

# Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows in the Real-time Data Plane

Jonathan Falk, Heiko Geppert, Frank Dürr, Sukanya Bhowmik, Kurt Rothermel  
Institute for Parallel and Distributed Systems, University of Stuttgart, Germany  
{firstname.lastname}@ipvs.uni-stuttgart.de

**Abstract**—Many networked applications, e.g., in the domain of cyber-physical systems, require strict service guarantees for time-triggered traffic flows, usually in the form of jitter and latency bounds. It is a notoriously hard problem to compute a network-wide traffic plan, i.e., a set of routes and transmission schedules, that satisfies these requirements, and dynamic changes in the flow set add even more challenges. Existing traffic-planning methods are ill-suited for dynamic scenarios because they either suffer from high computational cost, can result in low network utilization, or provide no explicit guarantees when transitioning to a new traffic plan that incorporates new flows.

Therefore, we present a novel approach for dynamic traffic planning of time-triggered flows. Our conflict-graph-based modeling of the traffic planning problem allows for the reconfiguration of active flows to increase the network utilization, while also providing per-flow QoS guarantees during the transition to the new traffic plan. Additionally, we introduce a novel heuristic for computing the new traffic plans. Evaluations of our prototypical implementation show that we can efficiently compute new traffic plans in scenarios with hundreds of active flows for a wide range of settings.

**Index Terms**—traffic planning, QoS, reconfiguration, time-triggered

## I. INTRODUCTION

Real-time communication with guarantees on delay and jitter has become an essential requirement for implementing time-sensitive networked systems from application domains such as manufacturing (Industrial Internet of Things), the automotive domain, or any kind of cyber-physical system, where physical processes are controlled through networked sensors, actuators, and controllers. For a long time, networking technologies for such time-sensitive systems have followed another trajectory than “traditional” computer networks resulting in field-buses and other distinct, commercial networking standards that were tailored to the specific real-time requirements for so-called operational technology (OT). However, to save cost and to benefit from the fast evolution of wide-spread networking technologies such as Ethernet, we observe a strong trend to merge these two networking domains. This trend manifests itself in activities of standardization bodies, namely, the IETF Deterministic Networking (DetNet) [1] initiative or IEEE Time-Sensitive Networking (TSN).

For example, TSN has brought several extensions that equip IEEE Std. 802.3 (Ethernet) LANs with mechanisms allowing for

deterministic bounds on network delay and jitter. In particular, the Time-Aware Shaper (TAS) (cf. IEEE Std. 802.1Q [2]) provides a mechanism to schedule the release times of frames in the bridges in the network. In combination with explicit routing, this allows to enforce a network-wide traffic plan consisting of routes and schedules for each individual flow that provides hard real-time guarantees for time-triggered flows. Here, we use the term time-triggered flow as the umbrella term for isochronous traffic or cyclic-synchronous traffic, which are traffic types originating in field-level OT with cycle times and jitter and latency bounds in the range of micro-seconds.

Computing such a traffic plan is well-known to be a (NP-)hard problem, for instance, related to the Job Shop Scheduling Problem [3]. So far, the traffic planning problem has predominantly been addressed for static, a priori known sets of flows with a focus on scalability to support larger scenarios.

However, a second general trend which is postulated both in industry white-papers [4], [5] and research articles [6], [7], [8] is a general movement away from static environments towards dynamic reconfigurable scenarios, e.g., when attaching or physically moving a sensor or machine in a smart factory. The term *plug-and-produce* is frequently used in the context of Industry 4.0 to describe the need for flexible production facilities where devices can be added quickly and configured automatically, which also includes the adaption of network schedules and routes to integrate new devices into the network.

Software-defined Networking (SDN) introduced the fundamental concept of a logically centralized network controller with a global view onto the network, in particular, to dynamically adapt routes. TSN similarly provides the concept of a centralized network controller (CNC) and associated management abstractions to automate network configurations including schedules. Thus, the basic primitives for dynamically changing routes and schedules of nodes in a softwarized time-sensitive network exist. However, algorithms for *traffic planning* that such a controller has to provide are still subject to research.

This applies especially to traffic planning that supports dynamic scenarios where flows can be added dynamically—which is more than just “static planning done fast”. In addition to the general challenges of network updates [9], we also have to consider the quality-of-service (QoS) requirements of the time-triggered flows during the transitions between traffic plans. For example, if we naively execute static traffic planning again from scratch each time new flows are to be added, this

This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 285825138.

provides no control over the service degradation experienced by an individual flow, and usually we cannot even guarantee that the new plan still includes all former active flows. Worse yet, we might even have to suspend the emission of new packets for some time to ensure that there is no interference induced by old packets—resulting in a “stop-and-go”-reconfiguration, or, if technically possible, we have to drop old packets. All of this boils down to the insight that dynamic traffic planning requires taking the current traffic flows into account when computing a new traffic plan.

In early approaches [6], [10] this is addressed by what amounts to defensive planning: Defensive planning does not change the configuration of active flows in order to add new flows. In other words, defensive planning takes the configuration of active flows as fixed and uses the remaining network resources for routing and scheduling new flows. While defensive planning never affects the quality of service (QoS) of active flows, it might utilize network resources in a sub-optimal way. For example, defensive planning might result in scenarios where new flows have to be rejected simply because the active flows have “fragmented” the remaining network resources, and we cannot “correct” the past scheduling and routing decisions.

Obviously, this issue can be mitigated by what amounts to offensive planning where (some) active flows can be reconfigured when adding new flows. However, this introduces additional challenges: While offensive planning allows for better utilization of network resources, the transitory effects of reconfigurations possibly introduce short-term QoS degradation [11], and the transition phase has to be temporally bounded to guarantee deterministic communication in the long run [12]. This means, offensive planning makes only sense if we can control both, the degree and duration of a QoS degradation, and it requires applications that can tolerate controlled fluctuations of QoS. Such applications, for instance, correspond to the cyclic-synchronous traffic pattern where the emphasis is on latency and some jitter is acceptable, cf. the upcoming IEEE/IEC 60802 TSN profiles [13].

In the literature, reconfigurations of time-triggered flows are addressed in [11], [14]. The approach in [11] considers the problem how to execute an update from one given, current traffic plan to another given, new traffic plan such that the number of lost packets is minimized. This is later extended with a method [14] for the computation of the new traffic plan, more specifically, a new schedule. The problem of service degradation is only addressed on the level of packet losses, i.e., the computation of a new traffic plan is subject to constraints which ensure that a loss-free packet update is possible.

In contrast, our traffic planning approach considers both, the temporal (scheduling) and spatial (routing) aspects. We also address the challenges of offensive planning for cyclic-synchronous traffic flows by providing a method to satisfy per-flow QoS degradation requirements in terms of packet timings (not just no packet drops). In detail, the contributions of this paper are twofold:

- We present a novel approach for dynamic traffic planning for isochronous and/or cyclic-synchronous traffic flows using the zero-queuing principle [3], [15]. Our approach supports offensive traffic planning where the transitions

from the old traffic plan to the new traffic plan, i.e., a traffic plan update, do not require artificial pauses of sender nodes or dropping packets, and our approach supports limiting the QoS degradation during the update according to per-flow user-specified bounds. This allows for a mixed-operation mode where a subset of active flows (e.g., of jitter-resilient applications) can be reconfigured, while the remaining active flows (e.g., of jitter-sensitive applications) are exempt from reconfigurations.

- We use a conflict-graph-based method [16] for the traffic planning. Conflict-graph-based modeling is well-suited for dynamic traffic planning because the conflict graph embeds a large portion of the knowledge about the solution space from the previous traffic plan, and thus reduces the effort required to compute the new traffic plan. Computing a new traffic plan requires to solve a variant of the independent colorful vertex set problem with weighted colors. We present a novel heuristic for the underlying optimization problem which we have to solve to obtain a new traffic plan. Our evaluations show that our novel heuristic can solve this problem for hundreds of flows in a fraction of the time required by a state-of-the-art optimizer.

Following the related work discussion in Sec. II, we introduce our system model, modeling concepts, and problem statement in Sec. III-IV. We describe the traffic planning approach in Sec. V-VIII, evaluate our approach in Sec. IX and conclude the paper in Sec. X.

## II. RELATED WORK

In this paper, we address dynamic traffic planning for isochronous or cyclic-synchronous traffic for packet-switched, meshed networks with hard real-time requirements in terms of end-to-end latency and jitter. In particular, we focus on time-triggered approaches that rely on explicitly scheduled transmissions throughout the network. This means, approaches that use a combination of traffic shaping and admission control [17] are considered out of scope, since they are more suited for jitter-tolerant applications with asynchronous traffic patterns. Work that is primarily concerned with aspects of interfaces or system architectures [8], [18], [19], [20] addresses a complementary problem, and usually relies on traffic planning functionality.

TSN and similar converged networking technologies such as TTEthernet (SAE AS6802), or Time-Sensitive Software Defined Networks (TSSDN) have sparked many traffic planning approaches for scheduling time-triggered isochronous and cyclic-synchronous traffic in static scenarios. These can be roughly classified into two categories: 1) approaches that rely on generic constraint programming frameworks such as Integer Linear Programming (ILP) [21], [22], [22], [23], [24] or Satisfiability Modulo Theories (SMT) [25], [26], [27], and 2) approaches that use custom heuristics [3], [28], [29] or a combination of both [16]. Out-of-the-box, most static planning approaches cannot be used for dynamic traffic planning, since—by nature—they do not account for old packets from a previous traffic plan which results in the timing violations during the transition from one traffic plan to another.

In principle, incremental approaches [6], [10] can be used for static scenarios and defensive planning in dynamic scenarios: “Freezing” the configurations of active flows ensures that “nothing bad” happens when new flows are added, because the configurations of active flows from each previous step remain immutable for the future. However, defensive traffic planning prohibits adjustments of any past decision that may turn out to have been sub-optimal with respect to schedulability when facing new flows. This can result in low network utilization if a flow that has been added early on obstructs new flows.

This could be overcome by offensive planning that allows for the reconfiguration of active flows. Reconfigurations of time-triggered flows are addressed in [11], [14]. An enhancement for two-phase network updates [9] with schedule-aware per-flow update times was presented in [11]. The proposed update mechanism aims to reduce packet loss—caused by missed timings due to the reconfiguration—when transitioning from the current traffic plan to a given, new traffic plan. While [11] considers traffic planning as external step, [14] presents an ILP-based approach for the computation/modification of schedules which eliminates the possibility of missed timings altogether. Two algorithms are described in [14] that both allow for packet-drop-free reconfigurations of active flows (offensive planning), by either computing the new schedule directly, or modifying an otherwise obtained, given schedule.

In contrast, we consider scheduling *and* routing when computing the new traffic plan. Our modeling of the problem and our novel heuristic are more efficient than constraint-based approaches because we can reuse most of the state which encodes the solution space from the previous step. From a performance perspective, our approach therefore is a step towards true “online”-reconfiguration capabilities in terms of runtimes.

So far, the existing approaches provide service guarantees during a reconfiguration either by prohibiting reconfigurations (defensive planning) or only in terms of packet drops rather than timing guarantees. In contrast, our approach is more powerful and allows for incorporating application-specific *per-flow* bounds on the (temporary) QoS degradation in terms of jitter and packet reorderings, i.e., our approach enables offensive traffic planning for time-triggered flows with heterogeneous QoS requirements, e.g., wrt. jitter-tolerances.

### III. SYSTEM MODEL

We consider traffic flows in packet-switched data networks. A traffic flow is a sequence of packets that is transmitted from a source node to a destination node through the network. The network consists of infrastructure nodes (switches, routers), source nodes, destination nodes, and a logically centralized controller. Infrastructure nodes forward data packets via point-to-point links which connect nodes. Source node and destination node are the origin and target of the packets of a flow, respectively. Packets are sent and forwarded according to the global *traffic plan*. The traffic plan is computed and disseminated by the planner, which is situated at the logically centralized controller and has global view onto the network.

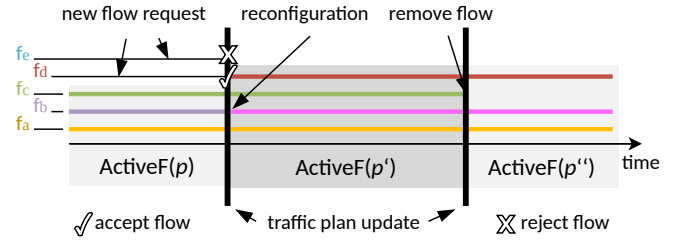


Figure 1: The planner computes a sequence of traffic plans. During the first traffic plan update, out of the two flow requests ( $\text{ReqF}(p') = \{f_d, f_e\}$ ), only  $f_d$  can be admitted (which requires a reconfiguration of  $f_b$ ) and  $f_d$  is added to the set of active flows  $\text{ActiveF}(p')$ , while  $f_e$  could not be scheduled and is rejected. The second update removes  $f_c$ , which is not needed by the application anymore.

#### A. Flow Dynamics

Whenever a set of new flows is to be added to the network, the planner computes a new traffic plan. Therefore, over time the planner computes a sequence of traffic plans, where a newly computed traffic plan becomes the current plan, which replaces the old one, cf. Fig. 1. A traffic plan, say  $p$ , defines for the set of flows admitted to the network when and along which routes the packets of the active flows are transmitted.

We will use  $\text{ActiveF}(p)$  to denote the set of active, i.e., admitted flows of plan  $p$ . For example, in Fig. 1,  $\text{ActiveF}(p) = \{f_a, f_b, f_c\}$ ,  $\text{ActiveF}(p') = \text{ActiveF}(p) \cup \{f_d\}$ , etc. Active flows can also be removed, e.g.,  $f_c$  in Fig. 1, which means that the source node stops sending packets and the planner is notified that it can reassign the resources previously reserved for that flow when computing the next traffic plan update. Removing active flows is trivial from the perspective of the planner, cf. Sec. VI-B. In the following, we therefore concentrate on the case that the planner only attempts to add new flows to the network.

Let  $\text{ReqF}(p)$  be the set of flows that the applications request to be added to the network, i.e.,  $\text{ReqF}(p)$  are the requests received by the planner while  $p$  is valid. By processing this request, the planner generates a new traffic plan  $p'$  with  $\text{ActiveF}(p') \setminus \text{ActiveF}(p) \subseteq \text{ReqF}(p)$ , where  $p$  is the plan preceding  $p'$ . In the following, we denote the current traffic plan by  $p$  (which becomes the old traffic plan after the update), and the new traffic plan by  $p'$ . For example, in Fig. 1,  $\text{ReqF}(p) = \{f_d, f_e\}$  and  $\text{ReqF}(p') = \emptyset$ .

In general, it can happen that a flow in  $\text{ReqF}(p)$  may not be admitted (cf.  $f_e$  in Fig. 1) due to network resource limitations. However, once a flow  $f$  has been admitted into the set of active flows, the end-to-end deadline constraints and jitter-bounds, i.e., real-time constraints, are guaranteed until the end of its lifetime. This explicitly includes all future traffic plans and possible reconfigurations of  $f$ . For example,  $f_b$  in Fig. 1 is an active flow whose lifetime spans over the two depicted traffic plan updates, the first one resulting in a reconfiguration of  $f_b$ : during the whole time, the source node of  $f_b$  can send packets which are guaranteed to arrive at the destination node in time.

We focus specifically on the problem of *computing* the

traffic plan. However, ultimately our traffic planning approach is but one building block of the controller, which likely will need to incorporate additional application-specific logic and interfaces. For instance, the event that triggers the computation of a new traffic plan might depend on the application: The traffic plan update could be initiated by applications issuing a flow request or a batch of flow requests. Alternatively, the planner could collect incoming flow requests and decide itself when to compute a new traffic plan, e.g., periodically, or when the number of flow requests reaches a certain threshold. Consequently, we do not give any guarantees on the response time for flow requests. In other words, our goal is a *real-time data plane*, not a real-time control plane, although we strive for a fast processing of requests through efficient planning. We will show later that we can parameterize our approach with regard to the trade-off between runtime and schedulability.

We assume that we do not have any prior knowledge as to how the flow set changes over time, else we could pre-compute a sequence of traffic plans. In the following, the planner processes a flow request only once, i.e., either admits the new flow, or rejects it, but our approach allows for implementing other strategies, for example, retries of flow requests, too.

### B. Time-Triggered Traffic Flow

The notion of a traffic flow represents a directed communication channel from one source node to one destination node. In the following, we focus on time-triggered flows that carry time-sensitive, important information requiring deterministic QoS and no packet drops under normal operation conditions, i.e., in absence of external failures such as power-loss, physical disturbances, etc.

Starting from the time of its activation, the source node of an active flow emits one packet per transmission cycle. A transmission cycle is an interval of length  $t_{\text{cycle}}$ . Transmission cycles start at  $k \cdot t_{\text{cycle}} + t_0$ ,  $k \in \mathbb{N}$  where  $t_0$  is a common reference point in time. Source, destination, packet-size, and  $t_{\text{cycle}}$ , and end-to-end delay are flow *parameters* and do not change during a flow's lifetime.

When an application requests a flow, it specifies the parameters in the request for the planner. Conversely, the planner assigns a *route* which connects source node and destination node, and the *phase*  $\phi$  to each active flow when computing the traffic plan. The value of  $\phi$  is used by the source node to schedule its packet transmissions at  $k \cdot t_{\text{cycle}} + \phi + t_0$ ,  $k \in \mathbb{N}$ . The valid range of  $\phi$  is restricted to  $[0, t_{\text{cycle}} - t_{\text{trans}}]$  with  $t_{\text{trans}} = \frac{\text{packet size}}{\text{bandwidth}}$  being the transmission delay. Thereby, the transmission of a packet on the outgoing link at a source node is confined within the transmission cycle boundaries. Once the source node has sent a packet, the packet is forwarded by infrastructure nodes to the destination node. Unless ambiguous, we omit the flow indices of flow properties, i.e., we use  $t_{\text{cycle}}$  instead of  $t_{\text{cycle}}(f)$ .

1) *Flow Configuration*: A flow configuration is one assignment of values to the phase and route of a flow. Since a time-continuous  $\phi$  yields infinitely many flow configurations, we make some practical assumptions: As proposed in [16], we use discrete time with appropriately mapped granularity, e.g., with 1  $\mu\text{s}$  resolution.

Secondly, we also restrict the routing options for each flow to a set of candidate paths. Each candidate path must be loop-free. Additionally, we enforce the end-to-end delay constraint of a flow by capping the length of its candidate paths. In Sec. III-D, we discuss how zero-queuing provides a simple mapping of the end-to-end delay bound  $t_{\text{e2e}}$  to the path length. Depending on the scenario, e.g., a highly meshed network and “large” end-to-end delay bounds, we might still end up with impractical many candidate paths which are “short enough” to fall below the end-to-end delay induced path length limitation. Therefore, for each flow, an absolute upper-bound on the number of candidate paths ( $n_{\text{path}}$ ) can be specified to further restrict the routing options.

Upon receiving a request to add a flow, the planner then *once* computes a set of up to  $n_{\text{path}}$  candidate paths for that flow, and assigns a path identifier  $\pi$  to each candidate path of the flow. In our case, the planner uses Yen's k-shortest path algorithm [30] to compute the candidate paths.

Hence, we can define a flow configuration with a tuple of integers from a finite set.

**Definition 1** (Flow Configuration). *The tuple  $(f, \phi, \pi)$  represents a flow configuration.*

Remember, offensive planning allows for the reconfiguration of active flows while a new plan is established. Consequently, a flow may have different configurations over time. The notion  $\text{config}(f, p)$  identifies the configuration of flow  $f$  while  $p$  is valid. If the planner decides to reconfigure  $f$  in a succeeding plan  $p'$ , then  $\text{config}(f, p) \neq \text{config}(f, p')$ . We say that each reconfiguration of a flow generates a new version of this flow, where each version exists as long as the corresponding plan is valid. To be able to distinguish the various flow versions, we will use the notion  $\langle f, p \rangle$ , where flow  $f$  is associated with plan  $p$ , i.e.,  $\text{config}(f, p)$  is the configuration of  $\langle f, p \rangle$ .

### C. Node Capabilities

We consider infrastructure nodes with store-and-forward behavior and FIFO queuing semantics per output port. Infrastructure nodes provide facilities to forward packets along *explicitly* defined routes, e.g., via SDN flow tables, VLAN tagging, explicit path control in IEEE Ethernet networks [31], etc.

Source nodes support time-triggered packet scheduling, i.e., it is possible to explicitly specify the time of a packet transmission. Such functionality can be implemented in hardware, e.g., the LaunchTime-feature in commodity NICs such as Intel's I210 controllers [32], or in software, e.g., TAPRIO or Earliest TxTime First (ETF) queuing discipline in Linux. Consequently, all nodes need synchronized clocks, e.g., using protocols such as Precision Time Protocol (PTP). Support for multiple traffic classes, and time-triggered packet scheduling (e.g., TAS, TTEthernet, etc.) in the infrastructure nodes can be used to protect the scheduled transmission windows from cross-traffic due to unscheduled flows. With time-triggered scheduling support in infrastructure nodes, unscheduled traffic, e.g., best-effort traffic, can be forwarded “outside” of these scheduled transmission windows. In practice, this requires additional

guard bands to ensure that the transmission of non-preemptible best-effort packets (or framelets) cannot continue into the transmission windows for time-triggered traffic.<sup>1</sup> The schedules and reserved transmission windows for mechanisms such as the TAS can effectively be derived from the phase  $\phi$  of the corresponding configuration [16].

Before a node sends or forwards packets of a particular flow  $f$ , the controller propagates the necessary flow information corresponding to  $\text{config}(f, p)$  to the respective nodes in the network. What this entails is technology-specific, but includes routing entries and packet transmission schedules. The flow information in the nodes, in particular routes and schedules, can be updated at runtime by the controller. Flow information updates shall not affect in-flight packets to prevent network update issues such as black-holing. This means, infrastructure nodes temporarily may have multiple sets of flow information for each version of a reconfigured flow and deliver each packet according to the flow configuration that was used by the source node when sending the packet. This can be achieved, e.g., by tagging packets with additional metadata, address schemes, etc. [9]

The controller lazily purges obsolete flow information sets from the nodes once all old packets have been received by the destination nodes. In this context, obsolete flow information sets refers to flows that either have been removed from  $\text{ActiveF}(p')$ , or have been reconfigured. In the latter case, while the flow itself remains active, the flow information sets for old flow versions  $\langle f, p \rangle$  are not used anymore. Practically, to perform this “garbage collection”, the controller has to track the flow information sets associated with each traffic plan, and then has to delete the corresponding routing entries, clear now unused scheduling entries, etc.

In principle, an out-of-band mechanism (e.g., via an additional, dedicated management network) may be employed to perform the modifications of the flow information sets in the nodes. However, we expect it to be more realistic to modify flow information at the network nodes using logical, in-band control channels, that run over the same network as the application data. In Sec. VII, where we detail the installation of a traffic plan, we discuss the requirements of such control channels, and sketch how to implement them in-band.

We consider a network where all links operate with the same transmission speed. We denote the processing delay of an infrastructure node by  $t_{\text{proc}}$  and the propagation delay on a link by  $t_{\text{prop}}$ . These values relate to hardware properties (switching fabric, cable length, etc.) and are therefore considered constants. While it is not necessary for our approach that all nodes and links induce the same processing delay and propagation delay, respectively, we omit node and link identifiers for  $t_{\text{proc}}$  and  $t_{\text{prop}}$  in the following, i.e., assume they have the same values network-wide.

#### D. Zero-Queuing

Our approach uses the zero-queuing principle [3], [15] where packets must not be buffered in the network. Instead, packets

<sup>1</sup>In the simplest case, we can create such guard bands by blocking unscheduled traffic for an interval of the length of MTU transmission before a transmission window for scheduled traffic.

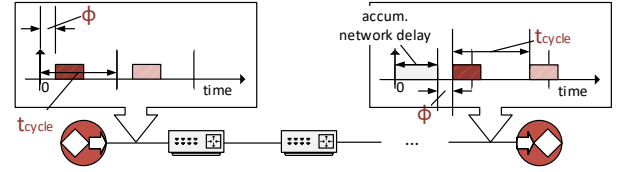


Figure 2: Zero-queuing: Packets accumulate deterministic delay on the route to the destination.

always enter empty queues at each network element and are immediately forwarded to the next hop. Zero-queuing can be achieved by a global coordination (here: in the form of the traffic plan) of packet transmissions. If two flows violate the zero-queuing constraint, we say they *interfere*.

Fig. 2 depicts a packet sequence of a time-triggered flow at different points in the network with zero-queuing. The source node sends one packet per cycle on its outgoing link starting at  $t_0 + \phi + k \cdot t_{\text{cycle}}$ ,  $k \in \mathbb{N}$ . At each hop, packets accumulate only the (inevitable) delay which consists of the time between receiving a packet and completing the forwarding of the packet on the next link. The per-hop delay  $t_{\text{perhop}} = t_{\text{trans}} + t_{\text{prop}} + t_{\text{proc}}$  can be computed from  $t_{\text{trans}}$  and the network properties  $t_{\text{prop}}$  and  $t_{\text{proc}}$ . Thus, zero-queuing enables transmissions with bounded end-to-end delay  $t_{\text{e2e}} = t_{\text{trans}} + t_{\text{prop}} + l \cdot t_{\text{perhop}} + t_{\text{src}} + t_{\text{dst}}$  that depends on the number of hops  $l$  in-between source and destination (i.e.,  $l$  is the number of infrastructure nodes on the route), and processing delays of source and destination node (denoted by  $t_{\text{src}}$  and  $t_{\text{dst}}$ ), respectively.

## IV. CONCEPTS AND PROBLEM STATEMENT

We use a conflict-graph-based approach to compute the new traffic plan. Conflict-graph-based modeling for static traffic planning was previously introduced by us in [16]. From a birds-eye view, conflict-graph-based modeling represents flow configurations and their relations by vertices and edges in a so-called conflict graph. A traffic plan can then be obtained from the conflict graph by searching for a set of conflict-free configurations.

Before we formally state our problem, we explain how these conflict graph concepts relate to our planning problem and extend these concepts for dynamic scenarios.

#### A. Configuration Conflict

Intuitively, we have a conflict between two configurations if packets would have to be transmitted *simultaneously* on the same link to arrive at their respective destination without queuing. Since packet transmission are serialized at each output port, we define configuration conflicts as violations of the zero-queuing constraint.

**Definition 2** (Conflict). *Let  $c_1$  and  $c_2$  be two different configurations. The two configurations are in conflict if any packet sent with  $c_1$  would be buffered due to a packet sent with  $c_2$ , or vice versa.*

This is illustrated in Fig. 3. There is a conflict between configuration  $a1$  for flow  $f_a$  and configuration  $b2$  for flow  $f_b$



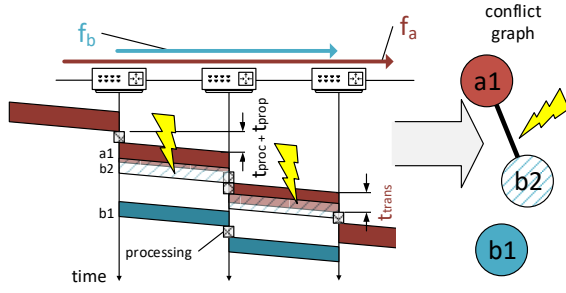


Figure 3: Example: The configuration  $a1$  for  $f_a$  conflicts with the configuration  $b2$  of  $f_b$  because  $f_b$ 's packets are scheduled too early, resulting in buffering. Both flows traverse all links without buffering if we select  $b1$  (which does not conflict with  $a1$ ) for  $f_b$  instead.

since the transmission at the source of  $f_b$  is scheduled too early, and packets sent with  $b2$  would have to be buffered until the transmission of packets from  $f_a$  is finished—which violates the zero-queuing principle. In contrast, configuration  $a1$  and  $b1$  are conflict-free due to the increased phase for packets of  $f_b$ .

The zero-queuing constraint allows us to conflict-check pairs of configurations independently in parallel. To compute whether two configurations conflict, we have to check if the scheduled packet transmissions are always temporally isolated on common links. Due to the cyclic property of the packet transmissions, we only check an interval of the length of a hyper-cycle of the two flows, i.e., the least common multiple of their  $t_{\text{cycle}}$ -values. However, packets may take longer than the (hyper-)cycle to reach the destination. Therefore, we use modulo arithmetic to “fold” transmissions crossing the cycle bounds back to the start of the interval.

The effort to check whether two configurations conflict in principle scales with the length of the candidate paths and the relative least common multiple (lcm) of the cycle times of the two flows (not the length of the global hyper-cycle which is the lcm of all cycle times). Since for each pair of flows  $f_a, f_b$  we expect to encounter the same candidate-path combination  $\pi_a, \pi_b$  for many phase-values  $\phi_a, \phi_b$ , we can use caching to store the common links for the candidate-path combinations. In practice, the effort therefore depends on the shared sub-paths of the candidate paths and the lcm of the cycle times.

### B. Conflict Graph

We encode the relations and constraints between the configurations in a conflict graph. A flow configuration is represented by a vertex in the conflict graph, and conflicts between configurations of *different* flows are represented by edges in the conflict graph. For example, in Fig. 3 there is an edge between configuration  $a1$  and configuration  $b2$ .

A new version of the conflict graph is generated by modifying the old one whenever a new traffic plan is to be computed. As will be shown below, a new version of the conflict graph is generated by adding/removing configurations to the previous version and/or locking/unlocking configurations. In the following,  $\mathbf{G}(p)$  identifies the version of the conflict graph

that is associated with plan  $p$ , i.e.,  $\mathbf{G}(p)$  has been used to compute  $p$ .

The conflict graph includes for each flow a set of potential configurations, which we will call the flow's candidate configurations, or candidates for short. We use  $\text{cand}(f, p)$  to denote the candidate set of flow  $f$  in  $\mathbf{G}(p)$ . Given that—depending, e.g., on the granularity of the time-discretization—there potentially might exist millions or more configurations for each flow, we discuss our strategy for a non-exhaustive conflict graph construction in Sec. V-B where  $\text{cand}(f, p)$  need not contain every single existing configurations of  $f$ .

We can interpret  $\mathbf{G}(p)$  as a colored graph.

**Definition 3 (Conflict Graph).** *The conflict graph  $\mathbf{G}(p)$  is an undirected, vertex-colored graph where all configuration vertices in  $\text{cand}(f, p)$  are colored by  $f$ . Two configuration vertices in  $\mathbf{G}(p)$  are connected by an undirected edge if and only if they belong to different flows and there is a conflict between the two corresponding configurations.*

To insert a configuration  $c$  for flow  $f$  to  $\mathbf{G}(p)$ , we add  $c$  to the vertex set of  $\mathbf{G}(p)$  and add an edge to all conflicting candidates of flows other than  $f$  in  $\mathbf{G}(p)$ . To remove a single configuration  $c$  from  $\mathbf{G}(p)$ , we delete all edges between  $c$  and other configurations in  $\mathbf{G}(p)$ , and remove  $c$  from the vertex set of  $\mathbf{G}(p)$ .

### C. Global Traffic Plan

From the perspective of the planning problem, a traffic plan  $p$  provides a phase and route for each flow  $f$  in  $\text{ActiveF}(p)$  such that all packet transmissions are isolated temporally and/or spatially, and thus satisfy the zero-queuing principle. In other words:

**Definition 4 (Traffic Plan).** *A traffic plan  $p$  is a set of conflict-free configurations for  $\text{ActiveF}(p)$  in  $\mathbf{G}(p)$  which contains exactly one configuration for each flow in  $\text{ActiveF}(p)$ .*

We say that traffic plan  $p$  admits flow  $f$  if  $p$  contains a configuration for  $f$ , i.e.,

$$\text{admits}(f, p) = \begin{cases} 1 & \text{if } \exists \text{ configuration for } f \in p \\ 0 & \text{if } \nexists \text{ configuration for } f \in p \end{cases} \quad (1)$$

**Definition 5 (Independent Set).** *Denote by  $\mathcal{V}$  the vertices, and denote by  $\mathcal{E}$  the edges in  $\mathbf{G}(p)$ , respectively.  $C \subseteq \mathcal{V}$  is an independent subset of vertices in the conflict graph  $\mathbf{G}(p)$  iff  $\forall u, v \in C : (u, v) \notin \mathcal{E}$ .*

With the correspondence vertex  $\leftrightarrow$  configuration and edge  $\leftrightarrow$  conflict, it is easy to see that a traffic plan corresponds to a set of independent vertices in the conflict graph [16].

Note that  $\mathbf{G}(p)$  possibly contains many sets of different conflict-free configurations. While it is intuitively clear that we prefer to find a set of conflict-free configurations that admits all flows, this may not always be possible. Therefore, we specify the objective that guides the search for the configuration in Sec. IV-E.

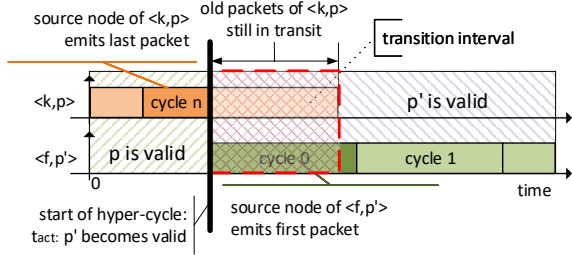


Figure 4: The possibility of transition interference between old packets of flow  $k$  from plan  $p$  and new packets of flow  $f$  from  $p'$  is limited to the *transition interval*.

#### D. Transition Interference

Assume that we have a new traffic plan  $p'$  and want to replace the traffic plan  $p$  with  $p'$ . We call the time a new traffic plan  $p'$  becomes valid its activation time  $t_{act}$ . To ensure that no source node is in the middle of a packet transmission at the activation time of  $p'$ , and since the flows in  $\text{Active}(p)$  may have different cycle times, we choose the start of a fresh hyper-cycle of all active flows in  $\text{ActiveF}(p)$  as activation time, i.e.,  $t_{act} = t_0 + k \cdot \text{lcm}_f(t_{\text{cycle}}(f))$ . From the perspective of an active flow it is perfectly fine for  $t_{act}$  to be very far—many transmission cycles—in the future, because all flows in  $\text{ActiveF}(p)$  remain active with their established configurations from  $p$  until  $t_{act}$ .

Then, at  $t_{act}$  the new traffic plan becomes valid. This means, from  $t_{act}$  on, we would like *all* the packets of each flow  $f$  in  $\text{ActiveF}(p')$  to be transmitted according to  $\text{config}(f, p') = (f, \phi', \pi')$ . If all packets sent according to the old plan  $p$  are delivered to the destination nodes before  $t_{act}$ , this would simplify our problem a lot. It implies that the network would be empty at  $t_{act}$  which would allow us to compute each new traffic plan  $p'$  independently of  $p$ . In other words, under these circumstances computing a new traffic plan is equivalent to solving a new static traffic planning instance.

However, this simplified “solution” is inadequate because it severely restricts the network utilization. Ultimately, for each packet to arrive within the transmission cycle it was sent by the source node,  $\phi$  would need to be less than  $t_{\text{cycle}} - t_{e2e} - t_{\text{trans}}$ . This means, the transmission frequency and network utilization are inherently coupled to the network size.

Therefore, we provide a solution for the general case where there can be remaining yet-to-be-delivered packets from  $\langle k, p \rangle$ ,  $k \in \text{ActiveF}(p)$  in the network when the new traffic plan  $p'$  becomes valid. In the general case, we have to account for the possibility of *transition interference*, i.e., flow interference between the remaining old packets from  $\langle k, p \rangle$ ,  $k \in \text{ActiveF}(p)$  and the first packets from  $\langle f, p' \rangle$ ,  $f \in \text{ActiveF}(p')$ , when computing the new traffic plan  $p'$ .

**Definition 6** (Transition Interference). *Let  $p$  denote the traffic plan valid before  $t_{act}$  and let  $p'$  denote the new traffic plan with activation time  $t_{act}$ . Transition interference is defined to be between  $\langle k, p \rangle$ ,  $k \in \text{ActiveF}(p)$  and  $\langle f, p' \rangle$ ,  $f \in \text{ActiveF}(p')$  if any packet sent by the source node of  $\langle k, p \rangle$  prior to  $t_{act}$  is buffered due to a packet sent by the source node of  $\langle f, p' \rangle$*

from  $t_{act}$  on, or vice versa.

We can algorithmically check whether transition interference occurs between  $\langle k, p \rangle$  and  $\langle f, p' \rangle$ : If the route of  $\langle k, p \rangle$  and the route of  $\langle f, p' \rangle$  have common links, and any packet from  $\langle k, p \rangle$  sent before  $t_{act}$  is scheduled for transmission at any point in time when any packet sent by  $\langle f, p' \rangle$  from  $t_{act}$  on is scheduled on a common link, transition interference occurs.

Transition interference is restricted to a possible zero-length time interval, the *transition interval*  $t_{\text{transit}}$ , which starts at  $t_{act}$  and is upper-bounded by the end-to-end delay of  $\langle k, p \rangle$ , cf. Fig 4. Now we can also concretize the “purging” of obsolete flow information sets (cf. Sec. III-C): It is easy to see that once the transition intervals of all flows in  $\text{ActiveF}(p)$  have expired, the controller can safely delete all unused flow information sets associated with  $p$  from the nodes in the network.

In the following, we only consider the case where transition intervals of different traffic plan updates are mutually exclusive, i.e., we consider traffic plan  $p$  and its immediate successor  $p'$ .

#### E. Problem Statement

Considering the many facets of dynamic traffic planning, our focus is how to actually compute the traffic plan and the traffic plan (updates) for an evolving set of real-time traffic flows.

Let  $p$  be the traffic plan for the flows in  $\text{ActiveF}(p)$  computed from  $\mathbf{G}(p)$ , i.e., the packets of flows in  $\text{ActiveF}(p)$  are transmitted according to  $p$ . Now applications request that a set of flows  $\text{ReqF}(p)$  is added to the network. Our goal is to compute a new traffic plan  $p'$ .

To compute  $p'$ , we have to construct the new conflict graph  $\mathbf{G}(p')$  and search for a set of conflict-free configurations (i.e., independent vertices) in  $\mathbf{G}(p')$ . In our case, we want to ensure that once the planner added a flow to  $\text{ActiveF}(p)$ , this flow remains active until the application itself indicates the flow can be removed. This means, the planner shall not unsolicitedly evict any flow from  $\text{ActiveF}(p)$ . Consequently, the new traffic plan shall include a configuration for *every* flow in  $\text{ActiveF}(p)$  and as many new flows as possible from  $\text{ReqF}(p)$ . The different importance of active flows and new flows results in the following objective.

**Definition 7** (Traffic Planning Objective). *Let  $\mathbf{G}(p')$  be a conflict graph which contains candidates for all flows in  $\text{ActiveF}(p) \cup \text{ReqF}(p)$  with  $\text{ActiveF}(p) \cap \text{ReqF}(p) = \emptyset$ . We want to find a new traffic plan  $p' \subseteq \bigcup_f \text{cand}(f, p')$  that maximizes the objective*

$$\max_{p'} \sum_{f \in \text{ActiveF}(p)} \text{admits}(f, p') + \sum_{f \in \text{ReqF}(p)} \frac{\text{admits}(f, p')}{|\text{ActiveF}(p) \cup \text{ReqF}(p)|}. \quad (2)$$

The factor  $1/|\text{ActiveF}(p) \cup \text{ReqF}(p)|$  in Eq. 2 discounts the relative importance of flows in  $\text{ReqF}(p)$ . This means, any flow in  $\text{ActiveF}(p)$  is more “valuable” than all flows  $\text{ReqF}(p)$  combined. From a graph-theoretic perspective, this is a specific colorful independent vertex set problem where we want to find a set of independent vertices and each color has either unit weight or a weight inversely proportional to the total amount of colors.

## V. CONSTRUCTING THE CONFLICT GRAPH

Next, we explain how the planner constructs the new conflict graph  $\mathbf{G}(p')$  when processing  $\text{ReqF}(p)$ . With offensive planning, the planner has to consider both flow versions  $\langle f, p \rangle$  and  $\langle f, p' \rangle$  for each active flow  $f$ , where  $p'$  indicates the new plan replacing  $p$ .

### A. Overview

Remember that after installing  $p'$ , packets from both,  $\langle f, p \rangle$  and  $\langle f, p' \rangle$ , may be in the network for some time. Therefore, it is not sufficient to only avoid conflicts between any new flow versions  $\langle f, p' \rangle$  which would suffice in the static case. Instead, the planner additionally has to ensure that any  $\langle k, p \rangle$  does not interfere with any other flow  $\langle f, p' \rangle$ . This leads to the following requirements:

- R1 For all  $k, f \in \text{ActiveF}(p')$ ,  $\langle k, p' \rangle$  does not interfere with  $\langle f, p' \rangle$  for  $k \neq f$ , i.e., active flows of the new plan do not interfere.
- R2 For all  $k \in \text{ActiveF}(p)$  and  $f \in \text{ActiveF}(p') \setminus \text{ActiveF}(p)$ ,  $\langle k, p \rangle$  does not interfere with  $\langle f, p' \rangle$ . That is, the active flows of the old plan do not interfere with the flows added to the new plan.
- R3 For all  $k, f \in \text{ActiveF}(p)$ ,  $\langle k, p \rangle$  does not interfere with  $\langle f, p' \rangle$ . That is, the active flows of the old plan do not interfere with their versions in the new plan. If  $f = k$ , then we consider the old and new version of the same flow,  $\langle f, p \rangle$  and  $\langle f, p' \rangle$ . Since packets from  $\langle k, p \rangle$  and  $\langle f, p' \rangle$  may populate the network at the same time, we have to ensure that no transition interference occurs.

R1 is a basic requirement, which is sufficient for defensive planning. Allowing for reconfigurations of active flows, however, we must add R2 and R3 to ensure the zero-queuing constraint. In order to fulfill R1 the planner has to make sure that the configurations associated with flows in  $\text{ActiveF}(p')$  are not in conflict. To this end, the planner first has to *add candidate configurations* for each flow in  $\text{ReqF}(p)$  to the conflict graph. The expanded conflict graph is the basis for solving the planning problem as stated above.

Now let us see, how additionally R2 and R3 can be guaranteed. Note that the interference of two flows/flow versions implies that their configurations are conflicting, while the opposite is not true. For example, if we ensure that the packets of one flow or flow version only enter the network after all packets of another flow have already left the network, these two flows do not interfere even if their configurations are conflicting. Therefore, we can fulfill R2 by *delaying the activation* of the added flows until all packets that belong to flow versions of  $p$  have been delivered: Assume the transmission of packets of each flow  $\langle k, p \rangle$  with  $k \in \text{ActiveF}(p)$  is stopped at time  $t$ . All packets sent by these flows have left the network by  $t + \tau$ , where  $\tau \geq \max_k (t_{\text{e2e}}(k) - t_{\text{cycle}}(k))$  is greater or equal to the largest transition interval  $t_{\text{transit}}$ . If the transmission of each  $\langle f, p' \rangle$  with  $f \in \text{ActiveF}(p') \setminus \text{ActiveF}(p)$  does not start before  $t + \tau$ , then no flow  $\langle k, p \rangle$  interferes with any flow added to  $p'$ , even if their configurations conflict. To put it differently, we prevent transition interference between flows from  $k \in \text{ActiveF}(p)$

and  $f \in \text{ActiveF}(p') \setminus \text{ActiveF}(p) \subseteq \text{ReqF}(p)$  from ever happening by adjusting the activation time of new flows, i.e., outside of the actual computation of  $p'$ . Simplified speaking: the controller tells the source nodes of the new flows that their actual activation time is not  $t_{\text{act}}$ , but  $t_{\text{act}} + \lceil \frac{t_{\text{transit}}}{t_{\text{cycle}}} \rceil \cdot t_{\text{cycle}}$  with  $t_{\text{transit}}$  the duration of the longest transition interval of any flow in  $\text{ActiveF}(p)$ . In the example in Fig. 4, the first “usable” transmission cycle for  $\langle f, p' \rangle$  is cycle 1. For the source node of a new flow this has the same effect as if it took a few  $\mu\text{s}$  longer to compute  $p'$ . Therefore, fulfilling R2 requires no special treatment when constructing the conflict graph.

Note that this only applies for transition interference involving flows that are newly added to  $\text{ActiveF}(p')$ . Fulfilling R3 the same way as R2 is definitely not desirable since delaying data packets of an active flow might degrade QoS substantially. Figuratively speaking, our solution for R2—“turning on the machines a bit later”—does not work for R3 because “the machines are already running”. Therefore, we have to make sure that the planner cannot select configurations for active flows which conflict with the old configurations of these flows. We achieve this by *locking those configurations* that the planner must not select for the new plan. Note that the configurations to be locked depend on the old configuration of the active flows. Since the configuration of an active flow may change from plan to plan, configurations are unlocked each time after the new plan has been computed.

### B. Adding Candidate Configurations

Since  $\mathbf{G}(p)$  only contains candidates for flows in  $\text{ActiveF}(p)$ , the planner has to generate and insert new candidate configurations for all flows in  $\text{ReqF}(p)$  to  $\mathbf{G}(p')$ . In principle, the planner could add all candidates for each flow to  $\mathbf{G}(p')$ . However, we know from the static flow planning problem [16] that we can find a traffic plan for all flows even if the conflict graph contains just a subset of all possible candidates of each flow. We exploit this and add only a limited number (upper-bounded by a parameter  $n_{\text{ub}}$ ) of candidate configurations for each flow in  $\text{ReqF}(p)$  to the conflict graph.

Due to the nature of the traffic planning problem, it is often inherently difficult to identify promising candidate configurations a-priori. In this paper, we therefore use the following heuristic:

For each flow, the planner has a stateful configuration generator that walks through the  $\phi$ - $\pi$ -space (cf. Fig. 5) and returns a new configuration each time it is invoked. We can, figuratively speaking, think of a flow’s  $\phi$ - $\pi$ -space as a two-dimensional plane, where a configuration is a discrete point at a coordinate formed by the phase and path identifier of the configuration. Each invocation of the generator then corresponds to “moving” to the next point, which, in turn, generates the respective configuration to be added to the conflict-graph. In this context, the current coordinate is our generator state, and the rule for determining the next coordinate defines the order in which the configurations of a particular flow are added to  $\mathbf{G}(p')$ .

In our case, starting with  $\phi = 0$ ,  $\pi = 0$ ,  $\pi$  is incremented until all configurations for the current value of  $\phi$  have been covered



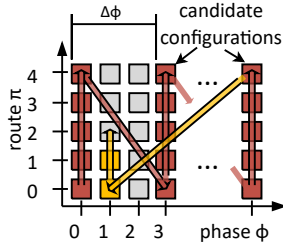


Figure 5: To obtain a new candidate configuration, the planner “moves” to a new point in the flow’s discrete  $\phi$ - $\pi$ -space and adds the visited candidate configuration corresponding to the current coordinates to the conflict graph. Our heuristic makes the planner walk as indicated by the arrows, i.e., the planner cycles through all candidate paths for the current phase  $\phi$ , before jumping to the next phase.

(i.e., moving “upwards” in the  $\pi$ -dimension), before increasing  $\phi$  by  $\Delta\phi$  (i.e., jumping in the  $\phi$ -dimension). We use for  $\Delta\phi$  the 75-th percentile of the transmission duration of all flows in  $\mathbf{G}(p')$ , hence we expect that one to two phase increments by the planner suffice to resolve a possible interference on a single link. If  $\phi + \Delta\phi$  exceeds the allowed range for  $\phi$ ,  $\phi$  is reset to the next, lowest uncovered phase-value, cf. Fig 5.

In other words, every time we need a new configuration for a particular flow, we invoke its configuration generator. The configuration generator then returns the next configuration in a sequence of this flow’s candidate configurations obtained by sweeping (possibly in multiple passes along the  $\phi$ -range) over the  $\phi$ - $\pi$ -space.

While it is obvious that we have to add candidates for flows in  $\text{ReqF}(p)$ , we can also increase the set of candidates for flows in  $\text{ActiveF}(p)$  in  $\mathbf{G}(p')$ . The larger  $\text{cand}(f, p')$ , the more of the solution space is covered, and the more alternatives for an active flow “to make way” for flow in  $\text{ReqF}(p)$  exist. Here, we add more configurations for each flow in  $\text{ActiveF}(p)$  when processing  $\text{ReqF}(p)$  until the configuration generator has traversed the  $\phi$ -range of a flow for the first time. After that point, new configurations for active flows are only added if the planner could not admit all flows from the previous request, i.e., a simple feedback loop controls the growth of the conflict graph.

In principle, we are free to choose how we want to traverse the  $\phi$ - $\pi$ -space when adding configurations to the conflict graph. This means, that the number of available configurations in the  $\phi$ - $\pi$ -space is decoupled from the size of the conflict graph. For example, if we want a time resolution of 1 ns instead of 1  $\mu$ s, we can scale  $\Delta\phi$  accordingly, e.g., instead of advancing three steps in the  $\phi$ -direction, we advance 3000 steps and end up with a similar sized conflict graph.

This decoupling is an important feature because the effort to add a single configuration  $c$  to  $\mathbf{G}(p')$  scales with the number of configurations  $\mathcal{V}$  already in the conflict graph ( $\mathcal{O}(\mathcal{V})$ ) since we have to check for each existing configuration already in  $\mathbf{G}(p')$  whether it conflicts with  $c$ . However, we do *not* have to pay the *total* cost of constructing a conflict graph with  $\mathcal{V}$  configurations, which scales with  $\mathcal{O}(\mathcal{V}^2)$ . When constructing  $\mathbf{G}(p')$ , we start

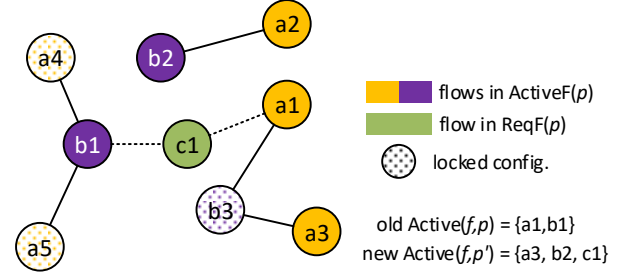


Figure 6: Example: Configuration  $c1$  for a new flow  $f_c$  conflicts with the active configurations  $a1$  and  $b1$  of  $f_a$  and  $f_b$ , respectively. To admit  $f_c$ , we need to reconfigure  $f_a$  and  $f_b$ . Due to transition interference, all 1-hop neighbors of active configuration  $a1$  and  $b1$  are locked in  $\mathbf{G}(p')$ , so only  $a2$ ,  $a3$ , and  $b2$  remain as options during the reconfiguration.

with  $\mathbf{G}(p)$ . Hence, the cost of adding a configuration effectively gets distributed over the lifetime of a flow. The planner could also add the additional configurations for flows in  $\text{ActiveF}(p)$  when idle, i.e., while no  $\text{ReqF}(p)$  has to be processed, to further accelerate the flow-request response time. Since the lifetime of active flows may differ, the planner should balance the number of candidates per flow in the long-term, e.g., by adding more configurations for younger flows.

### C. Locking Configurations

$\mathbf{G}(p')$ , generated as described in Sec. V-B, may include configurations that cause interference between the old and new versions of active flows, violating R3. We yet have to make sure that the planner cannot select configurations for active flows that violate R3. In detail, the following two conditions have to hold:

- 1) For all  $f \in \text{ActiveF}(p)$ , there exists no configuration  $c \in \text{cand}(f, p')$  that results in a new version  $\langle f, p' \rangle$  that interferes with  $\langle f, p \rangle$ , i.e., the planner cannot assign a new configuration to  $f$ , such that the old and new version of  $f$  interfere.
- 2) For all  $f, k \in \text{ActiveF}(p)$ ,  $k \neq f$  there exists no configuration  $c \in \text{cand}(f, p')$  that results in a new version  $\langle f, p' \rangle$  which interferes with  $\langle k, p \rangle$ . That is, the planner cannot assign a new configuration to  $f$  which results in interference with an old version of any other active flow  $k$ .

We achieve both conditions by locking all those configurations in  $\mathbf{G}(p')$  which would violate one of the conditions if selected by the planner. That is, informally speaking, we can think of a locked configuration in  $\mathbf{G}(p')$  as being “hidden” from the planner during the computation of  $p'$ .

To fulfill the first condition, for each  $f \in \text{ActiveF}(p)$  we have to visit each  $c \in \text{cand}(f, p')$  in  $\mathbf{G}(p')$  to check whether a new version of  $f$  that uses the visited candidate  $c$  results in transition interference with the old version that uses  $\text{config}(f, p)$ . If transition interference would occur, we have to lock the candidate. Obviously, this ensures that the planner cannot select a candidate  $c$  for  $\text{config}(f, p')$  which could cause interference with  $\langle f, p \rangle$ . Note that the configuration graph

includes only edges for conflicting configurations of different flows. Therefore, we maintain an additional data structure that provides efficient access to the candidates of a particular flow.

To fulfill the second condition, for each  $k \in \text{ActiveF}(p)$  we have to visit all 1-hop neighbors of  $\text{config}(k, p)$  in the conflict graph, i.e., all candidate configurations of any flow other than  $k$  that conflict with  $\text{config}(k, p)$ . If a visited neighbor  $c$  is a configuration of another flow  $f$  in  $\text{ActiveF}(p)$ , the planner checks for transition interference between a new version of  $f$  with configuration  $c$  and  $\langle k, p \rangle$ . If there would be interference, this configuration  $c$  is locked. Obviously, this prevents the planner from selecting for any other  $f \in \text{ActiveF}(p)$  a candidate configuration  $c$  for  $\text{config}(f, p')$  which would cause interference. Note, here we just have to follow the edges of  $\text{config}(k, p)$  to access the candidates potentially to be locked, cf. Fig. 6.

#### D. (Permanently) Pinning Flows

To prevent the planner from ever reconfiguring an active flow  $f$ , we can *permanently* remove all configurations of  $f$  other than  $\text{config}(f, p)$  from  $\mathbf{G}(p)$ —*pinning*  $f$  to its configuration for its whole lifetime. Obviously, this ensures  $\text{config}(f, p) = \text{config}(f, p')$  for future updates. In contrast, locking only temporarily excludes some candidates from  $p'$ .

### VI. COMPUTING THE NEW TRAFFIC PLAN

Once the new conflict graph has been constructed by adding additional configurations and locking currently “forbidden” candidate configurations, we have to compute the traffic plan. From a graph-theoretic perspective, we can reduce the maximum (colorful) independent vertex set problem to the problem of finding a traffic plan which maximizes Eq. 2. This means, unsurprisingly, that computing a new traffic plan remains an NP-hard problem [33]. Therefore, we present a novel heuristic, namely, the *Greedy Flow Heap Heuristic* (GFH). The name draws from the fact that the objective in Eq. 2 improves with every additional flow included in the new traffic plan. Next, we describe the GFH in detail, and explain our strategy to compute the new traffic plan.

#### A. Greedy Flow Heap Heuristic

From a birds-eye view, GFH is an iterative greedy approach. It splits  $\text{ActiveF}(p)$  and  $\text{ReqF}(p)$  into flow subsets and successively processes the different subsets in an order that is supposed to maximize the objective function (cf. Eq. 2). For every flow set, GFH iteratively selects the next flow, computes ratings for the flow’s configurations in  $\mathbf{G}(p')$ , and adds the configuration with the best rating to an intermediary set of conflict-free configurations  $C$ .

During the GFH execution,  $C$  is always an independent vertex set in  $\mathbf{G}(p')$ . For the same flow  $f$  there may be multiple configurations in the final  $C$  returned by the GFH, but we can only include one of these (we need only one route and phase) in a traffic plan. In this case, we arbitrarily select one configuration for every flow with multiple configurations in  $C$  since all configurations in  $C$  are conflict-free.

Before explaining the greedy strategy and how configurations are rated, we introduce some terminology:

- $C$  *admits* flow  $f$ , if  $C$  contains at least one candidate configuration for  $f$ .
- A configuration  $c \in \mathbf{G}(p')$  is *shadowed* if at least one of its neighbors is included in  $C$ . Thus, adding a configuration to  $C$  shadows all its neighbors. Shadowed configurations are excluded from being added to  $C$  since they conflict, i.e., are connected by an edge, with at least one of the configurations in  $C$ .
- A configuration is *solitary* if it has no neighbors (i.e., no edge to other configurations) in  $\mathbf{G}(p')$  and thus cannot conflict with any other configuration.
- A configuration  $c$  is *eligible* for selection by the GFH algorithm if and only if  $c$  is neither shadowed, nor  $c \in C$ .

In each run, the GFH attempts to find a  $C$  that admits all flows as follows: First,  $C$  is initialized by adding all solitary configurations. Thus,  $C$  already admits all flows with solitary configurations without reducing the solution space. Then, configurations for the remaining flows are iteratively added to  $C$ . We will discuss the configuration selection strategy in Sec. VI-A1 and the configuration rating in Sec. VI-A2, including a small GFH example.

If  $C$  does not admit all flows after its first run, the GFH has a *re-run* mechanism which attempts to find a  $C$  that admits more flows by additional *re-runs* with modified starting conditions. The number of re-runs (default  $n_{\text{re-runs}} = 3$ ) can be parameterized. We discuss the re-run mechanism in Sec. VI-A3. Finally, the complexity of GFH is addressed in Sec. VI-A4.

1) *Configuration Selection Strategy*: GFH being a greedy algorithm, the quality of the solution, i.e., how many flows can ultimately be admitted, relies on the strategy we use to add configurations to  $C$ . GFH uses a hierarchical, iterative strategy, first deciding on the next flow to process, and then selecting a configuration for that flow. Remember that any active flow contributes more to the objective than all new flows taken together. To prevent that any configuration of a new flow shadows any active flow’s configuration such that an active flow may not be admitted to  $C$ , we thus first process all flows in  $\text{ActiveF}(p)$ , before searching for a configuration for new flows from  $\text{ReqF}(p)$ .

The pseudo-code to process a particular (sub-)set of flows, say  $\text{searchF}(p') \subseteq \text{ActiveF}(p) \cup \text{ReqF}(p)$ , is given in the method `addConfigPerFlow` in Alg. 1.

---

#### Algorithm 1: addConfigPerFlow

---

```

input : flows set  $\text{searchF}(p')$ , conflict-free configurations  $C$ 
1  $\forall f \in \text{searchF}(p') : \text{check admits}(f, C);$ 
2 create heap of not admitted flows;
3 while heap  $\neq \emptyset$  do
4    $f_{\min} \leftarrow \text{pop } f_{\min} \text{ with least eligible configurations from heap};$ 
5    $c_{\text{sel}} \leftarrow \text{eligible } c \text{ of flow } f_{\min} \text{ with smallest}$ 
      $\text{computeShadowRating}(c);$ 
6   add  $c_{\text{sel}}$  to  $C$ , update eligibility of neighbors of  $c_{\text{sel}};$ 
7   update heap (remove completely shadowed flows and reorder);
8 return updated  $C;$ 

```

---

We start by creating a min-heap that contains all flows from  $\text{searchF}(p')$  not already admitted by  $C$  (cf. Alg. 1, line 1). The

flows in the min-heap are sorted according to the number of remaining eligible configurations, i.e., flows with less eligible configurations are on top of the heap. Ties are broken with the higher total degree of the configurations in  $\text{cand}(f, p')$ , because a high total degree suggests a high chance that all configurations of the flow will soon become ineligible. The unique flow identifier is used as final tie-breaker to get a deterministic algorithm.

Next, the heap is iteratively processed: The flow on top of the heap—the one with the least remaining eligible configurations—is removed from the heap, and we add the flow’s best-rated (cf. Sec. VI-A2) configuration to  $C$ . When a configuration is added to  $C$  the heap is adjusted. Any completely shadowed flow is removed from the heap and the remaining flows are reordered according to their new eligible configurations count.

2) *Rating Configurations*: The shadowRating value of a configuration is intended to capture the “cost” of selecting that configuration in terms of the remaining solution space. For example, adding a configuration to  $C$  that shadows huge portions of the conflict graph is “expensive”. Conversely, the cost is lower, the fewer neighbors are shadowed. If neighbors are shadowed, it is preferable to shadow configurations of those flows with lots of remaining eligible configurations. This intuition is encoded in the pseudo-code given in Alg. 2.

**Algorithm 2:** computeShadowRating

---

```

input : configuration  $c$ 
1 shadowRating = 0;
2  $\mathcal{F}_{\text{neig}} \leftarrow$  set of flows of all neighbors of  $c$ ;
3 foreach  $f_n \in \mathcal{F}_{\text{neig}}$  do
4   shadowCount  $\leftarrow$  number of eligible configurations of flow  $f_n$  in
   neighborhood of  $c$ ;
5   eligibleCount  $\leftarrow$  total number of eligible configurations in  $f_n$ ;
6    $\delta \leftarrow$  shadowCount / eligibleCount;
7   if  $\delta = 1$  then shadowRating +=  $\alpha$ ;
8   else shadowRating +=  $\delta$ ;
9 return shadowRating of  $c$ ;

```

---

We compute for each flow  $f_n$  with a configuration in the 1-hop neighborhood of  $c$  the share  $\delta$  of this flow’s remaining eligible configurations that would be shadowed by picking  $c$ . Configurations which are already shadowed are not taken into account. The shadowRating ordinarily amounts to the sum over the  $\delta$  values of  $c$ ’s neighborhood. If adding  $c$  to  $C$  shadows all remaining eligible configurations of a flow ( $\delta = 1$ ), a very large constant  $\alpha$  (default  $\alpha = 1000$ ) is added instead to discourage the selection of a configuration  $c$  which shadows the remaining configuration(s) of another flow. Note that the shadowRatings can be computed in parallel since we only have to read the neighbors of  $c$ . Further, the *eligibleCount* calculation (cf. Alg 2, line 5) can be optimized via caching.

To give a brief example of how GFH works, we revisit the conflict graph given in Fig. 6. As there are no solitary configurations, GFH starts with  $C = \emptyset$ . The active flows  $f_a$  and  $f_b$  are handled by the first addConfigPerFlow call. Hence, the heap consists of these two flows where  $f_a$  has three and  $f_b$  has two eligible configurations, respectively. First, the best configuration of flow  $f_b$  is calculated since  $f_b$  has less eligible configurations. Since  $b1$  shadows all configurations of  $f_c$ ,  $b1$  gets a very high shadowRating whereas  $b2$  gets the

best rating with 0.33 and is added to  $C$ . Second, flow  $f_a$  is processed. Here,  $a3$  has the lowest shadowRating (0.0 since  $b3$  is locked) and is added to  $C$ . Note that  $a4$  and  $a5$  are locked,  $a2$  is shadowed and  $a1$  would shadow all configurations of  $f_c$ . Finally, addConfigPerFlow is called a second time with the new flows  $\text{searchF}(p') = \{f_c\}$ . Thereby,  $c1$ , which is the only configuration of  $f_c$ , is added to  $C$ . Since all flows are admitted, there is no need to use the re-run mechanism in this example.

3) *Re-run Mechanism*: The re-run mechanism provided by the GFH (cf. Alg. 3) can be used to improve the number of flows admitted by  $C$ , e.g., if not all flows are in  $C$ . In principle, a single run in the GFH corresponds to executing the body of the loop in Alg. 3 (starting at line 3) once.

**Algorithm 3:** GFH

---

```

input : active flow set ActiveF( $p$ ), new flow set ReqF( $p$ ),  $n_{\text{re-runs}}$ 
1  $C \leftarrow \emptyset$ ; // init.  $C$  (first run)
2 cache  $\leftarrow \emptyset$ ;
3 for  $i = 0; i \leq n_{\text{re-runs}}; i++$  do
4   naActiveF  $\leftarrow$  ActiveF( $p$ ) \  $C$ ;
5   aActiveF  $\leftarrow$  ActiveF( $p$ )  $\cap$   $C$ ;
6   naReqF  $\leftarrow$  ReqF( $p$ ) \  $C$ ;
7   aReqF  $\leftarrow$  ReqF( $p$ )  $\cap$   $C$ ;
8   //  $C \leftarrow$  solitary configurations:
9    $C \leftarrow \{v \in \mathcal{V} : \text{degree}(v) == 0\}$ ;
10   $C \leftarrow$  addConfigPerFlow(naActiveF,  $C$ );
11   $C \leftarrow$  addConfigPerFlow(aActiveF,  $C$ );
12   $C \leftarrow$  addConfigPerFlow(naReqF,  $C$ );
13   $C \leftarrow$  addConfigPerFlow(aReqF,  $C$ );
14  if  $C$  admits (ActiveF( $p$ )  $\cup$  ReqF( $p$ )) then return  $C$ ;
15  else store  $C$  in cache;
16 return  $\arg \max_{C \in \text{cache}} \text{TrafficPlanningObjective}(C)$ ; // see Eq. 2

```

---

In each run, we call addConfigPerFlow once for every flow subset, using it as parameter  $\text{searchF}(p')$ . Remember that the order of the flow subsets for which we execute addConfigPerFlow implicitly assigns priorities to those subsets. Due to the greedy strategy, the earlier we include a flow  $f$  from  $\mathbf{G}(p')$  in  $\text{searchF}(p')$  the likelier it is that a configuration for  $f$  is added to  $C$ . We take advantage of this in the re-run mechanism to prioritize those flows which were not admitted to  $C$  in the previous run. Note that we still have to account for the higher importance of active flows compared to new flows (cf. Eq. 2). To this end, we split both, ActiveF( $p$ ) and ReqF( $p$ ), into two subsets each: one subset contains all flows which were admitted by  $C$  in the previous run (prefixed by “a”, i.e., aActiveF, aReqF in Alg. 3), and another subset contains those flows which could not be admitted previously (prefixed by “na”, i.e., naActiveF, naReqF in Alg. 3). Before each run, we reset  $C$  (cf. Alg. 3, line 8). Then, we first process the two subsets corresponding to ActiveF( $p$ ), before processing the two subsets corresponding to ReqF( $p$ ) (cf. Alg. 3 line 9-12). To be exact, we call addConfigPerFlow on the four flow subsets in the following order: 1) flows from ActiveF( $p$ ) previously not admitted by  $C$ , 2) flows from ActiveF( $p$ ) previously admitted by  $C$ , 3) flows from ReqF( $p$ ) previously not admitted by  $C$ , and 4) flows from ReqF( $p$ ) previously admitted by  $C$ .

The first run constitutes a special case where the flows admitted by  $C$  are not the result of a previous run but contain only those flows with solitary configurations. Solitary configurations do not shadow any eligible configuration and

consequently cannot affect the results during the re-runs. By convention, we do not count the first run as a re-run, i.e., the loop can be executed up to  $n_{\text{re-runs}} + 1$  times. Re-runs are performed either until  $C$  admits all flows, or we run out of re-runs (default  $n_{\text{re-runs}} = 3$ ). This can improve the objective value, but it is not guaranteed. If we use up all re-runs, the GFH algorithm returns the  $C$  with the highest objective value seen so far.

4) *Complexity*: The worst-case runtime complexity of a single GFH run, i.e., adding the solitary configurations to  $C$  and performing a single run of `addConfigPerFlow`, can be stated as  $\mathcal{O}(\mathcal{E} + \mathcal{V}\mathcal{F})$ . Here,  $\mathcal{E}$  is the number of conflicts,  $\mathcal{V}$  is the number of configurations in the conflict graph, and  $\mathcal{F}$  represents the number of flows. In the following, we derive this complexity.

Adding all solitary configurations is an  $\mathcal{O}(\mathcal{V})$  operation performed exactly once. Building the filtered heap is an  $\mathcal{O}(2\mathcal{F} \cdot \mathcal{V})$  operation (cf. Alg. 1, line 2). The heap updates later will be done in logarithmic time, resulting in  $\mathcal{O}(\mathcal{V} \cdot \log \mathcal{F})$ . We require the  $\mathcal{V}$  factor in the heap operations to count the eligible configurations for the flow comparisons every time. The `shadowRating` is calculated for every configuration once. Thus, we have  $\mathcal{O}(\mathcal{V}(d + \mathcal{F} + 1))$  computations, where  $d$  is the highest degree in the graph. The  $\mathcal{F}$  parameter comes from accumulating the `shadowCount`. This assumes that we calculate the `eligibleCount` (cf. Alg. 2, line 5) once for each flow in the heap and use the (cached) `eligibleCount` values for the computations of all configurations of this flow, which takes  $\mathcal{O}(\mathcal{F} \cdot \mathcal{V})$  time. Finding  $c_{\text{sel}}$  can be done space-efficient for every flow in  $\mathcal{O}(\mathcal{F} \cdot \mathcal{V})$  time. The eligibility update of the neighbors of  $c_{\text{sel}}$  (cf. Alg. 1, line 6) for every flow is an  $\mathcal{O}(d \cdot \mathcal{F})$  operation. When combining these sub-routines, GFH ends up with a total runtime complexity of  $\mathcal{O}(\mathcal{V}(d + 5\mathcal{F} + \log \mathcal{F} + 2) + d\mathcal{F})$ , which can be simplified to  $\mathcal{O}(\mathcal{E} + \mathcal{V}\mathcal{F})$ .

So far, we more or less considered a single run of the GFH. The re-run mechanism adds another factor  $r' = n_{\text{re-runs}} + 1$  resulting in a simplified overall complexity of  $\mathcal{O}(r'\mathcal{E} + r'\mathcal{V}\mathcal{F})$ .

### B. Rejecting and Removing Flows

If  $\text{ReqF}(p)$  contains flows that are not admitted by  $C$ , the planner rejects the corresponding flows. All candidate configurations for a rejected flow are consequently purged from  $\mathbf{G}(p')$ . Similarly, applications could indicate to the planner that active flows shall be removed from the network. In this case the planner also purges the corresponding candidate configurations from the conflict graph.

### C. Optimization: Progressive Strategy for Offensive Planning

The planner employs locking and GFH in a two-phase meta-strategy for offensive planning:

In the first, defensive phase, for each flow in  $\text{ActiveF}(p)$  every configuration in  $\text{cand}(f, p') \setminus \text{config}(f, p)$  is locked in  $\mathbf{G}(p')$ , i.e., we conserve the configuration of all active flows. Then  $\mathbf{G}(p')$  exposes only one configuration per active flow and all candidates for new flows to the GFH. If we already find a traffic plan  $p'$  for  $\text{ActiveF}(p) \cup \text{ReqF}(p)$ , we are done,

and usually saved computation time since the GFH considered fewer configurations.

Only if we cannot admit all new flows, we release the conservative locks (configurations which cause transition interference, cf. Sec. V-C remain locked) and expose the “full” conflict graph to the GFH. This second phase widens the search space for the GFH at the expense of a longer runtime, and active flows now may be reconfigured. If the GFH rejects any active flows in the second phase, we revert to the result from the first phase, which is guaranteed to include all active flows in the new traffic plan  $p'$ .

## VII. INSTALLING THE PLAN

After having computed the new plan, the controller propagates the sub-plans to the nodes in the network. The sub-plan sent to a particular infrastructure node defines how the nodes are supposed to route incoming packets, when to reserve transmissions windows for these packets, and when the new plan becomes active, i.e., the traffic plan’s activation time  $t_{\text{act}}$ . In other words, after having computed the new plan, it is installed in two steps:

Firstly, we send the sub-plans, flow-information sets, and the activation time for the new plan via *control channels* to all the nodes in the network while the old traffic plan is still active. Secondly, at the respective  $t_{\text{act}}$ , the nodes switch over to the new flow information sets. Such a time-based two-step update pattern is, e.g., specified in [2] to update the gate control lists of the Time-Aware Shapers.

A transition to the new traffic plan shall only happen if all nodes have agreed that they have received the new sub-plans and will put their sub-plan into action at  $t_{\text{act}}$ . This means, each traffic plan update requires to solve the well-known consensus problem with a deadline that equals  $t_{\text{act}}$  (minus the time nodes need to process update-protocol packets in the worst-case). However, it is well-known that it is impossible to solve the consensus problem in asynchronous systems [34]. In particular, termination within bounded time cannot be guaranteed in such systems. From a practical perspective, though, our specific network update problem has some properties which make this less of an issue:

Firstly, as soon as nodes or links fail, the consensus problem more or less becomes irrelevant since our network may not even work for the active flows anymore. To handle this case, we need additional counter-measures such as redundancy or fail-over on the networking level in the first place, and these have to be incorporated into the traffic plans as well, which is out of the scope of this paper.

Secondly, we can implement *static* real-time control channels between the controller and each network node. These channels could be established when the network is initialized. For example, we can even use the traffic planning approach described in this paper to set up these channels—before processing any time-triggered traffic flow requests issued by “regular” applications—as pinned time-triggered flows such that their associated QoS is not subject to degradation. With real-time channels and bounded update-protocol processing times in the nodes, we effectively execute the network update in a synchronous system.

Therefore, agreeing on the update within a deadline is possible. But even “cheaper” real-time control channel implementations in terms of occupied network resources, e.g., non-time-triggered flows with bandwidth guarantees, are “safe” since the controller can set the activation time to a hyper-cycle boundary “far-enough” in the future. Obviously, this involves a compromise between network resources spent on control channels and the time it takes to install an update.

As discussed in Sec. V, the controller has to postpone the activation time for *sources* of new flows after the transition interval whereas this is not necessary (and it would also be impractical) for infrastructure nodes that forward packets of new flows. This means, there is one activation time for the source nodes of flows from  $\text{ActiveF}(p)$  and the infrastructure nodes. The activation time for the source nodes of new flows however is delayed by  $\lceil \frac{t_{\text{transit}}}{t_{\text{cycle}}} \rceil \cdot t_{\text{cycle}}$  with  $t_{\text{transit}}$  the duration of the longest transition interval. While the flows from  $\text{ActiveF}(p)$  already use the new traffic plan, which also has been deployed to all infrastructure nodes by that time, the source nodes of new flows have to wait a little longer before they are allowed to start sending packets.

## VIII. QoS CONSIDERATIONS

The reconfiguration of active flows can degrade the QoS by introducing jitter. Since we aim for deterministic real-time communication, we must also quantify and possibly contain the QoS degradation caused by reconfigurations to a level acceptable by the applications. Next, we study how the reconfiguration of an active flow can degrade QoS. The level of degradation depends on the “distance” of the flow’s old and new configuration, i.e., the phase shift and the difference in the lengths of the routes. After showing how these properties affect QoS degradation, we present a way how applications can control the degree of degradation.

### A. Computing QoS Degradation

Without reconfiguration, the destination node of an active flow receives the next packet every  $t_{\text{cycle}}$  seconds after the reception of the previous packet, and packets are received in the order they were sent from the source node. Now assume, a new traffic plan  $p'$  supersedes  $p$ , and an active flow  $f \in \text{ActiveF}(p)$  is reconfigured, i.e.,  $\text{config}(f, p) \neq \text{config}(f, p')$ . W.l.o.g., the activation time of the new traffic plan  $p'$  is the start of the  $n+1$ -th cycle of  $f$ . Consequently, the point in time  $t'_{\text{rx}}$  when the destination node receives the  $n+1$ -th packet, which is sent with  $\text{config}(f, p')$ , may differ from the point in time  $t_{\text{rx}}$  when the destination would have received the  $n+1$ -th packet without reconfiguration, i.e., if the  $n+1$ -th packet had been sent with the old  $\text{config}(f, p)$ , cf. Fig 7. We call the magnitude of this deviation  $\Delta t = t'_{\text{rx}} - t_{\text{rx}}$  the (reconfiguration) *jitter*.  $\Delta t$  can be computed with the following theorem.

**Theorem 1** ( $\Delta t$ ). *After a reconfiguration of flow  $f$  from  $(f, \phi, \pi) = \text{config}(f, p)$  to  $(f, \phi', \pi') = \text{config}(f, p')$ , the arrival of the new packets at the destination node deviates by  $\Delta t = (\phi' - \phi) + (l(\pi') - l(\pi)) \cdot t_{\text{perhop}}$  from their respective scheduled arrival according to previous  $\text{config}(f, p)$  with  $l(*)$*

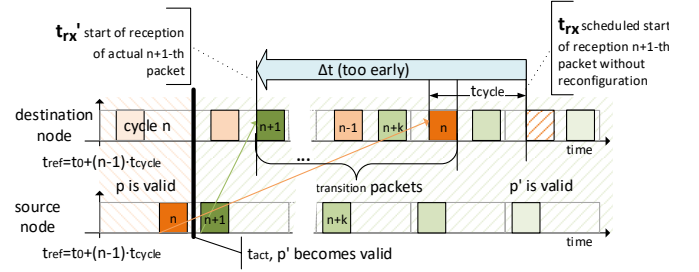


Figure 7: Reconfiguration of an active flow can temporarily cause jitter and packet reordering. Here, the packets with numbers  $n+1$  to  $n+k$  arrive “too early”, i.e., before packet  $n$ .

denoting the number of infrastructure nodes on the candidate path with index  $*$ .

This means,  $\Delta t$  is composed of a term  $(\phi' - \phi)$  expressing the phase-difference, and a term  $(l(\pi') - l(\pi)) \cdot t_{\text{perhop}}$  that expresses the difference between the traversal times of a packet on the old route and the new route.

*Proof.* W.l.o.g. we use the start  $n$ -th cycle as reference time  $t_{\text{ref}}$ . In the  $n$ -th cycle, the source node starts transmitting the  $n$ -th packet at  $t_{\text{ref}} + \phi$ . The reception of this packet by the destination node starts at  $t_{\text{ref}} + \phi + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi) \cdot t_{\text{perhop}} + t_{\text{dst}}$  (cf. Sec. III-D). If  $p$  remained valid, the reception of the  $n+1$ -th packet then would start  $t_{\text{cycle}}$  later at  $t_{\text{rx}} = t_{\text{ref}} + \phi + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi) \cdot t_{\text{perhop}} + t_{\text{dst}} + t_{\text{cycle}}$ .

However, the new plan  $p'$  and thus  $\text{config}(f, p')$  becomes valid at  $t_{\text{ref}} + t_{\text{cycle}}$ , which is the start of the  $n+1$ -th cycle. The reception of the  $n+1$ -th packet sent by the source node with  $\text{config}(f, p')$  starts at  $t'_{\text{rx}} = t_{\text{ref}} + t_{\text{cycle}} + \phi' + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi') \cdot t_{\text{perhop}} + t_{\text{dst}}$  at the destination node. By inserting these values, we get  $\Delta t = t'_{\text{rx}} - t_{\text{rx}} = (\phi' - \phi) + (l(\pi') - l(\pi)) \cdot t_{\text{perhop}}$ .  $\square$

If  $\Delta t > 0$ , there is an additional delay between the last packet of  $\langle f, p \rangle$  and the first packet of  $\langle f, p' \rangle$ . Intuitively, a reconfiguration that results in either a longer path or a phase increment (i.e.,  $\phi' > \phi$ ), or both, results in  $\Delta t > 0$ . If  $\Delta t < 0$ , the new packets from  $\langle f, p' \rangle$  arrive too early. This happens, if either the reconfiguration results in phase decrement ( $\phi' < \phi$ ), or a shorter route  $l(\pi') < l(\pi)$ , or both of these. Potentially, this can lead to a situation, where packets from  $\langle f, p' \rangle$  overtake the last packets from  $\langle f, p \rangle$  in the network if the relative end-to-end deadline is allowed to be greater than the cycle time.

We can upper bound the number of packets which possibly arrive in a different order than the order in which they were sent, and do not have an inter-arrival time  $t_{\text{cycle}}$  by  $n_{\text{transition}} \leq 2 \cdot \left\lceil \frac{|\Delta t|}{t_{\text{cycle}}} \right\rceil$ . The term  $2 \cdot \left\lceil \frac{|\Delta t|}{t_{\text{cycle}}} \right\rceil$  approximates the worst-case where old and new packets arrive interleavedly at the destination node until all old packets are delivered.

It is important to note that any individual packet that is sent by a source node will always arrive within the end-to-end deadline bounds. Reconfigurations only affect the relative inter-arrival times in a packet train.



Table I: Overview of evaluation scenario parameters.

Fig.	$ \text{ActiveF}(p) $	$ \text{ReqF}(p) $	flows to remove	$n_{\text{ub}}$	$t_{\text{cycle}}$ [μs]	$n_{\text{path}}$	network	$\Sigma$ ingress rate
8a	500	25 (poiss.)	25 (poiss.)	$\leq \{50, 100\}$	$\{250, 500, 1000, 2000\}$	3	ring(64,3)	~615MB/s
8b,8c	800	50 (poiss.)	50 (poiss.)	$\leq \{50, 100\}$	$\{250, 500, 1000, 2000\}$	3	ring(64,3)	~984MB/s
9, 10	250	25 (det.)	25 (det.)	$\leq 100$	$\{200, 250, 500\}$	3	ring(64,3)	~601MB/s
11a, 11b	500	50 (det.)	50 (det.)	$\leq 100$	$\{250, 500\}$	5	var., 49 nodes	~984MB/s
11a, 11c	500	50 (det.)	50 (det.)	$\leq 100$	$\{250, 500\}$	5	var., 81 nodes	~984MB/s
$t_{\text{trans}}$ (packet size)	$\in \{1\mu\text{s}, 3\mu\text{s}, 5\mu\text{s}, 12\mu\text{s}\}$ (corresponds to 125 B – 1500 B packets on 1 Gbit s <sup>-1</sup> links)							
flow clusters	$\in \{1, 2, 4, 8, 16, 32\}$ (subject to $ \text{ReqF}(p) $ and num. flows to remove)							
network parameters	processing delay $t_{\text{proc}} = 2\mu\text{s}$ , propagation delay $t_{\text{prop}} = 1\mu\text{s}$							

### B. Restricting QoS Degradation

Applications can provide the planner with QoS degradation constraints in the form of thresholds on  $|\Delta t|$  and the number of affected packets. If the destination node has no facilities to handle packet reorderings, e.g., has no buffer, then such constraints can be used to ensure that all packets are received in the sending order. The QoS degradation constraint results in the following condition for  $\mathbf{G}(p')$ :

**QoS** For each  $f \in \text{ActiveF}(p)$ , there exists no configuration  $c \in \text{cand}(f, p')$ , where the reconfiguration from  $\text{config}(f, p)$  to  $c$  exceeds the threshold on  $|\Delta t|$ .

Analog to Condition 1 from Sec. V-C, we can prevent reconfigurations that violate the QoS condition by *locking*. This can be achieved with minimal overhead since the planner anyway traverses  $c \in \text{cand}(f, p')$ , and only needs to check the QoS condition for each candidate  $c$  of  $f$  that is not locked already due to transition interference.

## IX. EVALUATION

We evaluated a prototypical C++-implementation of the planner. The evaluation scenarios consist of a sequence of planning rounds. In each round, the planner processes a set of flow requests which can include requests for new flows as well as requests for the removal of active flows. Besides the conflict graph adjustments, and the computation of the new traffic plan  $p'$ , the planner also validates the absence of configuration conflicts in the new traffic plan in each round.

### A. Setup and Parameters

By default, we used a desktop-grade computer with Intel i7-10700K (8 cores) and 16 GB RAM for the evaluations.

Table I gives an overview over the evaluation parameters which have been derived from typical industrial use cases [13], [35]. For each flow, we draw the values for  $t_{\text{trans}}$  and  $t_{\text{cycle}}$  uniformly at random from the respective set. The number of new flows to add, i.e.,  $|\text{ReqF}(p)|$  and the number of flows to remove is either drawn from a truncated Poisson distribution (poiss.) with averages as stated in Tab. I, or is deterministic (det.) for each planning round. The expected ingress traffic data rate, i.e., the aggregate of the traffic entering the network, is computed from the random distributions for  $t_{\text{cycle}}$  and  $t_{\text{trans}}$ . We place/remove the flows in clusters in the network to simulate control units connected to multiple sensors and actuators, where all flows in a cluster start or target one common “cluster” node.

The cluster sizes correspond to commonly found configurations of I/O bays of industrial control units. For example, to add 25 new flows the request may contain three clusters of size 1, 8 and 16 flows, respectively, or any other combination that adds up to 25 flows. By default, 20% of the new flows are pinned permanently, and we limit  $\Delta t$  for each remaining active flow to  $t_{\text{cycle}} - t_{\text{trans}}$  such that packets arrive in order at the destination node. We omit specifying an explicit value for  $t_{\text{e2e}}$ . Instead, we use a k-shortest path algorithm which provides us with the candidate paths with the lowest end-to-end delays that are possible within the given topology.

In Sec. IX-B and IX-C, we consider circular networks with  $n$  nodes where each node is connected to the next  $k$  nodes in both directions (denoted by  $\text{ring}(n, k)$ ). Besides being a common building block in, e.g., industrial automation networks [36] or sensor networks, the regularity of  $\text{ring}(n, k)$  topologies eliminates several sources of uncertainty that could, e.g., result in infeasible problem instances, or otherwise limit the problem space that we can evaluate. For example, think of topologies with a bottleneck link connecting two sub-networks that—depending on how many flows have to cross that bottleneck—will result in massively different numbers of “acceptable” flows for the same topology. In contrast, in  $\text{ring}(n, k)$  topologies, we can expect similar behavior for the flow placements since each node in the network has the same degree and there is an equal number of alternative paths between any pair of nodes for this topology. In Sec. IX-D, we investigate the effects of different network topologies considering additional graph models, too.

In Tab. I, column  $n_{\text{ub}}$  denotes the upper bound on how many candidates per flow the planner may add at most to  $\mathbf{G}(p')$ . Likewise,  $n_{\text{path}}$  is the upper bound on the number of candidate paths per flow. Both,  $n_{\text{ub}}$  and  $n_{\text{path}}$ , can be considered parameters of our planner which result in different trade-offs wrt. runtime and schedulability. For example, for meshed networks, it is intuitive to expect better results with more candidate paths at the price of longer runtimes [37], but we expect a diminishing return with regard to the number of candidate paths due to shared sub-paths on the k-shortest paths. Therefore, we use 3-5 candidate paths per flow, which already yields a very high schedulability in our evaluations (similar to [16]).

### B. Runtime

We evaluated 60 scenarios wrt. runtime for each combination of average active flows (500, 800) and per-flow candidate set

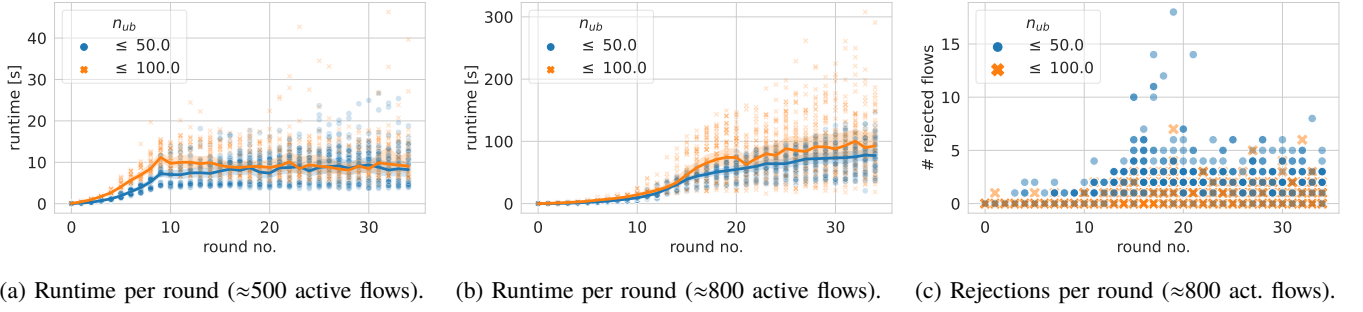


Figure 8: Runtimes and number of rejected flows per planning round (500 and 800 active flows).

increment limit (50, 100). Fig. 8 shows the results for 35 planning rounds.

We plot the *total* runtime for each planning round in Fig. 8a and 8b, which includes the time to compute the candidate paths for new flows, all the operations on the conflict graph (adding, locking, pinning and removing candidate configurations) and the GFH runtime. For these scenarios, we measured up to 9.072 GB RAM usage by the planner.

In the first 10 (16) rounds, we always issued requests to add 50 flows (no flow-removals) to initialize the conflict graph from scratch until  $\text{ActiveF}(p)$  contains  $\approx 500$  (800) flows. During these initialization rounds, we observe an increasing total runtime. After the initialization rounds, the planner receives requests to remove and add flows.

In the smaller scenarios with  $\approx 500$  flows (cf. Fig. 8a), the runtime per round plateaus (avg:  $8.24 \pm 3.58$  s,  $n_{ub}=50$ ;  $9.2 \pm 4.03$  s,  $n_{ub}=100$ ) after the initialization rounds. In other words, here it takes on average less than 10 s to exchanges 25 active flows (and their configurations) against 25 new flows (and the respective new configurations) even though the planner can reconfigure  $\approx 400$  active flows. Here, we also have a quite “stable” conflict graph size in the post-initialization rounds which results in the comparatively constant runtimes per round.

In Fig. 8b, the runtimes for the bigger scenarios with  $\approx 800$  active flows are depicted. Here, the runtime continues to grow after the 16 initialization rounds (max: 146.83 s for  $n_{ub}=50$ ; 307.72 s for  $n_{ub}=100$ ). This behavior can be explained with Fig. 8c where we plot for each scenario how many new flows the planner rejected in each round. As discussed in Sec. V-B, after the first pass over the  $\phi$ - $\pi$ -space the planner adds more candidates for *active flows* if flows from the previous request had to be rejected. This means, we actually observe the desired expansion of the solution-space since we have to fit  $\approx 300$  more flows in the same network as for the smaller scenarios, This causes the planner to grow the conflict graph more aggressively—resulting in longer runtimes per round.

The effects of varying  $n_{ub}$  can also be observed: if the planner may add up to 100 additional candidates per flow, it can admit more new flows, but generally takes longer: In the scenarios with 800 flows the planner rejected on average 0.84 (0.23) flows per request set after the initialization rounds for  $n_{ub} = 50$  ( $n_{ub} = 100$ ) per round.

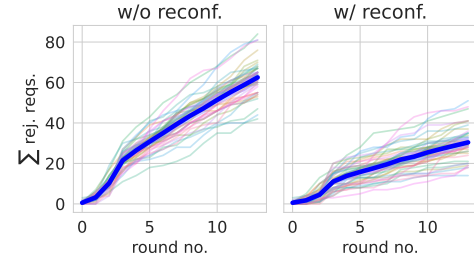


Figure 9: Defensive vs. offensive planning: Cumulative rejected flows (thin lines: per scenario, bold line: average).

### C. Comparison

Next, we compare defensive and offensive traffic planning wrt. schedulability, and show that the new algorithms such as the GFH are required to make offensive traffic planning feasible.

The evaluation scenarios vary with respect to how many active flows are on average permanently pinned to their configuration. We increase the ratio of pinned flows from 0% to 100% in 20%-increments and evaluated 40 scenarios for each round and pinning-setting. Here, there is no QoS restriction on  $\Delta t$  for each active flow that can be reconfigured.

1) *Defensive vs. offensive planning*: In Fig. 9, we plot the cumulative number of flows rejected by the planner over 14 rounds. By pinning every active flow (cf. w/o reconfiguration, permanently pinning 100% of flows in Fig. 9), our approach performs defensive traffic planning, which results in a total of 62.5 rejected new flows on average. In comparison, if the planner performs offensive traffic planning less than half as many flows—30.4 flows on average—are rejected (cf. w/ reconfiguration, permanently pinning 0% of flows in Fig. 9).

2) *Comparison with ILP*: Since the GFH is a heuristic, the question arises how “optimal” the GFH results are. To answer this, we compare the GFH against a drop-in integer linear programming implementation, which is given the same input as the GFH, namely, the conflict graph  $\mathbf{G}(p')$  and the flow sets  $\text{ActiveF}(p)$  and  $\text{ReqF}(p)$ , and has to compute  $p'$  which optimizes Eq. 2. To make it clear, this ILP does not encode the whole traffic planning problem itself, but requires the planner to encode the traffic planning constraints in  $\mathbf{G}(p')$ . The ILP constraints are given in Eq. 3-5, and, except for the objective, correspond to the ILP used as “sure” algorithm in [16]. We have two sets of binary decision variables: For

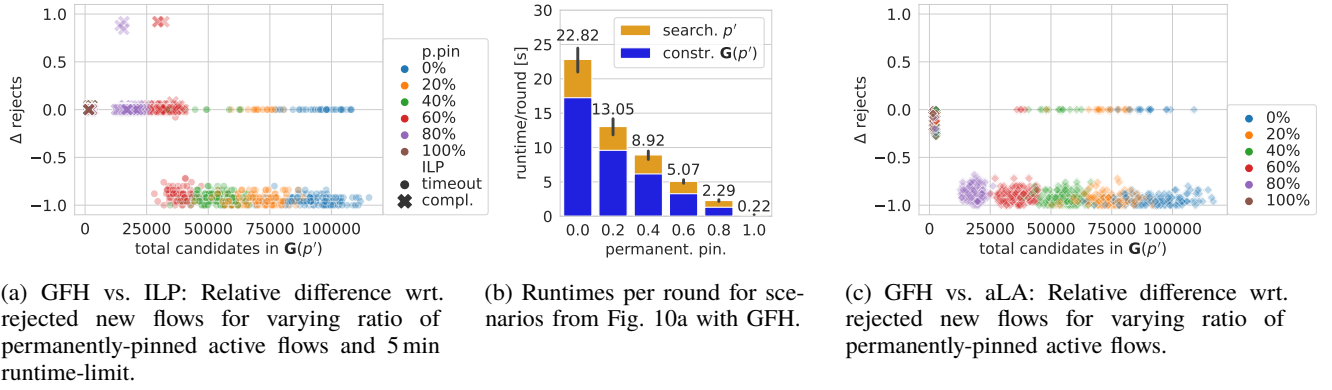


Figure 10: Comparison of schedulability and runtime for varying ratios of non-reconfigurable (permanently-pinned) active flows.

each flow,  $X_f[f] \in \{0, 1\}$  encodes whether flow  $f$  can be admitted. Similarly, a value of 1 for  $X_c[v] \in \{0, 1\}$  encodes that configuration  $v$  is added  $C$ .

$$\max \sum_{f \in \text{ActiveF}(p)} X_f[f] + \sum_{f \in \text{ReqF}(p)} \frac{X_f[f]}{|\text{ActiveF}(p) \cup \text{ReqF}(p)|} \quad (3)$$

$$\text{subject to: } \forall (v, u) \in \mathcal{E} : X_c[v] + X_c[u] \leq 1 \quad (4)$$

$$\forall f \in \text{ActiveF}(p) \cup \text{ReqF}(p) : X_f[f] \leq \sum_{v \in \text{cand}(f, p)} X_c[v] \quad (5)$$

For this comparison, we saved the conflict graph instances from the last four planning rounds from Fig. 9—presumably the largest conflict graphs in each scenario—to disk, and solved the corresponding ILP instances with a tool-chain using Julia [38], JuMP [39] and Gurobi v9.1.1 [40]. The ILP solver had a runtime limit of 5 min (for comparison, the max. GFH runtime was 8.3 s). Due to the higher memory requirements (up to 56.694 GB) of the ILP tool chain, we used a server-grade computer with two AMD EPYC 7401 processors (each 24 cores) and 256 GB RAM, but both, GFH and ILP solver, were limited to using max. 16 threads as in the other evaluations.

Fig. 10a plots for each planning round the relative difference in the number of rejected new flows  $\Delta \text{rejects} = \frac{\text{rejects GFH} - \text{rejects ILP}}{\text{new flows (total)}}$  over the number of candidate configurations in the conflict graph. If the ILP solver rejected fewer flows, i.e., computed a better result we have  $\Delta \text{rejects} > 0$ , and vice-versa. In Fig. 10a, the ILP solver could provide better solutions for small conflict graphs, and, except for a few outliers where GFH reverted to the result from the first phase, the advantage of the ILP over GFH is small. Yet, once conflict graphs contain 30,000 or more candidate configurations, the ILP solver frequently hits the time-limit (for 0% pinned flows: GFH avg=4.9 s  $\pm$  1.8 s; ILP avg=290.7 s  $\pm$  71.4 s) and would reject most new flows. This means even if we factor in the performance benefits of the conflict graph modeling itself (cf. [16]), the GFH algorithm pushes the boundaries for offensive traffic planning further out compared to state-of-the-art exact approaches (in this case, integer linear programming) which are limited to either small scenarios or a small fraction of reconfigurable flows (cf. pinning 80% of all flows).

For reference, we also depict the average runtimes of the planner using the GFH for each round of the scenarios from

Fig. 10a for the different ratios of permanently pinned flows (error-bars indicating the variance of the total runtime per round). Figure 10b highlights the flexibility of our approach: if we “re-interpret” flow pinning as a tuning parameter, e.g., used for probabilistic pinning of new flows by the planner, we can cover the full range between offensive planning with high schedulability and extremely fast defensive planning (cf. Fig. 10b: max. 250 ms for processing rounds with  $\approx 250$  active flows on the server-grade machine).

3) *Comparison with randomized heuristic:* We also compare the GFH against another heuristic. To the best of our knowledge, the “closest” competitor is the adapted version of Luby’s maximal independent vertex set algorithm [41], which has been presented in our prior work [16]. The adapted Luby’s algorithm (aLA) is a randomized, iterative heuristic for the computation of  $C$ . In each iteration of the aLA, eligible configurations are probabilistically added to  $C$ . How likely it is for an eligible configuration  $c$  to be added to  $C$  depends on the number of conflicts of  $c$  (less conflicts increases the probability), and the number of configurations already in  $C$  that belong to the same flow  $f$  as  $c$  are already included in  $C$  ( $f$  not admitted increases the probability). The latter property adds some level of flow-awareness which the original Luby’s algorithm lacks. We additionally modify the aLA heuristic to account for the different importance of flows in  $\text{ActiveF}(p)$  and  $\text{ReqF}(p)$  (cf. Eq. 2): To ensure that no configuration of  $\text{ReqF}(p)$  shadows a configuration of a flow in  $\text{ActiveF}(p)$ , we—similar to the GFH—first search configurations for active flows in the subgraph of  $G(p')$  induced by all the candidates of  $\text{ActiveF}(p)$ , i.e., the aLA “sees” only configurations and conflicts for flows in  $\text{ActiveF}(p)$ , before we search for configurations for  $\text{ReqF}(p)$  in the “full”  $G(p')$ . In other words, in the second phase of the heuristic the aLA “sees” configurations for all flows but can only select eligible configurations for flows in  $\text{ReqF}(p)$  that do not conflict with the configurations found during the first phase.

We implemented the aLA heuristic in Julia and evaluated it with same methodology introduced for the comparison of GFH and ILP. That is, we save the conflict graph instances to disk and use them as input for the aLA heuristic to compute  $p'$  and then compare the quality of the aLA heuristic and GFH in terms of admitted flows. The results for the relative number of rejected flows  $\Delta \text{rejects} = \frac{\text{rejects GFH} - \text{rejects aLA}}{\text{new flows (total)}}$  are plotted in

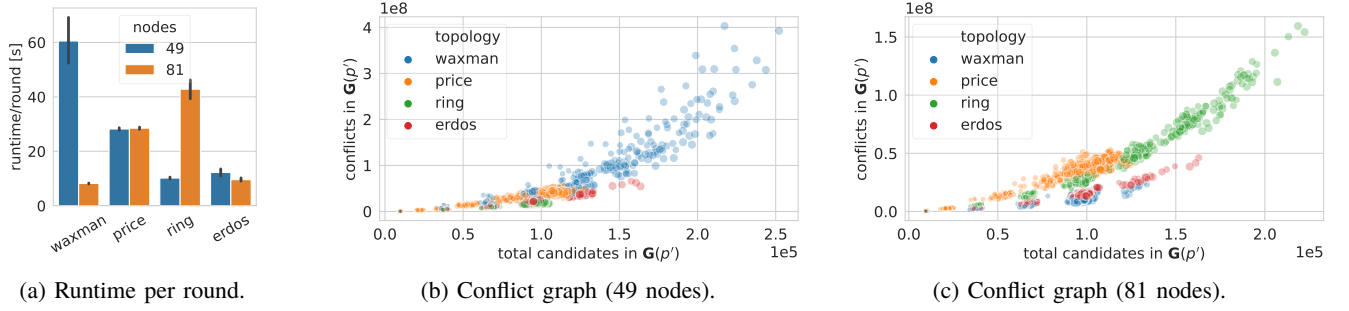


Figure 11: Impact of different network topologies on total runtime and conflict graph size.

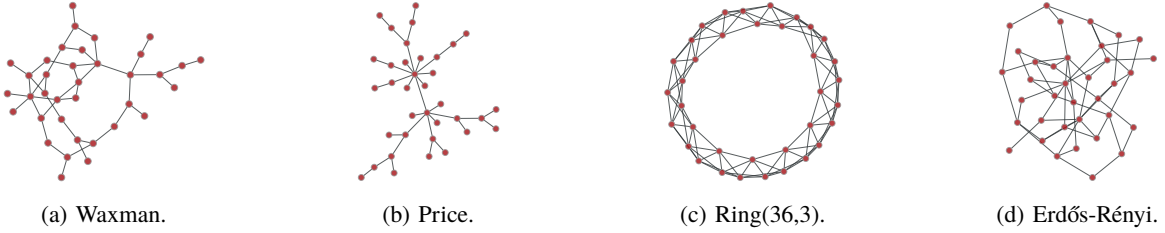


Figure 12: Example graph topologies for the different graph models with 36 nodes each.

Fig. 10c. We see that the aLA heuristic never outperforms the GFH, i.e., different to the ILP, the aLA heuristic was never able to admit more flows than the GFH. In absolute numbers, the aLA heuristic on average rejects 10.3 more than the GFH (rejected flows: GFH  $\text{avg}=3.7 \pm 4.1$ ; aLA  $\text{avg}=13.9 \pm 9.4$ )—given a total of 25 new flows in  $\text{ReqF}(p)$ . While for ties for smaller  $G(p')$ , we predominantly observed that both, GFH and the aLA heuristic, admit similarly many ( $\approx 20$ ) new flows, we observed that ties for larger  $G(p')$  predominantly occur if both heuristics cannot admit any new flow, i.e., they have similarly “bad” results. Remarkable is also the steep drop-off in the relative performance of the aLA once  $G(p')$  contains several thousand candidate configurations ( $\Delta\text{rejects}$ :  $<5000$  candidates  $\text{avg}=-0.10$ ;  $\geq 5000$  candidates  $\text{avg}=-0.85$ ). Considering that our aLA heuristic was implemented as a single-threaded Julia implementation, it is not surprising that it resulted in higher runtimes compared to the GFH (for 0% pinned flows: GFH  $\text{avg}=4.9 \text{ s} \pm 1.6 \text{ s}$ ; aLA heuristic  $\text{avg}=29.2 \text{ s} \pm 9.3 \text{ s}$ ). However, while it is reasonable to expect that—being much simpler—a multi-threaded aLA heuristic will match or even undercut the GFH runtime, this is paid for with much worse quality, especially for larger conflict graphs.

4) *Summary*: Firstly, we showed in our evaluations that reconfiguring active flows during a traffic plan update reduces the number of rejected new flows. Secondly, our evaluations highlight that the configuration selection strategy of the GFH is worth its complexity. The GFH mostly stayed within close distance to the quality of the exact ILP solutions for smaller conflict graphs, and the GFH continued to provide good solutions within seconds even for those “larger” conflict graphs where the ILP ran into the runtime-limit. On the other hand, in our evaluations, the simpler aLA heuristic at best managed to tie with the GFH, and on average the results of the aLA heuristic could admit only half the number of flows admitted by the GFH.

#### D. Network

To investigate the effects of network topology and network size, we generated scenarios for networks with 49 nodes and 81 nodes for different graph models, namely, Waxman, Price,  $\text{ring}(n, k)$ , and Erdős-Rényi, using graph generators from [42], [43]. Examples for the different network topologies are provided in Fig. 12. These networks cover a variety of different network characteristics, like diameter, number of alternative paths and scale-free property. Thus, we can evaluate the fitness of our approach for different network types. We evaluated 60 scenarios for each size-topology combination with 500 active flows on average. After the initialization, the planner tries adding 50 new flows and removing 50 active flows in each planning round.

The average runtimes after initialization and warm-up rounds are plotted in Fig. 11a. We observe that a larger network does not per-se result in higher runtimes or bigger conflict graphs, cf. Fig 11b, 11c. Waxman networks with 49 nodes result in the largest conflict graph sizes overall and highest average (60.5 s) and maximal (283.72 s) total runtime per round. In Price networks with only one path between every two nodes, the planner cannot resolve conflicts via routing. Compared to the other topologies, on average more than ten times as many new flows were rejected per round (49n.  $\text{avg}=10.4 \pm 5.12$  rej./round; 81n.  $\text{avg}=11.6 \pm 5.19$  rej./round) in scenarios with Price network topologies. In scenarios with Price networks, GFH often required all re-runs, and we measured the third-highest total runtimes. This suggests that our approach scales with the actual difficulty rather than just the network size.

## X. CONCLUSION AND OUTLOOK

In this paper, we presented a novel approach for offensive dynamic traffic planning, which allows for the reconfiguration of active flows to achieve better network utilization, and a novel algorithm for computing these traffic plans. We are able



to quantify and control the QoS degradation a flow may suffer during such reconfigurations. Our evaluations show that we can efficiently compute updated traffic plans for scenarios with hundreds of active flows.

Interesting directions for future work include investigating memory-access optimized conflict graph data-structures and the relaxation of the zero-queuing constraint, as well as protocols and concepts for error-handling while computing the traffic plan and during traffic plan deployment.

## REFERENCES

- [1] N. Finn and P. Thubert, "Deterministic Networking Problem Statement," RFC 8557, May 2019. [Online]. Available: <https://rfc-editor.org/rfc/rfc8557.txt>
- [2] IEEE Computer Society, "IEEE Standard for Local and Metropolitan Area Network-Bridges and Bridged Networks," *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, 2018.
- [3] F. Dürr and N. G. Nayak, "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. ACM, 2016, pp. 203–212.
- [4] KUKA, "Hello industrie 4.0 \_we go digital."
- [5] —, "Hello industrie 4.0 \_we connect you."
- [6] M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner, "Runtime reconfiguration of time-sensitive networking (TSN) schedules for Fog Computing," in *2017 IEEE Fog World Congress (FWC)*, 2017, pp. 1–6.
- [7] P. Danielis, H. Puttnies, E. Schweissguth, and D. Timmermann, "Real-Time Capable Internet Technologies for Wired Communication in the Industrial IoT—a Survey," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 266–273.
- [8] F. Prinz, M. Schoeffler, A. Lechler, and A. Verl, "Dynamic real-time orchestration of i4.0 components based on time-sensitive networking," *Procedia CIRP*, vol. 72, pp. 910–915, 2018.
- [9] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 323–334.
- [10] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental Flow Scheduling and Routing in Time-Sensitive Software-Defined Networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, 2018.
- [11] Z. Li, H. Wan, Z. Pang, Q. Chen, Y. Deng, X. Zhao, Y. Gao, X. Song, and M. Gu, "An Enhanced Reconfiguration for Deterministic Transmission in Time-Triggered Networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1124–1137, Jun. 2019.
- [12] V. Balasubramanian, M. Aloqaily, and M. Reisslein, "An SDN architecture for time sensitive industrial IoT," *Computer Networks*, vol. 186, p. 107739, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128620313256>
- [13] IEC SC65C/WG18 and IEEE 802.1 TSN TG. IEC/IEEE 60802 TSN profile for industrial automation |. [Online]. Available: <https://1.ieee802.org/tsn/iec-ieee-60802/>
- [14] Z. Pang, X. Huang, Z. Li, S. Zhang, Y. Xu, H. Wan, and X. Zhao, "Flow Scheduling for Conflict-Free Network Updates in Time-Sensitive Software-Defined Networks," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 1668–1678, Mar. 2021.
- [15] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A Centralized "Zero-queue" Datacenter Network," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. ACM, 2014, pp. 307–318.
- [16] J. Falk, F. Dürr, and K. Rothermel, "Time-Triggered Traffic Planning for Data Networks with Conflict Graphs," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 124–136.
- [17] J. W. Guck, A. V. Bemten, and W. Kellerer, "DetServ: Network Models for Real-Time QoS Provisioning in SDN-Based Industrial Environments," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1003–1017, 2017.
- [18] M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat, "Self-configuration of IEEE 802.1 TSN networks," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1–8.
- [19] A. Nasrallah, V. Balasubramanian, A. Thyagaturu, M. Reisslein, and H. ElBakoury, "Reconfiguration Algorithms for High Precision Communications in Time Sensitive Networks," in *2019 IEEE Globecom Workshops (GC Wkshps)*, 2019, pp. 1–6.
- [20] T. Gerhard, T. Kobzan, I. Blöcher, and M. Hendel, "Software-defined Flow Reservation: Configuring IEEE 802.1Q Time-Sensitive Networks by the Use of Software-Defined Networking," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2019, pp. 216–223.
- [21] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive Software-defined Network (TSSDN) for Real-time Applications," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. ACM, 2016, pp. 193–202.
- [22] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegl, and G. Mühl, "ILP-based Joint Routing and Scheduling for Time-triggered Networks," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017, pp. 8–17.
- [23] J. Falk, F. Dürr, and K. Rothermel, "Exploring Practical Limitations of Joint Routing and Scheduling for TSN with ILP," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2018, pp. 136–146.
- [24] A. Atallah, G. Bany Hamad, and O. Ait Mohamed, "Routing and Scheduling of Time-Triggered Traffic in Time Sensitive Networks," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2019.
- [25] W. Steiner, "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 375–384.
- [26] R. S. Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1Qbv Gate Control List Synthesis Using Array Theory Encoding," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 13–24.
- [27] W. Steiner, S. S. Craciunas, and R. S. Oliver, "Traffic Planning for Time-Sensitive Communication," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, 2018.
- [28] M. Pahlevan, N. Tabassam, and R. Obermaier, "Heuristic list scheduler for time triggered traffic in time sensitive networks," *ACM SIGBED Review*, vol. 16, no. 1, pp. 15–20, 2019.
- [29] A. Syed and G. Fohler, "Efficient offline scheduling of task-sets with complex constraints on large distributed time-triggered systems," *Real-Time Systems*, vol. 55, no. 2, pp. 209–247, 2019.
- [30] J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.
- [31] ISO, IEC, and IEEE, "ISO/IEC/IEEE International Standard – Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 1Q: Bridges and bridged networks AMENDMENT 1: Path control and reservation," pp. 1–122, 2017.
- [32] V. Dadwal, Intel Corporation. Adopting Time-Sensitive Networking (TSN) for Automation Systems. Intel. [Online]. Available: <https://www.intel.com/content/www/us/en/develop/articles/adopting-time-sensitive-networking-tsn-for-automation-systems-0.html>
- [33] M. R. Garey and D. S. Johnson, *Computers and Intractability*, ser. A Guide to the Theory of NP-Completeness. Freeman, 1979.
- [34] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
- [35] Industrial Internet Consortium. Time sensitive networks for flexible manufacturing testbed characterization and mapping of converged traffic types. [Online]. Available: [https://www.iiconsortium.org/pdf/IIC\\_TSN\\_Testbed\\_Char\\_Mapping\\_of\\_Converged\\_Traffic\\_Types\\_Whitepaper\\_20180328.pdf](https://www.iiconsortium.org/pdf/IIC_TSN_Testbed_Char_Mapping_of_Converged_Traffic_Types_Whitepaper_20180328.pdf)
- [36] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehr, and F. Dürr, "Scaling TSN Scheduling for Factory Automation Networks," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*, pp. 1–8.
- [37] J. Falk, F. Dürr, S. Linsenmayer, S. Wildhagen, B. Carabelli, and K. Rothermel, "Optimal routing and scheduling of complementary flows in converged networks," in *Proceeding of the 27th International Conference on Real-Time Networks and Systems*, ser. RTNS'19, 2019.
- [38] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: <https://doi.org/10.1137/141000671>



- [39] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.
- [40] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2020. [Online]. Available: <http://www.gurobi.com>
- [41] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.
- [42] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.
- [43] T. P. Peixoto, "The graph-tool python library," *figshare*, 2014. [Online]. Available: [http://figshare.com/articles/graph\\_tool/1164194](http://figshare.com/articles/graph_tool/1164194)



**Jonathan Falk** joined the Distributed Systems group of the Institute for Parallel and Distributed Systems of the University of Stuttgart after receiving his Master's degree in Electrical Engineering and Information Technology. His research interests include time-sensitive communication, time-triggered networks, and cyber-physical systems.



**Sukanya Bhowmik** received her doctoral degree from University of Stuttgart, Germany, in 2017. She is currently working as a postdoctoral researcher at the Distributed Systems research group of University of Stuttgart. Her research interests include stream/complex event processing, high performance communication middleware, in-network event processing, software-defined networking, and distributed graph processing, with a focus on scalability, line-rate performance, resource efficiency, and adaptability aspects.



**Kurt Rothermel** received his doctoral degree in Computer Science from University of Stuttgart in 1985. From 1986 to 1987 he was a Post-Doctoral Fellow at IBM Almaden Research Center in San José, U.S.A., and then joined IBM's European Networking Center in Heidelberg. In 1990, he became a Professor for Computer Science at the University of Stuttgart. From 2003 to 2011 he was head of the Collaborative Research Center Nexus (SFB 627), conducting research in the area of mobile context-aware systems. His research interests are in the field of distributed systems, computer networks, and mobile systems.



**Heiko Geppert** received his Master degree in Software Engineering from the University of Stuttgart, Germany, in 2019. He is currently working on his PhD at the Distributed Systems group of the University of Stuttgart. His research interests include distributed graph processing and network scheduling, with a focus on fast and scalable solutions.



**Frank Dürr** is a senior researcher and lecturer at the Distributed Systems Department of the Institute of Parallel and Distributed Systems (IPVS) at University of Stuttgart, Germany. He received both, his doctoral degree and diploma in computer science from University of Stuttgart. Frank Dürr is currently leading the mobile computing and the software-defined networking (SDN) & time-sensitive networking (TSN) groups of the Distributed Systems Department. Besides several scientific papers in the area of SDN and TSN, including for instance concepts

for scheduling and routing in TSN/SDN networks, he has given tutorials on SDN at several conferences. Besides SDN and TSN, his research interests include mobile and pervasive computing, location privacy, and cloud computing aspects overlapping with these topics like mobile cloud and edge computing, or data center networks.