computers

MDPI

Article

Addressing Bandwidth-Driven Flow Allocation in RINA†

Michal Koutenský [1,*], Vladimír Veselý [1] and Vincenzo Maffione [2]

[1] Department of Information Systems, Brno University of Technology, 61200 Brno, Czech Republic; koutenmi | veselyv@fit.vutbr.cz

[2] Dipartimento di Ingegneria dell'Informazione, Università di Pisa, 56122 Pisa, Italy; vincenzo.maffione@ing.unipi.it

* Correspondence: koutenmi@fit.vutbr.cz; Tel.: +420-541-14-1100

† This paper is an extended version of the conference paper: Koutenský, M.; Veselý, V., Maffione, V. Bandwidth-driven Flow Allocation Policy for RINA. In Proceedings of the 2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), Paris, France, 24–27 February 2020; IEEE Communications Society: New York, NY, USA, 2020; s. 51–56. ISBN: 978-1-7281-5127-4.

Received: 31 May 2020; Accepted: 6 August 2020; Published: 10 August 2020

Abstract: Effective capacity allocation is essential for a network to operate properly, providing predictable quality of service guarantees and avoiding bottlenecks. Achieving capacity allocation fairness is a long-standing problem extensively researched in the frame of transport and network layer protocols such as TCP/IP. The Recursive InterNetwork Architecture offers programmable policies that enable more flexible control on the mechanics of network flow allocation. In this paper, we present our version of one of these policies, which provides flow allocation according to the bandwidth requirements of requesting applications. We implement the bandwidth-aware flow allocation policy by extending rlite, an open source RINA implementation. Our evaluation shows how the policy can prevent links from becoming oversaturated and use alternate paths to achieve high total link data-rate use.

Keywords: RINA; TCP; congestion control; flow allocation; rlite

1. Introduction

Networks should be able to handle and accommodate different types of traffic, with diverse needs and requirements. This is especially true for the Internet, being central to most of our social interactions. With growing diversity of applications (e.g., desktop vs. mobile vs. IoT) and deployment scenarios (e.g., LAN vs. WAN vs. data-center), providing proper Quality of Service (QoS) becomes an essential goal.

Applications require different handling of traffic (e.g., reliability of delivery, packet dropping eligibility, minimal bandwidth, allowed maximal delay and jitter) depending on their purpose. Such handling poses often challenge for (non-)cooperating layers of the TCP/IP protocol suite, which do not provide a comprehensive solution for supporting QoS. For instance, following things need to be aligned across the TCP/IP stack to provide such QoS guarantees: (a) prioritization in ingress/egress interface buffers on physical layer; (b) PDUs recognition and marking on data-link and internet layers; (c) QoS signalization and traffic throttling on transport layer; and (d) APIs for integration with programs on application layer. QoS support is something that was added to best-effort IP/Ethernet and reliability-concerning TCP later (after they have been already standardized and codified in RFCs). Although the first congestion collapse happened in 1986, it took nearly the whole 1990s decade to address this issue and introduce QoS support in lower layers of TCP/IP. The results of this development are Integrated Services (IntServ) and Differentiated Services (DiffServ). IntServ

mandate precise control of every flow, which is subject to QoS. IntServ offer accurate capacity allocation planning and predictive traffic behavior, but they have problems with scalability. DiffServ focus not on flows, but traffic classes which group flows of similar QoS characteristics. On the one hand, DiffServ scale nicely; on the other hand, DiffServ allow only stochastic capacity allocation.

The problem of network capacity allocation can be approached in two ways [1]. First, in a proactive manner — such as conditional allocation — which attempts to avoid critical resource depletion and congestion. Second, with a reactive approach, where congestion is dealt with when it occurs, is used by many current transport layer protocols (e.g., TCP, SCTP). Nevertheless, these approaches are not mutually exclusive. Although capacity allocation is long-term, dealing with the congestion is a short-term task taking into account the flow lifecycle.

In this paper, we aim at investigating the possibilities of a proactive approach to flow allocation within the Recursive InterNetwork Architecture (RINA) [2,3]. A RINA layer, called Distributed Interprocess Facility (DIF), and the participating processes providing IPC (IPCPs), contain all the necessary information to decide whether allocating resources for a pending flow request could congest the network. Not only do we want to prevent congestion from occurring; we want to guarantee available bandwidth to allocated flows during their whole lifetime. This is achieved using bandwidth reservations — bandwidth that has been reserved cannot be used by another flow, even if it is not being used at the moment.

There are several RINA implementations available: IRATI [4] and rlite [5], two full-stack implementations targeting Linux-based operating systems; Ouroboros [6], a RINA-inspired implementation; ProtoRINA [7], a Java-based academic prototype; and RINASim [8], a simulation framework for OMNeT++. After designing our distributed Flow Allocator policy, we have decided to implement and to evaluate it experimentally in the frame of the rlite stack.

This paper contributes the design of a proactive flow allocation scheme based on bandwidth (or better would be "data-rate" in order to avoid misconception with terminology from physical layer) reservation. The flow allocation scheme has been implemented in rlite as a policy for the Flow Allocator. We show how this policy achieves the proposed goals of preventing congestion and guaranteeing bandwidth for flows as a form of QoS.

The main point of comparison is the existing Flow Allocator policy in rlite (called `local`). The `local` Flow Allocator does not support any QoS guarantees. All flow requests — assuming they are accepted by the destination application — succeed. Unlike our policy, it does not take into account the network state, does not employ admission control, and the accompanying routing policy does not attempt to use alternative network paths to better use available resources.

Our flow allocation scheme contributes a more complex solution that aims to provide such additional features. The goal is to experiment with an alternative approach to network resource management in an attempt to demonstrate the benefits of RINA with regards to the service provided by the network. By leveraging several of the features unique to RINA — such as being a unified, comprehensive architecture and the clear separation of mechanisms and policies — we have been able to design and implement a flow allocation scheme based on admission control providing bandwidth reservation for flows.

Section 2 contains a summary of some of the congestion control and resource allocation theory. Section 3 introduces basic RINA concepts for readers not familiar with the architecture. Section 4 provides an in-depth look at the design and implementation of our policy, with some background information on rlite. Section 5 discusses the experiments done and the results thereof. Section 6 sets the results in a broader context, while Section 7 maps out the directions for possible future work. Section 8 contains information about the technical details regarding the experiments done and how to reproduce them.

2. Background

Congestion control is one of the core problems that needs to be tackled when ensuring Quality of Service, as it has a negative impact on almost all aspects of network performance. Network congestion decreases the throughput of an application, increases latency and the rate at which packets are dropped; in the most severe cases, the service can become completely unusable.

Generally, mechanisms trying to avoid congestion can be categorized as either *open-loop* or *closed-loop*, or *proactive* and *reactive* respectively [1,9]. Proactive mechanisms apply *a priori* knowledge about the network to control the traffic; reactive mechanisms use feedback from the network to adjust to the current state.

The dominant paradigm in TCP/IP is reactive congestion avoidance in its various forms. The pioneering work of Van Jacobson regarding congestion control [10] has resulted in various versions of the TCP congestion avoidance mechanism, such as TCP Tahoe [11], CUBIC [12], BIC [13] and BBR [14]. These all work by interpreting some feedback from the network at the sender (usually packet loss rate) as a sign on congestion, and adjusting the sending rate accordingly. Supplementary mechanisms, such as Random Early Detection [15], use information available to gateways in the network (e.g., queue size) and use either explicit (notifications) or implicit (dropping packets) methods to signal congestion to the flow endpoints.

One issue with reactive congestion control mechanisms implemented as part of transport protocols is their behavior with each other, referred to as fairness. Some transport protocols, such as UDP, do not implement any form of congestion control. Even if two protocols, or two versions of the same protocol, implement congestion avoidance algorithms, they interpret the implicit feedback from the network in ways that might be incompatible, or react to the same feedback differently. This causes issues of fairness, where different flows might claim proportionally unequal shares of the bandwidth, which can even lead to starvation of one of the flows. Some algorithms are known to be unfair even with other flows using the same algorithm, regarding e.g., RTT, such as TCP BBR [16].

Proactive approaches traditionally use reservations and admission control and have been employed in telecommunication networks [17]; their usage in environments with heterogeneous traffic classes, such as IP networks, is much more difficult [18]. This difficulty partly lies in estimating the bandwidth requirements of applications, partly in lack of knowledge and control over the network, and partly due to scalability issues.

Congestion control is only one of the tools available to provide Quality of Service to flows; there are other factors that need to be considered, such as latency and jitter. In current IP-based networks, there exist two major systems that attempt to provide a more comprehensive QoS solution for applications. The main difference between them is that first one enforces QoS explicitly on every flow, whereas the second one implicitly groups flows into coarser QoS classes.

Integrated services [19] leverages Resource Reservation Protocol (RSVP, [20]), which is a signaling protocol controlling QoS in the network core (i.e., routers and switches). RSVP distinguishes traffic direction and treats data flows as simplex communication channels. RSVP is receiver-oriented, which means that the receiver is in charge of initiating and request a reservation. Among RSVP subsystem modules there are: policy control, to authenticate and authorize reservation requests; admission control, to account and administer available resources; packet classifier, to group flows into queues; and packet scheduler, to enforces QoS by shaping the outgoing traffic according to classification. For IntServ to operate, RSVP must be present on every device of the network infrastructure. Resource reservation is originated from the receiver and it propagates to the sender with the help of RSVP. On one hand, RSVP employs sophisticated strategies when merging reservations (e.g., two reservations sharing resources on a single link). On the other hand, the merge of reservation can cause starvation or denial of service. RSVP periodically (by default every 30 s) sends state refresh messages, for both path and reservation. Moreover, RSVP poses a significant overhead on QoS-enabled communication (e.g., delay in processing of many small reservations), which is a known scalability issue [21]. Due to

these limitations, IntServ is deployed to support small and predictable QoS use-cases such as VoIP telephony in a corporate network environment.

Differentiated services [22,23] uses neither dedicated signaling protocol nor coordinated QoS treatment among active network devices. In DiffServ, the edge network device (i.e., switch, VoIP-phone) classifies each flow into some traffic class, based on source/destination address/port or protocol, by marking packets. Special Ethernet and IP header fields are used to hold the marks, and their values may represent desired QoS properties (e.g., delay tolerance, delivery assurance under prescribed conditions). Once flow packets are marked, processing network devices (such as routers or switches) may honor the marks and apply policies such as rate-limiting and packet scheduling. The marks can also be ignored, and traffic overridden or reclassified to meet different QoS goals. When compared to IntServ, DiffServ is far more scalable since it does not leverage any signaling. Nevertheless, DiffServ offers only coarse control for similar traffic, whereas IntServ can be fine-tuned to accommodate the specific demands of each flow.

IntServ and DiffServ demonstrate different approaches to how QoS is being addressed in current TCP/IP networks. Both have their advantages and disadvantages, and their operational experience provides valuable insights. IntServ is closer to the topics of this paper because it handles resources reservation per each flow, similarly to the functionality of the Flow Allocator in RINA.

Mechanisms employed to achieve QoS in computer communication includes packet classification and marking, traffic policing and shaping, congestion management and avoidance. Each of these mechanisms is coupled with resource or, more precisely, capacity allocation problem (i.e., where "capacity" means measurable properties of network communication).

Classification identifies particular IP traffic as belonging to some specific class with the same criteria for packet handling. Altering relevant QoS-related bits in data-link (e.g., 802.1p CoS [24], MPLS EXP [25]), network (e.g., DSCP and IPP fields [22]), or transport layer headers is a way to preserve marking (i.e., sharing the classification between network devices). Marking may encode drop eligibility, class selectors, or assured/expedited forwarding in queues. Traffic is the subject of conditioning by policers and shapers. Although policers drop or mark excessive traffic exceeding agreed quota, shapers buffer and delay momentary peaks until bandwidth is available. Both policers and shapers leverage the token bucket algorithm and its mission-specific variants (such as single rate three-color policer [26]) to deal with violating traffic and keep it in conforming limits. Managing congestion in TCP/IP-based networks involves queuing and scheduling. Queuing triggers when the output interface of an active network device experiences congestion (i.e., when output interface is overloaded with incoming packets that cannot be dispatched in due time and must be put into the queue). Queuing is accompanied by scheduling, which decides how packets are reordered in the queue(s) to impose QoS and reflect capacity allocation. Various queuing algorithms do exist (e.g., first-in-first-out, round-robin, low latency queuing, class-based weighted fair queueing) that impact how packets are being enqueued and handled. Congestion avoidance mechanisms (e.g., tail dropping, or weighted RED [27]) periodically monitor queues and avoid potential congestion by dropping some packets preemptively to protect remaining flows.

We have chosen RINA as our target architecture (and rlite as the implementation) as we aim to leverage some of the features unique to RINA to implement a reactive flow allocation scheme based on admission control. Compared to IntServ, we need no additional signalization protocol; most nodes in the network need not store any additional state information, which, combined with RINA's simple DIF stacking model, greatly reduces scalability concerns; and we have a representation of the whole DIF available, allowing us to make better informed decisions.

RINA places a great importance on separating mechanism from policy; this allows us to easily reuse existing functionality to introduce new behavior. By rearranging and extending functionality that is already in place, we can focus on the overall design and the goals we want to achieve. In addition, the decomposition into cooperating components with clearly defined roles and responsibilities allow us to clearly identify which parts of the networking stack need to be modified. In our case, this would

be the Flow Allocator; during development, we have found that our design needs also support from the Routing component to function properly. The policy framework available in rlite allows us to specify this dependency relationship to enforce correct behavior. Likewise, it allows us to easily switch between such policies at run-time to simplify testing and comparison.

The uniformity of the architecture is likewise of principal benefit. There is only one transport protocol, one way to request flow allocation. Whether the application uses reliable or unreliable flows is of no concern to us, as they use the same underlying mechanisms. All application-level communication is done using the Common Distributed Application Protocol (CDAP), and as InterProcess Communication Processes (IPCPs) are applications providing IPC facilities to other applications, we can use this standard mechanism to exchange any additional information between nodes as required. It is not necessary to implement an additional control/signalization protocol. The Resource Information Base (RIB) of any RINA node stores information about the network that is available at our disposal to use during the decision-making process of our policy. The DIF — as the core building unit of RINA networks — gives us guarantees that the other nodes within will be using the same policy and therefore act in a cooperative manner.

The closest point of comparison in TCP/IP would be IntServ; however, this commonality is mostly surface-level. IntServ provides a general framework for managing QoS requirements by employing an additional signaling protocol which reserves resources along a path in a decentralized manner. It does not use any information about the global network state and depends on periodic signalization. Requests travel along the routed path and need to be confirmed by each router on the path; as a result, each router needs to store state information about the QoS reservation.

Our policy, in comparison, is solely focused on bandwidth (data-rate) guarantees. It works in a logically centralized manner where a distinguished node employs information about the state of the DIF to make decisions about received flow requests. Only after a decision is made are the necessary nodes informed with a new route; they do not store any additional state information — only the distinguished node keeps track of the reservations. Therefore, there is no additional periodic signalization, as all the necessary information is propagated through standard network events, such as the enrollment process and routing updates.

3. Recursive InterNetwork Architecture

We have designed and implemented our work in the context of the Recursive InterNetwork Architecture. As RINA uses concepts that often differ vastly from what is being used in TCP/IP, this section provides a brief introduction to the architecture.

The fundamental idea RINA derives the rest of architecture from is that *networking is IPC and only IPC* [28]. An Application Process (AP) is an instantiation of a program containing one or more tasks (Application Entities) and functions for managing allocated resources (processor, storage, IPC). A Processing System (PS) consists of hardware and software executing an Application Process capable of coordinating with a *test-and-set* instruction or its equivalent. A Computing System (CS) is a collection of Processing Systems under the same management domain. What distinguishes IPC within a Processing System from IPC within a Computing System is the IPC facility: the functions remain the same, but the internal mechanisms differ.

A Distributed Application Facility (DAF) is a collection of APs which communicate using IPC and maintain shared state. A Distributed IPC Facility (DIF) is a DAF that provides IPC, and is necessary for IPC between several Processing Systems. A DIF is a network layer and is the basic building block of RINA. The core distinction between traditional (OSI or TCP/IP) network layers and DIFs is that the traditional layer is distinguished by function; DIFs are distinguished by the communication scope, i.e., all DIFs (can) provide all the functions. A DIF may use the service provided by another DIF, thus giving a recursive structure.

To illustrate how a network can be built using DIFs, let us consider three PSs: A, B, and C, connected using physical links as shown in Figure 1.

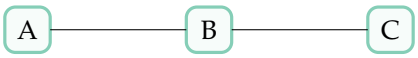

Figure 1. Three PSs *A*, *B* and *C* connected with physical links.

Every PS can instantiate an InterProcess Communication Process (IPCP — an AP providing IPC) for each of its links; a pair of IPCPs sharing a link is able to establish a DIF, as displayed in Figure 2. APs on PSs can use the IPC facilities provided by the new DIFs to communicate with APs on the remote end of a link. However, no communication between APs on *A* and *C* is possible, as *A* and *C* do not share a common DIF.

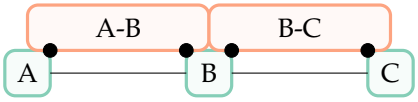

Figure 2. Three PSs *A*, *B*, and *C* connected with physical links. A DIF is created over every link allowing pairs of PSs to communicate.

Communication between all three PSs is made possible by creating another DIF, using the provided IPC facilities of DIFs *A*-*B* and *B*-*C*, shown in Figure 3. The IPCP on *B* will relay messages between *A* and *C* using the lower (link) DIFs.

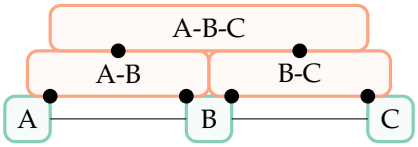

Figure 3. Three PSs *A*, *B*, and *C* connected with physical links. A DIF is created over every link allowing pairs of PSs to communicate. An additional DIF is created over the link DIFs, joining all three PSs in a shared communication domain.

This creates the recursive structure reflected in the architecture's name. There is no limit imposed on the recursion — DIFs can be stacked indefinitely [28].

A DIF (DAF) has the following functions, mirroring the functions used to manage local resources, albeit distributed:

- Task Scheduling
  The Task Scheduling function is responsible for coordinating processing load among the members of a DAF. In the context of a DIF, this would be routing and resource allocation.
- Memory Management
  The RIB contains the local view of the distributed application, i.e., is the storage. It provides access to this information for other members of the DAF when requested. It may provide replication, ACID properties, etc. as needed by a particular DAF. In the case of a DIF, the information stored might be, e.g., the connected neighbors of an IPCP.
- IPC Management
  This function manages the supporting IPC resources, such as multiplexing SDUs into flows or protecting SDU integrity and confidentiality.
- DAF Management
  DAF Management is concerned with the overall management of the DAF, e.g., enrollment into the DAF (employing access control).

DIFs provide IPC service with functions using common mechanisms; however, the strategy of how to use a mechanism will vary across environments. This is referred to as *separation of mechanism and policy*. Different DIFs may provide IPC service in different manner best suited to the scope and use-case using on the same set of operations.

Application Entities (AEs) are tasks within an AP involved with exchanging information with other APs. There are only six possible operations on remote objects: create/destroy, start/stop, and read/write. The differences between "application protocols" are mainly in the set of exposed objects. RINA therefore defines a single application protocol (The architecture does not *require* that a single application protocol is used, although there are benefits in doing so.) called Common Distributed Application Protocol (CDAP). As IPCPs are APs providing IPC, they communicate using this protocol too.

4. Design and Implementation

We have designed and implemented a set of policies for rlite to build a bandwidth reservation system for flows. The main goal of these policies is to guarantee a user-specified amount of bandwidth for each flow over its lifetime. Unlike the best-effort approach to capacity allocation used by TCP/IP, our system denies those flow allocation requests which are deemed not satisfiable. The possibility of negative responses to allocation requests allows us to prevent critical exhaustion of resources — and therefore congestion.

4.1. The Design of Rlite

Before describing how our policies have been implemented in rlite, we will first introduce rlite itself. Rlite is a free and open source RINA implementation for GNU/Linux systems. It consists of a set of kernel loadable modules (KLMs), and a set of user-space daemons (User-space IPCPs—UIPCPs) and tools, as shown in Figure 4.

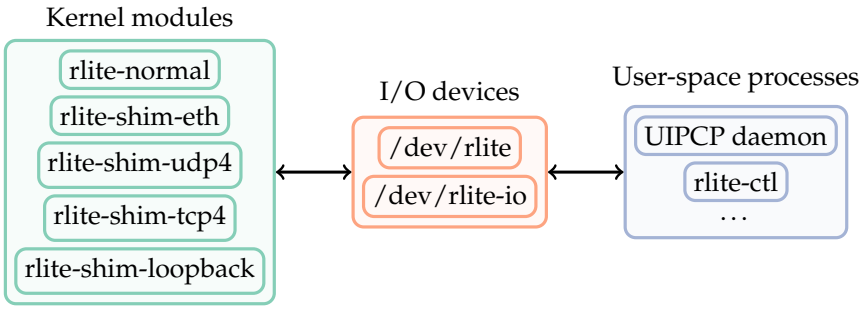

Figure 4. The rlite architecture

The tasks of kernel components is to provide: (i) a generic interface for user-space applications to transmit and receive PDUs; (ii) a set of management functions that allow the user-space components to manage the IPCP lifecycle, such as creation, configuration, inquiry and destruction; and (iii) functions to perform flow allocation and deallocation. Different types of IPCP types are supported, such as the normal IPCP, the shim IPCP over Ethernet, and the shim IPCP over UDP; each IPCP type is implemented as a separate kernel module. Shim IPCPs act as a compatibility layer, allowing deployments of RINA networks over existing Ethernet/IP infrastructure. The Data Transfer and Data Transfer Control functions of an IPCP (i.e., the DTP and DTCP protocols that manage data flow between DIF members) are implemented as a part of the normal IPCP module. Communication with the kernel modules is achieved through two special character devices exposed by the kernel: one serves as a packet I/O interface, the other provides management and control functionality.

The main part of the user-space toolkit, the UIPCP daemon, implements most of the management functionality of the normal IPCP. Functions such as routing, flow allocation, application registration and enrollment are the responsibility of distinct components within the daemon. The actual behavior of these components is configurable via run-time replaceable policies. Policies provide the capability to adjust and extend the operation of built-in components, and adapt the networking stack to the environment where it is deployed. Each IPCP runs in a separate thread with its own instances of all the components, and their policies can be changed at run-time.

4.2. Design of the Bandwidth-Aware Flow Allocation

The goal of the policies we implemented is to guarantee that a requested amount of bandwidth will be available to each flow, exclusively, for the whole lifetime of the flow. Currently, the policy can be applied to normal DIFs stacked directly on top of Ethernet shim DIFs. Additional normal DIFs can be created on top of the bandwidth-aware DIF, as long as they use the default flow allocation policy. Upon flow allocation, the requestor specifies the bandwidth deemed necessary for proper operation and functionality of its service. The Flow Allocator component determines whether the current state of the network (i.e., DIF) allows this request to be accepted; if it concludes that such resource allocation cannot be performed, the request is rejected and the requestor is informed via a negative response. In the case of a positive response, the Flow Allocator reserves some network bandwidth for the newly created flow. On flow deallocation, the bandwidth associated with the flow is freed to be reused by other flows.

No flow can use more than its allocated share, even if the current state of the network would be able to accommodate it. This constraint is enforced in a cooperative manner by means of rate-limiting each flow, implemented in the normal IPCP kernel module. If a flow does not specify the amount of bandwidth required for its operation, it is assigned a pre-defined, configurable amount. Alternatively, it is possible to configure the policy to outright reject such flow requests.

In this manner, the Flow Allocator can track the current state of bandwidth use across the whole DIF. A tightly coupled policy for the Routing component ensures that flows travel only through network paths that have been assigned to them, preventing a change in the network topology from rerouting a flow through a different link, possibly causing congestion to occur. By having full control over the flow allocation process, and enforcing rate limitations in accordance with our desired policy, it is possible to avoid link over-subscription in a proactive way.

The proposed flow allocation scheme can be decomposed into several sub-problems:

1. Querying the interface bandwidth
2. Propagating the interface bandwidth to IPCPs within a DIF
3. Determining whether a flow allocation request can be accepted
4. Making the flow allocation process distributed and ensuring fault-tolerance
5. Ensuring proper usage of network paths

We will discuss each of these problems in turn.

4.2.1. Querying the Interface Bandwidth

To properly manage a resource — in our case the network bandwidth — it is necessary to have up-to-date information about the resource and its state. We have selected the physical link speed as our ground truth; the assumption is that there are no rate-limiting mechanisms in place and every link can operate at their reported speed.

We use the fact that shim over Ethernet DIFs are created over an Ethernet interface, and are usually the lowest DIFs in a system. During DIF creation, we request the interface speed for the associated interface from the kernel. As rlite targets Linux-based systems, this is achieved using the `ioctl()` system call with the `SIOCETHTOOL` command. The reported interface speed is stored as a property of the newly created IPCP, to be used by other components.

4.2.2. Propagating the Interface Bandwidth to IPCPs within a DIF

A shim IPCP over Ethernet therefore knows its own link speed; this information is propagated to a N+1 IPCP during the enrollment process. To share bandwidth information with IPCPs within the same DIF, we use the messages exchanged as part of the routing process.

Rlite implements a link-state routing mechanism, using a Lower-Flow DataBase (LFDB). This shared database contains all the necessary information to provide a holistic view of the DIF

topology to each IPCP, as required by link-state routing to facilitate proper operation. As such, it is necessary that its state is kept up to date.

We can easily distribute bandwidth information about links to other IPCPs by extending the LFDB. Routing update messages will ensure the information is propagated through neighbors to all the IPCPs participating in the DIF.

4.2.3. Determining Whether a Flow Allocation Request Can Be Accepted

The flow allocation process begins with a flow allocation request and ends with one of two outcomes: the request is either accepted or rejected. A flow allocator policy determines how this decision is made.

For our purposes, we want this decision to be made based on the current network state with regards to bandwidth usage. The LFDB can be used to build a graph of the DIF, as this is necessary for the operation of link-state routing. We have already extended it with information about total lower-flow bandwidth (link speeds); we can extend it further with information about lower bandwidth flow usage. With the requested bandwidth being part of the flow request message, we now have all the components to model this as a graph problem with edge capacities.

We are looking for a path from source to destination which satisfies the capacity requirement. This will be the path for which we will reserve bandwidth for our flow. As an additional constraint, we would like to find the *shortest* path (in number of hops). This is useful to minimize latency.

Such a path can be found using Algorithm 1, which is a modified Breadth-First Search (BFS) with additional path capacity checks. By using BFS, we ensure that we select the shortest path among several alternatives. The three lines at the beginning of the algorithm contain variable initialization which consist of value assignments and therefore have a constant complexity of $O(1)$. The first cycle in the algorithm, labeled with the *Find the shortest viable path* caption, iterates over all the edges in the graph in a breadth-first manner. In the worst case, where the correct path is the last one tried, and the additional capacity condition does not cause any potential paths to be eliminated, this cycle iterates over all the edges, giving it a complexity of $O(E)$. The algorithm concludes with an additional cycle, labeled *Build the path*, which iterates over the found path from the target to the source to reverse it, which in the worst case of a linear graph also has a complexity of $O(E)$. This gives the algorithm a total complexity of $O(2 * E)$.

Algorithm 1 Algorithm for finding the shortest path of a given capacity. V is a set of vertices, E a set of edges constituting the graph. cap is a function $E \to \mathbb{N}$ describing edge capacities. s and t are the flow source and destination vertices, with b being the flow size.

Input: a flow network $G = (V, E, cap, s, t, b)$
Output: a viable path P

```
Let found := False
Let q be a FIFO queue
Push s to q
Let pred be an array holding the preceding edge on the current path
while not found ∧ q is not empty do                    ▷ Find the shortest viable path
    Pop w from q
    for all (u, v) ∈ E : u = w do
        if pred[v] = null ∧ (v, u) ≠ pred[u] ∧ cap(u, v) ≥ b then
            pred[v] := (u, v)
            Push v to q
if pred[t] ≠ null then                                 ▷ Build the path
    Let P := {t}
    for all (u, v) ∈ pred starting from pred[t] do
        Prepend u into P
        Continue to pred[u]
```

4.2.4. Making the Flow Allocation Process Distributed and Fault-Tolerant

The basic version of the flow allocation process only involves the two endpoints of the new flow, and as such, many flow allocations can run independently. Our proposed allocator modifies the global state (bandwidth reservations): we need an arbiter to avoid any inconsistencies. The delay in propagating information among nodes might result in them making decisions based on stale data that no longer reflects the current state of the network. As a result, some bandwidth could get reserved multiple times and thus lead to possible congestion.

It is clear that the flow allocation process can no longer be fully independent and must be carried out in a cooperative manner. However, delegating the responsibility to a single distinguished node results in a centralized design that is not resilient to node failures.

To balance these disagreeing requirements, we have opted to implement the flow allocation as a distributed process using the Raft [29,30] consensus algorithm. This allows us to reason about the process as if it were centralized, yet providing benefits of being distributed, such as fault-tolerance.

Raft is a consensus algorithm similar to Paxos, which ensures that all participating nodes eventually agree on the same value of some global variable. It operates by viewing the participating nodes as replicated state machines. A distinguished node, called the *leader*, is elected from among the cluster. The leader appends all state changes into a log which is distributed to the other nodes. Only after most nodes have confirmed receiving the update are the changes actually committed to the state machine. In the case of leader failure, the rest of the cluster can elect a new leader from the nodes which have the most recent log, who takes over the leader's responsibilities. The rest of the nodes act as *followers*, passively synchronizing their state with the leader and checking the leader's availability.

Our revised flow allocation process, using Raft, looks as follows:

1. An application requests a new flow from the local IPCP
2. The local IPCP sends the flow request to the Raft leader
3. The leader finds a viable path, reserves the bandwidth, and forwards the flow request to the destination IPCP
4. The destination IPCP relays the request to the target application, which decides whether to accept or reject
5. The reply is returned to the destination IPCP which forwards it back to the leader
6. The leader returns it to the source IPCP
7. The reply reaches the requesting application

In step 3., the leader does not forward the request further until it receives confirmation about the reservation entries being replicated. When communicating with the leader, nodes do not address it directly, instead they target the whole replica cluster; the replicas make sure only the leader processes the message.

To identify the flow whose reservation is being modified, the leader constructs a unique *flow-id* for each flow. This consists of the source an destination IPCP addresses together with the source port; the destination port is not included as it is not known during step 3., which is where the flow-id is first needed.

Requested bandwidth is considered to be in a *pending* state between steps 3. and 5. When evaluating any new flow requests, it is treated as unavailable. However, if the flow allocation process ends negatively, either due to rejection, timing out when waiting for a reply, or some other cause, the associated bandwidth is reclaimed. In case of flow deallocation, the process follows a similar set of steps as in the allocation case, with the difference that bandwidth is freed instead of being reserved.

4.2.5. Ensuring Proper Usage of Network Paths

For each flow, our distributed Flow Allocator picks a path through the network and reserves the necessary bandwidth. However, this information is only stored within the LFDB; the actual path any packet takes within the network is responsibility of the routing component.

Therefore, our implementation contains a routing policy that cooperates with the Flow Allocator. These policies are tightly coupled to prevent activating the flow allocator policy without the necessary routing policy.

The new routing policy routes each packet based on the flow the packets belongs to, rather than simply looking at the packet destination. Even flows sharing the same source and destination might be using different paths. It is the responsibility of the Raft leader to distribute routes after a flow allocation request has been accepted. Conversely, it cleans up the routes associated with a flow after the latter has been deallocated. This ensures that flows are transported through the network as intended.

4.3. Policy Operation and Configuration

The proposed flow allocation scheme has been implemented within rlite as two cooperating policies: *bw-res* for the Flow Allocator, and *bw-res-link-state* for the Routing component. The Routing policy contains supporting functionality with regards to graph operations and LFDB updates, while the main FA policy implements the distributed FA process itself. A diagram of the FA process is shown in Figure 5 below.

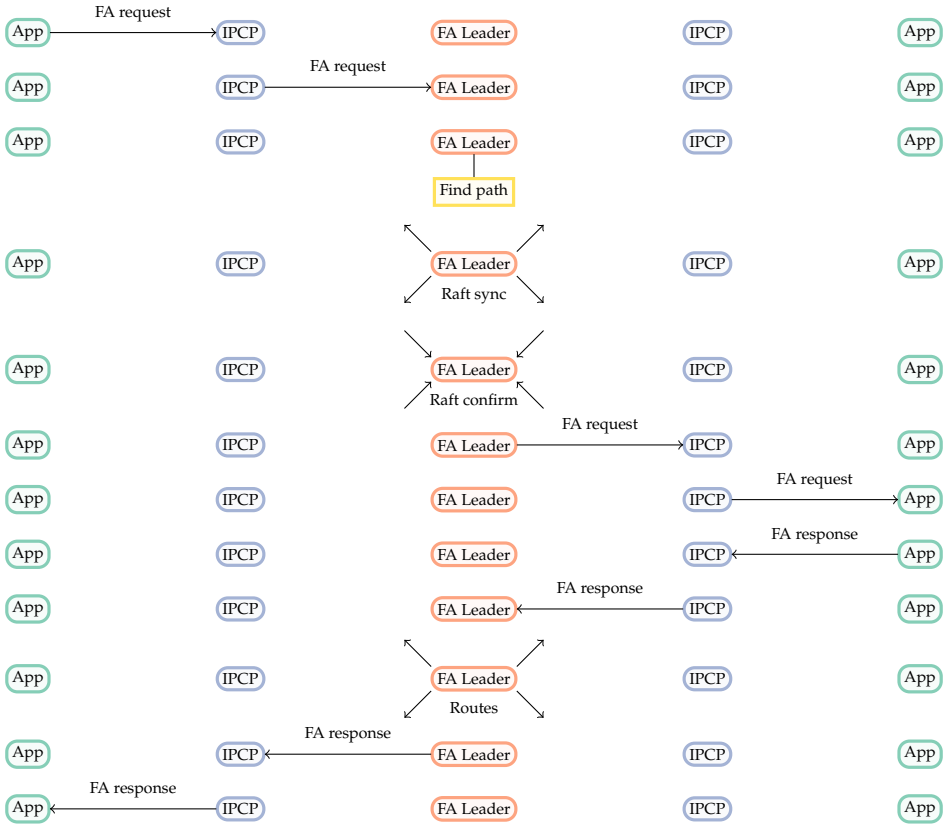

Figure 5. Overview of the flow allocation process. Horizontal arrows represent direct communication; diagonal arrows are communications with other replicas within the Raft cluster or between the Raft cluster and nodes along the reserved path; the yellow box is an action done within the IPCP.

The FA policy has a set of configurable parameters, shown in Table 1. The first two control the behavior of the Raft cluster, whereas the others affect the handling of flow requests that do not specify the required bandwidth. It is possible to either reject such requests, or to assign to them a pre-defined bandwidth value. The choice between these two options is left to the network administrator, as different deployment scenarios and environments might prefer one over the other.

Table 1. Overview of the configurable parameters.

Name	Description
`replicas`	Names of the IPCPs acting as part of the Raft cluster
`cli-timeout`	Timeout for the client request for the replicas
`reject-inf-bw`	Whether to immediately reject flow requests that do not set the expected average bandwidth
`default-bw`	The average bandwidth value used for flow requests that do not set it

The extra information about bandwidth in the LFDB has been exposed in the management interface of *rlite-ctl*. Every entry now contains two more values, the *Total* available bandwidth and the available *Free* bandwidth. In addition, printing the Resource Information Base (RIB) of an IPCP participating in the Raft cluster shows a table of flow reservations, with their flow-ids, endpoints, route and bandwidth values. An example can be seen in Figure 6 below.

```
Lower-Flow Database:
    Local: n1.b.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.b.IPCP, Remote: n1.a.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.a.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.c.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.c.IPCP, Remote: n1.d.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 20 Mbps

Supported flows (Src/Dst <Appl,IPCP,port>):
    [R], Src=<rinaperf-data|client,n1.a.IPCP,2>
         Dst=<rinaperf-data|server,n1.d.IPCP,2>
         Connections: [<SrcCep=1, DstCep=1, QosId=0> ]
    [R], Src=<rinaperf-data|client,n1.a.IPCP,1>
         Dst=<rinaperf-data|server,n1.d.IPCP,1>
         Connections: [<SrcCep=0, DstCep=0, QosId=0> ]

Flow reservation table:
     n1.a.IPCPn1.d.IPCP2: n1.a.IPCP->n1.d.IPCP
       (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,n1.d.IPCP,) : 40 Mbps
     n1.a.IPCPn1.d.IPCP1: n1.a.IPCP->n1.d.IPCP
       (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,n1.d.IPCP,) : 40 Mbps
```

Figure 6. The status of the RIB of a Raft replica while running a `rinaperf` test application, requesting 40 Mbps per flow, with two active flows. The flow reservation table lists two entries, and the available bandwidth has decreased accordingly. The output is not complete, as some parts have been edited or omitted for brevity.

5. Evaluation and Results

Testing has been done on the graph algorithm itself in isolation and the implemented policies as a whole. The graph algorithm tests presented are part of rlite unit tests; their main goal is verifying the expected functionality of the algorithm. We also implemented integration tests for our policies, to prevent future changes from introducing regressions into the implementation. In addition to those, several experiments have been carried out with the implemented policies to measure how they work with regards to bandwidth use. Technical details about the experimental setup are described in Section 8.

The graph algorithm tests consist of a pre-defined graph and a set of flow requests. Every test starts from the same initial graph; there are no tests that simulate previous flows being allocated.

Figure 7 shows the simplest test scenario, consisting of four nodes connected as a square. In test (**a**), a flow from *N0* to *N2* of size 5 is requested. There are two possible paths, and the algorithm returns one of them as expected. Restarting from the initial situation where all the links are free, test (**b**)

increases the flow to 10 and changes the destination to *N3*. Now, only one possible path exists — one that is longer than the shortest path from *N0* to *N3*. The algorithm correctly finds this longer path, as is necessary due to the capacity condition.

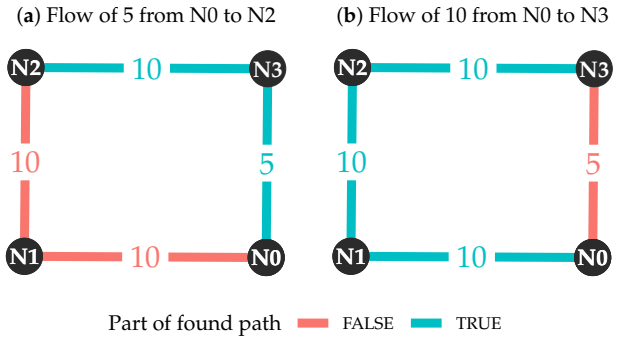

Figure 7. A basic test for the graph algorithm to check that it respects the capacity condition. (**a**) The result of searching for a path from *N0* to *N2* of size 5; (**b**) the result of searching for a path from *N0* to *N3* of size 10.

Figure 8 displays a similar test, with a bigger graph and many more possible paths between any two nodes. Again, the test has been intentionally designed to contain only one viable path that is much longer than the direct, shortest path. The algorithm can find this path.

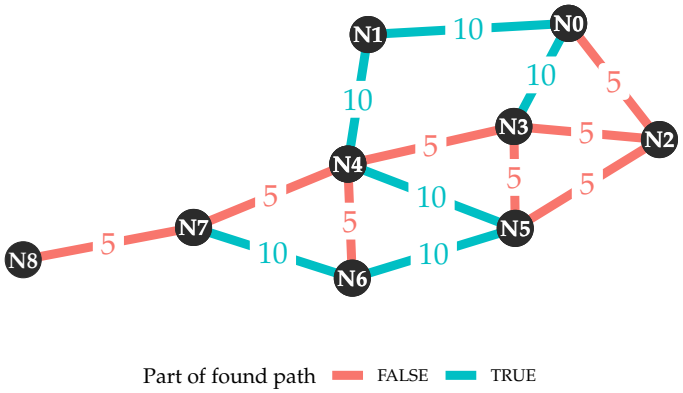

Figure 8. Another unit test for the graph algorithm. It successfully finds the only possible path from *N3* to *N7* of size 10.

The first policy test scenario, shown in Figure 9 consists of 6 nodes, representing two sites connected with redundant paths. Nodes *A* and *F* act as end devices and flow endpoints. Flows between these two sites can be routed through two possible paths: *B-C-E* or *B-D-E*. Links *A-B* and *E-F* have a capacity of 20 Mbps, the rest are 10 Mbps links.

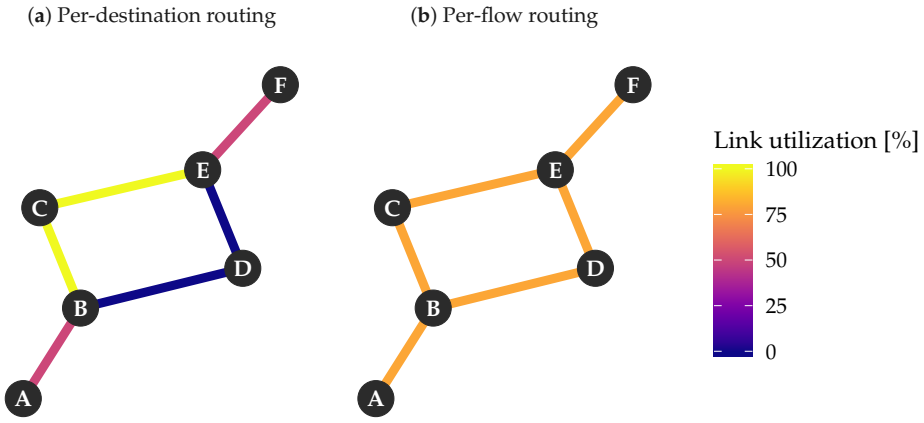

Figure 9. Better bandwidth use as a result of per-flow routing. (**a**) shows complete saturation of the inner links while an alternate path remains unused. In (**b**), there is almost double the traffic flowing through the network, as it can use all available paths.

In the first case, two flows have been allocated using `rinaperf`, each requiring 5 Mbps. As *B* routes both to *E*, it uses one of the available paths—in this case the one through *C*. This path becomes completely saturated, while the other path remains unused.

In the second case, we have increased the bandwidth of each flow to 8 Mbps. This results in a total of 16 Mbps, which would oversaturate and congest any of the inner links. The Flow Allocator is aware of this and routes one of the flows through the alternate path, resulting in a more even link use compared to the previous case as well as better overall network bandwidth use.

Two larger experiments have been carried out, differing in the network topology used and the number of `rinaperf` concurrent instances. In both experiments, each link has an available bandwidth of 10 Mbps. Each `rinaperf` instance uses two flows, and each flow requests 500 kbps of bandwidth so that one instance will use 1 Mbps in total. Every node in the network is connected to at least two other nodes. Single-homed nodes are not interesting in this scenario, since more available connections are needed to properly use alternate paths in the network.

The first of these experiments has been done with a network of 10 nodes, and 64 `rinaperf` instances have been used. Visualization of the experiment can be seen in Figure 10. Figure 11 displays additional data regarding link use and the success rate of flow allocation requests.

From this data, we can see that one of the links becomes fully saturated around the 35th iteration. This has no noticeable effect on the following flow allocation requests, as there are enough alternate paths that can be used. It is shortly before the 60th iteration that all three links connecting one part of the graph to another — seen as the three yellow edges going from the center node to the right in Figure 10d — that the average link use reaches a plateau and flow allocations start to fail. This is expected, as our network has effectively been segmented into two sub-networks and no flows allocations between these two segments will succeed.

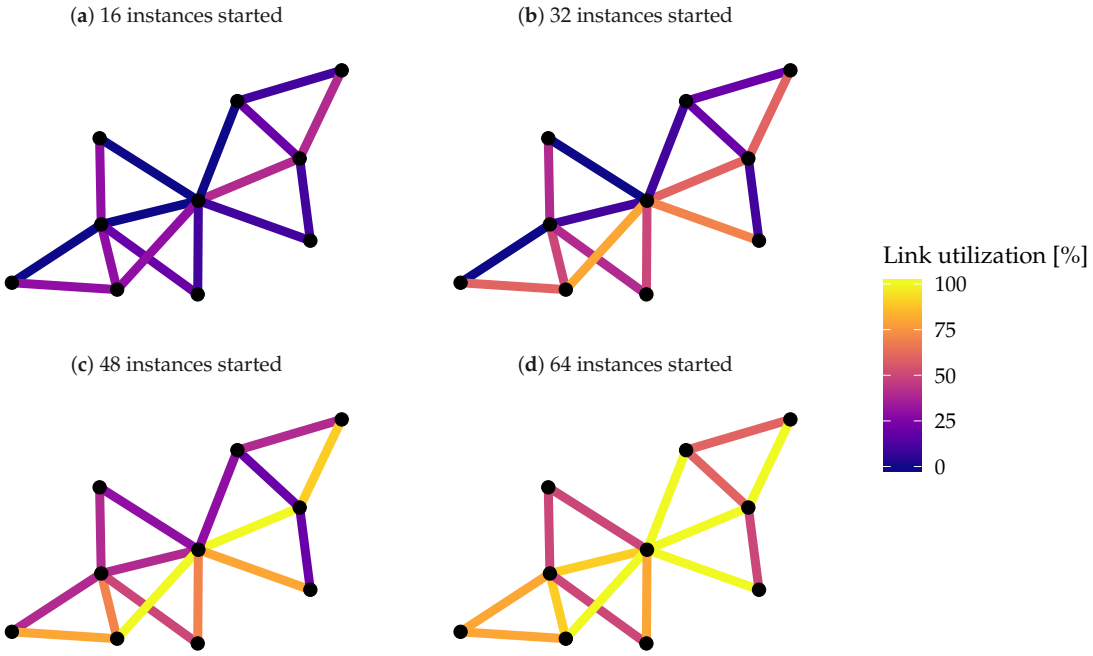

Figure 10. Scenario with 10 nodes: Visualization of link use after (**a**) 16, (**b**) 32, (**c**) 48, (**d**) 64 `rinaperf` instances have been started on randomly selected nodes.

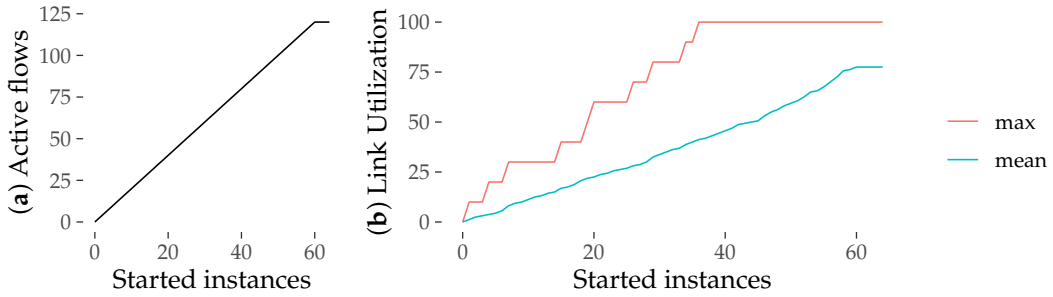

Figure 11. Scenario with 10 nodes: (**a**) Success of flow allocation requests over time; (**b**) link use over time. The value *max* shows the use of the link with the most traffic flowing through it at that point in time; *mean* is the average link use across the network.

The second experiment (Figures 12 and 13) has been carried out in a similar way, but with network of 20 nodes and 100 `rinaperf` instances. We observe a behavior analogous to the previous experiment. One of the interconnecting links — visible in orange-yellow in Figure 12a — becomes saturated shortly after the 25th iteration, with no effect on the overall behavior. Once again, the network partitions itself, this time around the 75th iteration. Use and flow count plateaus for some time, but since this network is larger than the previous one, some flow allocations succeed within their own segments.

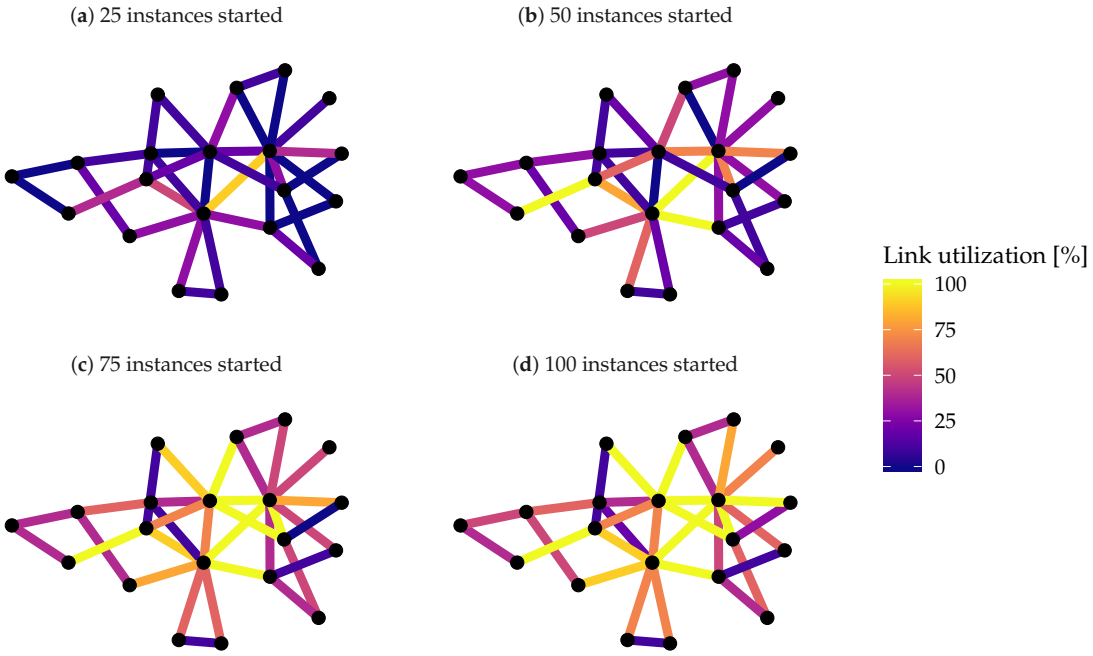

Figure 12. Scenario with 20 nodes: Visualization of link use after (**a**) 25, (**b**) 50, (**c**) 75, (**d**) 100 `rinaperf` instances have been randomly started.

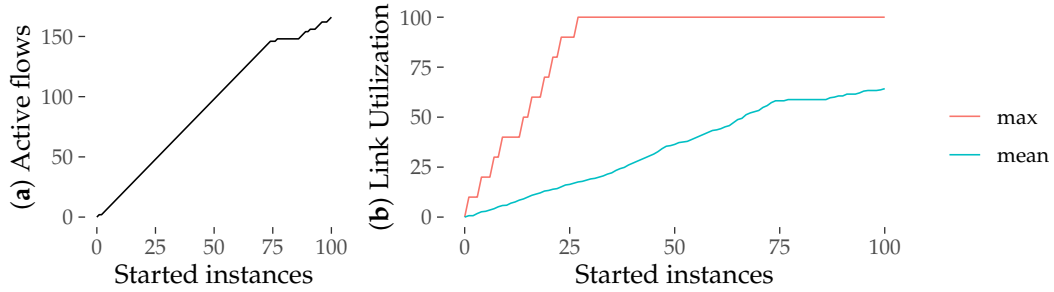

Figure 13. Scenario with 20 nodes: (**a**) Success of flow allocation requests over time; (**b**) link use over time. The value *max* shows the use of the link with the most traffic flowing through it at that point in time; *mean* is the average link use across the network.

Both experiments follow a similar pattern, where links in the center that interconnect the various edges of the network are the first to get saturated. As time progresses, links around these core links become more used, as they serve as alternate paths between the network parts. Link use is distributed fairly evenly, depending on the distance from the core. This agrees with our intuitive expectation that as more flows get allocated, the backbone of the network will get more strained.

As the policy can use alternate paths, it is able to deliver better a quality of service for some flows. This, however, comes at a price: as the selected alternate path might be considerably longer, this leads to increased latency and generally less predictable behavior.

Figures 14 and 15 display a comparison of maximum achievable flow between two ends of a network. The source and target nodes have had the capacity of their edges increased to infinity to act as "supersinks". In both figures, the first graph is the result obtained from running the Ford-Fulkerson maximum flow algorithm, which serves as the theoretical upper bound. The second graph contains the maximum achieved flow using our policy before flow allocation requests started being rejected. The maximum flow in both scenarios is 30 Mbps, which our policy successfully reaches, albeit using different paths.

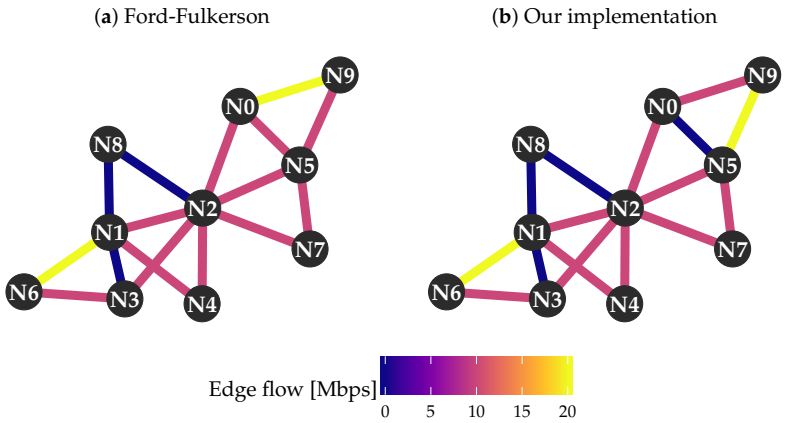

Figure 14. Comparison of maximum flow from N6 to N9 in the network from Figure 10. Both nodes act as "supersinks" and have had the capacity of their edges increased to infinity. (**a**) shows the results obtained using the Ford-Fulkerson maximum flow algorithm. (**b**) shows results obtained using our implemented policy.

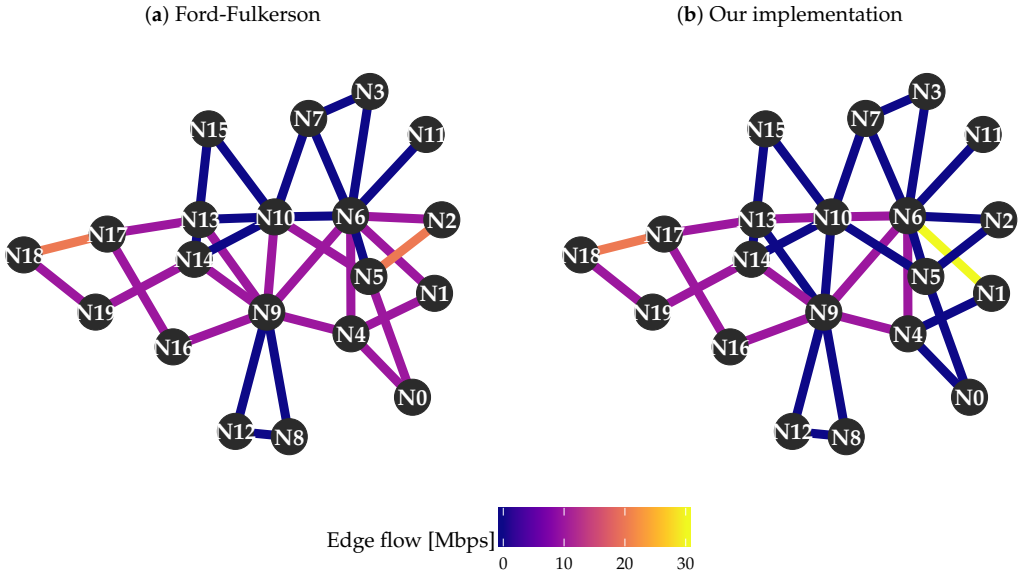

Figure 15. Comparison of maximum flow from N18 to N2 in the network from Figure 12. Both nodes act as "supersinks" and have had the capacity of their edges increased to infinity. (**a**) shows the results obtained using the Ford-Fulkerson maximum flow algorithm. (**b**) shows results obtained using our implemented policy.

Figure 16 shows the flow allocation delay for our *bwres* policy, as well as the default *local* policy available in rlite, in the scenario from Figure 10. The delay has been measured as the time period between the local IPCP receiving a FA request from an application (first row in Figure 5) and the same IPCP receiving a response, whether positive or negative, from the remote party (one before last row in Figure 5). The values measured for the *local* policy also serve as the lower bound of achievable FA delay, as the FA process consists of only the minimal signaling communication required for successful flow allocation.

The delay for local flow allocation (without delay introduced by the destination application making a decision of whether to accept or reject the request) can be computed as

$$d_{local} = RTT$$

where RTT is the round-trip time between the flow source and the flow destination.

For our *bwres* policy, the FA delay can be computed as

$$d_{bwres} = RTT_s + RTT_d + L$$

where RTT_s is the round-trip time between the flow requester and the Raft leader, RTT_d the round-trip time between the Raft leader and the destination node, and L the processing overhead of the Raft leader. L could be further computed as

$$L = G + C$$

where G is the cost of running the graph algorithm and C the overhead of replicating the Raft log. In an ideal case, where $L = 0$, the relation between the two delays is

$$RTT \leq RTT_s + RTT_d$$

depending on whether the Raft leader is on the shortest path between the flow source and flow destination nodes. The *local* policy therefore also serves as the lower bound on FA delay in this comparison.

As expected, the extra signaling overhead required for the policy's operation results in a higher median delay. The complexity of several parts of the flow allocation process (finding a path, distributing routes, etc.) is dependent on the length of the path/distance between the source and destination nodes, resulting in a greater variation in delay. However, for the test scenario, the third quartile of *bwres* reaches the values of the first outliers of *local* and is still significantly lower than the last outliers, suggesting that the policy's operation is in itself not the biggest source of possible delay.

Figure 17 shows the same data, this time for the scenario from Figure 12. The data follows a similar pattern as that from Figure 16. The larger network size has resulted in an increased allocation delay for both policies. For *bwres*, the outliers below the median value are probably allocation requests that have been rejected very fast, such as those whose source is the Raft leader; in such cases, the messages are processed on the same IPCP, thus avoiding any delay introduced by round-trip times. As the network has become larger, so has increased the delay variation for both policies. The difference between the median delays for the same policy in the two scenarios is significantly larger than the difference between the two policies within one scenario, again suggesting that the main source of delay is not the policy itself. Even with the added complexity and additional signaling the latency is comparable to that of the default, naive policy which provides no features at all.

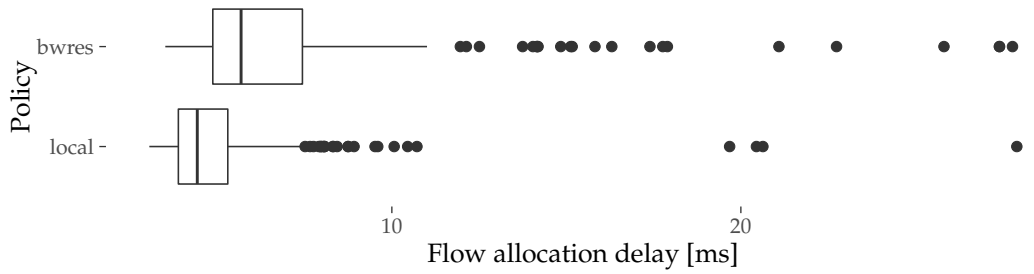

Figure 16. Comparison of flow allocation delay for the scenario from Figure 10. The box edges correspond to the first and third quartiles, with a bolder line representing the median value. Outliers are shown as dots.

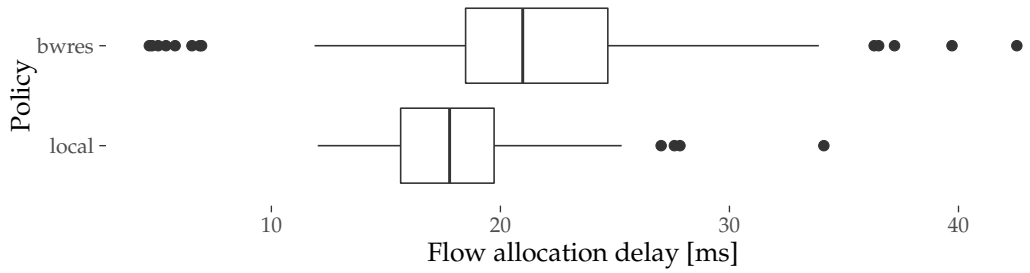

Figure 17. Comparison of flow allocation delay for the scenario from Figure 12. The box edges correspond to the first and third quartiles, with a bolder line representing the median value. Outliers are shown as dots.

6. Discussion

Our implemented flow allocation scheme achieves the stated goal of preventing a link from becoming oversaturated. It provides a reliable guarantee to flows regarding available resources. This flow allocation scheme is suitable for scenarios where bandwidth requirements can be estimated in advance, and its availability is of higher priority than predictable paths and latency between various parts of the network.

The implementation assumes the actors participating in the DIF are not malicious, and expects this to be enforced during DIF enrollment if necessary. There are no limits or checks to what an application might do: requesting an interface's worth of bandwidth for one flow is possible, but is probably closer to a DoS attack than a legitimate use-case. It is expected that applications work in a cooperative manner and do not request more bandwidth than is necessary. Such checks could be added, to e.g., limit the maximum bandwidth for one allocation, or the share of total bandwidth that one IPCP might have reserved at any point in time; it might in fact be necessary to do so to make the implementation production-ready, if only to prevent buggy applications from taking up too much bandwidth by accident. However, these restrictions would not change the design in any significant way, and might in fact prove to be limiting when running experiments.

A necessary side effect of our design is that flow allocation may fail, due to bandwidth limitations. Although this might seem as something undesirable at first, the extra information about the state of the network, coupled with better QoS assurance, enables creation of adaptive applications which can use these factors for their decision making and adjust their behavior. An application might select different compression algorithms for data transfer, trading local processing speed for bandwidth requirements. An alternative approach would be for the application to gracefully degrade the service it provides, based on the available bandwidth. The application could periodically probe the network state, trying to establish a better flow, and migrate its communication if successful.

7. Future Work

We have identified several areas where further work could be done to improve both performance and usability.

The first major concern is scalability: as it stands now, the policy is not able to fully use the features of the *Recursive* InterNetwork Architecture. In its current state, the implementation is a proof of concept of an alternative approach to flow allocation. Although some of its functionality leverages RINA's unique features, it currently does not fully integrate with the recursive stacking. This has been a conscious decision during implementation—not a result of any known design limitations—and plans exist to extend the implementation to fully support normal DIFs over normal DIFs.

The second area for improvements is reacting better to changes in the network. With the current design, the reserved path does change during the whole lifetime of a flow. This has negative consequences on the robustness of the network as well as being a potential source of performance

improvements. If a node along a reserved path goes down, the flow effectively dies as well, since routing rules do not reflect this new information. Rerouting such flows on link-state changes through alternate paths, if possible, would significantly improve the policy's resilience.

This idea can be pursued further: the FA might continually try to reroute flows through more optimal paths. Using a shorter path that has been previously unavailable would free up additional bandwidth and allow more flows to exist simultaneously. Besides significantly increasing the computational complexity of the FA process, rerouting flows too often and too fast could negatively impact the overall stability of the network. Likewise, a malicious actor could use this to launch a DoS attack by forcing the FA leader to constantly attempt to reroute all the flows.

We have, for the sake of simplicity, effectively avoided the issue of flows that do not specify their bandwidth. They are either ignored (by immediately rejecting them) or assigned a pre-defined bandwidth value and treated as any other flow. Depending on the configured value, the nature of the flows, and their number, this might result in a lot of bandwidth being wasted or conversely, being too restrictive.

One possible approach would be to dynamically divide the surplus bandwidth between all these flows. This would require updating the flow rate limits on each flow allocation. With this method, unlimited flows would be of lower priority than limited, as the bandwidth available to them would continually shrink to the advantage of the latter group. One of the undesirable assumptions — shared with the pre-defined bandwidth approach — is that every such flow could use the same amount of bandwidth. Given the great diversity between applications and their requirements, this is not ideal; however, this solution is simple to implement.

A more complex solution to this problem would be to limit each of these flows to the whole of the surplus bandwidth and let a flow control mechanism divide the bandwidth between flows. Proactive and reactive control mechanisms are not mutually exclusive [9]; they can be used in a complementary manner. This hybrid approach provides potentially better bandwidth use, as each flow would take up only as much bandwidth as needed. Besides the greater complexity, letting a reactive flow control mechanism come into play might negatively affect the limited flows, as there might be periods of time during which a link might be oversaturated.

Whether a flow allocation will succeed is dependent on the available bandwidth on each path between the source and the destination. There might be situations where a single path does not have enough bandwidth, but several of them combined do. Splitting the flow into sub-flows which will all be routed through a different path would enable the network to squeeze out several more successful flow allocations in a busy network. Once again, this approach significantly raises the complexity of the whole process. At the same time, different paths might vary greatly in their characteristics, such as latency and hop count. This might cause problems with the delivery, such as packet reordering and head-of-line blocking, negatively impacting the performance.

The final area for improvements, orthogonal to all the previous ones, is the graph algorithm itself. A more sophisticated algorithm might take into consideration additional properties of the available paths. Given several shortest path candidates, the algorithm could e.g., prefer the one with the largest bottleneck capacity:

$$cap(path) = min(\forall edge : cap(edge))$$

$$result = max(\forall path : cap(path))$$

This would prioritize paths with sparer bandwidth and delay the network becoming partitioned due to an important interconnecting link becoming saturated. An example is shown in Figure 18 below.

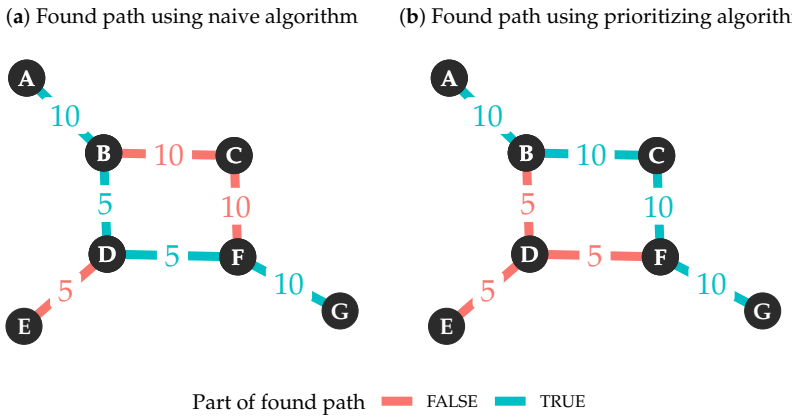

Figure 18. Improvements in bandwidth use effectiveness through algorithm modification. A network with three sites (*A*, *E* and *G*) connected using two redundant paths. In both cases, a flow from *A* to *G* of size 5 has been allocated. In (**a**), corresponding to the current algorithm, a path through *D* might be selected, depending on the traversal order. This completely cuts off site *E* from the rest of the network, even though there is enough bandwidth. Conversely, an algorithm with the proposed prioritizing modification in (**b**) avoids this by selecting the alternate route first.

8. Materials and Methods

The source code for our implementation is available at https://github.com/autrimpo/rlite, tagged with the `computers20` tag. Instructions how to build rlite are in the included `README.md` file.

Unit tests for the graph algorithm are defined in the file `user/uipcps/lfdb-bw-test.cpp` and can be ran using `build/user/uipcps/lfdb-bw-test`. Each test prints out information about the graph, the requested flow, as well as the found path, if any.

The bandwidth use measurements were done on a virtual QEMU network created using the included demonstrator tool. The demonstrator tool, located in `demo/demo.py`, takes a configuration file describing the desired network. Configuration files used in this paper are available at http://www.stud.fit.vutbr.cz/~koutenmi/Computers20/multipath.conf for Figure 9, http://www.stud.fit.vutbr.cz/~koutenmi/Computers20/stress-10.conf for Figure 10, and http://www.stud.fit.vutbr.cz/~koutenmi/Computers20/stress-20.conf for Figure 12. The second and third configurations already come with the bw-res policy set; in the case of the first one, manual adjustment of the running network is required.

Before creating the network, it is necessary to build a custom buildroot image using https://github.com/rlite/buildroot. The process is documented in the `update.sh` script: it requires proper configuration of the `package/rlite/rlite.mk` file to use the local rlite source code, setting the matching output path in the update script and running the script itself.

The demonstrator has been run with the following parameters: `./demo.py -b udp -e full-mesh -c <CONF_FILE>`. This generates several shell scripts that can be used to start (`up.sh`), stop (`down.sh`), and access (`access.sh`) the virtual machines. Server `rinaperf` instances have been started with `rinaperf -l -z <UNIQUE_NAME>` and the clients used `rinaperf -z <SERVER_NAME> -B 500000 -D 1000000`. Each client running requires its own server instance. Data has been gathered by dumping the RIB on a node participating in the Raft cluster with `rlite-ctl dif-rib-show`.

Author Contributions: Conceptualization, M.K. and V.M.; methodology, M.K.; software, M.K. and V.M.; validation, M.K.; formal analysis, M.K.; investigation, M.K.; resources, M.K., V.V. and V.M.; data curation, M.K.; writing–original draft preparation, M.K., V.V. and V.M.; writing–review and editing, M.K., V.V. and V.M.; visualization, M.K.; supervision, V.V. and V.M.; project administration, V.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding, the article publication was subsidized by Department of Information Systems by the Faculty of Information Technology of Brno University of Technology. The authors would like to praise Dušan Kolář for his generosity and open-mindedness to support RINA-related research.

Acknowledgments: This article has been supported by the Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (no. LQ1602).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AE	Application Entity
AP	Application Process
BFS	Breadth-First Search
CDAP	Common Distributed Application Protocol
CS	Computing System
DAF	Distributed Application Facility
DIF	Distributed Interprocess Facility
DiffServ	Differentiated Services
DTCP	Data Transfer Control Protocol
DTP	Data Transfer Protocol
DoS	Denial of Service
FA	Flow Allocator
IntServ	Integrated Services
IP	Internet Protocol
IPCP	InterProcess Communication Process
IoT	Internet of Things
KLM	Kernel Loadable Module
LAN	Local Area Network
LFDB	Lower-Flow DataBase
PS	Processing System
PDU	Protocol Data Unit
QoS	Quality of Service
RIB	Resource Information Base
RINA	Recursive InterNetwork Architecture
SCTP	Stream Control Transmission Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UIPCP	User-space InterProcess Communication Process
WAN	Wide Area Network

References

1. Welzl, M. *Network Congestion Control: Managing Internet Traffic*; John Wiley & Sons Ltd.: Chichester, UK, 2005.
2. Pouzin Society. RINA Specifications. 2019. Available online: https://github.com/PouzinSociety/RINASpecifications (accessed on 30 May 2020).
3. Day, J.D. *Patterns In Network Architecture: A Return to Fundamentals*; Prentice Hall: Upper Saddle River, NJ, USA, 2008.
4. IRATI. Available online: https://irati.github.io/stack/ (accessed on 30 May 2020).
5. Rlite: A Light RINA Implementation. Available online: https://github.com/rlite/rlite (accessed on 30 May 2020).
6. Ouroboros. Available online: https://ouroboros.rocks/ (accessed on 30 May 2020).
7. ProtoRINA. Available online: https://csr.bu.edu/rina/protorina/2.0/ (accessed on 30 May 2020).
8. RINASim. Available online: https://rinasim.omnetpp.org/ (accessed on 30 May 2020).
9. Keshav, S. Congestion Control in Computer Networks. PhD Thesis, University of California at Berkeley, Berkeley, CA, USA, 1991.
10. Jacobson, V. Congestion Avoidance and Control. In Proceedings of the SIGCOMM '88, Symposium Proceedings on Communications Architectures and Protocols, Stanford, CA, USA, 16–18 August 1988; ACM: New York, NY, USA, 1988; pp. 314–329. doi:10.1145/52324.52356.

11. Braden, R. Requirements for Internet Hosts—Communication Layers. In *RFC 1122*; IETF: Fremont, CA, USA, 1989.
12. Ha, S.; Rhee, I.; Xu, L. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.* **2008**, *42*, 64–74. doi:10.1145/1400097.1400105.
13. Xu, L.; Harfoush, K.; Rhee, I. Binary increase congestion control (BIC) for fast long-distance networks. In Proceedings of the IEEE INFOCOM 2004, Hong Kong, China, 7–11 March 2004; Volume 4, pp. 2514–2524. doi:10.1109/INFCOM.2004.1354672.
14. Cardwell, N.; Cheng, Y.; Gunn, C.S.; Yeganeh, S.H.; Jacobson, V. BBR: Congestion-Based Congestion Control. *Queue* **2016**, *14*, 20–53. doi:10.1145/3012426.3022184.
15. Floyd, S.; Jacobson, V. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.* **1993**, *1*, 397–413. doi:10.1109/90.251892.
16. Scholz, D.; Jäger, B.; Schwaighofer, L.; Raumer, D.; Geyer, F.; Carle, G. Towards a Deeper Understanding of TCP BBR Congestion Control. In Proceedings of the 2018 IFIP Networking Conference (IFIP Networking) and Workshops, Zurich, Switzerland, 14–16 May 2018.
17. Ramjee, R.; Nagarajan, R.; Towsley, D. On optimal call admission control in cellular networks. In Proceedings of the IEEE INFOCOM '96. Conference on Computer Communications, San Francisco, CA, USA, 24–28 March 1996; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1, pp. 43–50.
18. Kalmanek, C.R.; Kanakia, H.; Keshav, S. Rate Controlled Servers for Very High-Speed Networks. In Proceedings of the GLOBECOM '90: IEEE Global Telecommunications Conference and Exhibition, San Diego, CA, USA, 2–5 December 1990; pp. 12–20.
19. Braden, B.; Clark, D.; Shenker, S. Integrated Services in the Internet Architecture: An Overview. In *RFC 1633*; IETF: Fremont, CA, USA, 1994.
20. Braden, B.; Zhang, L.; Berson, S.; Herzog, S.; Jamin, S. Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. In *RFC 2205*; IETF: Fremont, CA, USA, 1997.
21. Mankin, A.; Baker, F.; Braden, B.; Bradner, S.; O'Dell, M.; Romanow, A.; Weinrib, A.; Zhang, L. Resource ReSerVation Protocol (RSVP)—Version 1 Applicability Statement Some Guidelines on Deployment. In *RFC 2208*; IETF: Fremont, CA, USA, 1997.
22. Nichols, K.; Blake, S.; Baker, F.; Black, D.L. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. In *RFC 2474*; IETF: Fremont, CA, USA, 1998.
23. Blake, S.; Black, D.L.; Carlson, M.A.; Davies, E.; Wang, Z.; Weiss, W. An Architecture for Differentiated Services. In *RFC 2475*; IETF: Fremont, CA, USA, 1998.
24. Group, I..W.; others. *IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks*; Technical Report; Technical Report Std 802.1 Q-2018; Revision; IEEE: New York, NY, USA, 2018; 1993p.
25. Faucheur, F.L.; Wu, L.; Davie, B.; Davari, S.; Vaananen, P.; Krishnan, R.; Cheval, P.; Heinanen, J. Multi-Protocol Label Switching (MPLS) Support of Differentiated Services. In *RFC 3270*; IETF: Fremont, CA, USA, 2002.
26. Heinanen, J.; Guerin, R. A Single Rate Three Color Marker. In *RFC 2697*; IETF: Fremont, CA, USA, 1999.
27. Baker, F.; Fairhurst, G. IETF Recommendations Regarding Active Queue Management. In *RFC 7567*; IETF: Fremont, CA, USA, 2015.
28. Day, J. The Interina Reference Model. 2014. Available online: https://github.com/PouzinSociety/RINASpecifications (accessed on 30 May 2020).
29. Ongaro, D.; Ousterhout, J.K. In search of an understandable consensus algorithm. In Proceedings of the USENIX Annual Technical Conference, Philadelphia, PA, USA, 19–20 June 2014; pp. 305–319.
30. Ongaro, D. Consensus: Bridging Theory and Practice. PhD Thesis, Stanford University, Stanford, CA, USA, 2014.