocation order heuristic on SimGrid are shown in Figure 6.7.

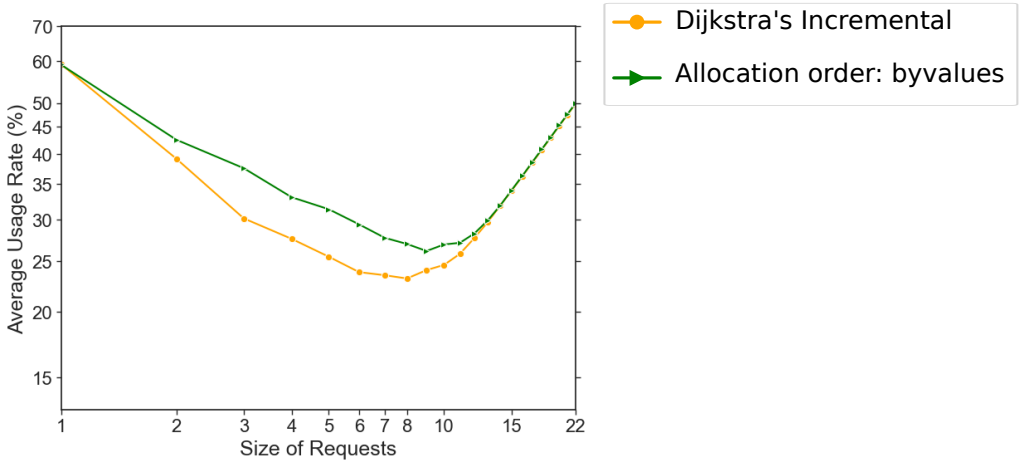
All metrics show that allocating the resources with the *byvalues* allocation order heuristic gives the best results. In Figure 6.7a an improvement of up to 20%, for requests of size 6, can be seen for the *Average Usage Rate*. Figure 6.7c shows that the *Average Waiting Time* for the *Incremental* algorithm is significantly worse than for the *byvalues* heuristic for requests of size 3 and more due to the domino effect.

As shown in Figure 6.7b, *byvalues* does not generate more messages than the *Incremental*. The *byvalues* heuristic requires one *ALLOC* message per resource in the request, plus an additional *ALLOC_ACK* at the end for a total of $r + 1$ messages where r is the size of the request, when the *Incremental* algorithm requires two messages per resource, an *ALLOC* and an *ALLOC_ACK* for a total of $2 * r$ messages. For requests of size 5 and less, the *byvalues* heuristic requires less messages but for requests of 6 and more the two have a similar number of messages, this is because of preemptions. The number of *PREEMPT* and*PREEMPT_ACK* messages increases reducing the difference in performance. This is similar to the difference between the *parallel* and the *byvalues* heuristics for the same reasons.

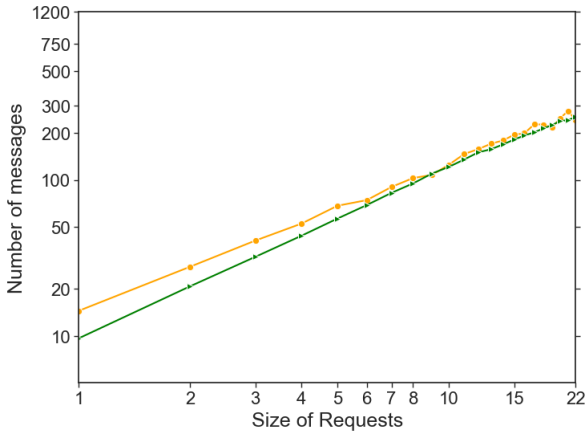
Real experiments with MPI

The results shown above from the simulator are confirmed with an implementation based on Open MPI version 3.1.3 [Gab+04]. The core of the experimental platform remains the same, Open MPI is implemented as an alternative execution platform from SimGrid. The implementation of the messages for this platform relies on Protocol Buffers [2] for

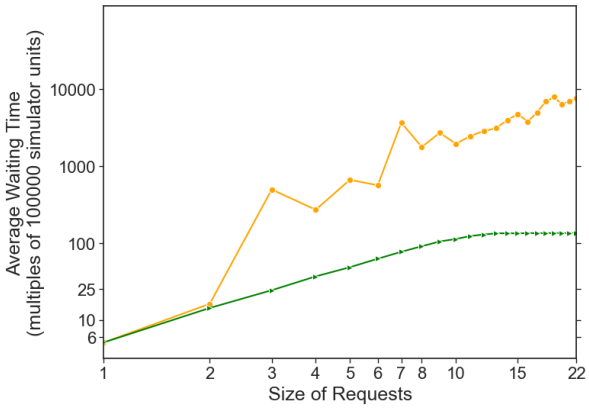
[2] `https://developers.google.com/protocol-buffers`



(a) Average Usage Rate



(b) Average Number of Messages



(c) Average Waiting Time

Figure 6.7: Comparison with Dijkstra's incremental algorithm in a system with one instance of 44 types of resources

the serialisation of the messages. Each node in the topology is run on a stand alone application. The applications are run on several servers on the Grid'5000 infrastructure [Bal+13]. The selected servers have two Intel Xeon E5-2630v3 8-core processors, 128 GiB of RAM, 1Gbps network interfaces.

The tests are done on the same *Cesnet200706* 44-node topology. The application instances are split across 4 servers. There is no correlation between the Grid'5000 servers used and the nodes of the topology, the nodes of the topology are distributed randomly on the 4 servers. Each Grid'5000 server runs 11 application instances. The network configuration is standard for Open MPI, no specific events are programmed. All communications use the MPI primitives to send and receive messages and are not optimised in case two nodes are on the same physical Grid'5000 server, but MPI may optimise these communications. However the network latency is not analysed for these tests. As for the simulation based on SimGrid, there are 44 types of resources and a single instance of each of them. As for the simulation the duration of the CS is constant, it is set to 20 seconds. The duration of each test is also constant and set to 30 minutes.

Table 6.1 shows some numerical results for the *Average Usage Rate*. There is no significant difference on how algorithms compare to each other from what is observed on the simulator. Numerical values vary from the simulator but remain comparable.

Request size	Dijkstra's Incremental		*reverse* heuristic		*byvalues* heuristic	
	MPI	SimGrid	MPI	SimGrid	MPI	SimGrid
5	21.56%	25.46%	27.18%	27.21%	**32.27%**	31.36%
6	17.14%	23.7%	20.35%	25.19%	**31.07%**	29.34%
7	17.64%	23.47%	25.74%	24.5%	**26.23%**	27.27%
8	21.41%	23.02%	24.46%	23.84%	**26.70%**	26.95%

Table 6.1: *Average Usage Rate* with Open MPI

6.3 Chandy-Misra DrPP algorithm

The Chandy-Misra DrPP algorithm [CM84] is detailed and evaluated in this section. It is based on their DiPP algorithm. The DrPP algorithm is one of the algorithms that have been evaluated as subroutines by Rhee for his algorithm [Rhe95] described in the next section 6.4.

6.3.1 Description of the algorithm

The DrPP is a generalisation of the DiPP in which requests are dynamic, i.e., a philosopher can request any subset of resources and this subset can change across requests. In this problem the resources are bottles that the philosophers need to drink from. The forks are not resources but are **auxiliary** resources that are required to collect the bottles and to enter the CS. Chandy-Misra solution relies on their DiPP algorithm. In their model, bottles are associated to the same edges of the *conflict graph* as forks.

The algorithm builds a graph which represents the precedence between pairs of conflicting processes (philosophers), called the **precedence graph** of the system. This graph is similar to the *conflict graph*, it has the same nodes and edges. However in the precedence graph, edges are oriented. They are oriented towards the philosopher holding the fork. The algorithm requires the precedence graph to be acyclic at any given time, including at the initialisation.

Requesting the CS

When a node wants to enter its CS, it first must request the forks from the philosophers holding them according to the same precedence graph used for the DiPP algorithm. When a philosopher holds all the forks he can decide to drink from the bottles that are on the same edges as the forks, or leave them. Another way to say it is that the philosopher needs to enter the DiPP CS first, which gives him a lock on the bottles that allows him to drink from them.

Releasing the CS

When a node releases its CS (resp. is done drinking), it can send the bottles it is holding to its neighbours.

Adaptation to the system

In Chandy-Misra model, bottles (resources) and forks (auxiliary resources) sit on each edge of the *conflict graph*. In the example given in the paper the forks and bottles are

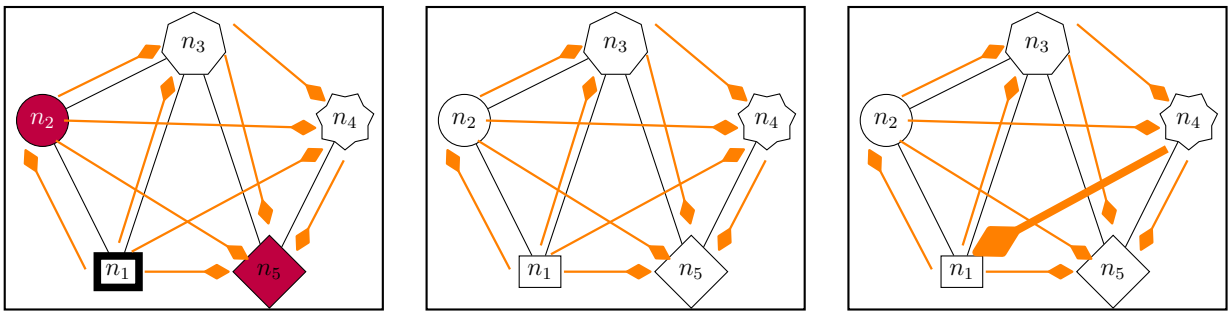
only shared by two philosophers (nodes). In the sample system described at the beginning of this chapter, any node can request any subset of resources. And this subset can change across requests. It is then necessary that each node shares a fork and a bottle with every other nodes. This means that the precedence graph becomes a **complete graph**. Figure 6.8a shows both the topology, with the black undirected edges, and the precedence graph where the oriented edges are represented by the orange forks. Both graphs share the same nodes.

Pros and cons

Chandy-Misra algorithm was the first to describe and solve the DrPP. Their solution allows multiple requests to allocate their resources concurrently when they do not share resources in the *conflict graph*. The major drawback of Chandy-Misra algorithm in the generalised setting, described just above, where any node can request any subset of resources, is that the precedence graph becomes a complete graph. So only one node can use resources at any given time.

6.3.2 Example of execution for the sample scenario

Initial state At t_0 the forks are held by the node with the lowest identifier. They are represented in Figure 6.8 with the handle of the fork facing the node holding it. For example in Figure 6.8a n_1 holds all its forks. The names of the fork is not shown in Figure for readability reasons but, like for the example for the DiPP algorithm, a fork is labelled f_{ij} where i and j are the identifiers of the two nodes sharing the fork. The bottles are the resources held by the nodes.



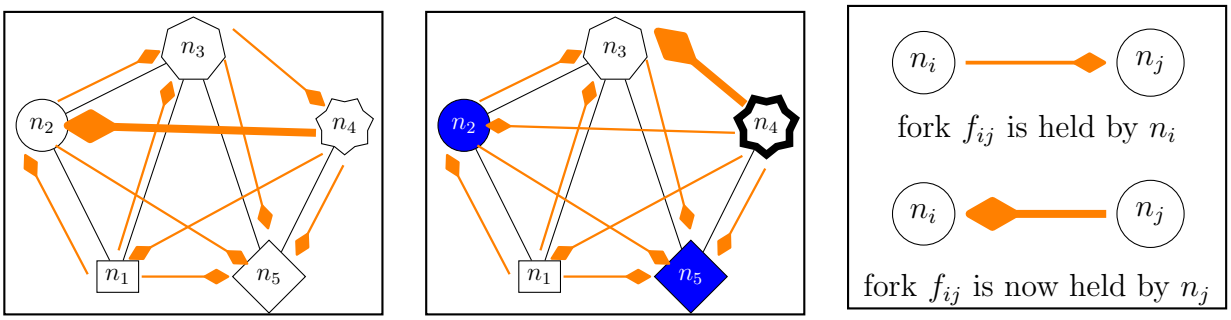
(a) t_1: initial stage, n_1 can enter CS for Req_A

(b) t_2: n_1 releases its CS, n_4 requests fork from n_1

(c) t_3: n_4 gets fork from n_1

Figure 6.8: Chandy-Misra DrPP algorithm 1/2

Execution Figures 6.8 and 6.9 show how the system evolves when n_1 and then n_4 want to enter their CS successively. Initially, at t_1, n_1 holds all the forks, it can clean them and enter its CS immediately to use the resources (bottles) n_2 and n_5 (Figure 6.8a). At t_2, n_1 releases its CS and stop using the resources (Figure 6.8b). At this moment, n_4 wants to enter its CS, it needs for that to gather the forks it does not currently hold from its neighbours in the precedence graph n_1, n_2 and n_3. As shown in 6.10, it starts by sending a *REQ_FORK* message to n_1. When it gets f_{14}, at t_3, it requests their shared fork f_{24} to n_2 (Figure 6.8c).



(a) t_4: n_1 gets fork from n_2 (b) t_5: n_4 gets fork from n_3 and can enter CS (c) Legend

Figure 6.9: Chandy-Misra DrPP algorithm 2/2

When n_4 gets f_{24}, at t_4, it then requests f_{34} to n_3 (Figure 6.9a). When it finally gets f_{34}, at t_5, n_4 holds all its forks and can enter its CS and start using resources n_2 and n_5 (Figure 6.9b).

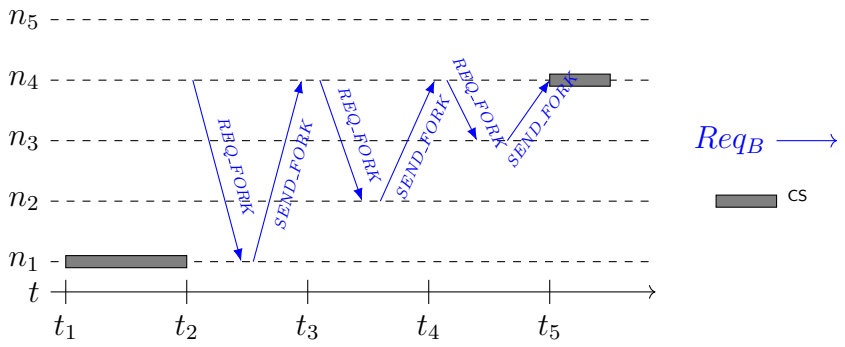


Figure 6.10: Timeline of Chandy-Misra DrPP algorithm

Message Complexity

The algorithm requires a node to send one message to each of the other nodes to request the shared forks. Then, each of these nodes needs to reply to this message to send the fork. If the node already holds a fork it does not need to send a message for this fork. The message complexity of Chandy-Misra DiPP algorithm is $O(N)$. This is summed up in Table 3.4 in Section 3.3.

6.3.3 Performance evaluation

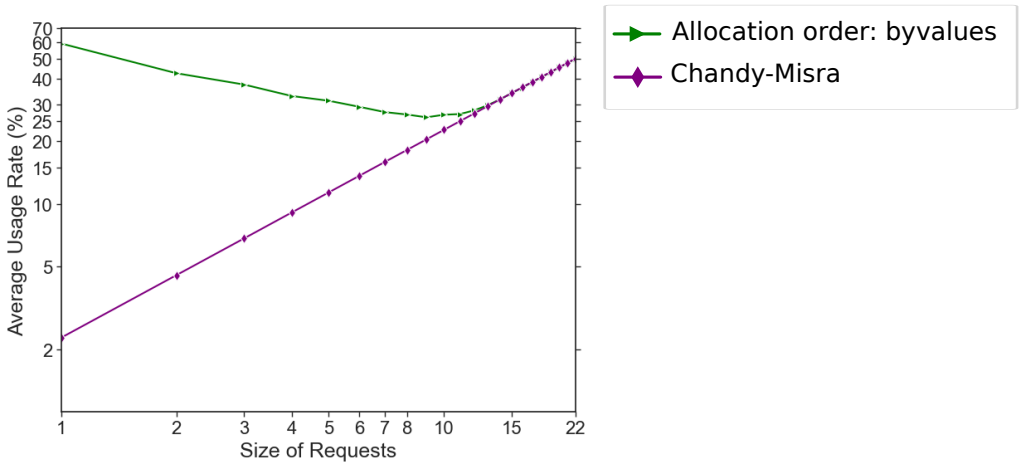
Figure 6.11 shows the results of the simulations run on SimGrid of Chandy-Misra DrPP algorithm [CM84] compared to the *byvalues* allocation order heuristic.

As Chandy-Misra algorithm allows only one CS at a time, only one request can be allocated at any given time so the *Average Usage Rate* is linear (Figure 6.11a). The usage rate can be approximated easily, for request of size 7, usage rate is approximately $7/44 \approx 15.91\%$. This linear graph highlights when the *Average Usage Rate* becomes linear for the *byvalues* heuristic. The two graphs rejoin shortly after a size of 12 for requests.

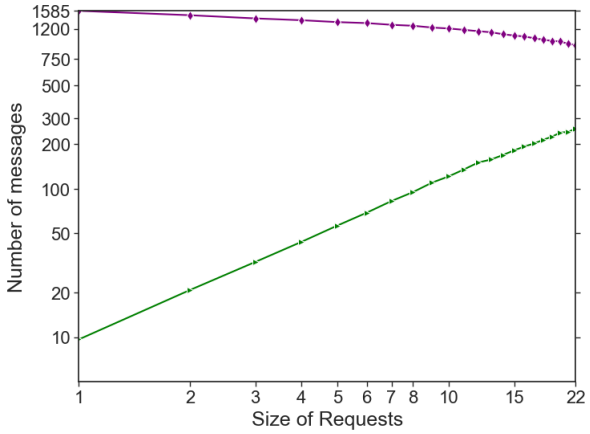
The *Average Number of Messages* is significantly higher for Chandy-Misra algorithm than for the *byvalues* heuristic, as shown in Figure 6.11b. This is due to the large number

of messages to manage the *forks* used to choose the node that enters the CS. As an example, *REQ_FORK* and *SEND_FORK* account for 97.8% of the total number of messages for requests of size 6.

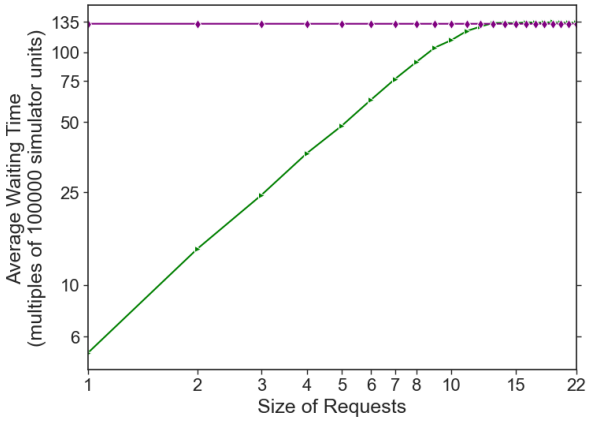
Figure 6.11c shows that the *Average Waiting Time* is constant for Chandy-Misra. Because a single request is in CS at a given time, all the other requests have to wait for their turn. As the algorithm is fair, they all wait for a same amount of time. Each node has to wait for 43 times the duration of a CS, 300000 in these simulations, so $43 * 300000 = 1.29 * 10^7$. It is an order of magnitude higher than for the proposed algorithm for smaller requests. This is consistent with the lower *Average Usage Rate* : as only one request is allocated at a given time, all the other requests wait for their turn. The two graphs rejoin for the same request size as for the *Average Usage Rate*. For requests of size 12 and more, both algorithms allocate requests sequentially and both are fair.



(a) Average Usage Rate



(b) Average Number of Messages



(c) Average Waiting Time

Figure 6.11: Evaluation of Chandy-Misra algorithm in a system with one instance of 44 types of resources.

6.4 Rhee's algorithm

Rhee's algorithm [Rhe95] is modular and composed of two subroutines.

The first subroutine can rely on any DiPP or DrPP algorithm to get an exclusive access to the nodes holding the resources for the request. It can be any DiPP or DrPP algorithm, the article has evaluated the performance using Chandy-Misra[CM84] and Choy-Singh [CS93] algorithms as subroutines. In terms of terminology, this algorithm requires to distinguish between the *subroutine CS*, i.e., the CS of the algorithm used for the first subroutine, from the *algorithm CS*. While in the *subroutine CS* the algorithm computes the position of the request in a **global distributed queue**. To achieve this, each node maintains a local queue of requests.

Next, during the second subroutine, requests are allocated according to the order in this queue. The queue is then updated when requests release their resources.

The intuition behind the algorithm is that by using a DrPP algorithm during the first subroutine to manage only the order in the distributed queue the lock on the processes is shorter and the performance of the selected DrPP algorithm is improved.

6.4.1 Description of the algorithm

Adaptation to the model

In Injong Rhee's model processes, called *users*, try to enter their CS by requesting resources to *resource managers*, each resource manager manages one resource. The topology issue is not addressed, it is assumed that every users can communicate with every resource managers and that every resource managers can communicate with their peers. To adapt it to the model introduced in Section 2.4.1 resource managers are mapped to nodes. Users are mapped to nodes as well because the model assumes each node can also be a requesting node.

Requesting the CS

When a node wants to enter its algorithm CS, the first subroutine (i.e., the algorithm that has been chosen for the first subroutine) is executed to get an exclusive access to the nodes holding the requested resources. The algorithm then computes a position for the request using *REPORT* and *MARKED* messages. *REPORT* messages are sent by the requesting node to get information about the current queues of nodes. A node replies with a *MARKED* message that contains the list of currently occupied positions in its local queue. The computed position is the same in all the local queues, it is computed to be the lowest common position available among all the nodes holding resources for the request. It then leaves the *subroutine CS*. If it is in the first position of the local queues, it can then enter the *algorithm* CS and start using the resources.

When a request is the first in the local queues of the nodes holding the requesting resources, which means it is also the first in the distributed queue, each of these nodes sends a *GRANT* to the requesting node. When the requesting node has received *GRANT* from all the nodes it enters its CS.

Releasing the CS

When a node releases its algorithm CS and the allocated resources, *ADV* or *DEC* messages are sent to inform other nodes that other requests can move up in their local queues. *DEC* messages are sent by the nodes holding the resources that just got releases to the requesting nodes of requests in their local queues that can now move up in the queue. If

a requesting node has received *DEC* messages from all the nodes in a request, it sends back *ADV* to inform them that they can move the request up in their queue. This is done asynchronously and this step does not require an exclusive access to the nodes.

Pros and cons

Rhee's algorithm improves the *Average Usage Rate* of the algorithms used for the subroutine evaluated in the paper: Chandy-Misra[CM84] and Choy-Singh [CS93]. However it requires a very large number of messages to manage the distributed queue.

6.4.2 Example of execution in sample system

Initial state Initially the local queues of all nodes are empty, hence the distributed queue is also empty.

Execution Figures 6.12 and 6.13 to 6.16 show the execution of the algorithm for the two requests Req_A and Req_B introduced in Figure 6.1. Table 6.2 shows the evolution of the local queues of each node at each step.

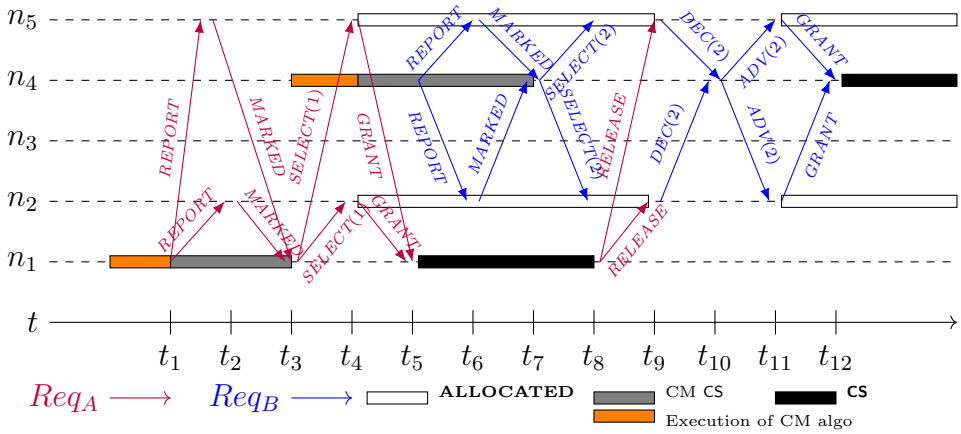
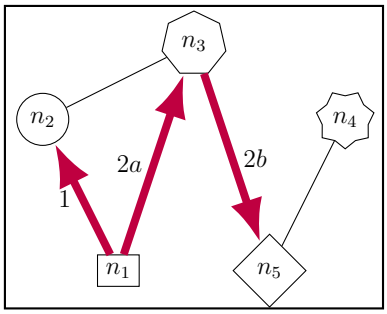
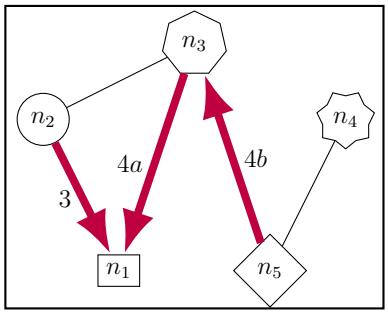


Figure 6.12: Timeline of sample execution of Rhee's algorithm

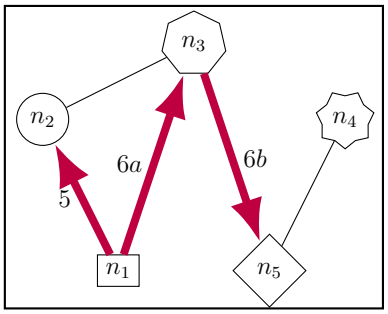
n_1 needs to execute Chandy-Misra (hereafter CM) algorithm to enter the CM CS for Req_A and n_4 needs to execute CM to enter the CM CS for Req_B. Section 6.3 showed that only one of them can enter the CM CS at any given time. In this example n_1 is the first to enter the CM CS. The messages and steps of the CM algorithm are not included here for the sake of readability.



(a) t_1: Req_A is in CM CS, n_1 sends *REPORT*



(b) t_2: resources reply with *MARKED* to n_1



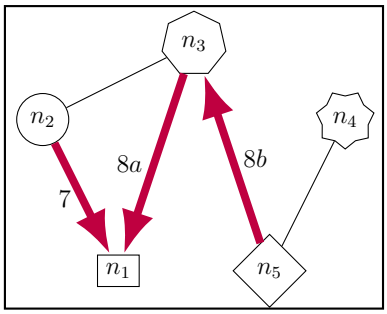
(c) t_3: n_1 selects position 1 for Req_A

Figure 6.13: Sample execution of Rhee's algorithm 1/4

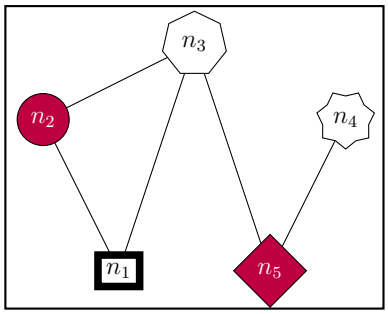
At t_1, n_1 starts in CM CS and n_1 sends a *REPORT* message to the nodes holding the resources, as shown in 6.13a. Upon reception of this message each node replies with a *MARKED* message at t_2 (Figure 6.13b) that contains the list of positions already occupied in its local queue. In this case, all the queues being initially empty, each message contains an empty list. Based on this information, n_1 can select position 1 for Req_A. Table 6.2 shows the evolution of the local queues during the execution of this example. n_1 sends a *SELECT* message at t_3 (Figure 6.13c) to n_2 and n_5 to inform them it has selected the first position. n_1 can then leave the CM CS as it has secured its initial place.

Node	time											
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
n_1	$\varnothing$											
n_2	$\varnothing$		position 1 : Req_A						position 1 : Req_A position 2 : Req_B		position 1 : Req_B	
n_3	$\varnothing$											
n_4	$\varnothing$											
n_5	$\varnothing$		position 1 : Req_A						position 1 : Req_A position 2 : Req_B		position 1 : Req_B	

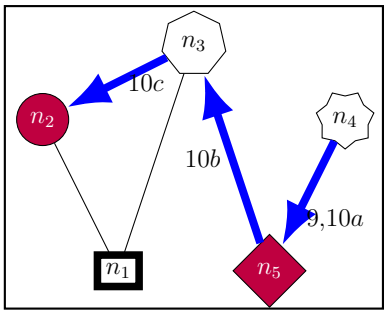
Table 6.2: Evolution of the local queues of the nodes with time



(a) t_4: position 1 is granted to n_1 for Req_A



(b) t_5: Req_A start using the resources

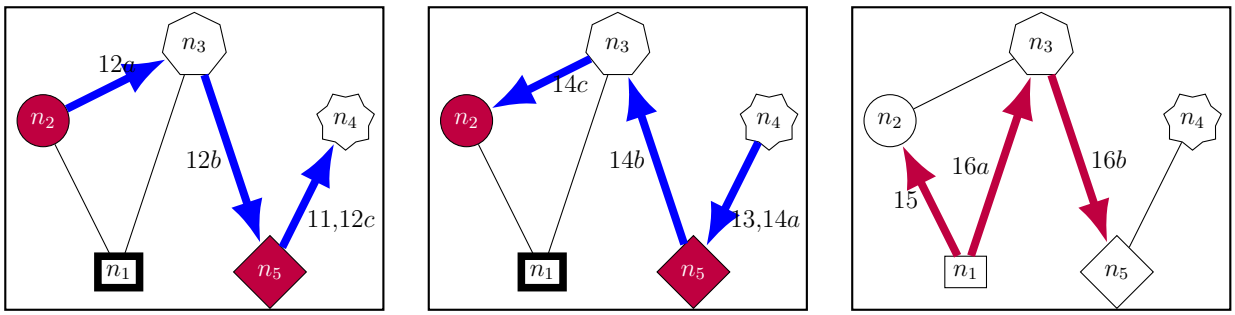


(c) t_6: Req_B is in CM CS. n_4 sends *REPORT*

Figure 6.14: Sample execution of Rhee's algorithm 2/4

When they receive the selection from n_1, the nodes grant the access to their resources with a *GRANT* message (Figure 6.14a). At t_5, n_1 can enter its *algorithm* CS and start

using the resources. As n_1 has left the CM CS at t_3, n_4 can execute CM and enter the CM CS. As for n_1, the messages for this part are not included here for the sake of readability. It is assumed that the CM algorithm has been executed between t_3 and t_6. Req_B can then follow the same steps as Req_A: it asks the nodes for a *REPORT* message of their occupied positions at t_6 (Figure 6.14c).



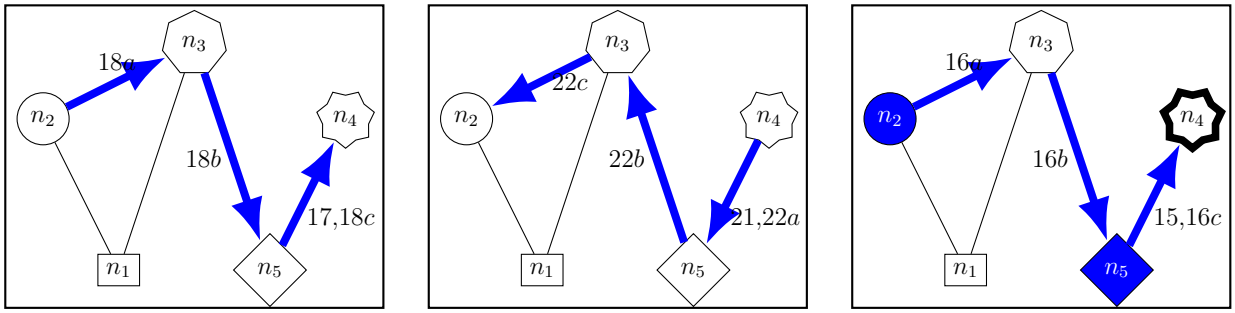
(a) t_7: resources reply with *MARKED* to n_4

(b) t_8: Req_B selects position 2

(c) t_9: Req_A releases resources

Figure 6.15: Sample execution of Rhee's algorithm 3/4

The nodes reply with *MARKED* at t_7 (Figure 6.15a) allowing n_4 to select a position in the queues (Figure 6.15b). As position 1 is already held by Req_A, it selects position 2 at t_8 (Figure 6.15b). When Req_A releases its CS, n_1 sends *RELEASE* messages to release the resources at t_9 (Figure 6.15c).



(a) t_{10}: *DEC* sent to inform n_4 it can change Req_B position

(b) t_{11}: n_4 sends *ADV* to change Req_B position

(c) t_{12}: position 1 is granted to Req_B

Figure 6.16: Sample execution of Rhee's algorithm 4/4

Once the resources are released, the next requests can advance in their local queues. n_2 and n_5 inform n_4 with *DEC* messages that its position in the distributed queue can decrease at t_{10} (Figure 6.16a). n_4 accepts to advance its position in the queues and informs them with *ADV* messages at t_{11} (Figure 6.16b). Req_B is now in the first position, it can enter its *algorithm* CS and start using the resources at t_{12}.

Message Complexity

Injong Rhee evaluated the performances of his algorithm using the Chandy-Misra [CM84], and Choy-Singh [CS93] DrPP algorithms. The results show that the performance of this algorithm on the Average Waiting Time is similar no matter the algorithm used as a

subroutine, with Chandy-Misra requiring less messages than Choy-Singh in his experiment settings. For the evaluation below, Rhee's algorithm relies on the Chandy-Misra [CM84] DrPP algorithm as a subroutine.

The algorithm relies on another algorithm as a first subroutine so the number of messages needed depends on the algorithm selected. For the rest of the execution the algorithm requires one *REPORT*, one *SELECT* and one *GRANT* per resource in the request. It also requires *DEC* and *ADV* for each conflict. The complexity is $O(Msg_A + r\delta\})$ where Msg_A is the number of messages of the first subroutine, r is the size of the request and δ the maximum number of conflicting requests at any time during the execution of the algorithm.

6.4.3 Performance evaluation

Figures 6.17 show, in red, the results of simulations of Rhee's algorithm [Rhe95] compared to the *byvalues* heuristic on SimGrid. These are the only figures where results are not shown for all the possible values of x due to the very high number of messages. The number of messages has a direct influence on the duration of a simulation. Because the SimGrid simulator is based on discrete events, each time a message is sent or received the execution of the thread is interrupted. So the more messages there are in a test the longer its simulation lasts. For example in the results below, for requests of size 8 the *byvalues* heuristic sends around 100 messages in average and Rhee's algorithm around 4325 messages (Figure 6.17b). The duration of the simulation for the algorithm in this case is around 6 minutes, while the simulation of Rhee's algorithm lasts 5220 minutes (or 4.5 days). Generating the whole graph takes weeks while providing no additional information. Only the simulations for the most significant size of requests have been evaluated on the experimental platform. The algorithm was also evaluated, along with the another algorithm described in this chapter, on other topologies with a smaller number of nodes.

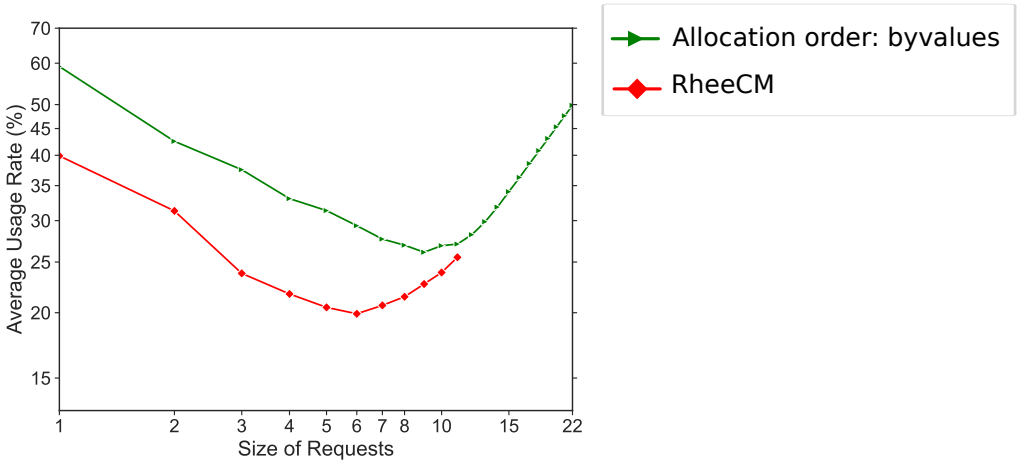
As expected, the algorithm improves the *Average Usage Rate* from the pure Chandy-Misra DrPP algorithm as shown in Figure 6.17a. But the *Average Usage Rate* is still significantly lower than with the *byvalues* heuristic. Rhee's original paper evaluated the algorithm with Chandy-Misra[CM84] and Choy-Singh [CS93] algorithms for the first subroutine. The results show that there is no significant difference to expect with a different algorithm because the time spent in the *subroutine* CS is orders of magnitude shorter than the time spent in the *algorithm* CS.

The *Average Number of Messages*, shown in Figure 6.17b, is worse than the Chandy-Misra algorithm because it adds new messages for the management of the distributed queue. It is the worst algorithm for that metric by several orders of magnitude. This metric could be improved if another algorithm than Chandy-Misra had been used for the first subroutine, however the results show that the *Average Number of Messages* for Rhee's algorithm grows exponentially due to the messages specific to the algorithm. Rhee's paper shows that using Chandy-Misra algorithm for the first subroutine requires less messages than when using Choy-Singh algorithm. It also shows that it requires less messages than Awerbuch-Saks [AS90] algorithm on most of the test cases. Rhee's own evaluation in his article show that using Choy-Singh algorithm [CS93] rather than Chandy-Misra results in a significantly larger number of messages, around 30%.

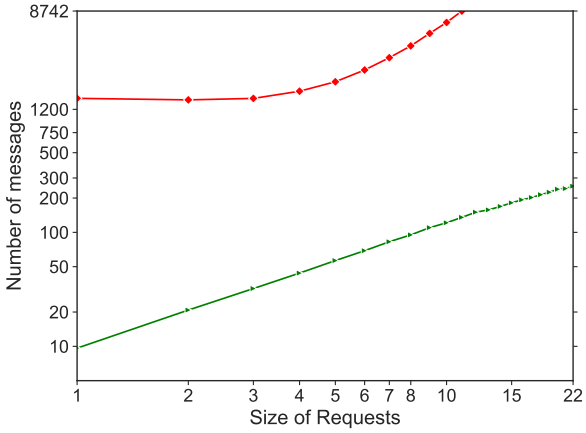
The *Average Waiting Time* is improving from the Chandy-Misra algorithm. It is however slightly higher than the *byvalues* heuristic (Figure 6.17c).

Using another algorithm than Chandy-Misra DrPP for the first subroutine could affect, and probably improve, the results for the *Average Usage Rate* and *Average Waiting Time*, but it would not reduce the number of messages required for the management of the distributed queue.

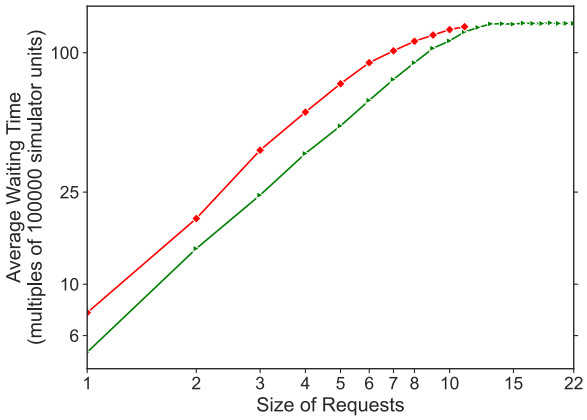
Rhee's solution to the distributed allocation problem is based on the construction of a distributed queue of the requests. It does not try to find a better scheduling, it simply is based on the order of arrival of the requests on each node. A similar approach could be used to build an optimised queue that would increase the *Average Usage Rate*, but that would require synchronising all the nodes on this queue. This can be achieved with a global view of the requests for example. However that could require even more messages, which is already the main drawback of Rhee's algorithm.



(a) Average Usage Rate



(b) Average Number of Messages



(c) Average Waiting Time

Figure 6.17: Evaluation of Rhee's algorithm in a system with one instance of 44 types of resources.

6.5 Bouabdallah-Laforest algorithm

The Bouabdallah-Laforest [BL00] algorithm extends the Naimi-Tréhel algorithm for the exclusive access to multiple types of resources. The algorithm is token-based with a unique *Control Token* in the system and one *Resource Token* per resource. The algorithm relies on the Chang [CHA90] improvement of the Naimi-Tréhel Mutual Exclusion algorithm

described hereafter to move the *Control Token* in an overlay tree built on top of the physical topology.

First the Naimi-Tréhel algorithm is described. Then, the Bouabdallah-Laforest algorithm is explained with the same example as the other algorithms.

6.5.1 Naimi-Tréhel Mutex algorithm

The Naimi-Tréhel Mutex algorithm [NT87] is a token-based algorithm for Mutual Exclusion, in systems with one instance of one type of resources. This algorithm builds two virtual structures: a dynamic tree and a queue. The head of the queue is the node holding the token and the tail the last node to have requested it. The virtual tree describes the path to the tail of the queue and its root is eventually the node holding the token. Each node maintains two variables:

- *father* stores the father of the node in the tree
- *next* stores the node to send the token to when leaving the CS, i.e., the next node in the queue

The algorithm uses two messages *REQUEST* and *TOKEN* to move the token in the tree. A node that wants to enter its CS requests the token with a *REQUEST* message. The token is sent in a *TOKEN* message.

The example below shows how the virtual structures are updated dynamically according to the requests of the nodes. These dynamic structures allow the algorithm to limit the number of messages. A request to enter the CS is sent only to a subset of all the nodes according to the current structure. Thank to this, the algorithm has a logarithmic complexity for the *Average Number of Messages*. This makes it one of the most efficient algorithms for Mutual Exclusion for this metric as shown in Section 3.3.

Example of execution

Figures 6.19 to 6.21 show a sample execution for the algorithm. The scenario is, given a global clock, n_1 requests to enter its CS, then n_3 and finally n_4. Figure 6.22 shows the virtual structures maintained by the algorithm. The messages sent are shown in Figure 6.18. Initially the root of the virtual tree is n_1 (Figure 6.22a).

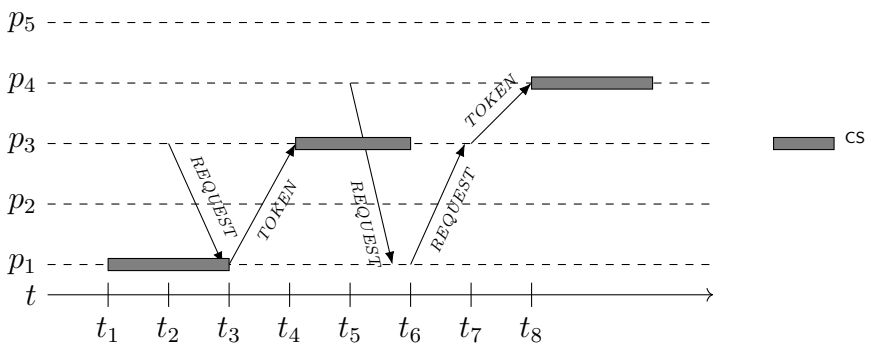
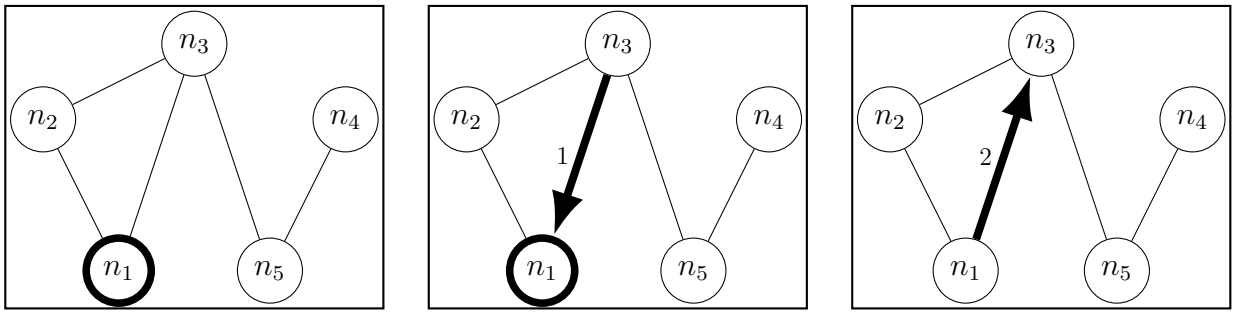


Figure 6.18: Timeline of sample execution of Naimi-Tréhel algorithm



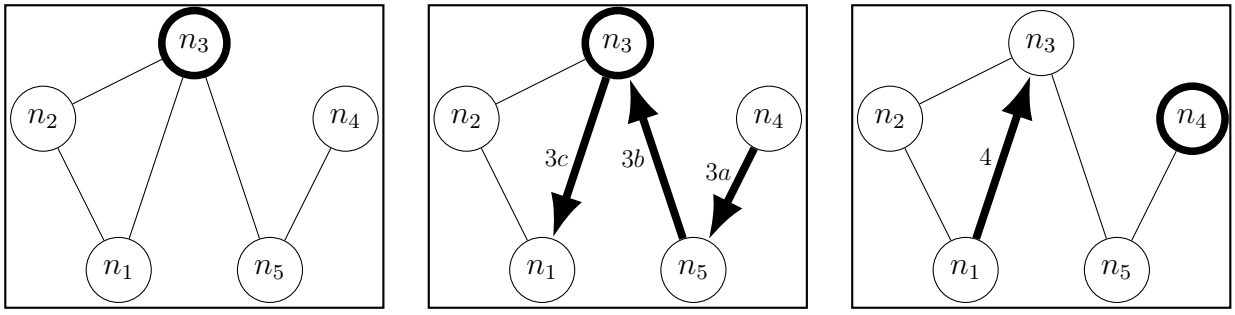
(a) t_1: n_1 enters its CS

(b) t_2: n_3 requests to enter its CS

(c) t_3: n_1 leaves CS, sends the token to n_3

Figure 6.19: Sample execution of Naimi-Tréhel algorithm 1/3

At t_1, n_1 requests to enter its CS and can do it immediately as it holds the token (cf. Figure 6.19a). Then at t_2 (Figure 6.19b), n_3 wants to enter its CS, to do so it sends a *REQUEST* message to n_1 which stores n_3 as its *father* and in its *next* variable (Figure 6.22b). When n_1 leaves its CS, at t_3, it sends the token using a *TOKEN* message to n_3 (Figure 6.19c) and clear its *next* variable (Figure 6.22c).



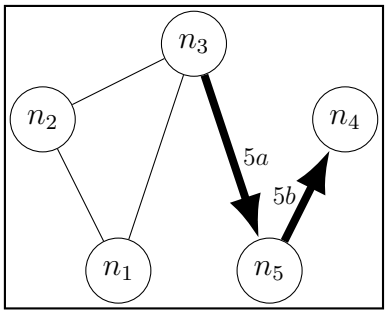
(a) t_4: n_3 enters its CS

(b) t_5: n_4 requests to enter its CS

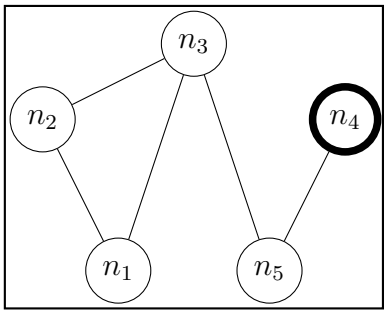
(c) t_6: n_1 forwards the request to n_3

Figure 6.20: Sample execution of Naimi-Tréhel algorithm 2/3

At t_4, n_3 can enter its CS because it has received the token (Figure 6.20a). When n_4 wants to enter its CS, at t_5 (Figure 6.20b), it requests the token to its *father* which is n_1. The virtual tree is updated so that n_4 is the new *father* of n_1 (Figure 6.22d). At t_6, when n_1 receives the request, it does not hold the token anymore so it forwards the request to its own *father* n_3 (Figure 6.20c). n_4 becomes the ex *father* and *next* of n_3, (Figure 6.22e).



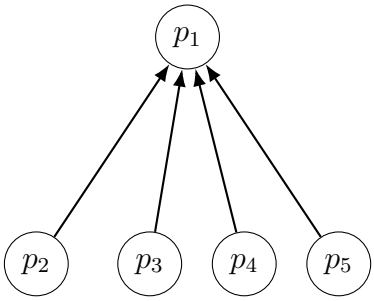
(a) t_7: n_3 leaves CS and sends the token to n_4



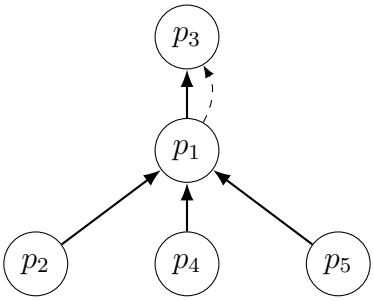
(b) t_8: n_4 enters its CS

Figure 6.21: Sample execution of Naimi-Tréhel algorithm 3/3

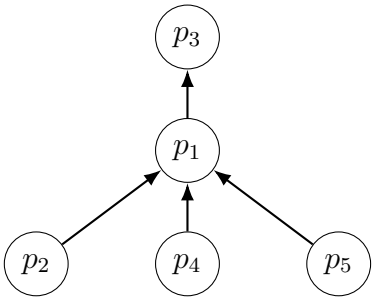
Finally when n_3 leaves its CS, at t_7, it sends the token to n_4 and clear its *next* (Figure 6.22f). At t_8, n_4 can enter its CS.



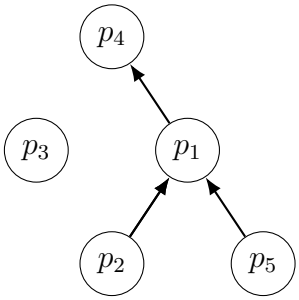
(a) t_1: p_1 is the root of the virtual tree



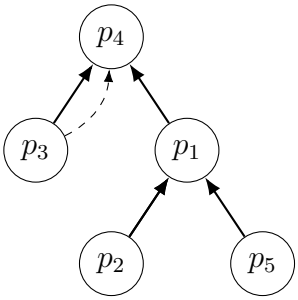
(b) t_2: p_3 becomes the root of the virtual tree and the *next* of p_1



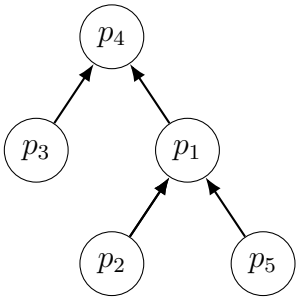
(c) t_3-t_4: p_1 has released its CS. It clears its *next* when sending the token to p_3



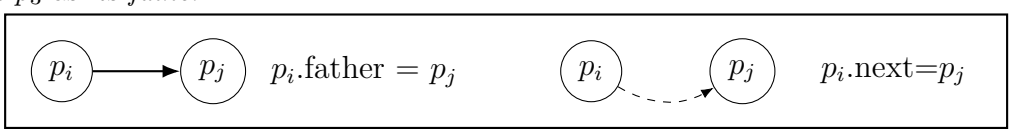
(d) t_5: p_4 wants to enter its CS. It becomes the root of the virtual tree. While p_1 forwards the request to p_3, no node has p_3 as its *father*



(e) t_6: p_4 becomes the root of the virtual tree and the *next* of p_3



(f) t_7-t_8: p_3 releases its CS and clears its *next*



(g) Legend

Figure 6.22: Evolution of the virtual structures during the execution of Naimi-Tréhel algorithm

6.5.2 Description of Bouabdallah-Laforest algorithm

Each node keeps a local list of the *Resource Tokens* it is holding.

The *Control Token* maintains two lists :

- a list A holding all the free instances,
- a (key, value) set B telling which node has last requested an instance. The keys are instances of types of resources, the values are nodes.

Adaptation to the system

With the Bouabdallah-Laforest algorithm, because it relies on the virtual structures of the Naimi-Tréhel, a node can send a message to any node of the communication graph, i.e. topology. For instance when a node requests the *Control Token*. This requires each node to have a full knowledge of the communication graph, when the system allows a node to have a partial view of the graph. However, in a system with one instance of multiple types of resources, according to the description in Section 4.2.2, each node has a knowledge on how to reach each type of resources. In this case, it means that each node knows how to reach any node of the communication graph allowing to use the same experimental setup. Evaluating the algorithm on a system with multiple instances would require additional development.

Requesting the CS

When a node wants to enter its CS it first asks for the *Control Token* using the overlay structure of the Naimi-Tréhel algorithm. Once it holds the *Control Token*, it can either use the resources to enter its CS, if all the requested resources are free, or it can request the missing resources by sending *INQUIRE* messages to the last requesters of these resources, according to the B set. In return these nodes update their *next* to be the node that sent the *INQUIRE* and reply with a *INQ_ACK1* messages. If a node holds some resources when it receives the *INQUIRE* message then it send the Resource Token in the *INQ_ACK1*. When the requesting node has received *INQ_ACK1* messages from all the nodes it can unlock the *Control Token*.

Releasing the CS

When a node releases its CS it informs the last requesters of the resources that they are not used anymore using *INQ_ACK2* messages.

Pros and cons

The management of the *Control Token* is based on the Naimi-Tréhel Mutual Exclusion algorithm that has a low message complexity. However these messages can be sent to any node of the system because the virtual structures are independent from the topology. Therefore it is possible that a message has to go through many nodes before reaching the node it is looking for.

However the requirement to hold the *Control Token* creates a global lock on the resources which gives poor results in terms of *Average Usage Rate* and *Average Waiting Time*.

6.5.3 Example of execution in sample system

Initial state At t_0, the virtual tree of Naimi-Tréhel is initialised with n_1 as the root and n_1 holds the *Control Token*. Also, at t_0, all resources are free so the list A contains all the resources $\{n_1, n_2, n_3, n_4, n_5\}$ and set B is empty.

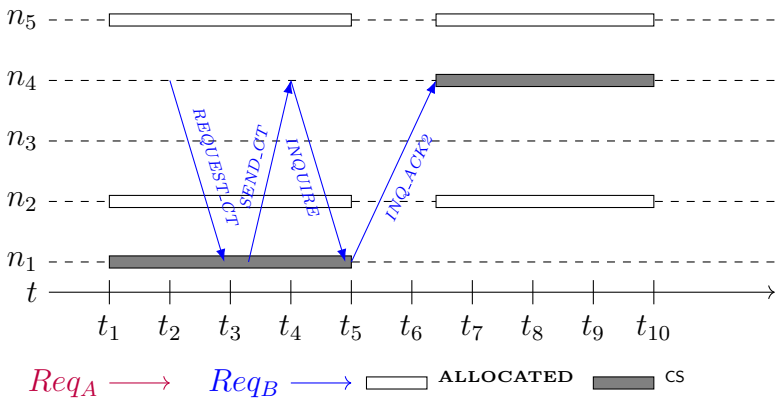
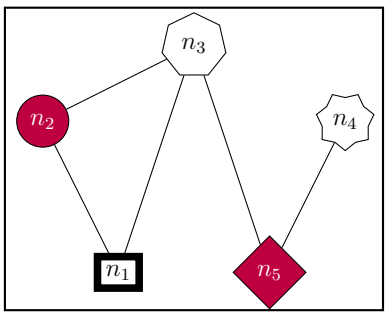


Figure 6.23: Timeline of sample execution of Bouabdallah-Laforest algorithm

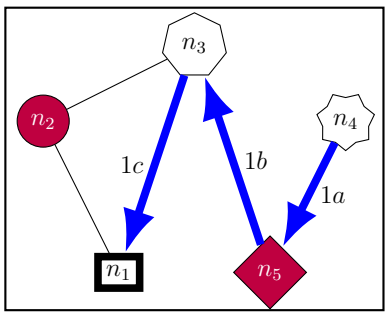
t	Holder	*list A*	*set B*
t_0	n_1	$\{n_1, n_2, n_3, n_4, n_5\}$	$\{\}$
t_1		$\{n_1, n_3, n_4\}$	$\{(n_1, \{n_2, n_5\})\}$
t_2			
t_3	n_4		
t_4			
t_5			$\{(n_4, \{n_2, n_5\})\}$
t_6			

Figure 6.24: Evolution of the Control Token

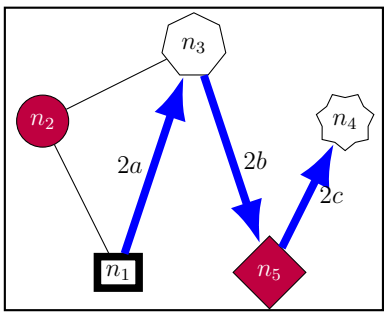
Execution Figures 6.23, 6.25 and 6.26 show an example of execution for the algorithm for the two sample requests Req_A and Req_B. Table 6.24 shows the evolution of the *Control Token* during the execution of the algorithm.



(a) t_1: Req_A is emitted. n_1 has the *Control Token*, it can enter its CS.



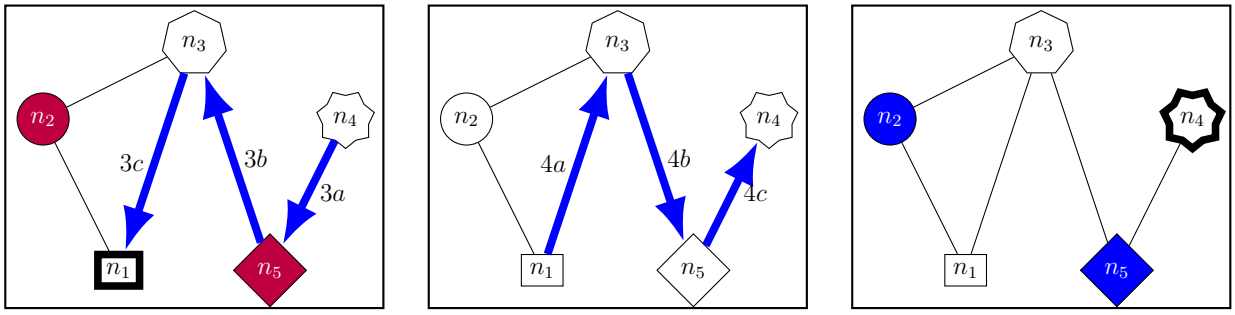
(b) t_2: Req_B is emitted. n_4 requests the *Control Token*.



(c) t_3: n_1 sends the *Control Token* to n_4

Figure 6.25: Sample execution of Bouabdallah-Laforest algorithm 1/2

When Req_A is emitted at t_1 (Figure 6.25a) n_1 updates the *Control Token* so that list A contains the used resources n_2 and n_5 and that set B reflects that they have been allocated for n_1. n_1 can enter its CS. When Req_B is emitted at t_2 (Figure 6.25b), n_4 requests the *Control Token* by sending a *REQUEST_CT* message to n_1 which is the root of the virtual tree.



(a) t_4: n_4 requests missing resources from n_1

(b) t_5: Req_A finished, n_1 sends tokens to n_4.

(c) t_6: n_4 enters CS, Req_B uses the resources.

Figure 6.26: Sample execution of Bouabdallah-Laforest algorithm 2/2

At t_3, in Figure 6.25c, n_4 receives the *Control Token* from n_1 in a *SEND_CT* message and becomes the root of the virtual tree. n_4 can now read in the *Control Token* that the resources it needs for Req_B are held by n_1 so it requests them with an *INQUIRE* message at t_4 (Figure 6.26a). When Req_A has finished using the resources at t_5, n_1 sends the tokens for the resources with a *INQ_ACK2* message to n_4, the node holding the *Control Token* according to the virtual tree (Figure 6.26b). n_4 then updates the *Control Token* accordingly to indicate that it holds the tokens for the resources (Table 6.24). Req_B can now enter its CS, at t_6 (Figure 6.26c).

Message Complexity

This algorithm has a low message complexity because Naimi-Tréhel has a logarithmic $log(N)$ complexity for the number of messages where N is the number of nodes. On top of the *REQUEST_CT* and *SEND_CT* messages needed by the Naimi-Tréhel algorithm to move the *Control Token*, at most three messages are required for each resource in a request: one *INQUIRE*, one *INQ_ACK1* and one *INQ_ACK2*. The algorithm requires between 0 and $3k$ messages, where k is the number of resources.

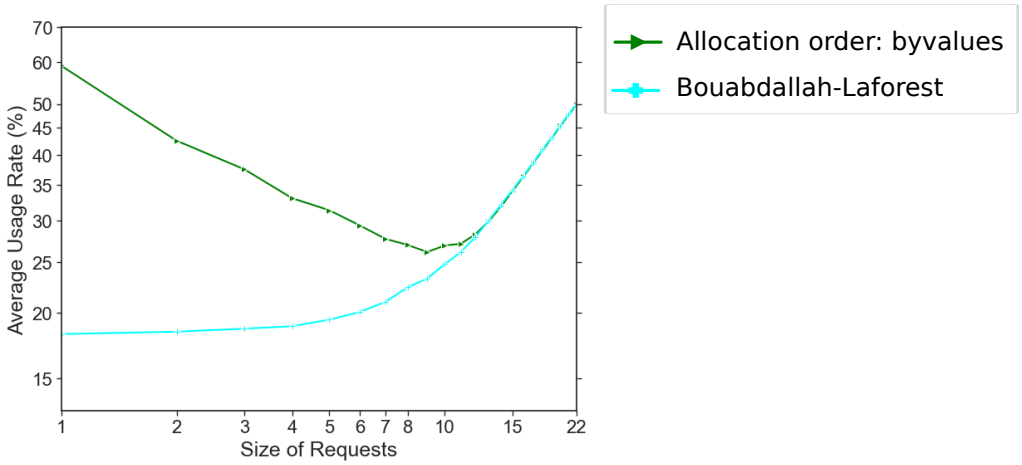
6.5.4 Performance evaluation

Figures 6.27 show, in blue, the results of the Bouabdallah-Laforest algorithm compared to the *byvalues* allocation order heuristic on simulation run on SimGrid.

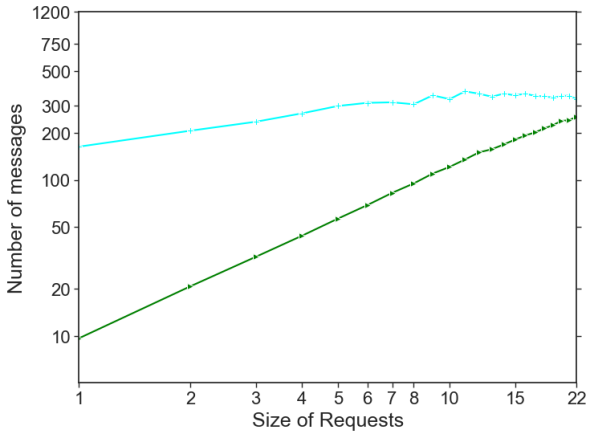
The algorithm relies on a *Control Token* to allow requests to allocate their resources. This *Control Token* introduces a global lock when resources are not available, which explain why the algorithm has worse performance in terms of *Average Usage Rate* (Figure 6.27a). The *byvalues* heuristic has a better concurrency, allowing non-conflicting requests to be allocated.

The Bouabdallah-Laforest algorithm also requires a larger number of messages in average (Figure 6.27b). Even if its message complexity is logarithmic, more than 80% of its messages are for the circulation of the *Control Token* because these messages can be

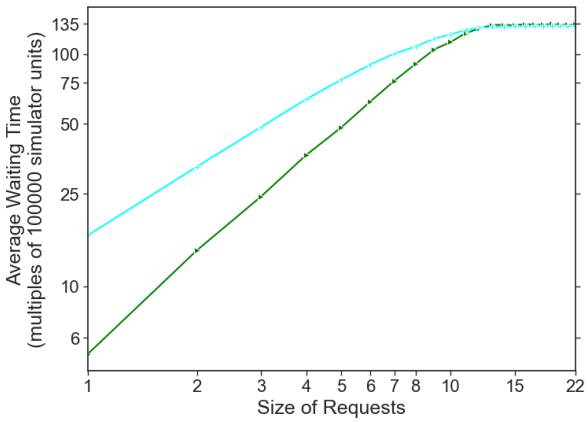
from any node to any node. The *byvalues* heuristic follows the path computed during the first subroutine and requires less messages in average. The message complexity is not sufficient to measure the difference of behaviour of the two algorithms, because it only considers the number of messages sent. This metrics shows the impact of the topology, a message might need to be forwarded by many nodes before reaching its destination.



(a) Average Usage Rate



(b) Average Number of Messages



(c) Average Waiting Time

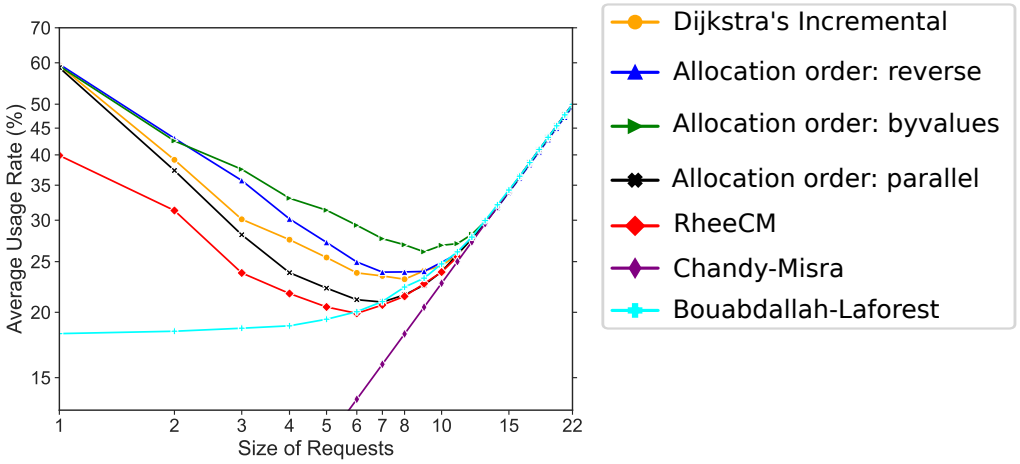
Figure 6.27: Evaluation of Bouabdallah-Laforest algorithm in a system with one instance of 44 types of resources

The *Average Waiting Time* metric is also higher with the Bouabdallah-Laforest algorithm (Figure 6.27c). Requests wait for a longer time to be allocated because the *Average Usage Rate* is lower.

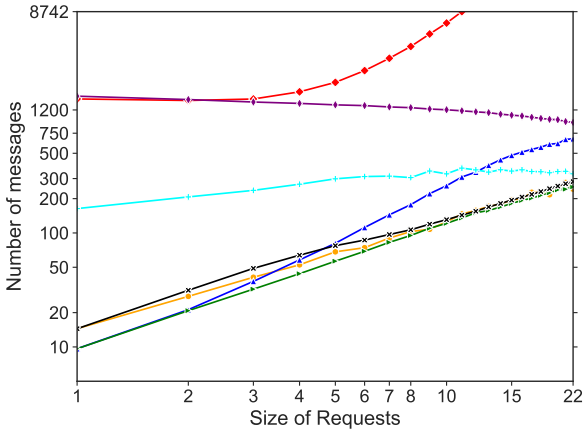
6.6 Summary

Few papers from the state of the art detailed in Section 3.2 include experimental evaluations of the algorithms. Page et al. [PJC93] compared the Average Response Times of their algorithms to Awerbuch-Saks[AS90] and Chandy-Misra[CM84]. Rhee [Rhe95] compared the Average Response Times and Average Message Complexity of his algorithm to Awerbuch-Saks[AS90], Chandy-Misra[CM84] and Choy-Singh [CS93]. Lejeune et al.[Lej+15] compared *Average Usage Rate* and Average Response Time of their algorithm

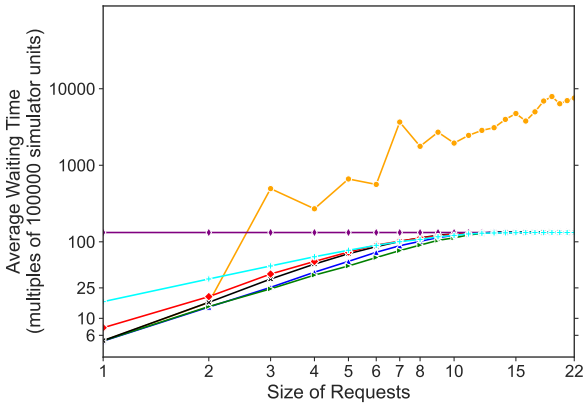
to Bouabdallah-Laforest [BL00] and an incremental algorithm using the Naimi-Tréhel algorithm for Mutual Exclusion.



(a) Average Usage Rate



(b) Average Number of Messages



(c) Average Waiting Time

Figure 6.28: Evaluation of all algorithms in a system with one instance of 44 types of resources

Figure 6.28 include all the results above in a single figure. It shows how all these results compare to each other. Sorting the algorithm by *Average Usage Rate* the best performance on all three metrics is obtained by the algorithm introduced in Chapter 4 using the *byvalues allocation order heuristic*. The *reverse allocation order heuristic* also gives better results than the other algorithms from the state of the art for the *Average Usage Rate*.

Figure 6.28c shows that all algorithms, except Dijkstra's incremental due to the domino effect, have the same *Average Waiting Time* once the *Average Usage Rate* becomes linear for requests of size 10 and more. The figure also shows that all the other algorithms compute a fair scheduling of the sequential requests. Since there are as many requests in the system as there are nodes (44), each request waits for the other 43 to end before it is allocated, as seen in the evaluation of Chandy-Misra algorithm in Section 6.3.3.

From the state of the art, Dijkstra's Incremental has the best *Average Usage Rate* but performs the worst by several orders of magnitude for the *Average Waiting Time*. Rhee's algorithm comes next but requires significantly more messages than the others.

Bouabdallah-Laforest is not efficient for requests of size 7 and less because of the global lock of the resource, it also requires more messages than most of the algorithms despite using the low complexity Naimi-Tréhel algorithm. Chandy-Misra can only allocate a single request at any given time and has the worst *Average Usage Rate* and also the second worst *Average Waiting Time*.

6.7 Conclusion

This chapter gives the results of experiments conducted with state-of-the-art algorithms for distributed resource allocation. They are compared to the performance of the algorithm introduced in Chapter 4. The experiments focus on systems with one instance of multiple types of resources as few algorithms address systems with multiple instances.

The results show that the proposed algorithm outperforms all the algorithms on the three metrics considered: *Average Usage Rate*, *Average Waiting Time* and *Average Number of Messages*. This shows that computing a total order with allocation vectors of counters as done by the LASS algorithm [Lej+15] removes the need for a global lock and allows for a better concurrency of requests. The drawback of the algorithm is that the number of messages depends on the number of preemptions. This information is usually not known in advance as it is dependant on the order of arrival of the messages which is typically difficult, if not impossible, to predict. This can make the algorithm non-deterministic on this metric. The *parallel* heuristic offers an intermediate solution because it does not require preemptions but with a lower performance on all metrics than the *byvalues* heuristic.

No results are shown in this chapter for systems with multiple instances. All the experiments uses the same first subroutine detailed in Chapter 4 to select the instances and compute a path so the output of this first subroutine is identical for all. A comparison between the two routing heuristics was done in the previous chapter and shows that it is the selection of the resource that changes the load-balancing and affects the performance. Bouabdallah-Laforest is the only algorithm from the state of the art and evaluated in this chapter that considered a generalisation for systems with multiple instances. It only considers that a requesting node can select one of the instances, it does not however address the method to select the instance.

Chapter 7

Conclusion

"There must be some way out of
here" Said the joker to the thief
"There's too much confusion,
I can't get no relief"

Bob Dylan, *All Along the Watchtower*

The problem addressed in this work belongs to the larger class of problems of the distributed allocation of resources. Here the focus is on requests for the allocation of chains of VNFs in network slices modelled as an allocation problem in systems with multiple instances of multiple types of resources. A new modular distributed algorithm is introduced to address this problem. Multiple heuristics are proposed and evaluated, each with their trade-offs between increasing the usage rate, reducing the waiting time and limiting the number of messages required. This algorithm is evaluated against four algorithms from the state of the art and against the *expected value*. This conclusion first lists the contributions of the work presented here. Then its limitations are listed, which give room for future work.

7.1 Contributions

The work presented here is at the intersection of two research domains: networks and distributed systems. These domains intersect on multiple research topics but there is limited interest for the class of algorithms under focus here. Networks often include a centralised manager, but as shown in Chapter 2 new multi-domain use cases have emerged recently with 5G making the case for a distributed management of network resources. Instead the focus is on the allocation of chains of VNFs for 5G slices which require the distributed allocation of ordered sets of resources, here the resources are VNFs. Three contributions are detailed:

A new distributed algorithm for the allocation of resources in networks. As shown in the state of the art in Chapter 3 the allocation of resources in distributed systems is mainly addressed as a variant of the Mutual Exclusion problem. However, few works address systems with both multiple types of resources and multiple instances and none addresses them specifically. The placement of the instances in the system and the order in which the resources are used are also not considered. The new algorithm presented

in Chapter 4 addresses these constraints. The instances are selected in a first subroutine to compute a path that satisfies the constraint on the order and contains instances of the requested types of resources while computing a vector of counters that allows the computation of a total order of the requests. Then, the second subroutine allocates these instances using a preemption mechanism to enforce the total order when multiple requests try to allocate an instance. Multiple heuristics are described for these two subroutines as well as their pros and cons. An extensive evaluation of the performance of multiple heuristics of the algorithm in Chapter 5 shows that the order followed to allocate the resources has a strong influence on the number of preemptions. The results show that reducing the number of preemptions improves the overall performance.

A numerical method to compute the *expected value* of the *Average Usage Rate* based on Markov chains. The method computes the *expected value* of the *Average Usage Rate* in the experimental settings where each node has always one request of constant size in the system. The performance of the algorithm is then compared with it. These ideal values could only be theoretically reached by an omniscient orchestrator informed instantly of the states of the nodes and the requests. However, the results show there is still room for improvement.

An experimental evaluation of the performance of the heuristics of the algorithm against algorithms from the state of the art The experimental comparison with algorithms from the state of the art in Chapter 6, in a simulator based on SimGrid, shows that the algorithm has better performance on the three considered metrics. A comparison using MPI on Grid'5000 is done with the algorithm of the state of the art with the best performance. It confirms that using allocation vectors of counters to compute the global order of requests gives better results than the other techniques. The results also show that the improvement does not come from the order itself but from the fact that the algorithm allows multiple requests to allocate resources at the same time if they are not trying to allocate the same instances. The trade-off of the algorithm is that the number of messages depends on the number of preemptions which can make it difficult to predict. An order based on the ordering of the nodes gives good result for the *Average Usage Rate* or the *Average Number of Messages* required but introduces a domino effect that makes the *Average Waiting Time* order of magnitude worse than for other methods. Other algorithms that require a global lock of the resources do not offer the same level of concurrency of the requests.

7.2 Limitations and future work

Further work is required to try to improve or evaluate in more details the performance of the algorithm. Adaptations to other systems or use cases can also be considered. Below is a list of possible future work.

7.2.1 Improving the performance

The *byvalues* heuristic offers a better performance than the algorithms of the state of the art that were evaluated, but it may be possible to improve the performance further.

Getting closer to an optimum A centralised solution with a global view of the system could be used to get closer to the *expected value Average Usage Rate* computed in Chapter 5. A solution that builds a global view of a distributed list of requests might improve the *Average Usage Rate* but the trade-off is the high number of messages that is orders of magnitude bigger than for the algorithm proposed. Both these methods might not be adapted to every system. A solution still has to be found to get closer to this maximum while keeping a low number of messages. A more thorough comparison with a global view-based solution could highlight the pros and cons of both methods. The *expected value* is not an optimum, depending on the actual set of requests on a given system it might be possible to achieve better or worse results. Comparing the performance of the algorithm with an optimum would allow the quantification of this difference. As the problem is NP-hard, it might be necessary to make this comparison on smaller topologies than the 44-node topology used in the experiments described here.

Identifying other heuristics more adapted to specific objectives Other heuristics can be considered for the algorithm. For the path computation subroutine, none of the heuristic proposed is trying to minimise the length of the path. A heuristic that looks for a compromise between load-balancing and the length of the path might be interesting for some use cases. For the allocation of resources, only totally random requests are evaluated while in a production systems the set of requests might not include all the possible permutations of resources, or some requests might be more frequent than others. In these situations the generic allocation order heuristics presented here might not be the most efficient and a heuristic tailored to the actual profile of requests would be more adequate. All the allocation order heuristics described in Section 4.5 are pessimistic: they expect deadlocks to happen. In production systems where requests are less random than on the experimental platform, there might be a lower risk of deadlocks. An optimistic approach would try to allocate the resources earlier and only handle deadlocks once they are detected.

7.2.2 Extending the system

The model proposed for the system is not adequate for every use cases and may need to be extended, two kinds of extension are proposed here.

Extending to dynamic topologies Among the assumptions made in the model used is that the resources are placed on the nodes of a static graph while the number and placement of resources of production platforms change over time: nodes can be added or removed, voluntarily when it is linked to capacity planning or involuntarily when this is because of failures. The algorithm cannot be used as is, adaptations are required when dealing with a dynamic graph. A fault tolerance mechanism could be added to deal with the loss of nodes.

Extending to heterogeneous systems and requests The simulations shown here take a lot of parameters. The influence of each parameter was studied in details. Still, for the sake of simplification some parameters are not studied specifically in the experiments. All the edges have the same weight of 1. Having different weights for the edges is not expected to change the overall results but a real system with multiple instances and with variations of the weight of the edges, i.e., the latency of the network links, might not

benefit from the algorithm if the length of paths has an impact on the service delivered by the resources. This needs to be considered when choosing the heuristic of the first subroutine to maximise the load-balancing among instances while minimising the length of paths. Also, even in systems with multiple instances, requests in the simulation are never for more than one instance for a given type of resource. This is not expected to have any influence on the results but has not been tested. Results show that the difference of performance between algorithms is linked to the load of the system.On systems with a low number of requests there might not be notable differences between algorithms.

7.2.3 Evaluating on a live 5G platform with actual workloads

Networks often are designed with a centralised manager and it does not make it simple to identify and implement a distributed use case other than on an experimental platform built on purpose like what was done during the course of this work. There are several steps forward left to make before we could be able to implement this algorithm on a multi-domain use case on a production-like platform. A target multi-domain architecture needs to be defined, for example a multi-domain 5G platform. The multi-domain use cases need to get more mature so research can focus on them. Then, the algorithm needs to be integrated in the multi-domain 5G platform for these use cases.

Bibliography

Bibliography Chapter 2

[3GP16] 3GPP. *Specification # 28.801 Telecommunication Management;Study on Management and Orchestration of Network Slicing for next Generation Network*. 2016. URL: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3091 (page 15).

[Ale17] Ana Cristina Aleixo. "Vertical Sector Requirements Analysis and Use Case Definition". In: (2017) (page 16).

[All16] NGMN Alliance. *NGMN 5G P1 Requirements & Architecture Work Stream End-to-End Architecture Description of Network Slicing Concept*. 2016. URL: https://www.ngmn.org/publications/technical-deliverables/page/3/ (page 15).

[Bar+15] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. "On Orchestrating Virtual Network Functions". In: *2015 11th International Conference on Network and Service Management (CNSM)*. 2015 11th International Conference on Network and Service Management (CNSM). 2015, pp. 50–56 (page 20).

[Ber+14] Pankaj Berde et al. "ONOS: Towards an Open, Distributed SDN OS". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6 (pages 21, 22).

[CDJ17] Francisco Carpio, Samia Dhahri, and Admela Jukan. "VNF Placement with Replication for Loac Balancing in NFV Networks". In: *2017 IEEE International Conference on Communications (ICC)*. 2017 IEEE International Conference on Communications (ICC). 2017, pp. 1–6 (page 20).

[CM84] K. M. Chandy and J. Misra. "The Drinking Philosophers Problem". In: *ACM Trans. Program. Lang. Syst.* 6.4 (1984), 632–646 (pages 7, 31, 34, 36, 38, 79, 87, 89, 91, 92, 94, 95, 103).

[Eur18] Eurecom. "Design and Prototyping of Integrated Multi-Domain SliceNet Architecture". In: (2018) (page 17).

[Fen+12] G. Feng, S. Garg, R. Buyya, and W. Li. "Revenue Maximization Using Adaptive Resource Provisioning in Cloud Computing Environments". In: *2012 ACM/IEEE 13th International Conference on Grid Computing*. 2012 ACM/IEEE 13th International Conference on Grid Computing. 2012, pp. 192–200 (page 20).

[Fra+18b] G. Fraysse et al. "Towards Multi-SDN Services: Dangers of Concurrent Resource Allocation from Multiple Providers". In: *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), short paper*. 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN). 2018, pp. 1–5 (pages 8, 20).

[Gre09] Kate Greene. *TR10: Software-Defined Networking*. 2009. URL: `http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/` (page 13).

[Gud+08] Natasha Gude et al. "NOX: Towards an Operating System for Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.3 (2008), 105–110 (page 21).

[HSS99] Mark Handley, Henning Schulzrinne, and Eve Schooler. *SIP: Session Initiation Protocol*. 1999. URL: `https://tools.ietf.org/html/rfc2543` (page 11).

[Inc12] MetroPCS Communications Inc. *MetroPCS Launches World's First Commercially Available Voice Over LTE Service and VoLTE-Capable 4G LTE Smartphone*. 2012. URL: `https://www.prnewswire.com/news-releases/metropcs-launches-worlds-first-commercially-available-voice-over-lte-service-and-volte-capable-4g-lte-smartphone-165336456.html` (page 11).

[ISG13] 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` (pages 13, 19).

[ISG14] ETSI NFV ISG. *ETSI GS NFV-MAN 001 V1.1.1 Network Functions Virtualisation (NFV); Management and Orchestration*. 2014 (pages 11, 19).

[ITU96] ITU-T. *H.323 : Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service*. 1996. URL: `https://www.itu.int/rec/T-REC-H.323-199611-S/en/` (page 11).

[Jai+13] Sushant Jain et al. "B4: Experience with a Globally-Deployed Software Defined Wan". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 3–14 (page 14).

[JS15] Brendan Jennings and Rolf Stadler. "Resource Management in Clouds: Survey and Research Challenges". In: *J. Netw. Syst. Manage.* 23.3 (2015), 567–619 (page 20).

[Jia+16] Yongzheng Jia et al. "Online Scaling of NFV Service Chains across Geo-Distributed Datacenters". In: (2016). arXiv: `1611.08086 [cs]` (page 20).

[Kop+10] Teemu Koponen et al. "Onix: A Distributed Control Platform for Large-Scale Production Networks". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 351–364 (pages 14, 21).

[Kre+15] D. Kreutz et al. “Software-Defined Networking: A Comprehensive Survey”. In: *Proceedings of the IEEE* 103.1 (2015), 14–76 (pages 13, 14).

[McK+08] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), 69–74 (page 13).

[Mij+16] R. Mijumbi et al. “Management and Orchestration Challenges in Network Functions Virtualization”. In: *IEEE Communications Magazine* 54.1 (2016), 98–105 (page 20).

[MFD11] K. Mills, J. Filliben, and C. Dabrowski. “Comparing VM-Placement Algorithms for On-Demand Clouds”. In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. 2011 IEEE Third International Conference on Cloud Computing Technology and Science. 2011, pp. 91–98 (page 20).

[Ope16] OpenDaylight Project, director. *Evolution of SDN in Google’s Network Infrastructure-Vijoy Pandey*. 2016 (page 13).

[Pos81] J. Postel. *Internet Protocol*. 1981. URL: `https://tools.ietf.org/html/rfc791` (page 10).

[PPP] 5G PPP. *5G PPP*. URL: `https://5g-ppp.eu/` (page 16).

[Sun+16] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. “Robotron: Top-down Network Management at Facebook Scale”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. New York, NY, USA: ACM, 2016, pp. 426–439 (page 16).

[Wic+15] J. A. Wickboldt et al. “Software-Defined Networking: Management Requirements and Challenges”. In: *IEEE Communications Magazine* 53.1 (2015), 278–285 (page 14).

[Wid+12] A. Widjajarto, S. H. Supangkat, Y. S. Gondokaryono, and A. S. Prihatmanto. “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 International Conference on Cloud Computing and Social Networking (ICCCSN). 2012, pp. 1–4 (page 20).

Bibliography Chapter 3: Shared-Memory Mutex algorithms

[dBru67] N. G. de Bruijn. "Additional Comments on a Problem in Concurrent Programming Control". In: *Commun. ACM* 10.3 (1967), 137–138 (pages 28, 36).

[Dij65] E. W. Dijkstra. "Solution of a Problem in Concurrent Programming Control". In: *Commun. ACM* 8.9 (1965), 569– (pages 7, 25, 26, 28, 36, 37).

[EM72] Murray A. Eisenberg and Michael R. McGuire. "Further Comments on Dijkstra's Concurrent Programming Control Problem". In: *Communications of the ACM* 15.11 (1972), 999 (pages 28, 36, 37).

[Knu66] Donald E. Knuth. "Additional Comments on a Problem in Concurrent Programming Control". In: *Communications of the ACM* 9.5 (1966), 321–322 (pages 28, 36).

Bibliography Chapter 3: Token-Based Mutex algorithms

[Gos89] A. Goscinski. "A Synchronization Algorithm for Processes with Dynamic Priorities in Computer Networks with Node Failures". In: *Information Processing Letters* 32.3 (1989), 129–136 (pages 30, 36).

[Gos91] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design*. Vol. 21. Addison-Wesley Sydney, 1991 (pages 30, 36).

[Gos90] Andrzej Goscinski. "Two Algorithms for Mutual Exclusion in Real-Time Distributed Computer Systems". In: *Journal of Parallel and Distributed Computing* 9.1 (1990), 77–82 (pages 30, 36).

[HPR88] J. M. Helary, N. Plouzeau, and M. Raynal. "A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network". In: *The Computer Journal* 31.4 (1988), 289–295 (pages 30, 36).

[Le 77] Gerard Le Lann. "Distributed Systems-towards a Formal Approach." In: *IFIP Congress*. Vol. 7. Toronto. 1977, pp. 155–160 (pages 25, 29, 36, 37).

[Mar85] Alain J. Martin. "Distributed Mutual Exclusion on a Ring of Processes". In: *Science of Computer Programming* 5 (1985), 265–276 (pages 29, 36).

[NT87] Mohamed Naimi and Michel Tréhel. "An Improvement of the logN Distributed Algorithm for Mutual Exclusion". In: *Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, September 1987*. IEEE Computer Society, 1987, pp. 371–377 (pages 30, 32, 36, 37, 97).

[Ray89b] Kerry Raymond. "A Tree-Based Algorithm for Distributed Mutual Exclusion". In: *ACM Trans. Comput. Syst.* 7.1 (1989), 61–77 (pages 30, 36, 37).

[RA83] Glenn Ricart and Ashok K Agrawala. "Authors' Response to "on Mutual Exclusion in Computer Networks" by Carvalho and Roucairol". In: *Communications of the ACM* 26.2 (1983), 147–148 (pages 29, 36, 37).

[SK85] Ichiro Suzuki and Tadao Kasami. "A Distributed Mutual Exclusion Algorithm". In: *ACM Trans. Comput. Syst.* 3.4 (1985), 344–349 (pages 29, 30, 36, 37).

Bibliography Chapter 3: Permission-Based Mutex algorithms

[Cad+17] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. "A Coded Shared Atomic Memory Algorithm for Message Passing Architectures". In: *Distributed Computing* 30.1 (2017), 49–73 (pages 29, 36).

[Cha94] Ye-In Chang. "Design of Mutual Exclusion Algorithms for Real-Time Distributed Systems". In: *Journal of Information Science and Engineering, Vol. 10* (1994), 527–548 (pages 30, 36).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565 (pages 25, 29, 30, 36, 37, 45).

[LT87] Nancy A Lynch and Mark R Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. 1987, pp. 137–151 (page 36).

[MS90] Shivakant Mishra and Pradip K Srimani. "Fault-Tolerant Mutual Exclusion Algorithms". In: *Journal of Systems and Software* 11.2 (1990), 111–129 (pages 30, 36).

[Mue98] F. Mueller. "Prioritized Token-Based Mutual Exclusion for Distributed Systems". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing. 1998, pp. 791–795 (pages 30, 36).

[Mue99] Frank Mueller. *Priority Inheritance and Ceilings for Distributed Mutual Exclusion - IEEE Conference Publication*. 1999. URL: `https://ieeexplore.ieee.org/document/818861/` (pages 30, 36).

[NM91] Mitchell L Neilsen and Masaaki Mizuno. "A DAG-Based Algorithm for Distributed Mutual Exclusion." In: *ICDCS*. 1991, pp. 354–360 (pages 30, 36).

[NLM90] Shojiro Nishio, Kin F. Li, and Eric G. Manning. "A Resilient Mutual Exclusion Algorithm for Computer Networks". In: *IEEE Transactions on Parallel & Distributed Systems* 3 (1990), 344–356 (pages 30, 36).

[RA81] Glenn Ricart and Ashok K. Agrawala. "An Optimal Algorithm for Mutual Exclusion in Computer Networks". In: *Commun. ACM* 24.1 (1981), 9–17 (pages 29, 30, 36, 37).

[TN87] Michel Tréhel and Mohammed Naimi. “A Distributed Algorithm for Mutual Exclusion Based on Data Structures and Fault Tolerance”. In: *Proc. IEEE Phoenix Conf. on Computer and Communications*. 1987, pp. 35–39 (pages 30, 33, 36).

[vdSne87] Jan L. A. van de Snepscheut. “Fair Mutual Exclusion on a Graph of Processes”. In: *Distributed Computing* 2.2 (1987), 113–115 (pages 30, 36, 37).

Bibliography Chapter 3: Quorum-Based Mutex algorithms

[AE91] Divyakant Agrawal and Amr El Abbadi. "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion". In: *ACM Transactions on Computer Systems* 9.1 (1991), 1–20 (pages 29, 30, 36, 37).

[Mae85] Mamoru Maekawa. "A N Algorithm for Mutual Exclusion in Decentralized Systems". In: *ACM Trans. Comput. Syst.* 3.2 (1985), 145–159 (pages 29, 36, 37).

[MNR91] Masaaki Mizuno, Mitchell L Neilsen, and Raghavendra Rao. "A Token Based Distributed Mutual Exclusion Algorithm Based on Quorum Agreements." In: *ICDCS*. 1991, pp. 361–368 (pages 29, 36).

[San87] Beverly A. Sanders. "The Information Structure of Distributed Mutual Exclusion Algorithms". In: *ACM Trans. Comput. Syst.* 5.3 (1987), 284–299 (pages 29, 36, 37).

Bibliography Chapter 3: k-mutex algorithms

[Bal94] Roberto Baldoni. "An O(NM(M+1/)) Distributed Algorithm for the k-out of-M Resources Allocation Problem". In: *14th International Conference on Distributed Computing Systems*. 14th International Conference on Distributed Computing Systems. 1994, pp. 81–88 (pages 33, 36).

[BV95] S. Bulgannawar and N. H. Vaidya. "A Distributed K-Mutual Exclusion Algorithm". In: *Proceedings of 15th International Conference on Distributed Computing Systems*. Proceedings of 15th International Conference on Distributed Computing Systems. 1995, pp. 153–160 (pages 33, 36).

[CE08] Pranay Chaudhuri and Thomas Edward. "An Algorithm for K-Mutual Exclusion in Decentralized Systems". In: *Computer Communications* 31.14 (2008), 3223–3235 (pages 33, 36).

[Kak+94] Hirotsugu Kakugawa, Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. "A Distributed K-Mutual Exclusion Algorithm Using k-Coterie". In: *Information Processing Letters* 49.4 (1994), 213–218 (pages 33, 36).

[Mak+92] K. Makki et al. "A Token Based Distributed k Mutual Exclusion Algorithm". In: *[1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. [1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing. 1992, pp. 408–411 (pages 33, 36).

[MT99] Y. Manabe and N. Tajima. "(H, k)-Arbiters for h-out of-k Mutual Exclusion Problem". In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003). 1999, pp. 216–223 (pages 33, 36).

[Ray89a] Kerry Raymond. "A Distributed Algorithm for Multiple Entries to a Critical Section". In: *Information Processing Letters* 30.4 (1989), 189–193 (pages 33, 36).

[Ray91a] 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 (pages 33, 36).

[SR92] Pradip K. Srimani and Rachamallu L. N. Reddy. "Another Distributed Algorithm for Multiple Entries to a Critical Section". In: *Information Processing Letters* 41.1 (1992), 51–57 (pages 33, 36).

Bibliography Chapter 3: Dining/Drinking philosophers algorithms

[AS90] B. Awerbuch and M. Saks. "A Dining Philosophers Algorithm with Polynomial Response Time". In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science. 1990, 65–74 vol.1 (pages 32, 36–38, 95, 103).

[BP92] Judit Bar-Ilan and David Peleg. "Distributed Resource Allocation Algorithms (Extended Abstract)". In: *Proceedings of the 6th International Workshop on Distributed Algorithms*. WDAG '92. Springer. London, UK, UK: Springer-Verlag, 1992, pp. 277–291 (pages 32, 36, 38).

[BL00] 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 (pages 7, 32, 34, 36, 38, 79, 96, 104).

[CM84] K. M. Chandy and J. Misra. "The Drinking Philosophers Problem". In: *ACM Trans. Program. Lang. Syst.* 6.4 (1984), 632–646 (pages 7, 31, 34, 36, 38, 79, 87, 89, 91, 92, 94, 95, 103).

[CHA90] Y. CHANG. "An Improved O(Log N) Mutual Exclusion Algorithm for Distributed Systems". In: *Proc. the 10th Int. Conf. on Distributed Computing Systems* (1990), 295–302 (page 96).

[CS95] Manhoi Choy and Ambuj K. Singh. "Efficient Fault-Tolerant Algorithms for Distributed Resource Allocation". In: *ACM Trans. Program. Lang. Syst.* 17.3 (1995), 535–559 (pages 32, 33, 36, 37).

[Dij71] E. W. Dijkstra. "Hierarchical Ordering of Sequential Processes". In: *Acta Informatica* 1.2 (1971), 115–138 (pages 7, 30, 31, 36, 38, 79–81).

[FR80] Nissim Francez and Michael Rodeh. "A Distributed Abstract Data Type Implemented by a Probabilistic Communication Scheme". In: *21st Annual Symposium on Foundations of Computer Science (Sfcs 1980)*. 21st Annual Symposium on Foundations of Computer Science (Sfcs 1980). 1980, pp. 373–379 (pages 31, 36).

[GSA89] David Ginat, A. Udaya Shankar, and A. K. Agrawala. "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 (pages 31, 34, 36, 38).

[Lam74] Leslie Lamport. “A New Solution of Dijkstra’s Concurrent Programming Problem”. In: *Communications of the ACM* 17.8 (1974), 453–455 (pages 28, 32, 36, 37).

[LR81] Daniel Lehmann and Michael O. Rabin. “On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’81. New York, NY, USA: ACM, 1981, pp. 133–138 (pages 31, 36).

[Lej+15] J. Lejeune, L. Arantes, J. Sopena, and P. Sens. “Reducing Synchronization Cost in Distributed Multi-Resource Allocation Problem”. In: *2015 44th International Conference on Parallel Processing*. 2015 44th International Conference on Parallel Processing. 2015, pp. 540–549 (pages 33, 36, 44, 79, 103, 105).

[Lyn80] Nancy A. Lynch. “Fast Allocation of Nearby Resources in a Distributed System”. In: *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*. STOC ’80. New York, NY, USA: ACM, 1980, pp. 70–81 (pages 31, 36, 37, 82).

[Mad97] 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 (pages 32, 34, 36).

[PJC93] Ivor Page, Tom Jacob, and Eric Chern. “Fast Algorithms for Distributed Resource Allocation”. In: *IEEE Transactions on Parallel and Distributed Systems* 4.2 (1993), 188–197 (pages 31, 36, 38, 103).

[Rhe95] Injong Rhee. “A Fast Distributed Modular Algorithm for Resource Allocation”. In: *Proceedings of 15th International Conference on Distributed Computing Systems*. Proceedings of 15th International Conference on Distributed Computing Systems. 1995, pp. 161–168 (pages 7, 32, 34, 36–38, 79, 87, 90, 95, 103).

[Rhe98] Injong Rhee. “A Modular Algorithm for Resource Allocation”. In: *Distributed Computing* 11.3 (1998), 157–168 (pages 32, 36).

[Sin89] M. Singhal. “A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems”. In: *IEEE Transactions on Computers* 38.5 (1989), 651–662 (pages 30, 36).

[SPS00] Paolo AG Sivilotti, Scott M Pike, and Nigamanth Sridhar. “A New Distributed Resource-Allocation Algorithm with Optimal Failure Locality”. In: *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems*. Vol. 2. 2000, pp. 524–529 (pages 32, 36).

[SP88] Eugene Styer and Gary L. Peterson. “Improved Algorithms for Distributed Resource Allocation”. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC ’88. New York, NY, USA: ACM, 1988, pp. 105–116 (pages 31, 36, 38).

[WPP91] E. B. Weidman, I. P. Page, and W. J. Pervin. “Explicit Dynamic Exclusion Algorithm”. In: *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing. 1991, pp. 142–149 (pages 32, 36, 38).

[WL93] Jennifer L. Welch and Nancy A. Lynch. “A Modular Drinking Philosophers Algorithm”. In: *Distributed Computing* 6.4 (1993), 233–244 (pages 31, 32, 36).

Bibliography Chapter 3: Other references

[BAI94] BR Badrinath, Arup Acharya, and Tomasz Imielinski. "Structuring Distributed Algorithms for Mobile Hosts". In: *14th International Conference on Distributed Computing Systems*. IEEE. 1994, pp. 21–28 (page 25).

[BVP02] R. Baldoni, A. Virgillito, and R. Petrassi. "A Distributed Mutual Exclusion Algorithm for Mobile Ad-Hoc Networks". In: *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*. Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications. 2002, pp. 539–544 (page 26).

[Ben+04] Mahfoud Benchaba, Abdelmadjid Bouabdallah, Nadjib Badache, and Mohamed Ahmed-Nacer. "Distributed Mutual Exclusion Algorithms in Mobile Ad Hoc Networks: An Overview". In: *ACM SIGOPS Operating Systems Review* 38.1 (2004), 74–89 (page 26).

[BHJ02] Azzedine Boukerche, Sungbum Hong, and Tom Jacob. "A Distributed Algorithm for Dynamic Channel Allocation". In: *Mobile Networks and Applications* 7.2 (2002), 115–126 (page 25).

[CR83] Osvaldo Carvalho and Gerard Roucairol. "On Mutual Exclusion in Computer Networks". In: *Communications of the ACM* 26 (1983), 146–147 (pages 29, 36, 37).

[CE06] Pranay Chaudhuri and Thomas Edward. "An o (vn) Distributed Mutual Exclusion Algorithm Using Queue Migration." In: *J. UCS* 12.2 (2006), 140–159 (page 33).

[CS93] Manhoi Choy and Ambuj K. Singh. "Distributed Job Scheduling Using Snapshots". In: *Distributed Algorithms*. International Workshop on Distributed Algorithms. Springer, Berlin, Heidelberg, 1993, pp. 145–159 (pages 36, 38, 91, 92, 94, 95, 103).

[CS92] Manhoi Choy and Ambuj K. Singh. "Efficient Fault Tolerant Algorithms for Resource Allocation in Distributed Systems". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing* (Victoria, British Columbia, Canada). STOC '92. New York, NY, USA: ACM, 1992, pp. 593–602 (page 38).

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), 269–271 (page 65).

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Communications of the ACM* 21.8 (1978), 666–677 (pages 29, 79).

[IEE04] IEEE. *IEEE 802.5 Web Site*. 2004. URL: http://www.ieee802.org/5/www8025org/ (page 29).

[Lam19] Leslie Lamport. *My Writings*. 2019 (page 28).

[Lej15] Jonathan Lejeune. "Algorithmique distribuéee d'exclusion mutuelle : vers une gestion efficace des ressources". Université Pierre et Marie Curie, 2015 (page 35).

[Lyn96] Nancy A Lynch. *Distributed Algorithms*. Elsevier, 1996 (page 27).

[Lyn81] Nancy A. Lynch. "Upper Bounds for Static Resource Allocation in a Distributed System". In: *Journal of Computer and System Sciences* 23.2 (1981), 254–278 (pages 31, 36, 38, 82).

[LF81] Nancy A. Lynch and Michael J. Fischer. "On Describing the Behavior and Implementation of Distributed Systems". In: *Theoretical Computer Science*. Special Issue Semantics of Concurrent Computation 13.1 (1981), 17–43 (page 26).

[Ran75] Brian Randell. "System Structure for Software Fault Tolerance". In: *Ieee transactions on software engineering* 2 (1975), 220–232 (pages 31, 82).

[Ray91b] Michel Raynal. "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms". In: *ACM SIGOPS Operating Systems Review* 25.2 (1991), 47–50 (page 28).

[Ray89c] Michel Raynal. "Prime Numbers as a Tool to Design Distributed Algorithms". In: *Information processing letters* 33.1 (1989), 53–58 (page 37).

[RWX04] Injong Rhee, Ajit Warrier, and Lisong Xu. *Randomized Dining Philosophers to TDMA Scheduling in Wireless Sensor Networks*. Citeseer, 2004 (page 25).

[Sin93] M. Singhal. "A Taxonomy of Distributed Mutual Exclusion". In: *Journal of Parallel and Distributed Computing* 18.1 (1993), 94–101 (pages 28, 29, 35).

[Sin92] Mukesh Singhal. *A Dynamic Information-Structure Mutual Exclusion Algorithm for Distributed Systems*. 1992. URL: https://www.computer.org/csdl/journal/td/1992/01/l0121/13rRUwI5TQs (page 37).

[Sop+05] Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. "A Fault-Tolerant Token-Based Mutual Exclusion Algorithm Using a Dynamic Tree". In: *Euro-Par 2005 Parallel Processing*. Ed. by José C. Cunha and Pedro D. Medeiros. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 654–663 (pages 30, 36).

[Vel93] Martin G Velazquez. *A Survey of Distributed Mutual Exclusion Algorithms*. Colorado State Univ., 1993 (pages 28, 37).

[WCM01] J Walter, Guangtong Cao, and Mitrabhanu Mohanty. "A K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks". In: *Proceedings of the First Annual Workshop on Principles of Mobile Computing*. Citeseer. 2001 (page 26).

[WK97] JE Walter and Savita Kini. "Mutual Exclusion on Multihop, Mobile Wireless Networks". In: *Texas A&M Univ., College Station, TX 77843-3112, TR97* 14 (1997) (page 26).

[WWV01] Jennifer E Walter, Jennifer L Welch, and Nitin H Vaidya. "A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks". In: *Wireless networks* 7.6 (2001), 585–600 (page 26).

[WW11] J. Welch and J. Walter. *Link Reversal Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011 (page 26).

Bibliography Chapter 4

[Fra+20] Guillaume Fraysse, Jonathan Lejeune, Julien Sopena, and Pierre Sens. "A Resource Usage Efficient Distributed Allocation Algorithm for 5G Service Function Chains". In: *Distributed Applications and Interoperable Systems*. Ed. by Anne Remke and Valerio Schiavoni. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 169–185 (pages 8, 40).

[Lam74] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". In: *Communications of the ACM* 17.8 (1974), 453–455 (pages 28, 32, 36, 37).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565 (pages 25, 29, 30, 36, 37, 45).

Bibliography Chapter 5

[Bal+13] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20 (pages 66, 86).

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), 269–271 (page 65).

[Gab+04] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmuller, Péter Kacsuk, and Jack Dongarra. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 97–104 (pages 7, 65, 85).

[Kar14] Samuel Karlin. *A First Course in Stochastic Processes*. Academic press, 2014 (page 72).

Bibliography Chapter 7

[Dijte] Edsger W. Dijkstra. "Over Seinpalen". N.d. (page 81).

[Gab+04] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmuller, Péter Kacsuk, and Jack Dongarra. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 97–104 (pages 7, 65, 85).

[Kar14] Samuel Karlin. *A First Course in Stochastic Processes*. Academic press, 2014 (page 72).

[Kni+11] S. Knight et al. "The Internet Topology Zoo". In: *IEEE Journal on Selected Areas in Communications* 29.9 (2011), 1765–1775 (page 65).

[Mae85] Mamoru Maekawa. "A N Algorithm for Mutual Exclusion in Decentralized Systems". In: *ACM Trans. Comput. Syst.* 3.2 (1985), 145–159 (pages 29, 36, 37).

Bibliography, others

[Cas+14] 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 (pages 1, 5, 7, 65).

List of Figures

List of Tables

Now I guess I'll have to tell 'em
That I got no cerebellum
Gonna get my Ph.D.
I'm a teenage lobotomy

Ramones, *Teenage Lobotomy*