Time-Triggered Traffic Planning for Data Networks with Conflict Graphs

Jonathan Falk
IPVS
University of Stuttgart
Stuttgart, Germany
jonathan.falk@ipvs.uni-stuttgart.de

Frank Dürr
IPVS
University of Stuttgart
Stuttgart, Germany
frank.duerr@ipvs.uni-stuttgart.de

Kurt Rothermel
IPVS
University of Stuttgart
Stuttgart, Germany
kurt.rothermel@ipvs.uni-stuttgart.de

***Abstract*—Traffic planning is the key enabler of time-triggered real-time communication in distributed systems, and it is known to be notoriously hard. Current approaches predominantly tackle the problem in the domain of the traffic planning problem, e.g., by formulating constraints on the transmission schedules for individual data streams, or the links used by the data streams. This results in a high degree of coupling of the configuration of an individual data stream and the global (network-wide) traffic configuration with detrimental effects on the scalability and runtime of the planning phase.**

In contrast, we present a configuration-conflict graph based approach, which solves the original traffic planning problem by searching an independent vertex set in the conflict graph. We show how to derive the configuration-conflict graph, and discuss the conceptual advantages of this approach. To show the practical advantages of the conflict-graph based traffic planning approach we additionally present a proof-of-concept implementation and evaluate it against a reference ILP-based implementation. In our evaluations, our proof-of-concept implementation of the conflict-graph based approach outperforms the reference ILP and is more memory efficient, making it a promising alternative to current constraint-based traffic planning approaches.

***Index Terms*—time-triggered traffic, conflict graph, routing, scheduling**

I. Introduction

Many distributed applications rely on guaranteed real-time communication. Examples of such applications can be found in many domains, e.g., vehicular on-board networks, industrial manufacturing, or power-grid automation. A well-established paradigm to achieve real-time communication is the combination of time-triggered transmissions with a network-wide coordinated configuration of routes and schedules.

Currently, the ongoing work of the Time-Sensitive Networking (TSN) working group has brought several mechanisms for real-time communication into standard IEEE Ethernet networks. In this paper, we focus on a key mechanism introduced by the TSN working group, the so-called Time-Aware Shaper (TAS) which is suitable for time-triggered traffic. The TAS allows to exert fine-grained temporal control over the forwarding process *in the switches*, i.e., TAS allows to schedule the transmissions at the switches according to a cyclic time-schedule. We use this combination of the TAS with meshed Ethernet networks as the technological backdrop of this paper and revisit the traffic planning problem, i.e., the problem of computing routes and time-schedules. In the traffic planning phase, different communication requirements, e.g., frequency of transmissions, amount of data per transmission, delay-bounds, etc., as well as the network topology have to be accounted for. In TSN networks TAS requires a time-schedule to be calculated for every switch. Calculating such schedules is NP-hard in general, as is the combined problem of joint routing and scheduling [1].

A large body of work has been dedicated to solve the traffic planning problem with constraint-based programming methods, such as (integer) linear programming (ILP), satisfiability theories (SAT, SMT), or constraint-based heuristics. In these approaches, the traffic planning problem is formulated directly as a set of constraints on the routing variables and scheduling variables. A solution to the traffic planning problem is then equivalent to an assignment of values to the routing and scheduling variables with which the hosts and the nodes in the network can be configured such that the conditions necessary for real-time communication are satisfied. Operating directly in the domain of the original traffic planning problem, i.e., formulating constraints directly for the routes of the packets and the transmission schedules at the network elements, results in a high-dimensional, highly-coupled problem. For example, if the variables for a single stream are changed somewhere in the network, this can trigger a cascade of constraint violations for many other streams which share the same network resources (queues, links), e.g., if the change inflicts collisions in the transmission schedules. Thus, solving the original traffic problem has the drawback of high coupling between a configuration of a single stream and the *global* configuration for all streams.

In contrast, we present a different, conflict-graph based approach to solve the problem of traffic planning for time-triggered traffic. Conflict graphs have been used in the context of scheduling problems before [2]–[5]. Compared to the classic constraint-based approach for time-triggered traffic planning, our conflict-graph based approach does not operate directly on the level of routing and scheduling variables, but on the level of stream *configurations* where one configuration is one possible assignment of the routing and scheduling variables

This paper has passed an Artifact Evaluation process. For additional details, please refer to http://2020.rtas.org/artifact-evaluation/.

Accepted for publication in Proc. 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia, April 21-24, 2020

for a particular stream. In our approach, we first construct a configuration-conflict graph from the original traffic planning problem. Then, we search an independent vertex set in this conflict graph. A set of independent vertices in the conflict graph represents conflict free stream configurations. If the independent vertex set contains stream configurations for all the traffic in the original traffic planning problem, we have solved the original traffic planning problem, too.

This approach has several advantages. Due to the reduced coupling compared to the established constraint-based approaches, we do not need to build the complete conflict graph covering the whole solution space right from the beginning. Instead, we can start with a small conflict graph built from a small set of initial configurations and perpetually grow the graph until a complete solution is found.

This does not only have the potential to speed-up the solving process, but also allows to quickly obtain feasible solutions for a subset of the streams which have to be placed in the network, if the solving process is aborted prematurely. This feature becomes important in dynamic scenarios with ephemeral network topologies. For example, consider a sensing scenario where a distributed wired sensor grid generates real-time data streams, e.g., using acoustic sensors. These data streams are evaluated in real-time by mobile nodes, e.g., with the purpose of tracing [6] some event. The mobile nodes connect wirelessly via access points to the sensor grid. Here, a network topology is only valid until the mobile nodes are handed-over from one access point to another. If the traffic rate of the sensor data streams is high compared to the hand-over of the mobile nodes, but the hand-over is "fast" compared to the time it takes to compute a complete network configuration, quick partial solutions where not all streams are scheduled/routed are highly valuable.

The conflict graph-based approach additionally allows to incorporate "domain-specific" knowledge. For example, if all streams start at the same node, it seems intuitively preferable to start with a set of configurations where the transmissions of different streams are evenly distributed in time.

Summarized, our contributions in this paper are threefold.

- We show how to solve the traffic planning problem for time-triggered traffic by a conflict-graph based approach.
- We present a proof-of-concept implementation for the conflict-graph based approach.
- We numerically evaluate our implementation of the conflict-graph based approach and compare its performance to a constraint-based approach which uses integer linear programming.

In the remainder of this paper, we first discuss the related work in Sec. II. We introduce the traffic planning problem and how to solve it with the conflict-graph based approach in detail in Sec. III. We describe our proof of concept implementation of the conflict-graph based approach in Sec. IV and evaluate it in Sec. V, before we conclude the paper in Sec. VI.

II. Related Work

Traffic planning for time-triggered traffic is not a new problem and has been extensively researched. Similarly, conflict graphs have been employed in scheduling problems, albeit for the purpose of traffic planning it is predominantly applied to wireless scenarios, cf. [4], [7]–[9].

In this paper, we use conflict graphs for traffic planning of time-triggered traffic in IEEE TSN networks, where we can identify two major focal points of the research, constraint-based programming approaches and heuristics.

The constraint-based programming approaches either consider only the scheduling aspect [10], [11], or scheduling and routing [12]–[14]. In these approaches, a set of constraints is formulated using the respective framework (satisfiability modulo theories, array coding, integer linear programming) in the domain of the network problem, e.g., constraints directly restrict the transmission schedule or the paths of packets.

Since traffic planning for time-triggered traffic is NP-hard, [15], [16] propose heuristics. The heuristics share the characteristic that a prematurely chosen, sub-optimal configuration for any individual stream can later on obstruct the placement of other streams, and the heuristics operate directly in the domain of the traffic planning problem.

In contrast to previous work, our approach solves the original traffic planning problem by searching independent vertex sets in a conflict graph derived from the original traffic planning problem. This transformation makes the problem suitable for heuristics, as well as exact algorithms, and lends itself naturally to iterative approaches, since the conflict graph allows a more intuitive reasoning about the relation of individual (stream) configurations to the global solution for all streams.

III. Time-Triggered Traffic Planning

Next, we explain in Sec. III-A the system model of the data network and the traffic streams and what exactly we refer to as the traffic planning problem. Then, we show in Sec. III-B how to translate the original traffic planning problem to the conflict graph. Finally, we establish fundamental relations between the conflict graph and the original traffic planning problem in Sec. III-C.

A. The Original Problem: Routing and Scheduling of Time-triggered Traffic in a Data Network

The original problem, which we solve with the concept of conflict-graph based traffic planning, is the problem of traffic planning for time-triggered traffic in a data network. In this paper, solving this problem requires finding transmission schedules and routes for a given set of data streams in a given network topology. Next, we explain these terms in more detail.

1) Data Network: The system model of the network and the traffic is derived from networks with IEEE Std 802.1Q-compliant *switches* with TSN capabilities (called bridges in [17]) with full-duplex IEEE Std 802.3 Ethernet links [18]. That is, the network is a packet-switched communication network, and we consider only a fully-switched scenario, i.e., physical links (e.g., "Ethernet cables") are point-to-point

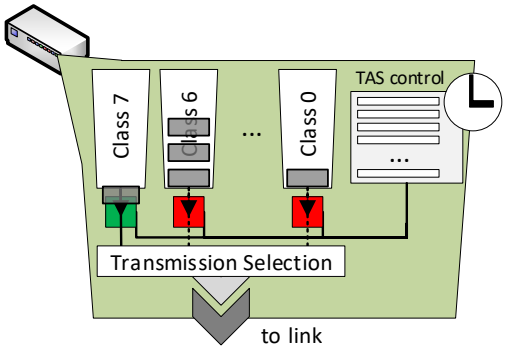

Fig. 1. Time-aware shaper for one port of an IEEE Std 802.1Q compliant switch. We use one class exclusively for time-triggered traffic.

connections. Communication between any pair of network elements is possible, if there exists sequence of links connected by switches between the source and destination network element, since switches are those network elements which can forward data packets. Packet forwarding can be controlled with the help of the switches' routing capabilities. We employ the routing capabilities to control the paths along which the packets of the individual streams are forwarded through the network. In IEEE Std 802.1Q-compliant networks this can be achieved, e.g., with VLAN tagging or explicit path control [19]. Switches can distinguish between different traffic classes and provide per-traffic class packet queuing with FIFO-semantics, cf. Fig. 1. This is a prerequisite for the switch's ability to control the exact point in time when the head-of-queue packets from each traffic class are eligible for transmission (cf. Fig. 1) using a cyclic, per-port switch-local time-schedule. The TAS from IEEE Std 802.1Q is a prominent example providing this functionality.

To control the forwarding process temporally across the network, a common time-base in the network is necessary. This can be achieved by clock synchronization algorithms, such as Precision Time Protocol (PTP).

Additionally, we assume, for the sake of simplicity, that all switches have the same processing delay and that the processing delay is constant. Likewise, links have the same propagation delay and data rate in the data network. Thus, without queuing delay we have the same per-hop delay for equal-sized messages.

2) Stream: The time-triggered traffic in the network consists of so-called *streams*. A single stream represents the directed information flow between two designated host nodes. For each stream, the so-called talker periodically generates packets and injects them into the network. We support different transmission periods for the streams. The network is supposed to forward the packets to the destination node which is called listener.

Table I summarizes the stream parameters. Stream parameters are externally given, e.g., derived from the requirements of a distributed application. The ingress network node and the egress network node are the source and destination of the packets of the stream, respectively. An ingress network node can be either the talker node, or the switch where the talker node is attached to, since the link between the talker

TABLE I
PARAMETERS FOR A SINGLE STREAM.

t_{cycle}	transmission period
t_{duration}	duration of the transmission of a single packet
t_{e2e}	end-to-end delay bound
ingress node	the first node in the data network for which a newly generated packet is considered in the traffic planning
egress node	the last node in the data network for which a packet of the stream is considered in the traffic planning

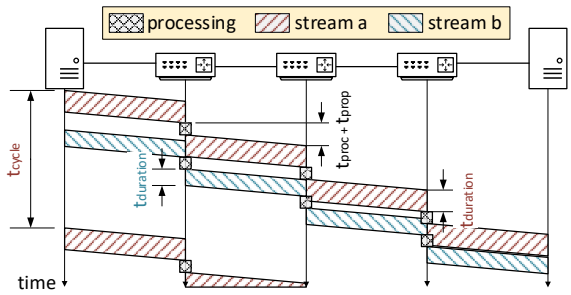

Fig. 2. Zero-queuing illustrated for two streams with different size and with the same talker (on the left) and same listener (on the right). We assume that processing delay is independent of the packet size.

node and the ingress switch of that stream is only a constant delay factor. The same holds for the egress. In case that the ingress network node and egress network node are switches, the end-to-end-delay bound has to be adjusted accordingly.

All streams share a single traffic class which is used exclusively for time-triggered traffic. This isolates time-triggered traffic, which is often of high-criticality, from other traffic types and keeps the number of "reserved" queues to a minimum.

With the model of the network and the streams, we can formally define a *stream configuration*.

Definition 1. *A configuration for stream s is the tuple of all scheduling variables and routing variables defined for stream s which fully define the talker behavior and the network behavior for stream s such that all constraints on the routes and schedules of the traffic planning problem are satisfied in the empty network.*

Our definition thus prohibits "invalid" variable assignments, e.g., paths which do not connect talker and listener.

3) Traffic Planning (Routing and Scheduling) for Time-Triggered Traffic with Zero-Queuing: In this section, we lay out the problem of traffic planning.

We restrict ourselves to "quasi-static" scenarios. This is a pragmatic simplification, since consistent network updates are challenging even without real-time constraints. How to update the configuration in the switches in a consistent way while maintaining required real-time guarantees and preventing other traffic, e.g., unscheduled traffic in a converged network scenario, to interfere with the time-triggered streams is even more difficult. In the following, we therefore consider the case that once route and transmission schedule are assigned to a stream, its configuration is not changed during its life-time.

The input for the traffic planning problem in this scenario consists of

- a network, with its topology and specification for switches (processing delay, transmission speed) and links (propagation delay),
- and a set $\mathcal{S}$ of streams with the parameters listed in Tab. I.

The traffic planning problem is to find a *global configuration* that *guarantees* that packets from each talker reach the respective listener within the required end-to-end delay. More specifically, we use the *zero-queuing* paradigm. With zero-queuing, the traffic planning problem becomes the problem of finding a global configuration for the streams which guarantees that packets of time-triggered streams traverse the network without ever being buffered on the way from talker to listener. Thus, the global configuration is the solution of the traffic planning problem and consists of stream configurations for *all* streams, and each stream configuration itself is comprised of a route and a schedule for the talker and switches. Next, we break down the properties of the problem and its solution:

- *Network constraint.* The network and its topology constrain the movement of a packet through the network. Packets can reach switches or hosts only via links. For each movement of a packet, a corresponding amount of time passes, e.g., the time necessary for processing, transmission and propagation of the respective packet. Combined with zero-queuing, the traversal of a packet is fully determined by the data network properties and the time it was injected into the network.
- *Temporal isolation constraint.* At any moment and every output port and link in the network, there is at most one packet in transmission, i.e., the transmission of packets is temporally exclusive.
- *Ordering constraint.* Due to FIFO semantics, packets which are transmitted over the same output port arrive in the order of transmission at the input port of the next hop, and packets are not reordered inside the switches.
- *Routing and delay constraint.* Packets of a specific stream enter the network at the talker network node and have to reach the listener network node where they leave the network. All packets of a stream are routed along the same path. Zero-queuing imposes a limit on the length of each stream's route via the end-to-end delay t_{e2e}.
- *Phase constraint.* The talker node of any stream completes the transmission of the first packet within its first cycle. This is a technical constraint to exclude solutions which "abuse" the mathematical concept of infinity.
- *Interarrival constraint.* The interarrival time of packets at the talker node of each stream is determined by t_{cycle}.

With zero-queuing, talker schedules and switch schedules are such that all packets enter an empty switch queue, are immediately selected for forwarding, and sent to the next hop (cf. Fig. 2). The talker schedule restricts when talkers emit packets, and switch schedules control when to forward packets of each traffic class. This eliminates queuing delay and binds the end-to-end delay of packets to the path length.

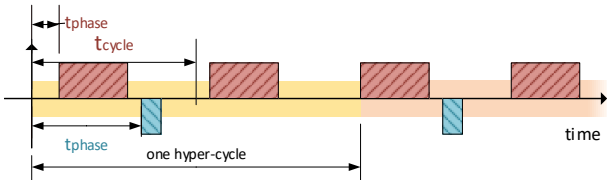

Fig. 3. Scheduling variable t_{phase} determines the transmission of packets along the network. Here, the transmission frequency of the upper stream (upper) is twice the transmission frequency of the lower stream, i.e., t_{cycle} of the lower stream is equal to the hyper-cycle.

We therefore define the *scheduling variable* t_{phase} (cf. Fig. 3). The phase t_{phase} of s is defined as the start of the transmission of the first packet of streams s after $t = 0$ at the talker. From t_{phase}, we can derive the network schedules for the TAS. For example, if transmission of first packet starts on outgoing link of talker at $t_0 = t_{\text{phase}}$, we have to allow transmission until $t_0 + t_{\text{duration}}$ when the transmission of this packet ends on this link via appropriate schedule entries. Similarly, we can derive the remaining schedule entries from the transmission intervals of the packets along its route. The transmission on the next link starts without queuing at $t_1 = t_0 + t_{\text{duration}} + t_{\text{prop}} + t_{\text{proc}}$, the transmission on the subsequent link starts at $t_1 + t_{\text{duration}} + t_{\text{prop}} + t_{\text{proc}}$, and so on, and the talker transmits the next packet at $t_0 + t_{\text{cycle}}$.

We analogously introduce the *routing variable* i_{path} to identify one of the *candidate paths* of stream s. Candidate paths are pre-computed paths from talker to listener, or, to be more exact, from the ingress node of s to the egress node of s. According to the *routing and delay* constraint and Def. 1, paths which are "too long" such that packets traversing them do not reach the egress network node within their stream's t_{e2e} must not be candidate paths.

B. Mapping the Original Problem to Cvertices and the Conflict Graph

Our first contribution is to show how we translate the original traffic planning problem into an independent vertex set problem in a conflict graph.

In the conflict-graph based traffic planning approach, we first construct a conflict graph from the original traffic planning problem. The nodes of the conflict graph are called configuration vertices (in short: cvertices). A single cvertex represents one possible configuration of a single *plannable*, i.e., routable and schedulable, entity (here: a stream). Edges between cvertices in the conflict graph encode a violation of the traffic planning constraints, cf. Fig. 4. The traffic plan can then be obtained from an independent vertex set in the conflict graph, which contains at least one cvertex for each plannable entity.

Next, we formally introduce the cvertices in the context of our concrete system model from Sec. III-A, and we explain how to build a conflict graph from these cvertices.

1) Cvertex (Configuration Vertex): A cvertex is the fundamental building block of the conflict graph.

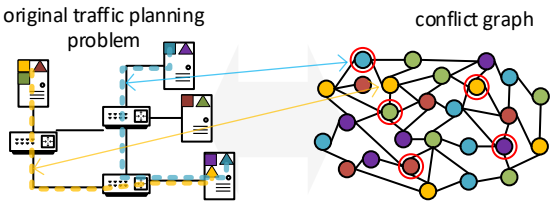

Fig. 4. Independent vertex set in conflict graph is a solution to the original traffic planning problem. Vertices in the conflict graph correspond to stream configurations (here: symbolized by routes).

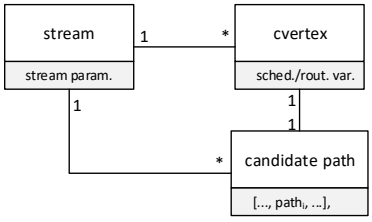

Fig. 5. Relation between stream and cvertices and candidate paths.

Definition 2. *A cvertex v_c is a vertex in the conflict graph which represents a single feasible configuration of a stream s.*

Since there can exist multiple cvertices for stream s, there exists a surjective mapping from cvertices to streams, cf. Fig. 5. In the following, we will say a cvertex v_c *belongs* to a stream s, iff v_c represents a configuration for s and similarly s *belongs* to v_c, iff v_c is a configuration for s. It is easy to see, that we can generate cvertices independently for each stream, just by taking into account the properties of the network and the respective stream.

2) Conflict and the Configuration Conflict Graph: Conflicts are the "glue" between cvertices and the original traffic planning problem.

Definition 3. *There exists a conflict between two cvertices v_c^1 and v_c^2, iff applying both configurations to the network and the talkers of v_c^1 and v_c^2 results in violating the constraints for the routes and schedules of the traffic planning problem.*

Algorithm 1 describes a method to check whether two cvertices are in conflict for our system model.

By definition, cvertices which belong to the same stream

Algorithm 1: Checking for conflict between two cvertices.

input : v_c^1, v_c^2
output : **true**, if v_c^1 and v_c^2 conflict, else **false**

1 **if** *v_c^1 and v_c^2 belong to the same stream s* **then return false**;
2 **else if** *candidate paths of v_c^1 and v_c^2 are link-disjunct* **then return false**;
3 **else if** *$\forall$ intersecting links of the candidate paths of v_c^1* ***and*** *v_c^2 the transmission of all packets of v_c^1 and v_c^2 is scheduled for mutually exclusive time intervals* **then return false**;
4 **else return true**;

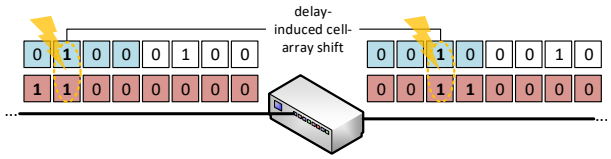

Fig. 6. Discrete time intervals are modeled via arrays, where each entry corresponds to a time-interval in the transmission period. Delay is modeled via circular shift of the array. If packets are scheduled such that the same time-interval is used by more than one stream, the temporal isolation constraint is violated.

cannot conflict (cf. line 1 in Alg. 1), since as per our original traffic planning problem, we apply only a single configuration for each stream. Due to output-port queuing and links being point-to-point connections between network elements, cvertices which are associated with link-disjunct paths (cf. line 2 in Alg. 1) are conflict-free, too, because packets are spatially isolated for contested network resources (output ports, links). At last, if there are links which occur in the candidate paths associated with both, v_c^1 and v_c^2, the two cvertices are conflict free, if packet transmissions are temporally isolated (cf. line 3 in Alg. 1). One possibility to check for temporal isolation is illustrated in Fig. 6 where time is discretized, and intervals where a link is occupied by a transmission are marked (here: by value 1). If t_{phase} for v_c^1 and v_c^2 is such that the same interval is marked by both streams, v_c^1 and v_c^2 conflict. Note that all of these conditions in Alg. 1 are computationally "cheap" to evaluate.

At last, we can define the conflict graph itself.

Definition 4. *A configuration conflict graph $\mathbf{CG}(\mathcal{C}, \mathcal{E})$ is an undirected graph where $\mathcal{C}$ is a set of cvertices, and $\mathcal{E} \subseteq \mathcal{C} \times \mathcal{C}$ is set of edges between cvertices in $\mathcal{C}$. Every edge $(v_c^1, v_c^2) \in \mathcal{E}$ represents a mutual conflict between the configuration associated with v_c^1 and the configuration associated with v_c^2.*

In this paper, we use configuration conflict graph and conflict graph as synonyms. However, the conflict graph is not to be confused with the graph representing the network topology of the original problem. $\mathbf{CG}$ is undirected, and an edge between two cvertices indicates a *mutual conflict* between the two configurations. In other words, two cvertices in $\mathbf{CG}$ are connected with an edge, if one or multiple of the constraints of the original traffic planning problem are violated if the solution includes both of these connected cvertices.

Note that Alg. 1 can also be used to construct a conflict graph from scratch from a set of cvertices $\mathcal{C}$ by adding an edge for any "conflicting" pair of cvertices $(v_c^1, v_c^2) \in \mathcal{C} \times \mathcal{C}$.

C. Relation between the Original Traffic Planning Problem and the Independent Cvertex Set in the Conflict Graph

From the previous section we know that a cvertex represents a configuration of a stream, i.e., defines when packets of that stream are traversing along which path through the network. The conflict graph $\mathbf{CG}$ is a graph where the vertices are cvertices, and there is an edge between v_c^1 and v_c^2 iff there is a

conflict between v_c^1 and v_c^2. Next, we establish the fundamental relations between the cvertices and the conflict graph on the one hand, and the streams and the data network of the original traffic problem on the other hand.

Theorem 1. *Let* $\mathbf{CG}(\mathcal{C}, \mathcal{E})$ *be a conflict graph for the original traffic planning problem for a set of streams* $\mathcal{S}$ *in a network. If* $\mathcal{C}_{\mathsf{ind}} \subseteq \mathcal{C}$ *is a set of independent cvertices in* $\mathbf{CG}$*, then* $\mathcal{C}_{\mathsf{ind}}$ *yields a conflict-free traffic configuration for* $\mathcal{S}_{feasib} \subseteq \mathcal{S}$*, iff* $\forall s \in \mathcal{S}_{feasib} : \exists v_c \in \mathcal{C}_{\mathsf{ind}} : (v_c$ *belongs to* $s)$.

Proof. (By construction.) If $\mathcal{C}_{\mathsf{ind}} \neq \emptyset$, we define $\mathcal{S}_{\text{feasib}} = \{s \in \mathcal{S} | \exists v_c \in \mathcal{C}_{\mathsf{ind}} : (v_c \text{ belongs to } s)\}$ and $\mathcal{C}_{\text{feasib}} \subseteq \mathcal{C}_{\mathsf{ind}}$: $\forall s \in \mathcal{S}_{\text{feasib}} : \text{card}\left(\{v_c \in \mathcal{C}_{\text{feasib}} | (v_c \text{ belongs to } s)\}\right) = 1$, i.e., $\mathcal{S}_{\text{feasib}}$ is a subset of $\mathcal{S}$ which contains all streams covered by $\mathcal{C}_{\mathsf{ind}}$, and $\mathcal{C}_{\text{feasib}}$ is a subset of $\mathcal{C}_{\mathsf{ind}}$ which contains exactly one cvertex for every stream $s \in \mathcal{S}_{\text{feasib}}$. Since $\mathcal{C}_{\mathsf{ind}}$ is an independent vertex set in $\mathbf{CG}$, $\mathcal{C}_{\text{feasib}}$ is an independent vertex set in $\mathbf{CG}$, too. From the definition of the independent vertex set property and the construction of $\mathbf{CG}$ follows directly that all $v_c \in \mathcal{C}_{\text{feasib}}$ are mutually conflict free. Thus, by configuring every stream $s \in \mathcal{S}_{\text{feasib}}$ according to the configuration associated with one corresponding $v_c \in \mathcal{C}_{\text{feasib}}$, it is guaranteed that none of the constraints on the routes, and schedules of $\mathcal{S}_{\text{feasib}}$ are violated. If $\mathcal{C}_{\mathsf{ind}} = \emptyset$, all traffic planning constraints are trivially satisfied since $\mathcal{S}_{\text{feasib}} = \emptyset$. □

Remark: An independent cvertex set $\mathcal{C}_{\mathsf{ind}}$ *covers* the stream set $\mathcal{S}_{\text{covered}}$, if $\forall s \in \mathcal{S}_{\text{covered}} : \exists v_c \in \mathcal{C}_{\mathsf{ind}} : (v_c$ belongs to $s)$.

Theorem 1 is the basis for heuristics which provide partial solutions to the original traffic planning problem. A partial solution to the traffic planning problem is a global configuration for a subset of streams $\in \mathcal{S}$. According to Theorem 1 any (non-empty) independent cvertex set provides us with valid configurations (schedules and routes) for at least those streams which it covers. This is an advantage over traffic planning in the original stream/data network domain which often searches for one variable assignment that satisfies all constraints at once, if the high-coupling results in an "all-or-nothing" solving process. The transformation of the original traffic planning problem to the conflict graph problem reduces this coupling, because we can decide whether two configurations are mutually exclusive just by evaluating the stream parameters and the configuration parameters of the two cvertices in question. Hence, this decision does not require knowledge about the configurations of any other streams or the global configuration.

The special case, where $\mathcal{C}_{\mathsf{ind}}$ covers $\mathcal{S}$ tells us how to solve the original traffic planning problem.

Theorem 2. *Let* $\mathbf{CG}(\mathcal{C}, \mathcal{E})$ *be a conflict graph for the original traffic planning problem for a set of streams* $\mathcal{S}$ *in a network. A set of independent cvertices* $\mathcal{C}_{\mathsf{sol}} \subseteq \mathcal{C}$ *in* $\mathbf{CG}$ *solves the original traffic planning problem, if there exists at least one* $v_c \in \mathcal{C}_{\mathsf{sol}}$ *for every stream* $s \in \mathcal{S}$.

Proof. For $\mathcal{S} = \emptyset$, $\mathcal{C} = \mathcal{C}_{\mathsf{sol}} = \emptyset$. If $\mathcal{S} \neq \emptyset$, then $\forall s \in \mathcal{S}$: $\exists v_c \in \mathcal{C}_{\mathsf{sol}} : (v_c$ belongs to $s)$, and all $v_c \in \mathcal{C}_{\mathsf{sol}}$ are mutually conflict free, because $\mathcal{C}_{\mathsf{sol}}$ is an independent cvertex set in $\mathbf{CG}$, i.e., $\forall (v_c^1, v_c^2) \in \mathcal{C}_{\mathsf{sol}} \times \mathcal{C}_{\mathsf{sol}} : \nexists (v_c^1, v_c^2) \in \mathcal{E}$. Thus, by selecting one $v_c \in \mathcal{C}_{\mathsf{sol}}$ for every stream $s \in S$ and applying its configuration to the network and the talkers, it is guaranteed that none of the constraints on the routes and schedules of $\mathcal{S}$ are violated. □

This means, we can solve the original traffic planning problem by searching for a set of independent cvertices which covers $\mathcal{S}$. We call the problem of finding $\mathcal{C}_{\mathsf{sol}}$ the *stream-aware independent cvertex set*-problem. The final independent cvertex set $\mathcal{C}_{\mathsf{sol}}$ returned by the algorithm may contain multiple cvertices for a stream. For example, we have $\mathcal{C}_A \subset \mathcal{C}_{\mathsf{sol}}$ for stream A. Then, only a single cvertex from $\mathcal{C}_A$ is selected and included in the global configuration which is finally applied to the network and talkers, since we need *one* route and *one* phase (schedule) for stream A. Here, any arbitrary choice from the cvertices in $\mathcal{C}_A$ is valid. If $\mathcal{C}_{\mathsf{sol}}$ contains multiple configurations for multiple streams, one might also pick particularly "suitable" configurations for the application at hand from $\mathcal{C}_{\mathsf{sol}}$, e.g., to optimize for best-effort traffic in converged networks.

The concept of conflict-graph based traffic planning is generic in the sense that it can be applied to traffic planning problems beyond our specific traffic planning problem, if they satisfy the following two conditions. Firstly, configurations of individual plannable entities need to be "additive" in the sense that it is possible to apply multiple configurations to individual network elements (here: switches) without "destroying" the previously applied configurations, and secondly it has to be possible to independently detect conflicts between any pair of configurations for the plannable entities. In our case, configurations are additive, for both routes—"addition" means adding a routing entry—and schedules. For the schedules, the additivity results from the temporal isolation constraint which allows us to merge the individual schedules on each switch.

IV. Exemplary Conflict-Graph Based Algorithm

In this section, we back-up our arguments for the conflict-graph based approach by presenting an implementation of a conflict-graph based algorithm that solves the original traffic planning problem. Fig. 7 gives an overview over this proof-of-concept conflict-graph based traffic planning (CGTP) algorithm.

Our CGTP algorithm is iterative. In each iteration, we first grow the conflict graph. Our method to grow the conflict graph is discussed in Sec. IV-A.

After growing the conflict graph, we use a combination of different algorithms to try to find the independent cvertex set $\mathcal{C}_{\mathsf{sol}}$ in the *current* conflict graph. By default, we use a quick algorithm (here: a modified maximal independent vertex set algorithm) in every iteration. This default algorithm (cf. Sec. IV-B) is computationally cheap, but its speed comes at the cost of giving no guarantee regarding the number of streams covered. Therefore, the execution of a second, slower algorithm (here: intermediate ILP) to try to compute $\mathcal{C}_{\mathsf{sol}}$ can be triggered. The second algorithm is computationally much more expensive, but it will surely find an independent cvertex set in the current

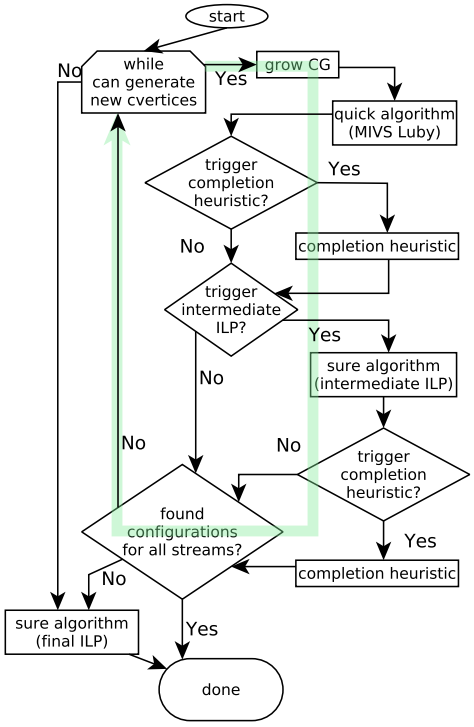

Fig. 7. Overview of the iterative, conflict-graph based traffic planning (CGTP) algorithm. The green, opaque arrow indicates one iteration.

graph which covers the maximum amount of streams given enough time. Hence, we call it the "sure" algorithm. However, intermediate executions of the sure algorithm have a runtime limit. We discuss this second algorithm and its trigger condition in Sec. IV-C.

Both, the quick algorithm and the sure algorithm can be followed up by the *completion heuristic*, cf. Sec. IV-D. When the completion heuristic is triggered, it tries to complete an almost complete independent cvertex set by generating new cvertices for those streams which have not been covered by the preceding algorithms.

All trigger rules make use of a history $\mathbf{h}_{\mathsf{found}}$ which is a list tracking the number of streams covered in the independent cvertex set in each iteration. While iterating, we also remember the previously computed maximal independent cvertex set which covers the most streams. This set is used to initialize the ILP and can be returned as partial solution if the CGTP algorithm is aborted.

We expect "simpler" problem instances in many applications (e.g., converged networks) where time-triggered traffic makes up only fraction of the network load, compared to scenarios where a diverse set of time-triggered streams shall saturate the network. However, since the underlying problem is known to be inherently NP-hard, we can end up with "difficult" instances ultimately requiring something alike an exhaustive search. Thus, if we did not find $\mathcal{C}_{\mathsf{sol}}$ in the "normal" iterations, even if the conflict graph contains all cvertices, i.e., encodes the whole configuration space, we execute the sure algorithm (here: ILP) one final time, with a relaxed runtime limit compared to the intermediate executions of the sure algorithm. Note that optionally, the final run of the sure algorithm can be executed without runtime limit. Without runtime limit, the sure algorithm is guaranteed to provide a solution covering the maximum amount of streams, barring practical problems such as running out of memory. This means, we can modify the parameters of the algorithm to guarantee that the CGTP algorithm yields an exact solution, i.e., a global configuration for as many as streams as possible for a particular problem instance.

After this overview, we present the details of the components of the CGTP algorithm. We start by detailing how the conflict graph is built. Next, we explain the two algorithms to find independent cvertex sets and the completion heuristic.

A. Building and Expanding the Conflict Graph

For the initial generation and the subsequent growth of the conflict graph in the CGTP algorithm, we define one *stateful generator function* per stream. Each generator function returns a new cvertex for its stream every time it is called as long as there exists a new cvertex for the respective stream. The state tracks for which parts of the feasible configuration space of the associated stream the generator function already created cvertices. The state is updated when the generator function generates a new cvertex. For our prototypical generator functions, the state comprises most importantly the path index and the phase of the last generated cvertex. For each invocation, a new cvertex is returned with an incremented i_{path} until all candidate path indices for the current t_{phase} are covered. If all candidates paths for the current t_{phase} have been covered, t_{phase} is incremented and the candidate path index is reset to $i_{\mathsf{path}} = 0$. This way, the generator functions sweep over the range of feasible values for their respective stream. With a discrete time representation, every generator returns a finite amount of cvertices.

In the CGTP algorithm, we start with an initial conflict graph that contains a single cvertex for each stream in the stream set, i.e., we invoke every per-stream generator function once, and construct the conflict graph from those cvertices by adding the corresponding edges.

During the iterations of the CGTP algorithm, we grow the conflict graph on-the-fly. This exploits the property, that it is possible to obtain a solution to the original traffic planning problem without a conflict graph including all cvertices for all streams. We grow the conflict graph in the beginning of each iteration by invoking the cvertex generator functions for the stream set and inserting the obtained vertices into the existing conflict graph (cf. Alg. 2).

B. Quick Algorithm: Adapted Maximal Independent Vertex Set Computation

According to Theorem 2, any independent cvertex set which covers all streams can be the solution $\mathcal{C}_{\mathsf{sol}}$ to the stream-aware independent cvertex set problem. Usually, $|\mathcal{C}| \gg |\mathcal{S}|$, thus there is a chance for any independent cvertex set $\mathcal{C}_{\mathsf{ind}}$ with $|\mathcal{C}_{\mathsf{ind}}| \geq |\mathcal{S}|$ to cover $\mathcal{S}$. In other words, any algorithm which

Algorithm 2: Cvertex insertion into conflict graph.

input : $\mathbf{CG}$, v_c
output : $\mathbf{CG}$

foreach $v_c^e \in \text{vertices}(\mathbf{CG})$ **do**
  if $\text{conflict}(v_c, v_c^e)$ **then** add edge (v_c, v_c^e) to $\mathbf{CG}$;
end
add vertex v_c to $\mathbf{CG}$;

Algorithm 3: Adapted Luby's algorithm for MIVS [20].

input : $\mathbf{CG}$, a
output : independent cvertex set $\mathcal{I}$

1 $\mathcal{I} \leftarrow \emptyset$; $\mathbf{CG}'(\mathcal{C}', \mathcal{E}') \leftarrow \text{copy}(\mathbf{CG}(\mathcal{C}, \mathcal{E}))$;
2 **while** $\mathcal{C}' \neq \emptyset$ **do**
3 $\mathcal{X} \leftarrow \emptyset$; // candidate set
4 **foreach** $v_c \in \mathcal{C}'$ **do**
5 $p_{\text{deg}} \leftarrow \frac{1}{(2 \cdot \deg(v_c))}$; $p_{\text{sc}} \leftarrow 1 - \frac{\mathbf{sc}[s]}{\max(\mathbf{sc})}$;
6 add to $\mathcal{X}$ with probability $p = a \cdot p_{\text{deg}} + (1-a) \cdot p_{\text{sc}}$;
7 **end**
8 $\mathcal{I}' \leftarrow \mathcal{X}$;
9 **foreach** $(v_c^1, v_c^2) \in \mathcal{I}' \times \mathcal{I}' : (v_c^1, v_c^2) \in \mathcal{E}'$ **do**
10 **if** $\deg(v_c^1) \leq \deg(v_c^2)$ **then** $\mathcal{I}' \leftarrow \mathcal{I}' - \{v_c^1\}$;
11 **else** $\mathcal{I}' \leftarrow \mathcal{I}' - \{v_c^2\}$;
12 update($\mathbf{sc}$) ;
13 **end**
14 $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{I}'$; $\mathcal{Y} \leftarrow \mathcal{I}' \cup \text{neighborhood}(\mathcal{I}')$;
15 $\mathbf{CG}'(\mathcal{C}', \mathcal{E}') \leftarrow$ subgraph on $\mathcal{C}' - \mathcal{Y}$;
16 **end**

returns a sufficiently large independent cvertex set may possibly solve the traffic planning problem.

Therefore, we adapt Luby's algorithm [20] for the computation of the maximal independent vertex set (MIVS). Luby's algorithm is intuitive to understand and can be parallelized.

Luby's algorithm is an iterative algorithm (cf. Alg. 3), which terminates with high probability in $\mathcal{O}(\log |\mathcal{C}|)$ rounds.

Each iteration of the algorithm consists of two steps. In the first step (cf. l. 4, Alg. 3), the algorithm randomly selects candidate cvertices. In our adapted algorithm (changes are highlighted in Alg. 3), the probability of cvertex v_c to become a candidate vertex depends on the degree of cvertex, and we consider how many cvertices that belong to the same stream as v_c are already part of the result set $\mathcal{I}$. For this, we use variable $\mathbf{sc}[s]$, which tracks the number of cvertices that belong to s and which were previously included in $\mathcal{I}$. The final probability accounts for both terms, weighted by a factor a with $0 \leq a \leq 1$ (by default $a = 0.7$) and thus can raise the probability of cvertices not yet covered in the result set $\mathcal{I}$ to become candidates. In step 2 (cf. l. 9, Alg. 3), cvertices from the candidate set are selected as cvertices for the maximal independent vertex set. The steps repeat until a maximal independent vertex set is found, i.e., all vertices are either in the result set $\mathcal{I}$, or neighbor vertices in the result set.

Unfortunately, the adapted MIVS algorithm returns "only" a maximal independent cvertex set. There is no guarantee that the maximal independent cvertex set will ever cover all streams.

C. Sure Algorithm: Finding Independent Cvertex Sets covering as many Streams as possible

Since the quick algorithm may not solve the problem completely, we employ an ILP as an alternative method. The ILP is guaranteed to find an independent cvertex set that covers as many streams as possible, albeit it may take very long.

Therefore, this sure algorithm is only executed in two cases, a) if it appears that the quick algorithm does not make any progress over multiple iterations, i.e., the number of covered streams is not increasing, or b) when the conflict graph contains all possible cvertices and still no solution has been found.

Next, we explain the ILP formulation that finds the independent cvertex set that covers as many streams as possible. Then we explain the trigger rule for the sure algorithm.

1) Integer-Linear Program for the Stream-Aware Independent Cvertex Set Problem: The ILP to compute the independent cvertex set that covers as many streams as possible is different from the ILP-based approaches for the original traffic planning problem in the related work which operate directly on the routing variables and the scheduling variables. It is much simpler and can be stated in the three lines.

$$\max \sum_{x_s \in \mathcal{X}_\text{S}} x_s \text{ subject to} \tag{1}$$

$$\forall (x_v^1, x_v^2) \in \text{edges}(\mathbf{CG}) : x_v^1 + x_v^2 \leq 1 \tag{2}$$

$$\forall x_s \in \mathcal{X}_\text{S} : \sum_{x_v \in \mathcal{X}_V^s} x_v \geq x_s \tag{3}$$

The ILP uses two sets of binary decision variables. Every decision variable $x_v \in \{0, 1\}$ in the decision variable set $\mathcal{X}_\text{cv}$ represents one cvertex in the conflict graph. If $x_v = 1$, then the associated cvertex is part of the independent cvertex set. Conversely, if $x_v = 0$, then the associated cvertex is not part of the independent cvertex set. Similarly, every decision variable $x_s \in \{0, 1\}$ in the second decision variable set $\mathcal{X}_\text{S}$ represents a stream $s \in \mathcal{S}$. If $x_s = 1$, the respective stream is covered by the independent cvertex set. Consequently, if $x_s = 0$ the associated streams s is not covered by the independent cvertex set. Additionally, we define the helper set $\mathcal{X}_V^s$, which includes all $x_v \in \mathcal{X}_\text{cv}$ which belong to a specific stream s. The set $\text{edges}(\mathbf{CG})$ contains all edges in the conflict graph $\mathbf{CG}$.

The objective function returns an independent cvertex set which covers as many streams as possible. If there is an independent cvertex set that covers all streams, the ILP will find it provided with enough computing resources. Note that the NP-hard maximum independent vertex set problem [21] is reduced to this ILP, if there is one cvertex per stream.

2) Intermediate ILP Execution: Even though the ILP is "just three lines" long, solving the ILP may take an unknown, large amount of time, which may be spent better by growing the graph and executing the quick algorithm. Therefore, we do not execute it in every iteration. However, it may happen, that the quick algorithm fails over the course of multiple iterations to return independent cvertex sets which cover an increasing number of streams despite a growing conflict graph. This can be caused e.g., by conflict graphs with an "unfortunate"

Algorithm 4: Trigger rule for sure algorithm.

input : history $\mathbf{h}_{\text{found}}$, window size w_{ILP},
output : trigger variable r_{ILP}, window size w_{ILP}

$\mathbf{window} \leftarrow$ slice of w_{ILP} last entries of $\mathbf{h}_{\text{found}}$;
$d_{\text{past}} = \sum \text{diff}(\mathbf{window})$;
if $d_{past} > 0$ **then**
  $r_{\text{ILP}} \leftarrow$ **false** ; window size $w_{\text{ILP}} \leftarrow w_{\text{ILP}} + 1$;
else $r_{\text{ILP}} \leftarrow$ **true** ; $w_{\text{ILP}} \leftarrow$ minimal window size ;
return r_{ILP}, w_{ILP};

cvertex degree distribution where the quick algorithm yields independent sets with many cvertices which all belong to a small set of few streams. If the number of covered streams found by the quick algorithm stagnates, or even decreases over time despite a growing conflict graph, then we want to execute the sure algorithm.

This rationale is encoded in Alg. 4. Alg. 4 evaluates the development of the number of covered streams in a (variable-sized) window of the history. Alg. 4 returns a Boolean r_{ILP} indicating whether the sure algorithm shall be executed and updates the windows size. The window grows, when the quick algorithm "makes progress", because we do not want a single "bad" iteration to trigger the sure algorithm. If the sure algorithm is triggered, we reset the window size, such that the trigger condition is again more sensitive to changes.

If the ILP is triggered, it is allowed to run only for a limited time (by default $5\,\text{min}$). Additionally, there is a limiter which suspends the sure algorithm for a certain number of iterations if it has been triggered successively for a certain number of iterations in the past. The motivation for limiting the number of successive ILP execution, i.e., suspending the ILP executions for some iterations, stems from the fact that there are two reasons for the ILP execution to not find an independent cvertex set in the current conflict graph, a) either finding the solution in the current conflict graph is so hard that we hit the time-out, or b) the current conflict graph does not contain an independent cvertex set which covers all streams (yet). In the latter case, the ILP execution wastes $5\,\text{min}$. However, we do not know which is the case. Thus expanding the graph (instead of executing the ILP) improves the chance that the graph contains a solution and that it can be found be the ILP in overall shorter time. By default, we suspend the sure algorithm for five iterations, if it has been triggered twice in succession.

D. Completion Heuristic

The *completion heuristic* can be triggered after both, the quick algorithm and the sure algorithm. It is triggered, if the number of missing streams, i.e., streams which are not covered in the independent cvertex set, in the current iteration is less or equal to a threshold. The completion heuristic invokes the generator functions to get a limited set of new cvertices for the missing streams and adds them to the current conflict graph, in the hope of finding cvertices without a conflict with the independent cvertex set found in the current iteration.

Algorithm 5: Update threshold for completion heuristic.

input : history $\mathbf{h}_{\text{found}}$, window size $w_{\text{cplt.}}$, threshold $p_{\text{thresh.}}$
output : windows size $w_{\text{cplt.}}$, threshold $p_{\text{thresh.}}$

1 $\mathbf{window}^{\text{old}} \leftarrow$ slice of $w_{\text{cplt.}}$ entries up to (including) penultimate entry of $\mathbf{h}_{\text{found}}$;
2 $\mathbf{window}^{\text{new}} \leftarrow$ slice of $w_{\text{cplt.}}$ last entries of $\mathbf{h}_{\text{found}}$;
3 $\bar{n}^{\text{old}}_{\text{found}} \leftarrow \text{round}(\text{mean}(\mathbf{window}^{\text{old}}))$;
4 $\bar{n}^{\text{new}}_{\text{found}} \leftarrow \text{round}(\text{mean}(\mathbf{window}^{\text{new}}))$;
5 **if** $\bar{n}^{\text{new}}_{\text{found}} < \bar{n}^{\text{old}}_{\text{found}}$ **then**
6 reduce $p_{\text{thresh.}}$; reduce $w_{\text{cplt.}}$ linearly ;
7 **else if** $\bar{n}^{\text{new}}_{\text{found}} < \bar{n}^{\text{old}}_{\text{found}}$ **then**
8 increase $p_{\text{thresh.}}$; increase $w_{\text{cplt.}}$ by $(w_{\text{max,cplt.}} - w_{\text{cplt.}})/2$;
9 **else** reduce $p_{\text{thresh.}}$;
10 **return** $p_{\text{thresh.}}$, $w_{\text{cplt.}}$;

A fixed threshold may result in excessive execution of the completion heuristic, once it is exceeded, and is difficult to determine a-priori. Therefore, we use a dynamic threshold. The threshold is adjusted depending on the past iterations by Alg. 5. Alg. 5 looks at a variable-sized window of the history $\mathbf{h}_{\text{found}}$. If the number of covered stream is increasing, the threshold is raised, i.e., more streams have to be covered, before the completion heuristic is triggered, and the window size grows, i.e., a single worse iteration has less impact. If the number of covered stream is decreasing, we do the opposite. The threshold is lowered to the effect that the completion heuristic is triggered with more missing streams, and the window size is reduced, i.e., the threshold adjustment becomes more responsive. If the change is too small (cf. the rounding in line 3 and line 4 in Alg. 5) indicating a lack of progress, the threshold is lowered, too.

Considering the case that the completion heuristic is triggered, but fails to cover the remaining streams, we can interpret the completion heuristic and its trigger rule as a secondary, optional growth-phase of the conflict graph in an iteration. This second growth-phase is restricted to streams which were left uncovered in the preceding conflict-checking step, thus the completion heuristic also serves as a primitive feedback mechanism to "guide" the conflict-graph growth.

V. Evaluation

In this section, we quantitatively evaluate the conflict-graph based traffic planning approach.

A. Evaluation Scenarios

We obtain problem scenarios by first generating a network topology and then creating a set of streams. The network topology is a ring graph where the nearest n-neighbors are connected (cf. Fig. 8). Besides its simplicity and close relation to ring topologies which are often found in industrial environments or in sensor arrays for direction-of-arrival tracking [22], this topology ensures that there exist multiple different candidate paths for every pair of streams.

Unless specified differently, $n = 3$ neighboring nodes are connected, processing delay is set to $2\,\mu\text{s}$ (here: time is discretized in $1\,\mu\text{s}$-intervals). Propagation delay is neglected.

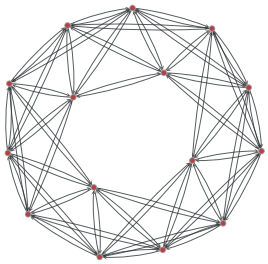

Fig. 8. Example of the network topology: ring graph with 15 nodes and $n = 3$ neighboring nodes (in each direction) are connected.

The stream parameters t_{cycle}, t_{duration}, and number of candidate paths are equal for all streams in an evaluation scenario. Talker and listener node for each stream are randomly selected among the nodes of the network. This keeps the number of parameters low, and reduces the probability of infeasible scenarios, e.g., caused by relative prime t_{cycle}. By default, we consider 3 candidate paths per flow. We write the network properties and the stream parameters to plain-text files which are ingested by the traffic planning programs.

B. Evaluation Methodology

We implemented the conflict-graph based traffic planning algorithm from Sec. IV in Julia [23] and refer to this implementation of the conflict-graph based traffic planning program as CGTP. CGTP uses the LightGraphs library for the computation of the k-shortest paths for each stream and interfaces with the ILP solver Gurobi [24] via JuMP [25].

Additionally, we also implemented an adapted version of a recently proposed, typical ILP [14] for the traffic planning problem to compare the conflict-graph based traffic planning approach to a constraint-based approach which operates directly in the domain of network and streams. We adapted the ILP to use the same system model of the traffic planning problem as CGTP, i.e., multiple candidate paths are considered, and we use a time-discretization to model temporal constraints. We refer to this constrained-based approach as *reference ILP* (short, RILP, not to be confused with the ILPs used for the sure algorithm in CGTP). To run the RILP, we use a Python program and the modeling library Pyomo [26] to build the RILP model for the traffic planning problem. Pyomo also interfaces with the ILP solver Gurobi, and we compute the k-shortest paths for each stream with the graph-tool library [27].

Tab. II summarizes the computing setup of our evaluations.

C. Evaluation Results

Next, we present the evaluation results, starting with a performance comparison of CGTP and RILP, before looking at different properties of CGTP. For all results in our evaluations the CGTP-algorithm yielded optimal results in the sense that it returned an independent cvertex set covering all streams.

1) Comparison CGTP vs. RILP: For the comparison of CGTP and RILP, we use a network with 50 nodes. We increase the number of streams in steps of 10, starting at 50 streams, with $t_{\text{cycle}} = 300\,\mu\text{s}$ and $t_{\text{duration}} = 5\,\mu\text{s}$ for all streams. We

TABLE II
SPECIFICATION OF COMPUTE NODES.

	PCsmall	PCbig
CPU	1 Intel Xeon E5-1650v3, 3.50 GHz	4 Intel Xeon E7-4850 v4, 2.1 GHz
RAM	16 GB	1 TB
host OS	CentOS Linux 7.7.1908, Kernel 3.10.0-1062,	Arch Linux, Kernel 5.2.5
container	docker v19.03, Fedora 30-based container image	
SW (CGTP)	Julia 1.2.0, LightGraphs, JuMP with Gurobi 8.1.0	
SW (RILP)	Python 3.7, graph_tool, Pyomo with Gurobi 8.1.0	

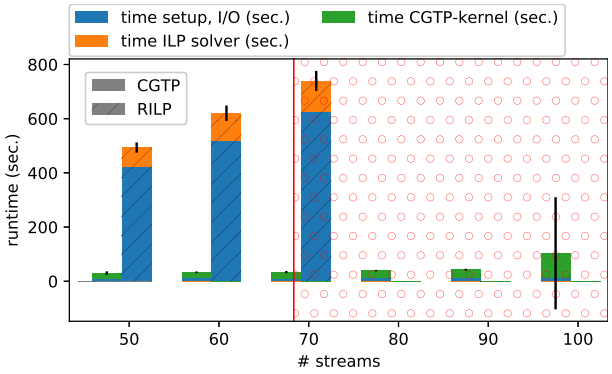

Fig. 9. Comparison of runtime on compute nodes of type PCsmall.

solve 20 problems per step, and each problem is solved once with CGTP and once with RILP on compute nodes of type PCsmall (cf. Tab. II).

The *wall clock* total runtime, i.e., the time from importing the problem from the files to writing the solution to disk for these scenarios is divided in the time for setup and I/O, and the time for the actual solving process. In case of CGTP, setup and I/O includes the time to read and write the files and to pre-compute the candidate paths for the streams. Time spent in the CGTP-kernel consists of the time for growing the conflict graph and searching for independent cvertex sets. For RILP, setup and I/O includes, besides file operations and path pre-computations, the time required for the construction of the ILP model. The time spent on solving the ILP is measured around the function call, which interfaces with the solver.

The average runtimes are depicted in Fig. 9. The vertical line on top of each composite bar indicates the standard deviation of the total runtime.

While it is apparent that CGTP solves the problem in much shorter time, e.g., the average total runtime of CGTP for 100 streams is with 103.2 s almost five times less compared to the average total runtime of RILP (493.4 s) for only 50 streams, we want to highlight another advantage of CGTP over constraint-based implementations. On the PCsmall compute nodes, the RILP software stack quickly hit the "memory bound". For already 70 streams, RILP could not solve 5 of 20 scenarios due to a shortage of memory and for 80 or more streams no scenario was solved as indicated by the missing bars in Fig. 9. Constraint-based approaches which solve the traffic planning

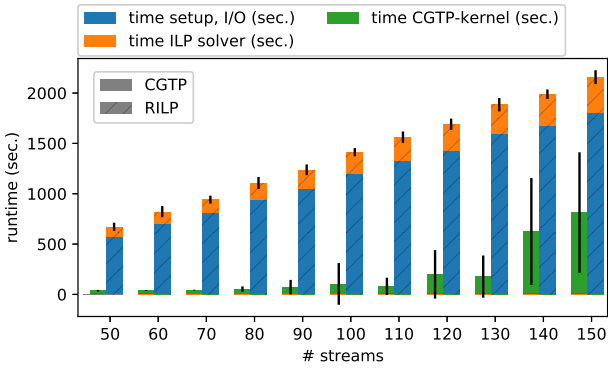

Fig. 10. Comparison of runtime on compute node PCbig.

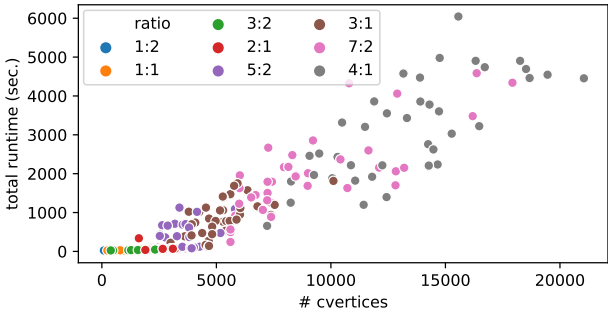

Fig. 11. Behavior for varying ratio of streams to nodes in the network.

problem in the network domain inherently result in a large set of constraints [1], and solvers often occupy a large chunk of memory for tracking the already covered solution space.

In contrast, CGTP runs to completion for the same evaluation scenarios on the PCsmall compute nodes. CGTP is more memory efficient during the solving process, because the cvertices have a high information-density compared to the individual variables in the RILP. To solve the stream-aware independent cvertex problem, we only need the cvertex identifiers, the edges in $\mathbf{CG}$ and a mapping of cvertices to stream identifiers, i.e., "a few" integers. All the details in the network domain are only necessary when constructing or growing the graph, or when finally assembling the solution. Thus, the conflict-graph based approach has the very practical advantage of pushing out the borders of tractable problems in scenarios with limited memory. Admittedly, out-of-core implementations can extend the memory bound for both approaches.

We re-ran the evaluations on a compute node of type PCbig (cf. Tab. II) with much more RAM. The results (averaged over 20 scenarios per step) are depicted in Fig. 10.

Up to 130 streams, the whole runtime of CGTP is on average faster than the time spend for just calling the ILP solver (excluding setup and I/O), even though the ILP solver is able to make use of all processor cores, whereas CGTP is implemented mostly single-threaded. While CGTP still outperforms RILP by a significant margin for all evaluation scenarios, we observe a steep increase in runtime and the variance of the total runtime for CGTP from 130 streams on. The average total runtime more than triples for CGTP from 130 streams (177.1 s) to 140 streams (625.0 s) and reaches 812.4 s for 150 streams.

In our evaluations, CGTP outperforms RILP, but we also observe that the region of scenarios which can be solved in practice with RILP is limited by the sheer size of the ILP model and the time spent on its construction. The iterative approach of CGTP does not exhibit this drawback.

2) Network Density: Next, we investigate the performance of CGTP and the previously observed increase in runtime and runtime variance for larger stream sets. We use the same network topology with 50 nodes and with the same properties as in the previous section. We vary the number of streams from 25 to 200 with an increment of 25 streams per step (each with $t_{\text{cycle}} = 1000\,\mu\text{s}$, $t_{\text{duration}} = 5\,\mu\text{s}$), effectively changing the total ratio of streams to nodes in the data network from 1:2 to 4:1. We generate and solve 40 problem instances per step on the PCsmall compute nodes.

In Fig. 11, we plot the total runtime over the number of cvertices of the conflict graph at the time when the solution was found. Each point in Fig. 11 represents an individual scenario. Points are colored according to the ratio of streams to nodes in the network. We observe comparably short mean runtimes for scenarios with 25 streams (12.0 s) to 100 streams (86.9 s) which form almost a line in the lower left corner of the plot. However, from 125 streams (ratio 5:2) on, individual runtimes increase strongly and spread farther apart, indicating also an increasing variance of the runtimes. From 125 streams (ratio 5:2) on, we measured the following average runtimes: 297.5 s for 125 streams, 851.1 s for 150 streams, 1982.3 s for 175 streams and 3134.7 s for 200 streams.

There are two factors contributing to this behavior. Firstly, by increasing the number of streams while maintaining the network size, the network load, and thus the difficulty of the traffic planning problem, is increasing, since conflicts become more likely. Secondly, we also see implementation-specific effects. As explained in Sec. IV, our proof-of-concept implementation of CGTP is an iterative approach which by default uses the adapted maximal independent vertex set algorithm in each iteration and, depending on the trigger condition, also executes an intermediate ILP. For scenarios with 100 or more streams, the intermediate ILP executions did not only get triggered multiple times, but we also observed several instance where the intermediate ILP solving had to be aborted due to hitting the runtime limit. The higher the number of streams, the more often this happened. Every intermediate ILP execution which was aborted due to the time-constraints therefore adds 300 s to the total runtime, i.e., for similar-sized conflict graphs the number of unsuccessful intermediate ILP executions can cause strongly varying total runtimes.

3) Scaling: Finally, we keep the ratio of streams to nodes in the network fixed to 1:1, i.e., the more streams, the larger the network. We vary the number of streams (and nodes in the network) from 50 to 400, (each with $t_{\text{cycle}} = 1000\,\mu\text{s}$, $t_{\text{duration}} = 5\,\mu\text{s}$). We generate and solve in total 80 problem instances per step on PCsmall compute nodes.

This time, we use two different settings for the ILP trigger

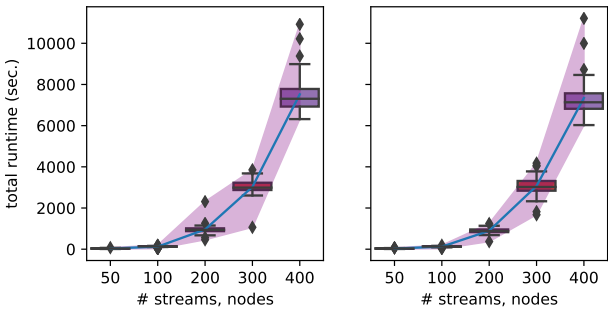

Fig. 12. Total runtime for increasing problem size with different intermediate ILP settings. Left plot: after two successive executions of intermediate ILP for at most 5 min, suspend intermediate ILP for 5 iterations, right plot: at most 1 execution of intermediate ILP every 10 iterations for at most 10 min.

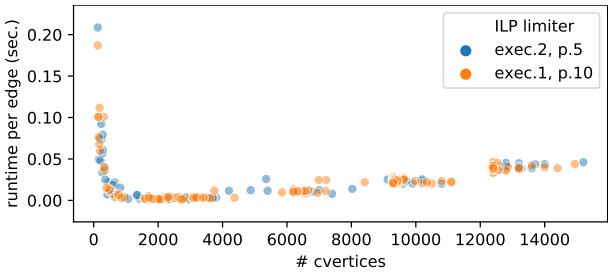

Fig. 13. Runtime normalized to the number of edges in the conflict graph for increasing problem sizes.

limiter. Half of the problem instances (left plot in Fig. 12) use the default setting where the intermediate ILP can be executed at most twice in succession, and the ILP solver runtime is limited to 5 min, before ILP triggering pauses for five iterations (labeled *exec.2, p.5*). For the other half of the problem instances (right plot in Fig. 12), the intermediate ILP can be triggered at most once every ten iterations and is allowed to run at most 10 min (labeled *exec.1, p.10*).

Despite the different settings for the ILP trigger limiter, we observe similar values for the average runtimes, as well as the variance of the runtime. The increase in runtime is dominated by two factors, namely the time it takes to grow the graph and the time it takes to search for the independent cvertex set. The growing of the conflict graph, or more exact, the time it takes to insert a single cvertex into the conflict graph grows with increasing number of cvertices in the conflict graph (cf. Alg. 2). The time required for searching the independent cvertex set also grows with an increasing conflict graph, but has the possibility to vary much stronger, depending on the actual structure of the conflict graph, the rounds in the quick algorithm, and the time spent in the sure algorithm.

In Fig. 13, we plot the total runtime normalized to the number of edges in $\mathbf{CG}$ over the size of $\mathbf{CG}$. Also, here the two settings for the intermediate ILP behave similarly. For small numbers of cvertices ($\sim 10^2$ to 10^3 cvertices), the overhead (I/O, path computation) dominates, hence we see a decline for increasing number of cvertices for these problem instances. For larger conflict graphs, the time spent for growing the conflict graph and finding the solution dominates, and we

TABLE III
MEAN AND STANDARD DEVIATION OF CONFLICT GRAPH SIZES FOR INCREASING PROBLEM SIZES, CF. FIG. 13.

$\lvert\mathcal{S}\rvert$,# nodes	50	100	200	300	400
mean($\lvert\mathcal{C}\rvert$)	1.297E3	3.043E3	6.256E3	9.438E3	1.267E4
std($\lvert\mathcal{C}\rvert$)	824	616	464	762	584
mean($\lvert\mathcal{E}\rvert$)	1.445E4	3.633E4	8.368E4	1.333E5	1.838E5
std($\lvert\mathcal{E}\rvert$)	1.338E4	1.038E4	1.187E4	2.293E4	2.000E4

observe a similar clustering (visible, e.g., around the average number of cvertices for 200 and more stream, cf. Tab III) in Fig. 11. The average size of the conflict graph is similar for both settings (cf. Tab. III), and there are few outliers in Fig. 13. This indicates that the combination of the quick algorithm, the sure algorithm, and the respective trigger rules succeed to consistently finding solutions without "over-growing the graph".

VI. CONCLUSION

In this paper, we presented an approach to solve the traffic planning problem for time-triggered traffic in data networks with conflict graphs. We explained how to derive these conflict graphs from the original traffic planning problems, and we showed how to solve the traffic planning problem by finding independent cvertex sets which cover all streams. With the conflict-graph based approach it is comparably cheap to get feasible (partial) solutions, since there exist efficient algorithms for finding independent sets even in large graphs.

Furthermore, we presented a proof-of-concept implementation of a traffic planning algorithm for the conflict-graph based approach and evaluated its performance. The performance evaluations showed practical advantages of the conflict-graph based approach, which can find solutions of the traffic planning problem faster and more memory efficiently.

The concept of conflict-graph based traffic planning problems can be advanced in several directions in future work. The most obvious one are performance improvements, e.g., by increasing the parallelization of the current implementation during construction of the conflict graph or when searching for independent vertex sets, or tuning trigger rules. Also, it seems promising to replace Luby's algorithm entirely by a bespoke algorithm which searches specifically for independent cvertex sets covering all streams, e.g., using techniques from [28]. Exploring, if the conflict-graph based approach for traffic planning is also suitable for different system models, e.g., without zero-queuing assumption, is another direction of future work.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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*, Hakodate, Japan, Aug. 2018, pp. 136–146.

[2] N. E. H. Tellache and M. Boudhar, "The two-machine flow shop problem with conflict graphs," *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 1026–1031, Jan. 2016.

[3] A. D'Ariano, M. Pranzo, and I. A. Hansen, "Conflict Resolution and Train Speed Coordination for Solving Real-Time Timetable Perturbations," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 2, pp. 208–222, Jun. 2007.

[4] P. Djukic and S. Valaee, "Link Scheduling for Minimum Delay in Spatial Re-Use TDMA," in *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, May 2007, pp. 28–36.

[5] C.-C. Hsu, K.-F. Lai, C.-F. Chou, and K. C.-J. Lin, "ST-MAC: Spatial-Temporal MAC Scheduling for Underwater Sensor Networks," in *IEEE INFOCOM 2009*, Apr. 2009, pp. 1827–1835.

[6] V. Cevher and J. McClellan, "Tracking of multiple wideband targets using passive sensor arrays and particle filters," in *Proceedings of 2002 IEEE 10th Digital Signal Processing Workshop, 2002 and the 2nd Signal Processing Education Workshop.*, Oct. 2002, pp. 72–77.

[7] H. Li, Y. Cheng, C. Zhou, and P. Wan, "Multi-dimensional Conflict Graph Based Computing for Optimal Capacity in MR-MC Wireless Networks," in *2010 IEEE 30th International Conference on Distributed Computing Systems*, Jun. 2010, pp. 774–783.

[8] L. Jiang, D. Shah, J. Shin, and J. Walrand, "Distributed Random Access Algorithm: Scheduling and Congestion Control," *IEEE Transactions on Information Theory*, vol. 56, no. 12, pp. 6182–6207, Dec. 2010.

[9] O. Kondrateva, H. Döbler, H. Sparka, A. Freimann, B. Scheuermann, and K. Schilling, "Throughput-optimal joint routing and scheduling for low-earth-orbit satellite networks," in *2018 14th Annual Conference on Wireless On-Demand Network Systems and Services (WONS)*, Feb. 2018, pp. 59–66.

[10] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16, 2016, pp. 183–192.

[11] 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)*, Apr. 2018, pp. 13–24.

[12] 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. Brest, France: ACM, 2016, pp. 193–202.

[13] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, 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.

[14] Jonathan Falk, Frank Dürr, Steffen Linsenmayer, Stefan Wildhagen, Ben Carabelli, and Kurt Rothermel, "Optimal Routing and Scheduling of Complemental Flows in Converged Networks," in *Proceeding of the 27th International Conference on Real-Time Networks and Systems*, ser. RTNS'19, Toulouse, France, Nov. 2019.

[15] 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, 2016, pp. 203–212.

[16] 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, May 2018.

[17] 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, Jul. 2018.

[18] ——, "IEEE Standard for Ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, Aug. 2018.

[19] 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," *ISO/IEC/IEEE 8802-1Q:2016/Amd.1:2017(E)*, pp. 1–122, Jul. 2017.

[20] M. Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: ACM, 1985, pp. 1–10.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability*, ser. A Guide to the Theory of NP-Completeness. New York, NY: Freeman, 1979.

[22] R. Chellappa, G. Qian, and Q. Zheng, "Vehicle detection and tracking using acoustic and video sensors," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, May 2004, pp. iii–793.

[23] 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

[24] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2018. [Online]. Available: http://www.gurobi.com

[25] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.

[26] W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Pyomo: modeling and solving mathematical programs in python," *Mathematical Programming Computation*, vol. 3, no. 3, pp. 219–260, 2011.

[27] T. P. Peixoto, "The graph-tool python library," *figshare*, 2014. [Online]. Available: http://figshare.com/articles/graph_tool/1164194

[28] W. Zheng, Q. Wang, J. X. Yu, H. Cheng, and L. Zou, "Efficient Computation of a Near-Maximum Independent Set over Evolving Graphs," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Apr. 2018, pp. 869–880.