Bandwidth-driven Flow Allocation Policy for RINA

Michal Koutenský
Dept. of Information Systems
Brno University of Technology
Brno, Czech Republic
koutenmi@fit.vutbr.cz

Vladimír Veselý
Dept. of Information Systems
Brno University of Technology
Brno, Czech Republic
veselyv@fit.vutbr.cz

Vincenzo Maffione
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Pisa, Italy
vincenzo.maffione@ing.unipi.it

***Abstract*—The assignment of available network bandwidth to application flows is a typical resource allocation problem. Although the nature of this problem appears simple, the inherent dynamism of networks makes it hard to achieve. The TCP/IP protocol suite lacks the capability to flexibly reserve bandwidth based on the available network resources. We investigate flow allocation with guaranteed bandwidth on the Recursive InterNetwork Architecture, which allows the deployment of arbitrary resource allocation strategies thanks to its inherent programmability. We present a distributed Flow Allocator policy that respects the bandwidth requirements of user applications. We have implemented and evaluated this policy in the frame of a RINA open source implementation.**

***Index Terms*—RINA, TCP, congestion control, flow allocation, rlite**

I. Introduction

Internet as a communication platform should be able to accommodate traffic from different kinds of devices, users and processes. The Internet backbone should be able to recognize and treat accordingly various types of traffic. However, this is easier said than done. With so many heterogeneous applications (e.g., desktop vs. mobile vs. IoT) and deployment scenarios (e.g., LAN vs. WAN vs. data-center), Quality of Service (QoS) becomes an even more important trait for the network operation than before.

The TCP/IP protocol suite of (non-)cooperating layers does not have a full-fledged solution providing support for QoS. Data-link and network layer protocols (such as Ethernet and IP) support: (a) best-effort delivery, which does not give guarantees; (b) integrated services, which suffers from non-scalable signaling; (c) differentiated services, which is based on static resource allocation. Transport layer protocols provide end-to-end congestion-management and flow control at best. However, congestion is a natural outcome of any system missing a complete QoS architecture. None of above-mentioned techniques offers true QoS-aware functionality in the sense of RFC 2990 [1].

Differently from TCP/IP, RINA design considers QoS in all relevant components, as detailed in the specifications [2]:

- Application Process, which can specify the QoS parameters (e.g., average/peak bandwidth, delivery order, maximum delay, maximum jitter, maximum bit error rate) necessary for the application to provide a satisfying service experience;
- Flow Allocator, which translates requested QoS parameters into a set of QoS-cubes and governs flow life-cycle (e.g., permit/deny flow allocation based on available resources);
- Resource Allocator, which coordinates components and enforces various policies to guarantee QoS targets (e.g., shape data throughput according to changing conditions).

There are two approaches for resource control: (a) a proactive approach (e.g., conditional flow allocation) tries to avoid congestion or critical depletion of resources; and (b) a reactive approach such as transport layer protocols (e.g., TCP, SCTP) manages the congestion upon its occurrence. The RINA layer, also known as Distributed InterProcess Communication Facility (DIF) has all the information needed to check whether it has enough capacity to guarantee bandwidth (as one of the QoS parameters) for a new flow on the whole path from source to destination. With this ability, allocated flow can be accepted or rejected to prevent congestion or critical depletion of resources. Our research aims at investigating the proactive approach.

Multiple RINA implementations are currently available: IRATI [3] and rlite [4] (full RINA stack implementations), Ouroboros [5] (RINA-inspired decentralized packet network), ProtoRINA [6] (Java-based prototype) and RINASim [7] (OMNeT++ simulation framework). This paper contributes the design of a proactive bandwidth reservation scheme for flow allocation and its implementation as a Flow Allocator policy for the rlite stack.

This paper is organized as follows. Section II introduces relevant theory behind resource reservation and flow allocation in RINA. Section III outlines the relevant software components of the rlite implementation. Our contribution is evaluated in Section IV, which describes the validation and verification tests that were conducted. Future research developments in this area are outlined in Section V; Section VI summarizes the contents of the paper.

II. Theory

Resource reservation is a common problem when operating a system with limited capacities. This section outlines such problem in the frame of TCP/IP and RINA.

A. Resource Reservation

Current IP-based computer networks can use either integrated or differentiated services in order to provide QoS to

978-1-7281-5127-4/20/$31.00 ©2020 IEEE

applications. The main difference between them is that first one enforces QoS explicitly on every flow, whereas the second one implicitly groups flows onto more coarse QoS classes.

Integrated services (IntServ, [8]) leverages Resource Reservation Protocol (RSVP, [9]), 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 the 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 seconds) sends state refresh messages, for both path and also 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 [10]. 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 (DiffServ, [11], [12]) 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.

B. *Flow Allocator*

This section contains a few references to the RINA theory and terminology. In-depth information is available in the specification [2] and the RINA book [13].

An IPC Process (IPCP) is an instance within a DIF, which allows the computing system to do IPC with other DIF members. Each IPC process performs (secure/reliable) data transport, (authenticated) enrollment, (de)allocation of resources, routing, management, etc. IPC Processes provide APIs to the upper layers, which can request its services. The basic IPCP API offers four operations: *allocate*, to allocate communication resources; *deallocate*, to release previously allocated resources; *send*, to pass SDU to an IPCP for transmission; and *receive*, to retrieve SDU received from an IPCP.

The Flow Allocator (FA) processes allocate/deallocate IPC API calls and further management of all IPCP flows. A Flow Allocator Instance (FAI) is created upon *allocate request*, and it manages a given flow, including its QoS parameters, for its whole lifetime. The FAI handles error and flow control while managing a single flow's connection. Upon successful flow allocation, the FAI returns a port identifier to the requesting application. The port identifier can be used as a handle to reference the flow. The FAI maintains a mapping between flows local port and connection endpoint identifiers. The FA can be programmed with multiple policies, to specify how to evaluate access control rights, or how to translate flow QoS requirements into QoS-cubes.

III. Implementation

The bandwidth reservation system has been implemented within the *rlite* [4] project, a Free and Open Source RINA implementation for GNU/Linux systems. The rlite software is composed of a set of kernel-space loadable modules (KLMs) and user-space components. The main kernel modules provides: (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. The Abstract Factory design pattern is used to support different types of IPCPs, such as the normal IPCP, the shim IPCP over Ethernet, or the shim IPCP over UDP, with a separate kernel module for each IPCP type. As an example, the Data Transfer and Data Transfer Control functionalities of the IPC Process (i.e., the DTP and DTCP protocols) are implemented as a part of the normal IPCP kernel module. The integration with the Linux network device layer is implemented within the shim IPCP over Ethernet. The packet I/O interface is provided through a special character device exposed by the kernel. A second character device provides the other management and control functions.

The management functions of the normal IPCP are implemented in the context of the *uipcp* user-space daemon process. Those functions, such as routing, flow allocation, application registration, and enrollment, are provided by a separate thread for each IPCP. Despite providing the same services, the behaviour of each management function can be customized by modifying its *policies*. Policies make the RINA stack programmable, so that it can adapt to the environment where it is deployed. The rlite stack allows policies to be changed at run-time.

A. Flow Allocation With Support for Bandwidth Reservation

This paper contributes the implementation of a bandwidth reservation scheme for rlite, in the form of a set of policies. The goal is to guarantee that a given amount of bandwidth will be available to each flow for its whole lifetime. Upon flow allocation, the flow requestor specifies the amount of bandwidth that it needs to provide a proper service. In response, the Flow Allocator layer management function will try to meet this request if the current network resources allow it. Otherwise, the flow allocation request is denied. This approach is substantially different from how flow allocation works in TCP/IP, where resources are allocated on a best-effort fashion. The policy assumes a single DIF configuration on top of eth shims.

In our proposal, if a flow does not specify the requested bandwidth it receives a default (configurable) amount. No flow is allowed to use more than its allocated share, even if there currently exists unused bandwidth in the network. This constraint is enforced in a cooperative manner, by means of sender rate-limiting implemented in the normal IPCP kernel module. With this approach, the Flow Allocator is able to track the total used bandwidth in the network — and, by extension, the free available bandwidth.

By denying flow allocation requests that would exceed the capabilities of the network, and enforcing allocated bandwidth shares, network congestion can be fully avoided. Such a *proactive* approach of congestion control is reminiscent of circuit-switched networks, such as ATM.

In addition to the Flow Allocator, it is also necessary to provide a policy for the Routing function, so that each flow can be routed through the network using a dedicated route, as detailed in the next section.

B. Bandwidth-aware LFDB

The default Routing in rlite uses a Link-State algorithm. Neighbor IPCPs in the DIF exchange Link State Advertisment (LSA) messages with each other. The LSAs are stored in a Lower Flow Database (LFDB), which is then turned into a graph on the network that is used to compute shortest path (with the Dijkstra algorithm). To support bandwidth reservation, the default Routing has been extended to add available bandwidth information to each N-1-flow in the graph.

The bandwidth of Ethernet N-1-flows can be queried in Linux using `SIOCETHTOOL ioctl()` commands. We start by observing that shim IPCPs over Ethernet are usually the lowest DIFs in the system, so that interface speed can be queried during its creation. It is then easy to propagate the bandwidth information to upper DIFs during enrollment, and to neighbor IPCPs within the DIF using LSA messages.

The LFDB with bandwidth attributes can provide the Flow Allocator with the information required for the decision process. Given IPCPs A and B and bandwidth *B*, the Flow Allocator needs a method to find a path between A and B, where each edge (N-1-flow) has at least *B* units of available bandwidth, if such a path exists. If several such paths exists, it is desirable to find the one with the shortest number of hops.

This graph problem can be solved with a modified Breadth-First Search (BFS), that includes an additional check for flow capacity. The benefit of using BFS is that it ensures that shortest path — if such a path exists — is selected. The algorithm is described in Alg. 1, and it has a worst case run time with is linear in the number of edges. More specifically, the algorithm is $O(2 * E)$, as we might iterate over all the edges twice in case of a linear graph.

Algorithm 1 Algorithm for finding the shortest path of a given capacity

```
Input: a flow network $G = (V, E, s, t, cap)$ and a required flow $B$
Output: a viable path $P$
  Let $found := False$
  Let $q$ be a queue
  Push $s$ to $q$
  Let $pred$ be an array holding the preceding edge on the current path
  while not $found \wedge q$ is not empty do        ▷ Find the shortest viable path
      Retrieve $w$ from $q$
      for all $(u, v) \in E : u = w$ do
          if $pred[v] = null \wedge (v, u) \neq pred[u] \wedge cap(u, v) \geq B$ then
              $pred[v] := (u, v)$
              Push $v$ to $q$
  if $pred[t] \neq null$ then                        ▷ Build the path
      Let $P := \{t\}$
      for all $(u, v) \in pred$ starting from $pred[t]$ do
          Prepend $u$ into $P$
          Continue to $pred[u]$
```

On flow allocation requests, the Flow Allocator asks the Routing component to run this algorithm and find a suitable path.

C. Distributed Flow Allocation Process

In the default rlite flow allocation procedure, only the source and the target IPCPs participate in the process. The source application contacts the target application, which will either accept or reject the flow. As a result, each flow allocation process in the network can be carried out independently of the others, and no DIF-global coordination is necessary. Conversely, the bandwidth reservation scheme does require DIF-global coordination.

The notion of DIF-global state (e.g., used and available bandwidth), makes it necessary to coordinate the flow allocation process to keep the DIF in a consistent state. Without coordination, applications in the DIF may try to concurrently allocate more bandwidth than it is actually available, as they would all start from an initial situations with no bandwidth allocated, and be unaware of the other pending allocations.

To solve the problem we introduce a distinguished entity, the coordinator, that helps avoid race conditions resulting from the inherent parallelism of the flow allocations. The coordinator acts as a mediator between the two flow endpoints, and as a guardian of the global state. Only the coordinator is allowed to access or update the global state, effectively acting as a *monitor*, and all flow allocations must be approved by it, updating the state in the process. Since the coordinator handles all the pending allocations, it can include the pending bandwidth reservations as unavailable as far as the following requests as concerned. If the target application rejects a flow allocation request, the associated bandwidth is reclaimed and can be reused by subsequent requests.

A drawback of this approach is that the coordinator is a single point of failure within the DIF. If it becomes unavailable, no new flows can be allocated. To overcome this limitation, the function of the coordinator should be distributed across a *cluster* of IPCPs. In the case of a failing IPCP, others in the cluster will have all the necessary information to keep the flow allocation service available.

The coordinator can be implemented as a fault-tolerant replicated state-machine with the help of a *consensus* algorithm such as Paxos [14] or Raft [15]. One of the nodes in a Raft cluster is elected as a *leader*. The leader is in charge of actively applying changes to the replicated state machine. The other nodes act as *followers*, as they passively synchronize their replica of the state-machine with the leader. In case of leader fault, the surviving replicas elect a new leader to fulfill the role. The Raft algorithm is available in rlite as an internal library, so that it can be used as a general fault-tolerance algorithm for implementing policies. A consensus algorithm provides the robustness and reliability of a distributed system while allowing the developers to reason about behaviour in a logically centralized manner, greatly simplifying the design.

Raft is based on an *append-only log*. The leader appends to the log every operation that clients ask to perform on the replicated state machine. Log entries then get distributed to the followers. By applying the actions in the log, each replica is able to eventually arrive at the same state as the leader. However, appending an entry does not mean it gets applied — the leader decides to commit an entry only after it is sure that the entry has been replicated to a majority of nodes.

D. Per-Flow Routing

The last component missing in the picture is a reservation-aware Routing process. None of the mechanisms described above change how the traffic moves through the network; they all work on a local representation of the network. With our bandwidth reservation it is no longer possible, in general, to route all the PDUs destined to the same address in the same way. Each allocated flow can be routed differently, so that its PDUs follow only the path with reserved bandwidth. This result has been achieved by: (i) extending the forwarding table logic to be able to take into account more than just the destination address; and (ii) letting the coordinator install per-flow routes along the path during the flow allocation process.

E. The Whole Is Greater Than the Sum

By combining all of the elements described in the previous sections, we are now able to build a set of policies that provide bandwidth guarantees to allocated flows. Two interdependent `bw-res` policies have been implemented in rlite; one for the Flow Allocator and one for the Routing component. The rlite policy system has been extended in such a way to take into account this dependency, so that the two policies can only be loaded together, and never separately.

The process starts with the Flow Allocator receiving a flow allocation request from a local application. The request is forwarded as is to the current Raft leader, which uses this information to create a *Flow-id* uniquely identifying the flow. The leader then invokes the Routing component to find a network path with enough bandwidth to satisfy the request. If no suitable path is found, a negative response is sent back to the source Flow Allocator, and then forwarded to the requesting application. If a viable path does exist, the leader preemptively reserves it, appending the corresponding entries (one per link) to the Raft log. The request is finally delivered to the target IPCP only after the final entry has been committed. The target Flow Allocator forwards the request to the target application; the response, whether positive or negative, is returned to the Raft leader.

If the response is positive, the Raft leader proceeds with distributing the necessary per-flow routes across the reserved path. In any case, the response is forwarded to the source Flow Allocator and then delivered to the requesting application.

The flow deallocation procedure is similar: the Raft leader acts as proxy for the request, freeing up the reserved bandwidth and removing the per-flow routes.

The Flow Allocator `bw-res` policy exposes two tunable parameters, in addition to the ones that control the Raft behaviour. The first one controls what happens to flow allocation requests that do not specify a bandwidth value. The first option is to simply reject such requests immediately. The alternative is to assign to them a predefined bandwidth value, which is controlled by the second parameter.

IV. Testing

The implementation has been tested in two parts. The first part covers the bandwidth reservation graph algorithm and verifies that it produces correct output. The second part describes a stress test over an example topology to measure overall bandwidth utilization.

A. Finding a Viable Path

Given an input graph with link capacities and a desired source-destination-bandwidth triple, the algorithm should return *any* shortest path if such a path exists.

Figure 1 shows a simple cyclic graph of 4 nodes with non-uniform edge capacities. We can see that the algorithm

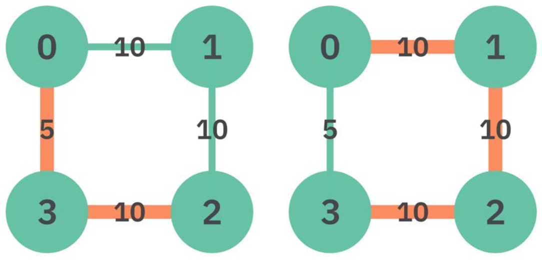

Figure 1. *Left:* Flow from 0 to 2 with capacity 4. *Right:* Flow from 0 to 3 with capacity 7. Found path in orange.

correctly finds a path that satisfies the capacity condition instead of returning just the shortest path. The algorithm does not try to do any optimization — in the first example, it might be better to take use the *0-1-2* path, as it leaves more surplus bandwidth along the path for future flows.

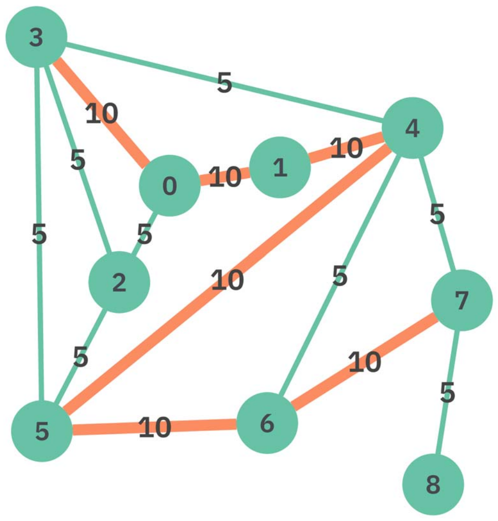

Figure 2. Flow from 3 to 7 of capacity 7. Found path in orange.

Figure 2 displays a more complicated graph with multiple possible paths between non-adjacent (and even some adjacent) nodes. The algorithm correctly finds the viable path, despite it being twice as long as the shortest possible path connecting those two nodes.

B. Bandwidth utilization

Bandwidth utilization was measured by running stress tests on multiple topologies. Each link had a capacity of 10Mbps and flows of 1Mbps were allocated at random. Snapshots of link utilization were taken at regular intervals, as can be seen in Figures 3 and 4. Every node is connected to the network with at least two links. Note that single-homed end stations are not interesting in this scenario, as there are no alternative links to be used for communicating with them to showcase how the policy works.

Both figures show a similar pattern. Link saturation is highest — and begins — in the center which interconnects different

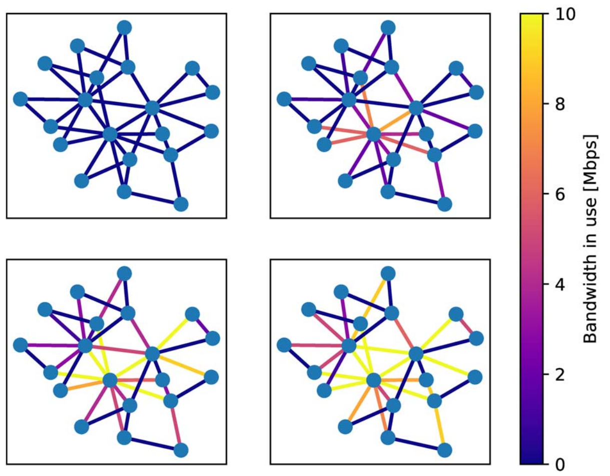

Figure 3. Timelapse of network link utilization by allocating flows at random on the first test topology. (Snapshot every 50 allocations)

parts of the network. As time progresses and new flows get allocated, saturation spreads to further edges of the network. This agrees with the intuitive observation that as more flows are created, the backbone of the network gets more strained. The load among non-saturated edge links is distributed quite evenly, with links closer to the center having higher utilization.

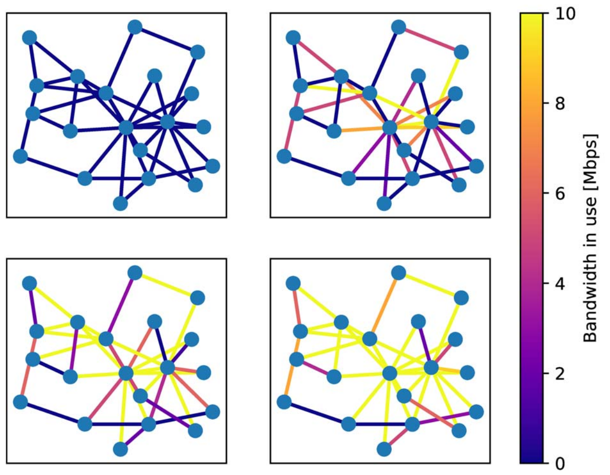

Figure 4. Timelapse of network link utilization by allocating flows at random on the second test topology. (Snapshot every 100 allocations)

An interesting observation to be made regarding the networks presented is that certain parts of the network, which could be considered edge branches, can be used to loop from the core back at a further point if the direct route has been saturated. As the bandwidth reservation policy is a form of multi-path routing, such edge links can be used to squeeze more bandwidth from the network at the cost of possibly increased latency and less predictable routes. This approach equalizes some of the asymmetry of current networks, where backbone links serve a distinguished role and require speeds orders of magnitude larger than links connecting end-users to the network.

V. Future work

We have identified several areas where further experimentation could be done to potentially improve performance.

As it stands now, once a flow is allocated it uses the same path for the whole lifetime. If one of the nodes along the reserved path fails, the flow effectively becomes unusable because the per-flow routes are never changed. The FA could react to link state changes and try to reroute the flow.

Another way to increase network utilization is to conditionally reroute active flows to use better paths and distribute the traffic more evenly. Besides significantly complicating the flow allocation process, this approach opens itself to possible DoS attacks and could lead to negatively impacting the overall stability of the system.

Flows which do not specify a bandwidth requirement get assigned a pre-defined amount. In some scenarios, this can be needlessly restrictive; it would instead be better to dynamically divide the surplus bandwidth between such flows so that they can fully utilize the available resources. This requires updating the rate limits on each new flow allocation, and while not optimal — as it assumes all unlimited flows could use the same amount of bandwidth — it is easy to implement.

A different approach would be to limit each of these unlimited flows to the whole surplus bandwidth and let a flow control mechanism divide the bandwidth. While potentially better adjusting to the needs of these flows, it might negatively impact flows with reservations, as the assumption about bandwidth in use never being greater than the available bandwidth could be broken occasionally while the flows adjust.

Another extension of our work would be to use multi-path routing not only at the destination level but also at the flow level; that is, we could split the flows into several sub-flows to achieve the required bandwidth. Aside from the additional complexity, there are specific issues to consider — as the paths might vary in their characteristics (latency, number of hops, etc.), problems such as packet reordering or head-of-line blocking could occur, negatively impacting the performance.

Orthogonal enhancements can be done to the graph algorithm, such as taking the path bottleneck into account when given multiple shortest path options.

VI. Conclusion

In this paper we have demonstrated a proactive approach to avoiding congestion in the network and providing a quality of service assurance to applications. Building on RINA's separation of mechanisms and policies, we have implemented a set of policies that allow reserving bandwidth for flows for the duration of their whole lifetime.

The Flow Allocator engages in a distributed computation managing the creation of new flows. By leveraging the Raft consensus algorithm, the implemented solution can be thought as being logically-centralized while being fault tolerant.

Each flow request carries with it an expected average bandwidth; the FA, before responding to the request, looks for a path that satisfies the bandwidth constraint. If such a path is found, the requested bandwidth gets reserved for the new flow. If no viable path exists, the request gets denied.

By denying flows that would exceed the capacity of the network, the possibility of congestion occurring is eliminated. To ensure that flows only use paths that they have reservations for, routes which forward packets differently for each flow are installed along the path. As such, flows cannot be automatically rerouted as a result of link failure. This is one area where further improvements are possible, as discussed in Section V. Another noteworthy problem is cooperation with flows which do not require bandwidth reservation, to which multiple solutions have been proposed.

The allocation graph algorithm has been tested to verify that it is able to find paths meeting the requested criteria. Experiments carried out with the implemented policies show that allocation requests for flows that could cause congestion are denied and that it is possible to achieve fairly high network utilization. The policy distributes link utilization evenly across the network, with the utilization being highest in the interconnecting core and declining further towards network edges.

Acknowledgment

This article has been supported by Dept. of Information Systems belonging to Faculty of Information Technology. Authors would like to thank the author and the contributors of rlite project for their effort.

References

[1] G. Huston, "Next steps for the ip qos architecture," Internet Requests for Comments, RFC Editor, RFC 2990, November 2000.
[2] Pouzin Society, "Rina specifications," https://github.com/PouzinSociety/RINASpecifications, 2019.
[3] "IRATI." [Online]. Available: https://irati.github.io/stack/
[4] V. Maffione and M. Koutenský, "rlite: A light rina implementation," 2019. [Online]. Available: https://github.com/rlite/rlite
[5] "Ouroboros." [Online]. Available: https://ouroboros.rocks/
[6] "ProtoRINA." [Online]. Available: https://csr.bu.edu/rina/protorina/2.0/
[7] "RINASim." [Online]. Available: https://rinasim.omnetpp.org/
[8] B. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture: an overview," Internet Requests for Comments, RFC Editor, RFC 1633, June 1994.
[9] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource reservation protocol (rsvp) – version 1 functional specification," Internet Requests for Comments, RFC Editor, RFC 2205, September 1997.
[10] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O'Dell, A. Romanow, A. Weinrib, and L. Zhang, "Resource reservation protocol (rsvp) – version 1 applicability statement some guidelines on deployment," Internet Requests for Comments, RFC Editor, RFC 2208, September 1997.
[11] K. Nichols, S. Blake, F. Baker, and D. L. Black, "Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers," Internet Requests for Comments, RFC Editor, RFC 2474, December 1998.
[12] S. Blake, D. L. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Internet Requests for Comments, RFC Editor, RFC 2475, December 1998.
[13] J. D. Day, *Patterns In Network Architecture: A Return to Fundamentals*. Prentice Hall, 2010.
[14] L. Lamport, "The part-time parliament," in *ACM Transactions on Computer Systems 16, 2*, 1998, pp. 133–169.
[15] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, 2014, pp. 305–319.