Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Conflict Graph-based Time-triggered Stream Scheduling with Multicast

Simon Naß

Course of Study:	Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Heiko Geppert, M.Sc.
Commenced:	January 19, 2023
Completed:	Juli 19, 2023

Abstract

For the industry it is important to have deterministic reaction times for time-critical applications. These applications run on distributed systems that are connected via a network. Real-time communication is therefore essential to ensure that the real-time behavior is maintained. Thereby, identical information needs to be distributed to multiple members of the network. Multicast is great for sending the same data to multiple destinations while keeping the network traffic minimal and saving resources. In this thesis we integrate multicasting in a conflict graph-based routing and scheduling approach for time-triggered streams. We especially look at the routing of the multicast streams. We present different methods to construct the route of multicasts on the data-link layer for joint routing and scheduling and empirically evaluate these methods. We discover that calculating up to four candidate path-trees already solves our joint routing and scheduling problem in most cases. Further, scheduling multicast streams becomes harder with an increasing number of destinations per stream. We confirm that multicast streams reduce the traffic compared to multiple unicast streams in the conflict graph-based routing and scheduling approach. We can route and schedule 842 multicast streams in 25 seconds for networks with 81 nodes. Traffic plan updates with 25 additional streams are generated in less than 3.3 seconds.

Kurzfassung

In der Industrie ist es sehr wichtig deterministische Reaktionszeiten für zeit-kritische Anwendungen zu haben. Diese Anwendungen laufen auf verteilten Systemen, die über ein Netzwerk verbunden sind. Echtzeitkommunikation ist daher essenziell, um sicherzustellen, dass das Echtzeitverhalten aufrechterhalten wird. Dabei müssen identische Informationen an mehrere Teilnehmer des Netzwerks verteilt werden. Multicast ist sehr gut, um dieselben Daten an mehrere Empfänger zu senden und gleichzeitig den Netzwerkdatenverkehr minimal zu halten sowie Resourcen zu sparen. In dieser Arbeit integrieren wir Multicasts in einen konfliktgraph-basierenden Routing und Scheduling Ansatz für zeitgesteuerte Streams. Besonders untersuchen wir das Routing der Multicaststreams. Wir präsentieren verschiedene Methoden um Multicastrouten im Data-Link Layer für das Routing und Scheduling zu bilden und evaluieren diese Methoden empirisch. Wir stellen fest, dass die Berechnung von vier Kandidat-Pfad-Bäumen unser kombiniertes Routing und Scheduling Problem bereits in den meisten Fällen löst. Des Weiteren wird das Scheduling von Multicaststreams mit ansteigender Zahl an Zielen pro Stream schwieriger. Wir bestätigen, dass Multicaststreams im Vergleich zu mehreren Unicaststreams den Datenverkehr im konfliktgraph-basierenden Routing und Scheduling Ansatz reduzieren. Für Netzwerke mit 81 Knoten verarbeiten wir 842 Multicaststreams in 25 Sekunden. Datenverkehrsplanänderungen mit 25 zusätzlichen Streams werden in weniger als 3,3 Sekunden generiert.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ACK acknowledgment. 24

CNC centralized network controller. 15, 27

DVMRP Distant-Vector Multicast Routing Protocol. 19

GFH Greedy Flow Heap Heuristic. 16

IGMP Internet Group Management Protocol. 19

IIoT Industrial Internet of Things. 15

ILP Integer Linear Programming. 16

IoT Internet of Things. 24

IT information technology. 15

KSP k-shortest path. 25

MAC Media Access Control. 19

OT operational technology. 15

PIM Protocol Independent Multicast. 24

PTP precision time protocol. 27

PubSub Publish Subscribe. 19

QoS quality-of-service. 17

RPF Reverse Path Forwarding. 19

TAS Time-Aware-Shaper. 17

TSN Time-Sensitive Networking. 15, 17

VLAN virtual local area network. 17

1 Introduction

Today's industry contains application domains such as manufacturing, automotive, smart grids, and other kinds of cyber-physical systems. All these applications include time-critical functionalities. These time requirements can be hard or soft. Hard time-critical requirements need to always be fulfilled to assure the functionality. Actuator brakes are one example for safety-critical components that have hard time requirements, since a lost or delayed signal might lead to a crash. If soft time-critical requirements are not met, functionality or quality is degraded. Video streaming services show soft time-critical requirements since in the worst case the video quality decreases or skips some pictures. The industrial systems that are of interest for this thesis usually have hard real-time requirements.

The industry often requires multiple systems to communicate with each other. Therefore, industrial local area networks with real-time communication are essential. Deterministic delay and jitter is an important characteristic of real-time communication. Industrial Internet of Things (IIoT) systems originally had their own separate networks that were just responsible for the operational technology (OT). The information technology (IT) that has no timing constraints would be on a separate network with best effort transmission.

With developments like the Time-Sensitive Networking (TSN) the OT is integrated in the IT of the Ethernet domain for the data-link layer. The collection of IEEE 802.1 standards for TSN provides the ability for real-time guarantees but does not include routing and scheduling algorithms. For TSN systems explicit routing with a network wide traffic plan is very appealing because buffering of frames can be minimized and controlled for deterministic end-to-end delays. For synchronized routing approaches avoiding buffering is essential to achieve end-to-end delays that are as short as possible. A traffic plan entails the route and the schedule for every frame. The traffic plan is calculated in a centralized network controller (CNC) and then distributed to the network participants. This allows decentralized communication between the network participants according to the traffic plan.

In the IIoT we deal with processes that are managed by networked sensors, actuators, and controllers. Sensor and actuator data need to be distributed to multiple members of the network. One example is a sensor value in a car that is sent separately to the controller unit, a display, and a database for logging. Another example would be a temperature value that is sent to multiple cooling units nearby. In such cases, where the same data is sent to multiple members of the network, using multicast is convenient.

Using multicast on the network layer or a higher layer requires to sent unicasts in sequence to all destinations. Depending on the used network links this potentially increases the end-to-end delay. Sending multiple unicasts requires for each destination the same traffic in the network as a single unicast. If multicast is implemented on the data-link layer, only one frame needs to be sent, which is duplicated along the route if necessary to reach all destinations. This leads to less network traffic.

Therefore, sending multicast frames saves resources. Because these multicasts reduce traffic, they are also more scalable. Overall, using data-link layer multicasts has a positive effect on the network traffic.

For many IIoT systems it is beneficial to have a dynamic schedule that adapts to the current communication needs. Such plug-and-produce scenarios need the CNC to adapt the traffic plan on the fly. Otherwise, the traffic plan has to be statically recomputed offline from scratch and deployed with a system and network restart. In general, dynamic scheduling is faster than a static approach and prevents downtime.

There are already static joint multicast routing and scheduling algorithms. Schweissguth et al. [STP+20] as well as Yu and Gu [YG20] use an Integer Linear Programming (ILP) solver to generate a traffic plan. However, both approaches do not support dynamic routing and scheduling. The conflict graph-based solver by Falk et al. [FGD+22] can dynamically schedule traffic plans for unicast streams with the Greedy Flow Heap Heuristic (GFH). But this approach does not support multicast. To the best of our knowledge, there is no dynamic joint routing and scheduling solver for multicast streams for the data-link layer yet. We extend the conflict graph-based solver to handle multicasting. Therefore, this thesis has the following contributions:

- We extend the conflict graph-based approach to allow routes to multiple destinations
- We present several algorithms to find candidate routes to multiple destinations
- We provide changes regarding the GFH to improve the solver in multicast scenarios
- We empirically evaluate our approaches with different multicast distributions and network topologies with 81 nodes

In Chapter 2 we offer background knowledge about TSN, Ethernet, multicast, graphs, and the independent colorful set problem. Then we present related work in Chapter 3. Afterwards, we introduce our system model in Chapter 4. In Chapter 5 the problem statement is formally established. Chapter 6 explains differences in the path of unicasts and multicasts, the data structure representing multicast routes, and the construction of the multicast routes, followed by the GFH solver heuristic considerations in Chapter 7. Finally, we reach the evaluation in Chapter 8. We end with the conclusion and future work in Chapter 9.

2 Preliminaries

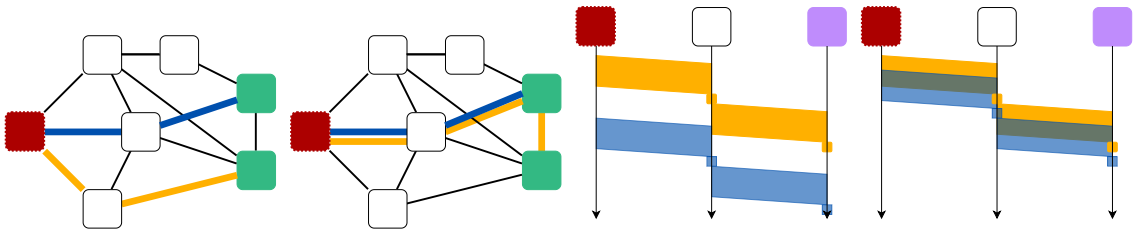
In this chapter, we present background knowledge and technologies necessary to understand our contribution to conflict graph-based time-triggered stream scheduling by adding multicast support. We will talk about Time-Sensitive Networking (TSN), different concepts regarding multicast, and conflict graphs.

2.1 Time-Sensitive Networking and Ethernet

Time-Sensitive Networking (TSN) is a collection of standards extending the Ethernet standard to share a single network for different types of traffic. It adds real-time guarantees to ensure that time-critical frames arrive on time. The bounded network delays are a quality-of-service (QoS) property. TSN is implemented on the data-link layer. The traffic in TSN is classified with priorities. These priorities are realized with virtual local area network (VLAN) tags. The tags are used to handle the different priorities over the same physical network while maintaining to keep them separate from each other.

In time-triggered TSN the delay is kept deterministic to ensure it is predictable. The standard IEEE 802.1Qbv [16] introduces a shaper, known as Time-Aware-Shaper (TAS). The shaper controls the behavior of the gates. This allows precise regulation of which frame a switch forwards next. TAS contains eight queues per outgoing port to separate the traffic classes. The TAS uses a gate control list to determine a subset of open queues to forward elements of one of these traffic class queues at a time. The gate control list is a sequence of instructions with a timestamp for each list entry. It changes for each gate if it is open or closed. The time the next gate control list instruction is triggered depends on a state machine. Frames in queues that are closed are not permitted to be selected as the next forwarded frame. From the remaining open queues, a frame is selected for transmission by the priority of the traffic class queue. If necessary, a guard band prevents frames of protected and unprotected traffic classes from overlapping when the gate control list changes the open queues. This is achieved by closing queues earlier than opening the next queues. By doing so, a guard band as a separation in time between the protected and unprotected traffic classes occurs. In the guard band transmissions of frames that have already started can be finished safely.

How long a frame is queued depends on if a gate of the queue is open and the amount of frames that arrived earlier in the queues. This queuing delay of each frame needs to be as deterministic as possible to ensure a reliable scheduling. If a frame arrives at an almost full queue, the delay is higher than if the queue is empty because the frame needs to wait until all frames in front of it are forwarded. Other open queues with higher priority also have an influence on the delay of a frame. The delay of an arriving frame is smaller when the gate control list opens the corresponding queue than when the queue has just been closed. This is because it requires time before the gate control list reopens. In this paper, we consider the no-wait approach also known as zero queuing, similar



(a) spatial separation **(b)** spatial conflict **(c)** temporal separation **(d)** temporal conflict

Figure 2.1: Examples of spatial and temporal separation and conflict

to Dürr and Nayak [DN16] to keep the queuing delay constant. Zero queuing can be realized by choosing the schedule appropriately so that the queues are always empty, and the frame is forwarded immediately.

Communication is organized in streams from one sender to one or many receivers. A stream is a sequence of frames that has a traffic class and a scheduled time frame. Streams conflict with each other if two frames are trying to simultaneously use the same network link in the same direction. Sending two streams over the same route at the same time is not possible because frames can only be forwarded one at a time, which would result in one being delayed, and in turn violating the no-wait constraint we adopt from Dürr and Nayak [DN16]. Therefore, precise routing and scheduling are necessary to avoid conflicts. Ensuring spatial or temporal separation for the streams prevents conflicts. Two streams are spatially separated if their routes do not overlap. Meaning that the routes do not share any network links. Two streams are temporally separated if they are at different places at all times. Implying that if they share network links, their frames are forwarded over these at different points in time. In Figure 2.1 the difference between separation and conflicts is presented.

2.2 Multicast

Unicasts (1:1) send single frames from one source to one destination (cf. Figure 2.2a) and broadcasts (1:all) send frames to all participants of the network (cf. Figure 2.2d), multicasts (m:n) differ by sending the frames to a selection of n destinations in the network (cf. Figure 2.2c). The m in multicast (m:n) stands for the number of sources that send different frames to the n destinations independently of each other. If we put no restrictions on which senders are allowed to send to a multicast group, we end up with multicast (all:n). Multicasts reduce the amount of frames sent across a network by combining the same frames sent to different destinations into one single frame sent to multiple destinations.

This paper deals with multicast (all:n) on the data-link layer. Depending on the network topology and the application, multicast is also implemented at other layers. Not all underlying networks support multicast, so there are higher-layer implementations that are independent of lower layer implementations. To take full advantage of multicast, and not just as an abstraction, it is necessary to implement multicast dependent on the multicast implementation of lower layers.

On the application layer multicast can be realized by sending unicasts to all destinations at the same time (cf. Figure 2.2b). Another way to implement multicast on a higher layer is with a Publish Subscribe (PubSub) system.

Multicast can also be part of the network layer. Here the destination IP-Address is not mapped to a single destination but to a dedicated multicast group. Such multicast IPv4-Addresses (Class D) have their unique range from 224.0.0.0 to 239.255.255.255. With the help of the Internet Group Management Protocol (IGMP) devices can join and leave a multicast group.

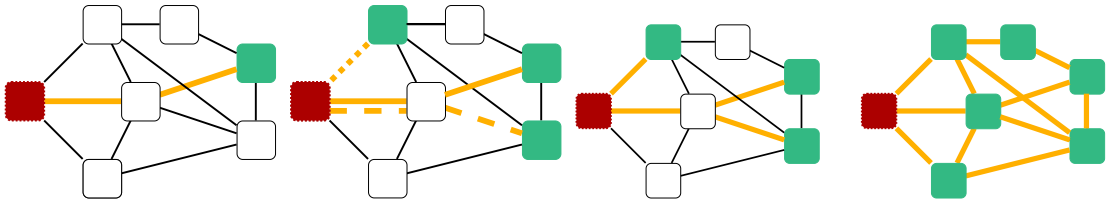
The routes of multicasts in the network layer are divided into two categories called source-based tree and shared-tree. In source-based trees, a unique spanning tree from the source to all multicast destinations is generated for every source. Most methods that construct the multicast route use the network information that is gathered from the unicast routing protocol. One way to generate a route is with Reverse Path Forwarding (RPF). It floods the network along a broadcast spanning tree. The Distant-Vector Multicast Routing Protocol (DVMRP) described in [88] improves RPF. Here, unnecessary subtrees are removed by sending prune packets. Moy [Moy94] defines a multicast extension to OSPF (MOSPF) that has a global state with its link-state-routing and therefore every network node can calculate source-based trees. For shared-tree routing, one tree per multicast destination group is generated, not one tree per source. These trees have one core node as a reference point in the network. The sources sent the packets to the core and distribute them to the destinations onward from the core node.

IP multicast needs an underlying support for multicast on the data-link layer. This is done by having specific multicast Media Access Control (MAC) addresses. A MAC address is represented by 48 bits (six bytes), but can be extended to 64 bits (eight bytes) [14]. Figure 2.3 depicts the structure of a MAC address. Multicast addresses are separated from unicast addresses by the least significant bit of the first byte. The so called I/G-bit (individual/group address depicted with M) is 1 for multicast. The next bit is to separate globally administrated from locally administrated addresses. This U/L-bit (universal/local address depicted with X) is 1 for locally administrated addresses. If it is set to 1, the next two bits are for separation of different address subspaces defined in the Structured Local Address Plan (SLAP) [17]. (Y=0, Z=1) is for Extended Local Identifier (ELI), (Y=1, Z=1) is for Standard Assigned Identifier (SAI) by P802.1CQ, (Y=0, Z=0) is for Administratively Assigned Identifier (AAI) meaning company internal assignments, and (Y=1, Z=0) is reserved. The rest of the six byte address is for the actual address number and subspace dependent specifications.

Switches typically broadcast multicast MAC addresses because they cannot identify a single port to forward the frame to. However, this network flooding is an undesirable behavior. In order to mitigate this, the switch requires a mapping between the ports and multicast MAC addresses. Then, known multicast frames are forwarded only to the ports that require the frame. Unknown multicasts are still broadcasted. IGMP snooping is a function that enables switches to look inside of IGMP packets passing through them. This allows them to generate forwarding tables containing the mapping between ports and MAC addresses.

2.3 Graph Concepts

This section introduces some definitions of graphs that are required for this paper. It includes colored conflict graphs and independent sets.



(a) unicast (1:1) **(b)** multiple unicasts **(c)** multicast (1:3) **(d)** broadcast (1:all)

Figure 2.2: Example of the different ways to send frames to multiple destinations

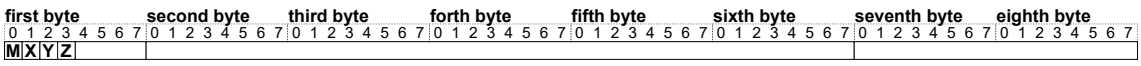


Figure 2.3: MAC address block

2.3.1 Colored Conflict Graph

A colored conflict graph $G(V, E, C)$ is an undirected, vertex colored graph, where V is a set of vertices, $E \subseteq \{(v_1, v_2) | v_1, v_2 \in V\}$ and $\forall (v_1, v_2) \in E : (v_2, v_1) \in E$ is a set of edges between vertices in G, and C is a set of colors. Each vertex can be mapped to a color with the function $color : V \rightarrow C$.

In our case, each vertex $v \in V$ represents a stream configuration and has exactly one color $c \in C$ corresponding to the stream to which this configuration belongs. Each edge $(v_1, v_2) \in E$ represents a conflict between the stream configuration of vertex v_1 and the stream configuration of v_2. The stream configuration contains a path and a phase. A potential conflict arises if two vertices are not spatially separated. This occurs when, the vertices have an overlap in their configured path. An actual conflict occurs when there is also no temporal separation. This occurs when there is a path segment that both stream configurations want to use at the same time.

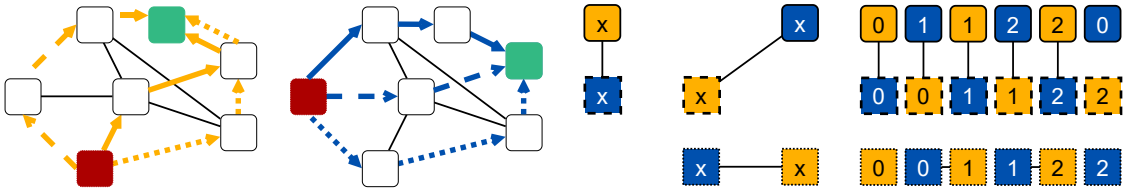
2.3.2 Independent Set

An independent vertex set $S \subseteq V$ of a colored conflict graph G is a subset of vertices without direct connections. For all pairs of two vertices that are contained in the set, there is no edge between them. $\forall s_1, s_2 \in S : (s_1, s_2) \notin E$

An independent colorful set is an independent set containing vertices of different colors $\forall s_1, s_2 \in S : color(s_1) \neq color(s_2)$. The independent colorful set is considered a tropical independent set if it contains each color exactly once $|S| = |C|$.

The joint routing and scheduling problem can be reduced to the independent colorful set problem as Section 2.4 describes in more detail. Therefore, an independent colorful set is a possible way how we can configure our streams without having conflicts between them.

Figure 2.4 shows an example with two streams s_1 and s_2 each having nine stream configurations combined out of tree different paths and tree different phases. Figure 2.4c shows where the routes overlap independent of the phases. Figure 2.4d is the actual resulting conflict graph. One solution for the independent colorful set is s_1 with the lined route and phase 0 and s_2 with lined route and phase 0.



(a) routes of stream s_1 **(b)** routes of stream s_2 **(c)** spatial conflict graph **(d)** total conflict graph

Figure 2.4: Example of a conflict graph

2.4 Network Problem Mapped to Graph Problem

In this section, the reduction of the joint routing and scheduling problem in the network to a conflict graph problem is described in more detail.

For each stream, there is a set of possible routes and schedules. Any combination of a route and a schedule for a stream configuration is a valid routing and scheduling for just this one stream. But we need to select stream configurations so that they are all spatially or temporally separated from each other. With such combinations, we have a valid traffic plan.

By mapping each stream configuration to a vertex, we can represent all combinations in a graph. We evaluate whether the stream configurations are separate or in conflict by comparing each vertex with all other vertices. Stream configurations that are spatially separated are not in conflict with each other, and therefore the vertices are not directly connected. If the stream configurations have a spatial conflict but are temporally separate, the vertices are also not directly connected. A temporal conflict between the stream configurations implies a conflict between the corresponding vertices in the graph. If two vertices are in conflict, then there is an edge between them. Otherwise, the vertices are not linked. This results in a conflict graph representing stream configurations either working or not working together in a traffic plan.

When there is a new stream that needs to be added, vertices that represent the stream configurations of this specific stream are added to the graph. All vertices that are mapped from the same stream have the same color. Then all existing vertices are compared to the new ones to find where edges need to be added.

By solving the independent colorful set problem on the conflict graph, we receive a selection of stream configurations. Per definition the selection contains just one configuration per stream because the independent colorful set selects one vertex per color. These stream configurations are spatially or temporally separated, since the independent colorful set chooses vertices that are not connected with each other. We can use the solution of the independent colorful set problem as a solution for our routing and scheduling problem. It fulfills the requirement to choose one stream configuration per stream so that they are all separated from each other. It was shown by Falk et al. [FGD+22] that partial conflict graphs already contain possible joint routing and scheduling solutions. Therefore, not all stream configurations have to be included from the beginning. The conflict graph can later be extended as required.

3 Related Work

The collection of IEEE 802.1 standards for TSN specifies deterministic, time-triggered network traffic. However, it does not define routing and scheduling algorithms and strategies to calculate traffic plans. Accordingly, research has been done in recent years to find ways to compute these traffic plans. First, we introduce papers that use conflict graphs to solve the scheduling and routing in a centralized manner. Then we discuss research that use constrained-based solvers. Finally, we present papers that do not deal with schedulers but other important information that is relevant in this thesis.

Falk et al. [FGD+22] present a configuration-conflict graph-based approach to solve the traffic planning problem. It is a dynamical version of the approach in [FDR20] where they route and schedule unicast streams in TSN networks with cyclic time-triggered traffic. They use a CNC unit that considers the streams of the current traffic plan p_i for the calculation of the new traffic plan p_{i+1}. They find independent routes and schedules for the traffic plan by solving the independent colorful set problem of a conflict graph. We try to build on this approach and add the capability to route and schedule multicast streams. This increases the amount of use-cases for conflict graph-based routing and scheduling since multicast is often used in TSN to distribute controller instructions and sensor data.

An approach to schedule multicast streams with conflict graphs is shown by Traskov et al. [THM+08]. They consider network coded multicast traffic in wireless environments. Here, all packets are broadcasted from the sender to all neighbors. If it is a unicast or multicast depends on the number of receivers that choose to listen to the packet. In their conflict graph, a conflict occurs if a node has to actively listen for two or more packets at the same time. A conflict also occurs if a node has to send two or more different packets at the same time. A conflict also occurs if a node has to listen and send packets at the same time. We have a different concept of conflicts in the non-wireless networks that is not as restricted as their concept of conflicts. In our network, a conflict occurs if two frames need to be sent over the same egress port at the same time. A switch, however, can listen to two packets simultaneously and forward them to two different ports. Additionally, in contrast to our approach Traskov et al. [THM+08] do not include the routing of the streams in the conflict graph.

There are approaches that solve routing and scheduling for multicast streams without conflict graphs. Schweissguth et al. [STP+20] adjust an ILP solver from [SDT+17] to support multicast. They also optimize resource constraints to have a smaller ILP model and flexible trade-offs between runtime and quality of the chosen schedule. Their multicast model requires up to 3.45x the average runtime of the unicast model. Because of the added functionality, we expect an increase in time, resulting in more complex routing and scheduling for each stream. They show a scenario with 40 streams, on 12 switches, and solve it in about 46 seconds. Depending on the chosen objective, the runtime can take more than 1125 seconds. Schweissguth et al. [STP+20] present an approach for small networks and not many concurrent streams. Our approach is able to scale more appropriately and therefore handle more streams simultaneously.

Yu and Gu [YG20] also use an ILP for their adaptive group routing and scheduling of multicast streams. They use a preprocessing phase to prune unnecessary nodes in the network topology. Additionally, link grouping is used to simplify the possible network paths a stream can take. This helps to reduce the complexity of certain constraints in the routing and scheduling phase of the ILP solver. They evaluated their approach in a network with 24 switches and reached a maximum of 450 streams they can schedule. It took them about 83 minutes (50 s per stream) to schedule 100 streams. We aim to be more scalable since our foundation the unicast approach of Falk et al. [FGD+22] shows results in bigger networks with 49 switches and more.

Another constrained based solver is described by Santos et al. [SSN19]. They use a different solver based on Satisfiability Modulo Theories (SMT) named Z3 from [MB08]. This approach uses stream fragments to encode multicast streams and is able to consider latency and jitter in the scheduling result. Their approach was evaluated with 10 switches and reached a maximum with 10 senders and 73 receivers. It took them about two hours to schedule streams to 40 receivers with streams performing tree hops. This approach needs time to calculate the routing and scheduling. Our approach is faster.

Pahlevan et al. [PTO19] generate routing and scheduling solutions by mapping directed acyclic graphs that represent the streams onto the network topology. They scheduled about 85 out of 100 requested streams on a mesh topology with ten switches and five end devices per switch in 1.58s. This approach is capable of multicasting, but they just evaluated with sets of unicast streams. Since we use zero queuing our approach has a shorter end-to-end delay.

Next we present research related to multicasting. Brooks and Uludag [BU18] show one way to realize multicasting in a higher layer for the Internet of Things (IoT). They use OPC UA PubSub over TSN. PubSub traffic routing in classic Ethernet routers can be realized with the Protocol Independent Multicast (PIM) system. PIM provides two different methods. The dense mode [NAS05] and the sparse mode. While PubSub network traffic with a broker does not congest the network as much as multiple unicasts, the quantity of duplicate network traffic depends on the location of this broker.

Kotachi et al. [KSSO19] present a multicasting routing protocol for low-power and lossy networks. Their protocol called REMI approaches the routing with clusters.

In the context of multicast MAC addresses the IEEE 802.1 working group has a currently ongoing project called: P802.1CQ Multicast and Local Address Assignment. Among other topics they address the assignment of devices to the multicast MAC addresses in the Standard Assigned Identifier. They integrate address assignment using Block Address Registration and Claiming (BARC) from Marks [Mar22]. In our paper the CNC knows all devices that are part of a multicast group. We do not make a difference between manually assigned multicast addresses and BARC or another address assignment protocol.

One important multicast problem is the acknowledgment (ACK) flooding. Generally the multicast frames do not arrive simultaneously making it very hard to send an ACK back differently than as an unicast. This has the effect of an increasing amount of small frames traversing through the network. The network might not be capable of handling this since multicasts are used to reduce network traffic in the first place. We support an ACK by separately adding unicast streams from the destinations to the source.

Another related research topic are k-shortest path (KSP) finding algorithms. They find k different loop-free paths to a destination while keeping the paths as short as possible. The most well-known KSP algorithm is described by Yen [Yen71]. Other KSP algorithms and Yen's algorithm improvements are presented by Eppstein [Epp94], de Queirós Vieira Martins and Pascoal [QP03], Hershberger et al. [HMS07], and Chen et al. [CCCL20] [CCCL21]. In our paper we consider the Dijkstra Overlap algorithm of Geppert et al. [GDBR23] as a relaxed version of KSP where we expect just k short path and not the k shortest. The Dijkstra Overlap algorithm executes Dijkstra's algorithm but changes the weight of all links of the already found paths after each search. This allows the Dijkstra Overlap algorithm to find a new short path with each search. The Dijkstra Overlap is one way to extend Dijkstra's algorithm. It does not ensure that these short paths are the k shortest, but it does produce candidate paths that are less similar as the resulting paths of KSP algorithm. Dijkstra Overlap is for example used by Zhang et al. [ZSQ23].

4 System Model

In this chapter, we describe our system model which is derived from Falk et al. [FGD+22] [FDR20]. We build upon their system model because we adapt their algorithm to support multicasting. For our time-triggered traffic, we consider centralized traffic planning. Therefore, the system model includes the essential network topology, the format of the streams, and the traffic planning. Table 4.1 provides an overview and description of all consecutively introduced variables.

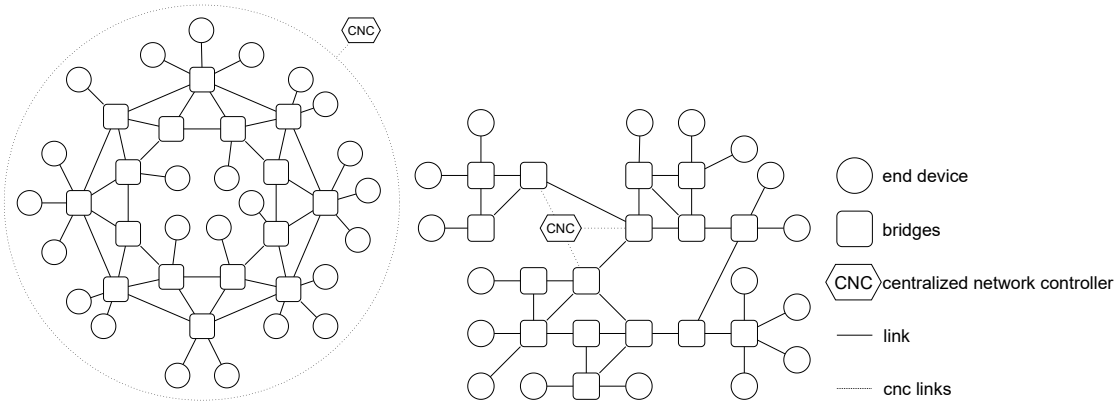
4.1 Network Topology

The network has a static structure, meaning its topology does not change over time. It consists of end devices, bridges, point-to-point links between nodes, and a centralized network controller (CNC). Figure 4.1 shows some network examples. We assume a homogeneous network topology to ensure all nodes and links have the same physical constraints.

End devices are nodes capable of sending and receiving frames. The source and destination of a stream is always an end device. If an end device wants to establish new streams, it sends a scheduling request to the CNC which processes them in batches. This schedule request is transmitted separately to the TSN traffic and is not part of the time-critical traffic. Each sender has a processing delay t_{src}. The t_{src} is the same for all nodes independent of the capabilities of each end device. Each destination node has a processing delay t_{dst}. These t_{dst} are also the same for all nodes.

Bridges are either switches or routers that forward frames over point-to-point links. These point-to-point links connect two bridges or a bridge with an end device. We assume full duplex point-to-point links. This allows us to send in both directions at the same time without worrying about conflicts. We have the same bandwidth B and the same propagation delay t_{prop} in all our full duplex Ethernet links. The processing delay t_{proc} for all bridges is constant. Our switches and routers need to be TSN capable. The IEEE 802.1Qbv complient switches described in [SCS18] satisfy this constraint. We assume these bridges to have a queuing delay of zero $t_{que} = 0$ because of the no-wait approach similar to Dürr and Nayak [DN16]. To ensure a consistent time reference between all nodes, the precision time protocol (PTP) is used. The precision we plan with is 1 µs, sometimes called a macrotick. We represent the synchronization in time by making our formulas relative to the common reference point in time t_0.

Our paper considers a network where a CNC routes and schedules the traffic. Because of this, no communication between routers and switches is needed to determine which paths the multicast frames take. Since we do not want switches to flood the network with multicast frames, we assume the CNC is capable to produce forwarding tables similar to the IGMP snooping. We also assume that the CNC knows all devices with their unique address that want to listen to a multicast address



(a) Example ring(16,2) **(b)** Example generic **(c)** legend

Figure 4.1: Example of possible network topologies

before it calculates the routing of the multicast frames. This ensures that senders just have to tell the CNC to what address they want to send independent of the type of address. Additionally, the senders do not necessarily need to know all listeners of the multicast group they send to.

The CNC has a global view of the network internally represented by an undirected graph. It computes and distributes a global traffic plan. Every node in the network follows this traffic plan to send, receive, and forward frames appropriately. The CNC can talk to every node of the network to distribute the traffic plan and to receive schedule requests. This communication is not part of our streams, but can be realized using the separated best effort traffic in our network or a second communication network.

4.2 Stream Structure

A stream s is represented by the tuple $s = (\text{source, group destination, } t_{cycle}, \text{ frame size, } t_{\max})$. It represents the communication from one source node to $\text{dstNo}(s) \in \mathbb{N}^+$ destination nodes. If $\text{dstNo}(s) > 1$, a group destination stands for the destination nodes as a multicast address. The distribution of different $\text{dstNo}(s)$ follows a probability distribution $P(X = \text{dstNo}(s))$. Each stream entails time-sensitive data and needs to reach all destination nodes of the group within bounded time to fulfill its QoS requirements. Under normal conditions the stream does not drop frames and all destinations of the group receive the frames. Each stream is a one directional communication with no ACK from the destinations to the source for flow control. If a two directional communication is necessary, separate streams from the destinations to the source need to be scheduled.

The source node of an active stream can transfer one frame at every transmission cycle. Each stream's transmission cycle has a length of $t_{cycle}(s)$. All frames sent over a stream have the same frame size. The end-to-end delay t_{e2e} describes the time a frame takes to reach all destination nodes. The deadline $t_{\max}$ describes the maximal time a frame is allowed to take to reach all destination nodes in order to fulfill the QoS. If there is no short enough route in the network that leads to a smaller or equal transmission time t_{e2e} for a frame than $t_{\max}$, then the QoS cannot be fulfilled for this stream.

With the same bandwidth across the network and the same frame size for all frames in a stream, the transmission time over a link can be computed as $t_{trans} = \frac{\text{frame\_size}}{B}$. The combined time a bridge needs to process, forward, and transmit the frame defines the time we need per hop $t_{perhop} = t_{proc} + t_{prop} + t_{trans}$. With a known per hop delay we can calculate the delay up to any link in the path to a destination with $t_{dst}(h) = (t_{src} + t_{prop} + t_{trans}) + (t_{perhop} \cdot h)$. The end-to-end delay of a path to a destination that contains h bridges results in $t_{e2e}(dst) = t_{dst}(h) + t_{dst} = (t_{src} + t_{prop} + t_{trans}) + (t_{perhop} \cdot h) + t_{dst}$.

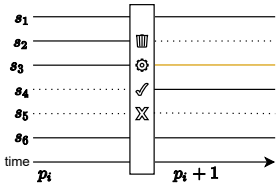
The parameters that define the stream as a tuple do not change during its runtime. If we want to have a unicast stream, the tuple does not change except the group destination. We replace the group destination with the actual destination node. The taken route and the scheduled phase are parameters of a stream that can change from one traffic plan to the next. Therefore, they are specified in the configuration of the stream $config(s, p_i) = (\phi, \pi)$.

The route of a stream starts at one node and ends at one or multiple destination nodes. Therefore, a route is not just a single loop free path but a tree structure to describe the path to all destination nodes simultaneously. We call this source-based tree structure describing the paths a path-tree. A stream with only one destination is a special case of a path-tree with a branching factor of one. We have to deal with highly unbalanced trees. The children in the path-tree are not ordered and the amount of children is not bound. The path-tree structure mainly preserves paths without link duplication. This leads to a similar structure to what multicast packets take in the network layer. Additionally, the path-tree easily tells us how many bridges we have already passed during the path traversal by the tree depth. We limit the routes that can possibly be selected by the stream to n_{path} candidate paths. Each member of the candidate paths is identified by a different π. Path-trees that have a longer t_{e2e} than $t_{\max}$ can never be a candidate path since they do not fulfill the QoS of the corresponding stream. We will later describe different approaches to determine which links create the path-tree. These approaches generally merge paths of one source to one destination together to one path-tree that reaches multiple destination nodes.

We determine multiple phases ϕ also called sending delays for each stream. The frames are emitted by a sender over a link at $t_0 + k \cdot t_{cycle} + \phi, k \in \mathbb{N}$ with $\phi \in [0, t_{cycle} - t_{trans}]$. This allows two streams to take parts of the same path at different points in time. The starting time is the only way to manipulate the time a path-tree takes a link if bridges cannot buffer frames. In our paper this is the case since we assume zero queuing similar to Dürr and Nayak [DN16]. Routes that have a larger or equal transmission time after the phase delay $t_{e2e} + \phi$ than $t_{\max}$ also do not fulfill the QoS for this stream.

4.3 Traffic Plan

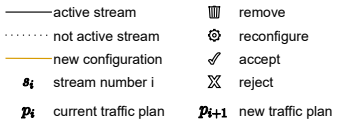
A traffic plan is a global representation of how the whole traffic needs to be routed and scheduled in the network. At any time, the network obeys to exactly one traffic plan called the current traffic plan p_i. When the CNC has a set of schedule requests corresponding to a set of new streams that need to be added, it computes a new traffic plan. We refer to the new streams that need to be added to the current traffic plan p_i with $\text{ReqS}(p_i)$. Over the time, calculating new traffic plans by the CNC leads to a sequence of traffic plans. The current traffic plan p_i is replaced with the newly calculated one p_{i+1}. It is not possible to precalculate the traffic plans $p_{i+2}, \ldots$ in advance since we are unable to predict the schedule requests of the future.



(a) Example p_i to p_{i+1} graphic

x	p_i	p_{i+1}
ReqS(x)	$\{s_4, s_5\}$	{}
ActiveS(x)	$\{s_1, s_2, s_3, s_6\}$	$\{s_1, s_3, s_4, s_6\}$
RejS(x)	{}	$\{s_5\}$
RemS(x)	$\{s_2\}$	{}
config(s_1,x)	(π_1, ϕ_1)	(π_1, ϕ_1)
config(s_2,x)	(π_1, ϕ_2)	(-,-)
config(s_3,x)	(π_1, ϕ_1)	(π_3, ϕ_4)
config(s_4,x)	(-,-)	(π_2, ϕ_5)
config(s_5,x)	(-,-)	(-,-)
config(s_6,x)	(π_1, ϕ_1)	(π_1, ϕ_1)

(b) Example p_i to p_{i+1} table



(c) legend

Figure 4.2: Example of a traffic plan change

The current traffic plan contains a set of active streams, meaning the streams that are admitted in the network to transmit frames. We refer to these active streams of the current traffic plan p_i with ActiveS(p_i). If a stream s is admitted in the traffic plan p_i, the expression admits(s, p_i) is one, otherwise it is zero. If a new traffic plan p_{i+1} is calculated, the current traffic plan p_i is extended with the added streams. Possibly, there are streams that do not fit. This is the case if we cannot find a configuration that is temporally separated to all other streams in the traffic plan. These streams have to be rejected. Streams that do not fulfill the QoS need to be rejected as well. We refer to the rejected streams of a traffic plan p_i with RejS(p_i). All streams of ReqS(p_i) end up either in ActiveS(p_{i+1}) or in RejS(p_{i+1}) after the new traffic plan p_{i+1} was calculated. Once a stream is active it can no longer be rejected in the future during the calculation of a new traffic plan. But if it is clear that a stream is no longer needed, it can be removed. We call these RemS(p_i). The active streams can no longer be rejected, but they can be rescheduled to use a different configuration in the new traffic plan. This can help close holes that the RemS(p_i) left behind and optimizes the routing and scheduling of ActiveS(p_i) streams. The active streams of the new traffic plan p_{i+1} are overlapping the current active streams like ActiveS(p_{i+1}) = (ActiveS(p_i)\RemS(p_i)) ∪ (ReqS(p_i)\RejS(p_{i+1})).

The best effort traffic can be transmitted over a link in the time when the traffic plan is empty. In this time the traffic plan has not scheduled any stream to take this link. Therefore, the best effort traffic cannot conflict with the scheduled time-sensitive traffic.

In Figure 4.2 an example of the changes that can occur with the calculation of a new traffic plan is shown.

variable	description
t_0	common reference point in time
t_{src}	processing delay of source nodes
t_{dst}	processing delay of destination nodes
t_{proc}	processing delay of bridges
t_{prop}	propagation delay of a network link
t_{que}	queuing delay of bridges = 0
t_{cycle}	the length of the transmission cycle of stream s
$t_{\max}$	max time a frame is allowed to take for the transmission
t_{e2e}	time a frame takes for the transmission to all destinations
$t_{e2e}(dst)$	time a frame takes for the transmission to a single destination dst
$t_{dst}(h)$	time a frame takes for the transmission to the h-th hop in a path to a single destination dst
t_{trans}	time it takes a frame to travel along a link
t_{perhop}	total time needed per hop
$config(s, p_i)$	the configuration of the stream s in the traffic plan p_i
n_{path}	number of candidate path-trees
π	the identifier of the chosen path-tree for this configuration of the stream
ϕ	the identifier of the chosen phase for this configuration of the stream
$frame\ size$	the frame size of frames sent over stream s
B	the bandwidth, maximum rate of data transfer across a link
dstNo(s)	number of destinations of the stream
p_i and p_{i+1}	current traffic plan and new traffic plan
s_i	stream i
ReqS(p_i)	the requested streams to add to traffic plan p_i
ActiveS(p_i)	the active streams of traffic plan p_i
RejS(p_i)	the rejected streams of traffic plan p_i
RemS(p_i)	the streams to be removed from traffic plan p_i
admits(s, p_i)	tells if a stream s is admitted in traffic plan p_i

Table 4.1: System model variables

5 Research Problem

In this chapter, we present our research problem on conflict graph-based time-triggered stream scheduling with multicast. In addition, we want to formulate our optimization goals for this problem with respect to multicasts.

We want to solve the routing and scheduling dynamically for multicast and unicast streams. Since the routing and scheduling can be solved dynamically with a conflict graph by solving the independent colorful set problem [FGD+22], we need to find an appropriate model to represent multicast streams in the conflict graph. The dynamic GFH algorithm also needs to be adapted to account for the multicast.

The solver of Falk et al. [FGD+22] has the objective of Equation (5.1) to put as many streams as possible in the traffic plan. The first sum of the function is for the rescheduling of already active streams. The second sum tries to maximize what streams to admit out of the requested ones. Since the already active streams must continue to be active until stated otherwise, they have a much higher importance than all new streams together. This is made sure by $\frac{1}{|\text{ActiveS}(p_i) \cup \text{ReqS}(p_i)|}$.

$$F(\text{ActiveS}(p_i), \text{ReqS}(p_i)) = \max_{p_{i+1}} \sum_{s \in \text{ActiveS}(p_i)} \text{admits}(s, p_{i+1}) + \sum_{s \in \text{ReqS}(p_i)} \frac{\text{admits}(s, p_{i+1})}{|\text{ActiveS}(p_i) \cup \text{ReqS}(p_i)|} \tag{5.1}$$

Our solver optimizes the amount of multicast streams. We can differentiate between unicast and multicast streams s by their destination amount. Therefore, the weight function $w : \{s : \text{dstNo}(s) \leq n \wedge n \in \mathbb{N}^+\} \rightarrow [0, ub], ub \in \mathbb{R}^+$ is introduced. The weight function maps each stream to the value this stream provides. The parameter n represents the maximum number of destination nodes a stream can send to in the network. The weight function must be monotonically increasing between $\text{dstNo}(s) = 1$ and $\text{dstNo}(s) = n$ because the solver should always prefer more destinations when comparing which of two streams to add. Additionally, the weight function has an upper bound ub to make sure that all new streams together are less valuable than an already active stream with a low weight function output.

The weight function in Equation (5.2) that is plotted in Figure 5.1 is chosen since it does not prefer one group of streams over another as long as the same amount of destination nodes are reached. The weight function $w(s)$ is monotonically increasing dependent on the destination amount. It also fulfills the upper bound with $ub = \max\ w(s) = \max\ \text{dstNo}(s) = n$.

$$w(s) = \text{dstNo}(s) \tag{5.2}$$

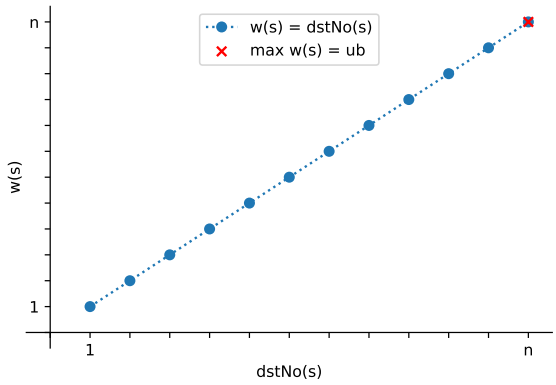


Figure 5.1: Weight function $w(s)$

Our optimization function shown in Equation (5.3) fulfills the extended goal to consider multicasts. It sums up the weight for each stream that is part of the new traffic plan p_{i+1}.

$$
\begin{aligned}
(5.3)\quad & OF(\text{ActiveS}(p_i), \text{ReqS}(p_i)) \\
&= \max_{p_{i+1}} \sum_{s \in \text{ActiveS}(p_i)} w(s) \cdot \text{admits}(s, p_{i+1}) + \sum_{s \in \text{ReqS}(p_i)} \frac{w(s) \cdot \text{admits}(s, p_{i+1})}{ub \cdot |\text{ActiveS}(p_i) \cup \text{ReqS}(p_i)|} \\
&= \max_{p_{i+1}} \sum_{s \in \text{ActiveS}(p_i)} \text{dstNo}(s) \cdot \text{admits}(s, p_{i+1}) + \sum_{s \in \text{ReqS}(p_i)} \frac{\text{dstNo}(s) \cdot \text{admits}(s, p_{i+1})}{n \cdot |\text{ActiveS}(p_i) \cup \text{ReqS}(p_i)|}
\end{aligned}
$$

A second goal is to keep the dynamic multicast approach scalable. The runtime of our solution should be as small as possible. One comparison are scenarios with only unicasts that represent the approach not to implement multicast at all. As a second comparison, we use scenarios with multiple unicasts representing the worst way to implement multicasting. This leads to z unicasts for a multicast of z destinations. Our solution should be between the two as close as possible to the only unicast scenarios.

We want to ensure that the routing of the multicast streams is as good as possible. This means that the route should be as short as possible, and that it needs to split as late as possible. To achieve this, the path-tree must use as few links as possible while maintaining a reasonable depth. Details about how to accomplish such path-trees is explained in the next chapter.

6 Path-tree Considerations

This chapter illustrates different aspects of path-trees. As presented in the system model (cf. Section 4.2), we change the route from a linear path to a tree to represent multicast streams in the conflict graph. First, we explain the necessary changes when using trees instead of linear paths. Then the data structures for the path-trees are described. Afterwards, we present how linear paths are integrated into a path-tree. Finally, methods for constructing path-trees are stated in detail.

6.1 Comparing Linear Paths with Path-trees

A stream can have an arbitrary amount of destinations since in our model $\text{dstNo}(s) \in \mathbb{N}^+$ holds. Therefore, the route of the stream is not just a single linear loop-free path, but a tree structure connecting the source to all destinations. This type of multicast routing is called source-based tree routing. A shared-tree route is not suitable for our case because it produces longer path that use the same links. The destinations could be arbitrarily positioned, causing a path-tree to have a different number of children for each node. This leads to a range of branching factors. In contrast, a linear path has a branching factor of one everywhere.

In addition, the destinations are at different distances from the source node, resulting in an unbalanced tree. Similar to the linear path, we are more interested in the tree level a link is positioned in than the order of the children. So reordering the children would be possible, but rebalancing the tree destroys the information stored in a path-tree.

An interesting metric is the current number of passed network links during the path traversal. In a linear path, this is indicated by how many links were taken so far. The current depth indicates the number of links that were taken in a path-tree. We can use the current number of passed network links to calculate $t_{dst}(h)$ since this metric correlates with the number of hops we have taken at this point.

We can still display unicasts as a specific form of multicast in a path-tree with $\text{dstNo}(s) = 1$. The path-tree structure in this particular case is equivalent to a linear path because there is a branching factor of one and one child for each inner node of the tree.

Using a path-tree instead of a linear path for the route of the stream has the disadvantage of requiring more complex and more costly comparison algorithms. In order to ensure spatial separation between two streams s_1 and s_2 in the case of linear paths, all links of s_1 must be compared to all links of s_2. To calculate all spatial conflicts between candidate routes of two streams, the algorithm depicted in Algorithm 6.1 needs to be executed. In the worst case of a path-tree, $\text{depth1} \cdot \text{dstNo}(s_1)$ links of s_1 need to be compared with $\text{depth2} \cdot \text{dstNo}(s_2)$ links of s_2 to guarantee spatial separation.

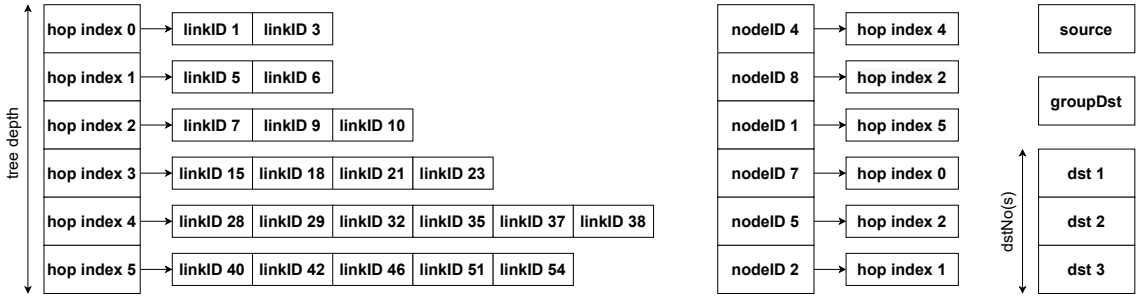
Algorithm 6.1 Spatial separation between two streams

```
procedure calculateSpatialOverlap(pathTree1, pathTree2)
    overlap = [];
    for all hop1 : pathTree1 do
        for all link1 : hop1 do
            for all hop2 : pathTree2 do
                for all link2 : hop2 do
                    if link1 == link2 then
                        overlap.add(link1);
                    end if
                end for
            end for
        end for
    end for
    return overlap;
end procedure
```

Algorithm 6.2 Temporal separation between two streams

```
procedure isTemporalSeparated(config1, config2)
    spatialOverlap = calculateSpatialOverlap(config1.route, config2.route);
    for i = 0; i < spatialOverlap.size(); i++ do
        time1 = computeTraversalTime(config1.route, spatialOverlap[i]);
        time2 = computeTraversalTime(config2.route, spatialOverlap[i]);
        if overlapInAnyFrame(time1,time2) then
            return false;
        end if
    end for
    return true;
end procedure
```

Temporal separation can be ensured by finding all overlapping links and checking if each overlapping link is taken at the same time in both streams. Algorithm 6.2 illustrates this check for temporal overlaps. The maximum number of overlapping links for linear paths is the amount of links in the smaller path of s_1 and s_2. In the case of a path-tree, the maximum number of overlapping links needs to be extended to $\min(\text{depth1}, \text{depth2}) \cdot \min(\text{dstNo}(s_1), \text{dstNo}(s_2))$ links. This is different from just taking the smaller path-tree. We could have one long, slim path-tree to a few destinations, while the other path-tree is short but reaches many destinations. The overlap would then be just the first couple of links of the slim path-tree that are also taken by the broad path-tree.



(a) Example of path-tree data structure

Figure 6.1: Example of path-tree data structure

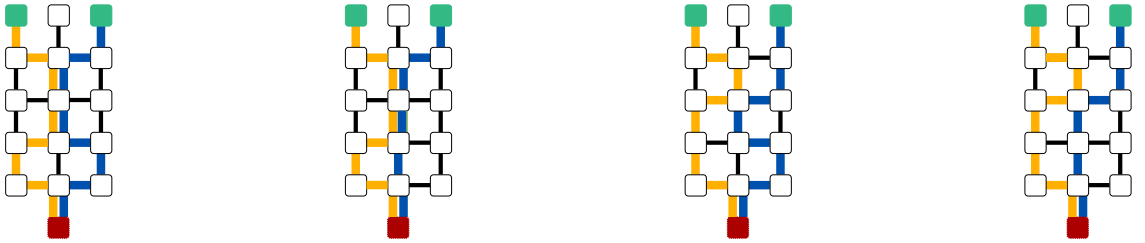
6.2 Path-tree Data Structure

The path-trees are represented as a structured collection of all the links used in the route of a stream. Traditionally, trees are represented through a structure with pointers or with one dimensional arrays. However, these representations are inappropriate in our case because they require a memory overhead for unbalanced trees. Instead, we derive our path-tree data structure from the link-list representation of a graph.

To structure the collection of links, we split them into smaller arrays of links that share one characteristic. The shared characteristic is the hop when the link would be taken during the traversal from the source to all destinations of a stream. Storing the links like this allows direct access to all links that are traversed in the i-th hop in constant time. In addition, the tree depth is directly available by the size of the outer array. Alternative graph data structures like an adjacency list would have the disadvantage of traversing from the start every time we want to know the links in the i-th hop or the tree depth. We use std::vector in our c++-implementation to represent the arrays. Vectors have the advantage of maintaining a direct index based access while having a flexible length.

An alternative data structure could be a list of all linear paths. But this alternative has the disadvantage that it would contain duplicate links. These duplicate links waste memory and contribute to more expensive spatial and temporal overlap calculations.

In summary, a vector that consists of vectors that contain links is the right data structure for us. The link collection data structure has the disadvantages that we need to traverse over all hops to find out if a new link connects to the already existing path-tree part. To mitigate this, we store all intermediate nodes of the path-tree. We implement this with a map using the nodes as keys and the hop leading to the node as value. This way we do not just know if a new link connects to a path-tree but also in which depth level of the tree. Additional to all used links and an intermediate node map, we store the source, the destination nodes and the group destination address. Figure 6.1 provides an example for the whole path-tree data structure. Overall our data structure needs O((tree depth + 1) · dstNo(s) + map(tree depth · dstNo(s))) space for storing a route of a stream.



(a) case 1.1 **(b)** case 1.2 **(c)** case 2.1 **(d)** case 2.2

Figure 6.2: Examples of invalid path-trees

6.3 Integrating a Linear Path into a Path-tree

In the following section, we want to explain how we integrate linear paths to become a path-tree. This has the purpose to recursively combine path to reach more destinations. How linear paths are found is not the focus of this section. For now, we assume that these are just the linear paths we have found for this path-tree. During linear path integration the path-trees can become invalid. In the following, we explain what that means. Then we present three different integration approaches.

6.3.1 Invalid Path-trees

We want to ensure that we do not construct invalid path-trees. The path-trees are considered invalid if they violate the properties of a tree. This is possible since we decide to use independent linear candidate paths and integrate them into a path-tree. Examples for such violations are loops or multiple branches that lead to the same node. We differentiate the following two cases of invalid path-trees.

Path-trees that have two different branches that split and then later reconnect again are invalid. Examples for this case are shown in Figure 6.2a with the same length and in Figure 6.2b with different lengths of their separated branch parts. Also, path-trees that have two different branches that split and then later cross each other in a node are invalid. This case is illustrated in Figure 6.2c with the same length and in Figure 6.2d with different lengths between their shared nodes.

We can detect all of these cases by checking the nodes of a path-tree. We count the amount of links that end in a node. If the count of a node is greater than one (> 1) then we have found a node where branches are reconnecting or crossing each other.

To correct invalid path-trees we need to remove links from this path-tree. More precisely, we need to remove links until the amount of links that end in a node is one or zero for every node. Additionally, all predecessor links have to be removed as well if they have no other successor link. This needs to be continued until the predecessor link is part of another branch to a destination again. Algorithm 6.3 is the corresponding pseudocode.

Algorithm 6.3 Code for resolving invalid path-trees

```
procedure correctInvalidPathTree(pathTree)
    seenNodes = [];
    newPathTreeDataStructure = [];
    for all link : pathTree do                    // Iterate through hops starting from the source
        if not link.first in seenNodes then
            continue;
        end if
        if link.second in seenNodes then
            continue;
        end if
        seenNodes.add(link.second);
        newPathTreeDataStructure.add(link);
    end for
    seenNodes = path-tree.destinations;
    for all link : pathTree do                    // Iterate through hops starting from the back
        if not link.second in seenNodes then
            newPathTreeDataStructure.remove(link);
        end if
        if not link.first in seenNodes then
            seenNodes.add(link.first);
        end if
    end for
    return newPathTreeDataStructure;
end procedure
```

6.3.2 Integration Approaches

In this section we present the integration approaches called *naive*, *rewrite*, and *end*. During these approaches, we can already look out for invalid path-trees and prevent them from occurring in the first place.

The first approach that is depicted in Algorithm 6.4 is called *naive* integration. In this approach, we naively insert all the links of the linear path without checking for parallel or shorter branches. It is important to note that this approach does not prevent invalid path-trees and therefor the properties of a tree are disobeyed. This violates our system model of a tree structure for the path-trees and downgrades the path-trees to a directed acyclic graph structure. As a result of that, potentially occurring frame duplicates also lead to violations of the zero queuing assumption. We still evaluate this integration approach to have a baseline comparison.

The second approach is called *rewrite* integration and tries to resolve the invalid path-trees after the *naive* insertion. This is done by adding the entire linear path, followed by optimizing this path-tree. Algorithm 6.5 contains the corresponding pseudocode. The *rewrite* approach removes parallel branches and searches for possible shorter branches in this path-tree.

Algorithm 6.4 Code for naive approach integration of a path into a path-tree

```
procedure INTEGRATE(pathTree, linearPath)
    for i = 0; i < linearPath.size(); i++ do
        pathTree[i].add(linearPath[i]);
    end for
    return pathTree;
end procedure
```

Algorithm 6.5 Code for rewrite approach integration of a path into a path-tree

```
procedure INTEGRATE(pathTree, linearPath)
    for i = 0; i < linearPath.size(); i++ do
        pathTree[i].add(linearPath[i]);
    end for
    pathTree = correctInvalidPathTree(pathTree);
    return pathTree;
end procedure
```

The third approach shown in Algorithm 6.6 is called *end* integration. This approach prevents invalid path-trees by the way it inserts the links instead of correcting the path-trees afterwards. Here, links are added starting from the end of the linear path. Adding the links is stopped as soon as we reach a node that is already contained in the path-tree. Integrating a path from the end does solve the problem of having two parallel branches, but it does not necessarily mean that this is now the shortest tree possible to create out of the given linear path links.

Algorithm 6.6 Code for end approach integration of a path into a path-tree

```
procedure INTEGRATE(pathTree, linearPath)
    for i = linearPath.size()-1; i >= 0; i– do
        if pathTree.contains(linearPath[i].firstNode) then
            firstAdd = i;
            hop = pathTree.findHop(linearPath[i].firstNode);
            break;
        end if
    end for
    for i = 0; i < (linearPath.size() - firstAdd); i++ do
        pathTree[hop + i].add(linearPath[firstAdd + i]);
    end for
    return pathTree;
end procedure
```

6.4 Path-tree Construction Methods

This section discusses different methods for constructing path-trees. We start with searching for a good path for each destination. They are called linear paths since each one is just a path to one destination. A path-tree is then built from the linear paths. Any integration approach can be used to build the path-trees. Similar to the KSP algorithms, we want to construct n_{path} candidate path-trees based on the input of the source node and destination nodes. Later, we empirically evaluate the construction methods against each other to verify the different emphasis on space costs, runtime costs, and quality of the routing of the multicast streams.

First, we discuss the destination order impact on the construction methods. Then, we present methods for constructing path-trees from precalculated linear candidate paths. Next, a method for constructing path-trees with a modified Dijkstra Overlap is presented. After that, we introduce an iterative method that does not precompute all linear candidate paths.

6.4.1 Destination Order During Construction

In this section, we take a look at the order in which the destinations are considered during a path-tree construction. The selection order is important because it influences the next steps of the construction methods and therefore the resulting path-trees.

It is possible to select the next destination randomly, based on the IDs, or the distance from the source. This selection process can vary independently of the rest of a path-tree construction method.

To increase randomness in the *random* destination order approach we create an additional variant that calculates a new random order before each candidate path-tree creation instead of reusing the same random order for all candidate path-tree constructions (*randomly per path-tree*). The ID based destination order approach can be *ascending* or *descending*.

The distance between the source and destination is the shortest path between the nodes. Depending on the method, this shortest path is already calculated or needs to be specifically calculated. This distance of two nodes is only based on the network. Any additional information of the construction method like link weight modifications are explicitly not taken into account. For the distance order, we can either select the *nearest* destination or the most *distant* destination next. We will later evaluate the impact of these six order selection approaches.

6.4.2 Constructing Path-trees from Linear Candidate Paths

This section describes different methods to construct path-trees from linear candidate paths. First, all of these methods generate a selection of linear paths once for each destination. The paths in these selections are called linear candidate paths. Then we select a linear candidate path for every destination and integrate them into a path-tree. We repeat this until we have n_{path} path-trees. The linear candidate paths are precomputed just one time since they are reused for all path-trees. The methods in this section differ from one another in their linear candidate path selection approaches. Algorithm 6.7 presents the high-level concept.

Algorithm 6.7 Abstract base concept of the linear candidate path methods

```
procedure FINDROUTES(src, dstOrder, n_linPath, n_path)
    candidatePathsPerDst = findCandidatePathsOneDst(src,dstOrder,n_linPath);
    candidatePathTrees = [];
    for i = 0; i < n_path; i++ do
        pathTree = PathTree();
        for j = 0; j < dstOrder.size(); j++ do
            selectedDst = next(dstOrder);
            selectedPath = selectLinearPath(candidatePathsPerDst[selectedDst], pathTree);
            pathTree = integrate(pathTree, selectedPath);
        end for
        candidatePathTrees.add(pathTree);
    end for
    return candidatePathTrees;
end procedure
```

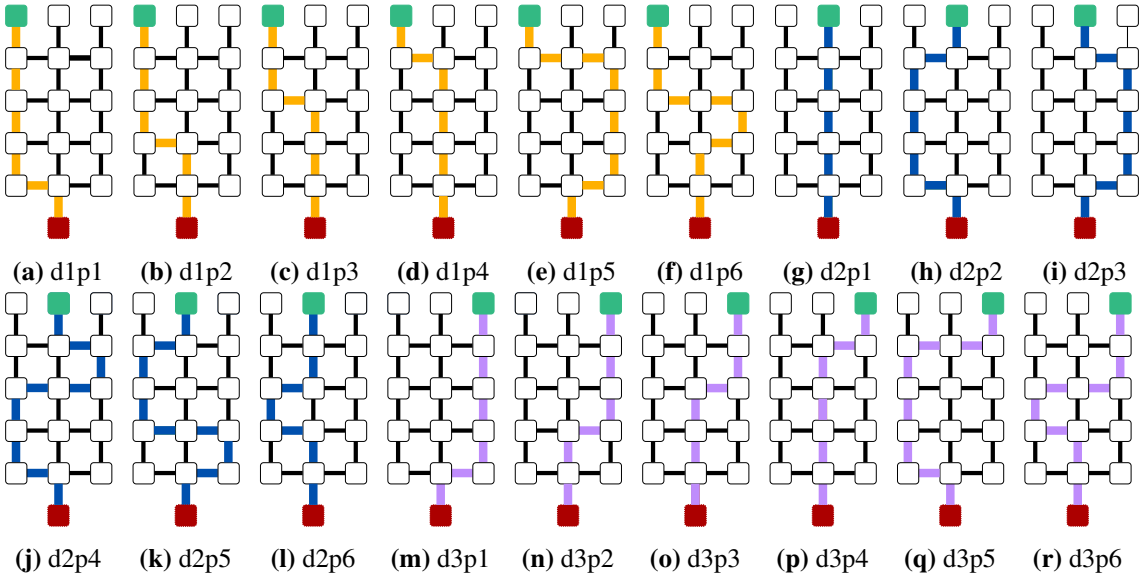


Figure 6.3: Example of selection of linear candidate paths from Dijkstra Overlap

The first step of the methods generates linear candidate paths for each destination. We precompute these $n_{linPath}$ linear candidate paths for each destination using the Dijkstra Overlap algorithm of Geppert et al. [GDBR23] because then the linear candidate paths are quite distinct from each other.

For clarity and visual understandability, we consider a continuous example for all methods. The example is based on a three by four grid network with three additional destination nodes at the short side and an extra source node on the other side. The destination order for all methods is considered to be from left to right. The example generates $n_{linPath} = 6$ linear candidate paths for each of the three destinations and $n_{path} = 6$ candidate path-trees. Figure 6.3 shows the resulting linear candidate paths of the Dijkstra Overlap for the example.

As second step, we select the next destination to work with according to the destination order. For the following linear candidate path selection step of the methods, different parameters can be considered. The selected linear candidate path is the path that optimizes a parameter the best. Such parameters can be the number of links that overlap with a path-tree or the number of parallel links of a resulting path-tree also described as the tree split positions. These two most important parameters are considered in the linear candidate path selection approaches that are described in the following subsections. Other potential parameters like the linear path length, the tree depth, or the number of links of a resulting path-tree are only indirectly considered.

As a special case, the selection of the first linear candidate path needs to be different. The reason for this is that the considered parameters of the methods depend on a path-tree being partially done already, but at this point in time this path-tree is still empty. After we integrated a linear path for the first destination, we can start using the parameters because this path-tree is no longer empty. Since we do not want just one shortest spanning path-tree but n_{path} different path-trees, we always select a different linear candidate path for the first destination. This ensures different starting configurations for the path-trees.

At last, the chosen linear candidate path is integrated into this path-tree. There are three different integration approaches called *naive*, *end*, and *rewrite* (cf. Section 6.3.2). Then we continue with the next destination.

Selection Approach: Index Based

The *index based* linear candidate path selection approach does not consider any parameters of the paths. It simply chooses the next unused linear candidate path by the index. For every destination, this leads to the i-th linear candidate path to be chosen for the i-th path-tree. When we start a new path-tree, we increase the index. Figure 6.4 presents the resulting candidate path-trees.

This *index based* selection approach clearly has no additional runtime or memory cost. Although the costs are low, there are disadvantages too. Selecting based on the index and not considering any characteristics of the linear paths and a path-tree leads to arbitrary trees. This means that we have no control over when a path-tree splits its paths and how long each path is.

Selection Approach: Least Links

In the *least links* linear candidate path selection approach, the overlap with a path-tree is considered. The goal is to minimize the number of links that do not overlap. As mentioned before, the path to the first destination is selected according to the path-tree index to ensure different starting configurations. For each next destination, the linear candidate path with the least number of links that are not already used in this path-tree is selected. This is done by separately merging the linear candidate path links into a set of the links of this path-tree and then counting the number of links. The linear path that creates the temporary set with the fewest links is selected. If a tie occurs the first one we found is selected. This correlates to the insertion order of the linear candidate path. We use sets for these overlap comparisons because sets naturally have no duplicates. Additionally, we do not want to consider any effects that the different integration approaches have on this comparison. For example the number of links in the path-tree after the integration can vary depending on the integration approach.

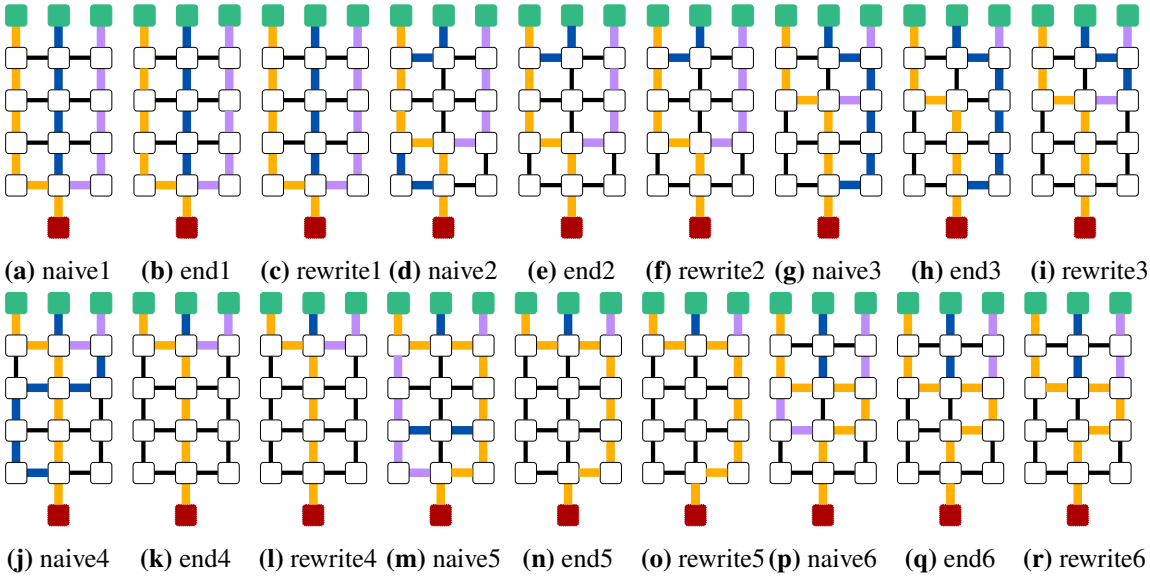


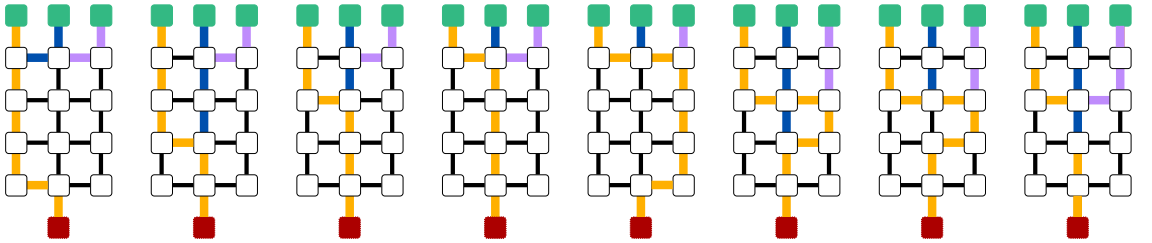
Figure 6.4: Examples of index based path-trees

Algorithm 6.8 Code for least links approach

```
procedure SELECTLINEARPATH(candidatePaths, pathTree)
    pathTreeSet = Set(pathTree);
    currentBestPath = candidatePaths[0];
    currentBestScore = Max;
    for all linearPath : candidatePaths do
        afterMerge = pathTreeSet.add(linearPath);
        if afterMerge.size() < currentBestScore then
            currentBestPath = linearPath;
            currentBestScore = afterMerge.size();
        end if
    end for
    return currentBestPath;
end procedure
```

The *least links* approach has an additional runtime cost per destination of the initial set generation and $n_{linPath}$ times a set copy and merge. For memory cost, this approach adds two sets and two integers. Algorithm 6.8 shows this approach and Figure 6.5 presents the resulting candidate path-trees. The resulting candidate path-trees of the *end* and *rewrite* integration approaches are just shown if they differ from the candidate path-tree of the *naive* integration approach.

Under the assumption that the first links are most likely to be used anyway, we can create a heuristic to reduce runtime that does not check the whole path, but only the last links per path.



(a) naive1 **(b)** naive2 **(c)** naive3 **(d)** naive4 **(e)** naive5 **(f)** naive6 **(g)** end6 **(h)** rewrite6

Figure 6.5: Examples of least links path-trees

Selection Approach: Similar Start

The *similar start* linear candidate path selection approach considers the number of parallel links of the resulting path-tree. The goal is that the path-trees branch as late as possible. As described previously, the path to the first destination is selected with the path-tree index to ensure different starting configurations. For every further destination, the linear candidate path is selected that has the most links at the start of the path that match with the path-tree. The approach is realized by iterating along the path-tree links starting with the source. We discard linear candidate paths if their next taken link does not match. Checking for matching links continues until only one or several similar linear paths remain.

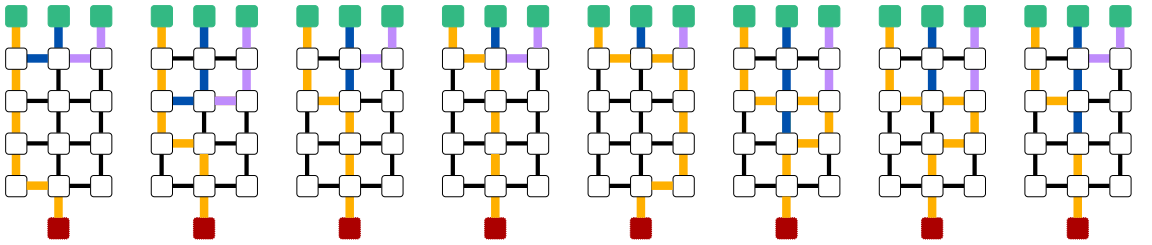
This approach has an extra runtime cost of $n_{linPath} \cdot pathLength \cdot \text{dstNo}(s)$ and a memory cost of two arrays containing $n_{linPath}$ integers. Algorithm 6.9 shows this approach and Figure 6.6 presents the resulting candidate path-trees. The resulting candidate path-trees of the *end* and *rewrite* integration approaches are just shown if they differ from the candidate path-tree of the *naive* integration approach.

We can create a heuristic to reduce runtime that does not check every consecutive link but only every 2^i-th link. The underlying assumption is that if a path follows a path-tree for a while we can skip links taken in between and still have matching links on the 2^i-th link. This heuristic has a similar result than checking every link. It is better if the 2^i-th link is the same than the 2^{i-1}-th link being the last matching link. A disadvantage of this heuristic is that it only takes effect if the linear paths are following a path-tree for a while. Additionally, this heuristic is prone to ignore detours between the 2^i-th link and the 2^{i-1}-th link.

The *similar start* approach, which considers the number of parallel links of the resulting path-tree, keeps the path-trees slim near the source. On the other hand, this approach prefers longer linear candidate paths because they follow a path-tree for the longest time. In the worst case, linear candidate paths are chosen that first travel to the farthest distant destination of this path-tree and then to their actual destination. We can address this problematic side effect by different alternatives. The first option is to only consider the linear candidate paths that are shorter than the path-tree depth plus some defined number of links. The second alternative is to avoid weird path-trees by stopping the path-tree traversal before the depths is reached and choosing the shortest path of the remaining ones. For example, we can stop at a quarter or at half of the depth. This assumes that having a similar start is good enough and having a more direct path is more important than keeping the linear paths together. In practice, this problem is solved indirectly by the number limit $n_{linPath}$, which ensures that the linear candidate paths are usable.

Algorithm 6.9 Code for similar start approach

```
procedure selectLinearPath(candidatePaths, pathTree)
    previousPaths = candidatePaths;
    stillpossiblePath = [];
    for all i = 0; i < pathTree.depth; i++ do
        for all path : previousPaths do
            if pathTree[i] == path[i] then
                stillpossiblePath.add(path);
            end if
        end for
        if stillpossiblePath.empty() then
            break;
        end if
        previousPaths = stillpossiblePath;
        stillpossiblePath = [];
    end for
    return previousPaths[0];
end procedure
```



(a) naive1 **(b)** naive2 **(c)** naive3 **(d)** naive4 **(e)** naive5 **(f)** naive6 **(g)** end6 **(h)** rewrite6

Figure 6.6: Examples of similar start path-trees

6.4.3 Constructing Path-trees with Modified Dijkstra Overlap

This section describes a method that generates path-trees by extending the principle of the Dijkstra Overlap for trees. In the Dijkstra Overlap, found linear paths influence the link weights. Similar to that, the modified Dijkstra Overlap uses the path-trees to influence the link weights to achieve diversity between the candidate path-trees. This method has no precomputed linear candidate paths.

First, the method chooses the next destination. Then, based on adjusted link weights, the method searches for the shortest weighted linear path to the selected destination. Next, we integrate the linear path into the path-tree, adjust additional temporal link weights, and proceed with the next destination. When all destinations are integrated, the link weights of this path-tree are readjusted and then the method continues with the next path-tree. The pseudocode is presented in Algorithm 6.10 and Figure 6.7 is an example showing the path-trees after each linear path integration using simplified link weight values (÷100).

For every path-tree we construct, the already generated candidate path-trees for this stream are considered to be known. Links of those candidate path-trees gain weight to be less appealing. The next path-tree we construct selects links with low weights and therefore prefers links that the existing path-trees do not use. This achieves diversity between the candidate path-trees.

The temporal link weights are separated from the link weights because they only temporarily reduce the link weights during the linear path search. This influences these links to be more appealing during a single path-tree construction. We desire the construction of multicast path-trees to prefer links that already exist in this path-tree since we want to minimize the amount of links in each path-tree. Strictly speaking, we want to have low diversity inside the path-trees. The temporal link weights are not part of the considerations during the later link weight adjustments of the candidate path-trees.

Before the first path-tree is constructed, we initialize all link weights with 100 and all temporal link weights with zero. For each path-tree construction, we first select the next destination based on the selection order.

In the next step, we use Dijkstra's algorithm with link weights to find the linear path with the lowest weight sum to this destination. The total link weight for this search is the sum of the link weight and the temporal link weight. This makes the total link weights the parameter in the search for the shortest weighted linear path.

The chosen linear path is integrated into the path-tree we currently construct, using one of the integration approaches (*naive*, *end*, and *rewrite*). The method then adjusts the temporal link weights of the chosen linear path. The temporal link weights are reduced by 50 to make them more appealing. This has the effect that it costs less if the link is already used in the path-tree we want to construct. Using the temporal link weights influences the path finding algorithm to prefer a high overlap of links between the linear paths and the path-tree. One option is to not adjust all links of the linear path (*whole*), but just the links of the linear path that are not yet part of the path-tree (*added*). Another option (*zero*) is to not reduce the temporal link weights, but to set them to the negative value of the link weight, so that the total link weight of the path-tree links is zero.

At the end of each path-tree construction, after we iterated over all destinations, we adjust the link weights. For each link in the finished path-tree, the weight is increased by 200. This has the effect, that links which are already used in another path-tree are less appealing for future path-trees because they cost more. In addition, the temporal link weights are reset to zero.

This method has an extra runtime cost of $dstNo(s) \cdot (pathLength + pathTree.depth)$ times a link weight adjusts per path-tree construction as well as $dstNo(s) \cdot n_{path}$ Dijkstra's algorithm runs and a memory cost of two link weight maps.

We can create a heuristic to reduce complexity and the influence of the destination order. This can be realized by calculating the path-trees with a shortest spanning tree algorithm that uses weighted links. After the calculation of a path-tree, we increase the link weights of the links used in this path-tree. This heuristic ignores the potential impact of any destination order considerations. The runtime also benefits if the used shortest spanning tree algorithm is faster than $dstNo(s)$ times a Dijkstra search. We call this heuristic *ignore* since the temporal link weights are unused and always stay at zero.

Algorithm 6.10 Code for modified Dijkstra Overlap method

```
procedure findRoutes(src, dstOrder, n_linPath, n_path)
    candidatePathTrees = [];
    weights = [all links = 100];
    temporalWeights = [all links = 0];
    for i = 0; i < n_path; i++ do
        pathTree = PathTree();
        for j = 0; j < dstOrder.size(); j++ do
            selectedDst = next(dstOrder);
            linearPath = Dijkstra(src, selectedDst, weights + temporalWeights);
            for all link : linearPath do
                temporalWeights[link] = temporalWeights[link] - 50;
            end for
            pathTree = integrate(pathTree, linearPath);
        end for
        for all link : pathTree do
            weights[link] = weights[link] + 200;
            temporalWeights[link] = 0;
        end for
        candidatePathTrees.add(pathTree);
    end for
    return candidatePathTrees;
end procedure
```

Furthermore, we want to note that this method can also be applied to graphs that already use link weights. In this case, initializing the link weights to 100 is skipped. Additionally, the increase and decrease of the link weights and temporal link weights need to be in the same scale as the link weights that are already used in the graph to ensure the desired amount of impact.

6.4.4 Constructing Path-trees Based on Intermediate Node Distance

This section describes a method that finds a linear path from any intermediate node to the destinations. The general idea is to add the fewest links possible to reach a destination. Here, we do not need to precompute all linear candidate paths, nor do we consider any link weights.

The method that is presented in Algorithm 6.11 executes the following steps for each path-tree. First, the next destination is chosen based on the selection order. The method then finds the shortest linear path from each intermediate node to the chosen destination. Intermediate nodes are considered to be all nodes in the path-tree so far. From this set of linear paths, we select the shortest. Next, we integrate the selected linear path into the path-tree. The method then proceeds with the next destination. Figure 6.8 shows an example of the path-trees after each linear path integration and the distance to the chosen destination in each intermediate node.

Next, we want to describe the step in which the method considers the shortest paths from any intermediate node to the chosen destination in more detail. The parameter that describes the length of a path from one node in a path-tree to a destination is used. We will select the shortest of these

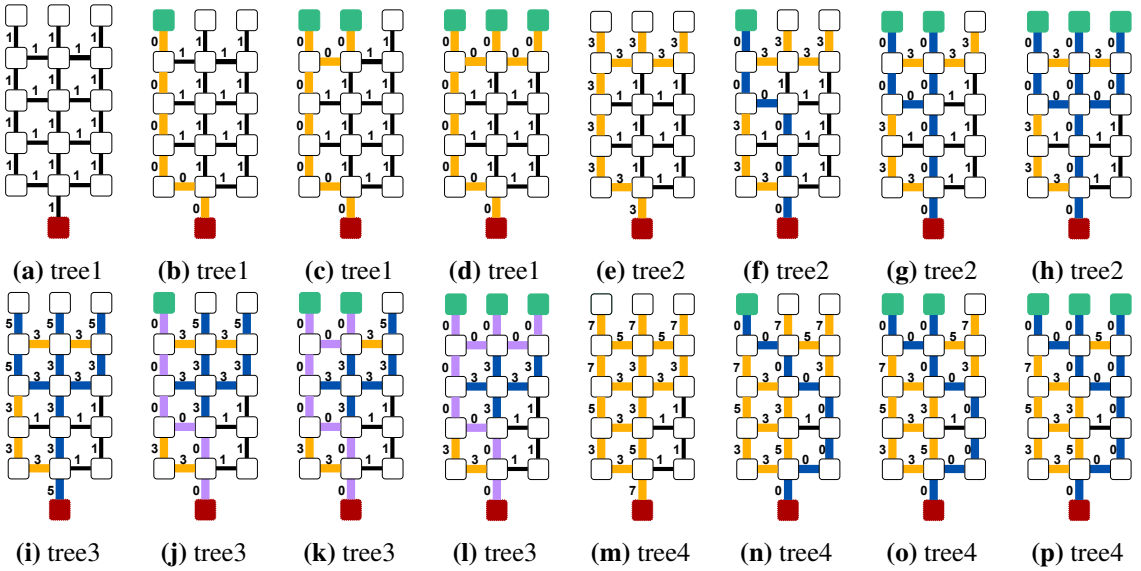


Figure 6.7: Examples of modified Dijkstra Overlap path-trees

linear paths. This is realized by calculating the shortest path from the first intermediate node of a path-tree to the destination and remembering the current optimal path length. Now, for every other intermediate node, we iteratively repeat the following steps. First, we calculate the shortest path from this node to the destination. Then we compare the linear path length to the current optimal one. If the new linear path is shorter, we replace the current optimal path. If we iterated over all intermediate nodes, we found the shortest linear path from this path-tree to the chosen destination. The distance from the source to the intermediate node of the shortest linear path is not considered.

As a special case, the linear path for the first destination needs to be found differently. This is because we want n_{path} different path-trees, we always select a different linear path for the first destination. This ensures different starting configurations for the path-trees. A solution that does not require a special case for the first destination is to use a different random destination order for each path-tree. After we integrated a linear path for the first destination, the normal intermediate node distance procedure is starting.

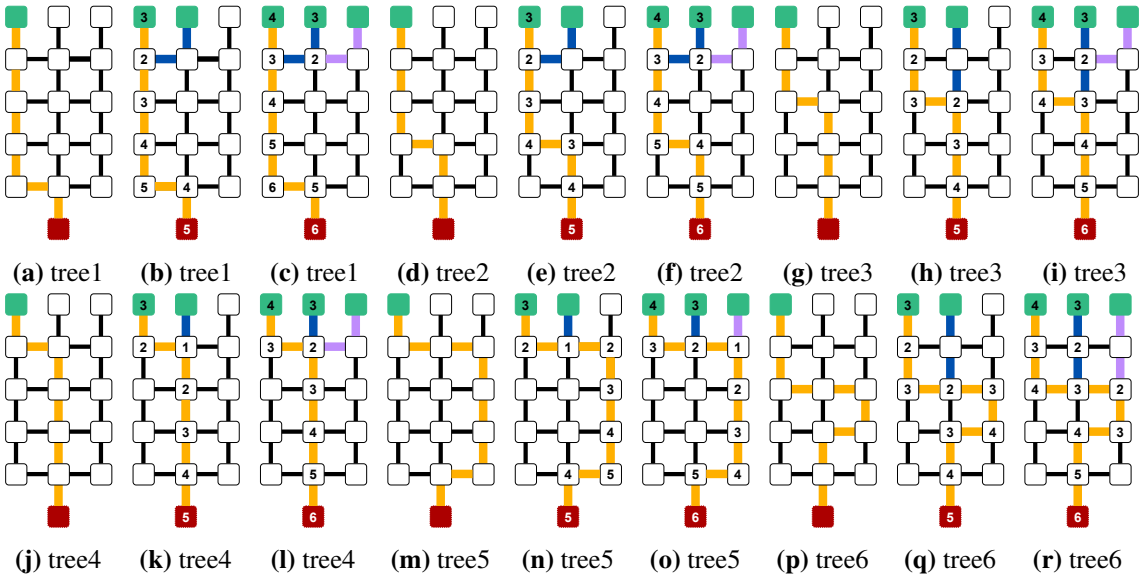
Now we take a look at how the method integrates the chosen linear path into a path-tree. In this method, the tree approaches called *naive*, *end*, and *rewrite* generate all the same result. This is because the linear path the method found has no overlap with the path-tree we construct. The integration is always realized similar to the *end* approach. Here, links are added starting from the end of the linear path. Adding the links is finished by the end of the linear path. This is exactly when we reach a node that is already contained in the path-tree.

This method has an extra runtime cost of $pathTree.depth \cdot dstNo(s)^2$ times a path finding algorithm and a memory cost of remembering two paths.

To speed up this process, we can use an algorithm solving the single destination shortest path algorithm. Such a spanning tree is calculated starting from the destination. The algorithm holds as soon as we reach one of the path-trees intermediate nodes.

Algorithm 6.11 Code for path-trees based on intermediate node distance method

```
procedure FINDROUTES(src, dstOrder, n_linPath, n_path)
    candidatePathTrees = [];
    for i = 0; i < n_path; i++ do
        pathTree = PathTree();
        for j = 0; j < dstOrder.size(); j++ do
            selectedDst = next(dstOrder);
            currentBestPath = findRoute(src, selectedDst);
            for all intermediateNode : pathTree do
                curentPath = findRoute(intermediateNode, selectedDst);
                if curentPath.size() < currentBestPath.size() then
                    currentBestPath = curentPath;
                end if
            end for
            pathTree = integrate(pathTree, currentBestPath);
        end for
        candidatePathTrees.add(pathTree);
    end for
    return candidatePathTrees;
end procedure
```



(a) tree1 **(b)** tree1 **(c)** tree1 **(d)** tree2 **(e)** tree2 **(f)** tree2 **(g)** tree3 **(h)** tree3 **(i)** tree3

(j) tree4 **(k)** tree4 **(l)** tree4 **(m)** tree5 **(n)** tree5 **(o)** tree5 **(p)** tree6 **(q)** tree6 **(r)** tree6

Figure 6.8: Examples of intermediate node distance path-trees

7 GFH Adaptions

This chapter describes the considerations that are made regarding the solver. We modify the optimization function to account for the amount of destinations. First, we briefly explain the high level concept of the GFH by Falk et al. [FGD+22]. Next, we explain our changes to consider multicasts.

Let us first recap how the GFH solver of Falk et al. [FGD+22] works on a high level. By mapping the joint routing and scheduling problem to a graph problem, the solver can solve an independent colorful set problem. For this purpose, the solver looks at a conflict graph containing multiple configurations for each stream. Each configuration contains a route and a phase that represent a possible solution on how to route and schedule the corresponding stream. The solver input contains the conflict graph, all the streams that are in the current traffic plan (ActiveS(p_i)), and all streams that are requested to be added to the traffic plan (ReqS(p_i)). The streams that are in ActiveS(p_i) are rescheduled first to find more optimal solutions that arise from the gaps of removed streams. For each of these two stream collections, a heap is generated to have a processing order for them. A stream is placed higher in the heap according to a lower eligible count, meaning that the stream has fewer schedulability options left. The eligible count represents how many configurations of a stream are not shadowed. A node shadows its neighbors in the conflict graph if its configuration is in the traffic plan. The first tiebreaker is a higher total degree, meaning the stream has more neighbors over all configurations together. If the total degree is equal, the configuration ID decides what is considered first. For each stream, the configuration that has the lowest shadow score is chosen for the traffic plan. The shadow score calculates how many configurations of each other stream are shadowed percentage-wise by this configuration and sums it up over all streams.

We generally want to leave the solver as it is, since finding streams that are easy to schedule next is more important than making sure the scheduled destination amount is higher. So the weight function $w(s) = dstNo(s)$ is considered in the stream scheduling order after the other established ordering parameters. For that reason, we insert the destination amount in front of the ID tiebreaker. This approach leads to the destination amount being another tiebreaker in the scheduling order. The tiebreaker prefers to pick streams with higher destination counts first. Algorithm 7.1 presents the resulting comparator function for the heap. This approach does not change the general functionality of the solver but still respects the weight function during the schedule. It does not add a significant bias to GFH.

We also consider a more aggressive alternative by moving the destination amount in front of the eligible count and the total degree of the streams. Then streams with a high destination amount are picked first, even if they have a higher eligible count. Algorithm 7.2 shows the comparator function for this aggressive alternative. This changes the fairness of the solver to prefer a high destination amount.

Algorithm 7.1 Comparator code for GFH heap sort with destination tiebreaker

```
procedure COMPARE(stream1, stream2)
    if stream1.eligibleCount() != stream2.eligibleCount() then
        return stream1.eligibleCount() > stream2.eligibleCount();
    end if
    if stream1.totalDegree() != stream2.totalDegree() then
        return stream1.totalDegree() < stream2.totalDegree();
    end if
    if dstNo(stream1) != dstNo(stream2) then
        return dstNo(stream1) < dstNo(stream2);
    end if
    return stream1.ID < stream2.ID;
end procedure
```

Algorithm 7.2 Comparator code for GFH heap sort aggressive destination alternative

```
procedure COMPARE(stream1, stream2)
    if dstNo(stream1) != dstNo(stream2) then
        return dstNo(stream1) < dstNo(stream2);
    end if
    if stream1.eligibleCount() != stream2.eligibleCount() then
        return stream1.eligibleCount() > stream2.eligibleCount();
    end if
    if stream1.totalDegree() != stream2.totalDegree() then
        return stream1.totalDegree() < stream2.totalDegree();
    end if
    return stream1.ID < stream2.ID;
end procedure
```

With this aggressive alternative, we want to absorb some natural disadvantages of streams with many destinations. Streams that have a high amount of destinations are harder to integrate into the traffic plan since their path-tree is larger. Therefore, there are more spatial and temporal conflicts with other streams. Scheduling these streams first can lead to them being integrated instead of being rejected since there are not as many other already selected configurations yet. Unicast streams are easier to schedule later on compared to large multicasts. This aggressive alternative increases the average amount of destinations per stream in the traffic plan.

The aggressive alternative is not as far from GFH as it seems. If a stream has many destinations then it also has many links in the path-tree. This leads to more spatial and temporal conflicts and therefore the configurations of this stream have many edges in the conflict graph. With more edges in the conflict graph, it is more likely that a configuration is shadowed by one of its neighbors and in consequence reduces the eligible count of the stream. Also, the total degree is higher for routes with many destinations.

8 Evaluation

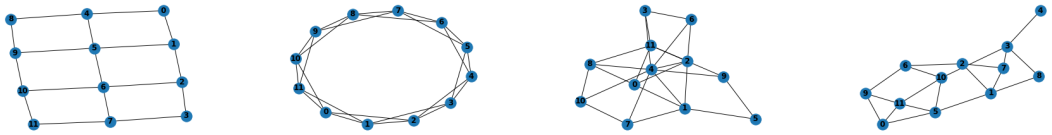
In this chapter, we empirically evaluate our work. To generate traffic plans that solve the routing and scheduling problem, we execute a C++ program. This planner program solves our problem for multiple time steps i based on a sequence of stream requests $\mathrm{ReqS}(p_i)$ specified in a scenario. First, we describe our evaluation setup and then the metrics we use for the evaluation. Second, we present the evaluation results for different path-tree construction methods. Next, we explain the evaluation results of different input parameters. We also compare the different GFH variants. Then we evaluate some static scenarios. Afterwards, we analyze domain-specific differences. Finally, we compare multiple unicasts to multicasts and explain the meaning of the results. At last, we discuss our evaluation results and compare them to related work.

8.1 Setup

This section specifies the setup that is used for our evaluation. The evaluation environment is running on a server with Ubuntu 22.04.2 LTS and a 5.15.0 kernel. It contains two AMD EPYC 7413 24-Core Processors as CPUs and 257 GB of RAM. This is a more powerful computer than what the planner program needs so that we are able to run seven independent evaluation environments in parallel on this one computer without using the full CPU capacity. Even with seven evaluation environments we do not use more than 20 GB of the RAM at any point in time. We use a logger in the planner program to collect data that is relevant for the evaluation. The logger contains a clock to keep track of how much time the planner needs. This clock is implemented with `std::chrono::high_resolution_clock;`. It is precise enough for our needs.

We execute the planner program four times with the same input parameters to reduce the variance of the evaluations. The twelve input parameters are the network, the scenario, the linear path routing method, the path-tree routing method, the conflict graph builder, the maximum number of configurations per stream n_{cps}, the maximum number of created linear candidate paths $n_{linPath}$, the maximum number of created candidate path-trees n_{path}, the solver heuristic, the conflict graph expansion method, the number of used threads, and output specifications. Next, we describe each parameter in detail.

We want to ensure that our methods generally work independently of the network's structure. Therefore, we evaluate with different network topologies. We evaluate networks containing $n = 81$ bridges. The links between the bridges and the end devices are abstracted here by setting the start and the destinations directly on bridges. We use networks with different topologies for the bridges, specifically grid, ring (n, k), Erdős-Rényi, and Waxman. Figure 8.1 displays the topologies of these networks. For the grids, we use every combination of two numbers that multiply to n as width and height except one and n itself. In the ring (n, k) we evaluate every k between one and four. In such a ring (n, k) every node has links to its k neighbors in every direction of the ring.



(a) grid 3x4 **(b)** ring (12,2) **(c)** Erdős-Rényi **(d)** Waxman

Figure 8.1: Network topology examples $n = 12$

network name	topology	nodes	links	diameter
grid27,3	grid width = 27, height = 3	81	132	28
grid9,9	grid width = 9, height = 9	81	144	16
ring81,1	ring (81,1)	81	81	40
ring81,2	ring (81,2)	81	162	20
ring81,3	ring (81,3)	81	243	14
ring81,4	ring (81,4)	81	324	10
erdos81,3	erdos	81	243	5
erdos81,4	erdos	81	324	4
waxman81,1	waxman	81	192	7
waxman81,2	waxman	81	392	4
waxman81,5	waxman	81	843	3

Table 8.1: Used network topology sizes

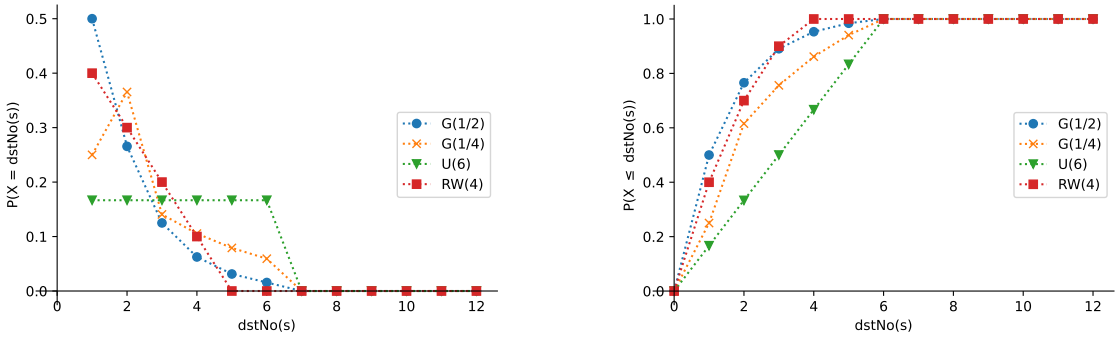
For Erdős-Rényi, and Waxman we generate multiple random networks and pick some that have a comparable amount of edge to the ring (n, k) and the grid. We make sure that each chosen network is a strongly connected graph. Table 8.1 lists the networks we ended up with. The networks are created with the python library networkX [HSS08]. In all tests, all network nodes have a processing delay of 4 µs, a propagation delay of 1 µs, a queuing delay of 0 µs, and a bandwidth of 1 Gbit s^{-1}. These delays are adopted from [FGD+22]. This leads to a transmission delay that is just depending on the frame size. Table 8.2 presents the resulting delays.

To make sure that our methods work independently of the scenario, we evaluate with multiple different scenarios. To be able to evaluate against multiple unicast scenarios we generate each scenario with only unicast streams (*only_u*), with multiple unicast stream sets (*multiple_u*), with all multicast streams (*only_m*), and with a combination of multicast and unicast streams (*some_m*). Only unicast streams is the special case of multicast where every stream has just one destination. The scenarios with *multiple_u* and *only_m* reach the same destinations except that *multiple_u* does that with a set of unicasts instead of a multicast. For most evaluations, we only use scenarios with a combination of multicast and unicast streams (*some_m*).

The probability of $dstNo(s)$ for the stream s in the multicast and unicast stream scenarios varies. We choose four different probability distributions. For each $dstNo(s)$ probability distribution we generate three scenarios. First, we use a geometric probability distribution $G(p) = P(X = dstNo(s)) = (1-p)^{dstNo(s)-1} \cdot p$ with $p = 0.5$ and $p = 0.25$ for $dstNo(s) \leq 6$ with the remaining probability added to $X = dstNo(s) = 2$. The expected value of a geometric probability distribution is

variable	value
t_0	$0 \mathrel{\widehat{=}}$ program start
t_{src}	4 µs
t_{dst}	4 µs
t_{proc}	4 µs
t_{prop}	1 µs
t_{que}	0 µs
B	bandwidth = 1 Gbit s^{-1}
t_{trans}	$\lvert frame\ size \rvert \cdot$ 8 ns
t_{perhop}	5 µs $+t_{trans}$
$t_{dst}(h)$	$t_{perhop} \cdot (h+1)$
$t_{e2e}(dst)$	$(5\ \mu s + t_{trans}) \cdot (h+1) + 4\ \mu s$

Table 8.2: Resulting delays



(a) probability distribution

(b) cumulative distribution function

Figure 8.2: Destinations probability distribution

$E(X) = \frac{1}{p}$ and with our cutoff after six this leads to $E(X) = 1.40625$ for $p = 0.5$ and $E(X) = 2.20117$ for $p = 0.25$. Next, we use a uniform probability distribution $U(6) = P(X = dstNo(s)) = \frac{1}{6}$ for $dstNo(s) \leq 6$ else zero. $E(X) = \frac{(1+2+3+4+5+6)}{6} = 3.5$ is the expected value of a uniform probability distribution. At last, we adopt a probability distribution from Schweissguth et al. [STP+20]. They evaluate with a probability distribution of $RW(4) = P(X = dstNo(s)) = \frac{1}{10} * (5 - dstNo(s))$ for $dstNo(s) \leq 4$ else zero. The expected value of this probability distribution is $E(X) = 1 \cdot \frac{4}{10} + 2 \cdot \frac{3}{10} + 3 \cdot \frac{2}{10} + 4 \cdot \frac{1}{10} = 2$. Figure 8.2 displays the different probability distributions. We also include scenarios that use a geometric probability distribution and a uniform probability distribution containing multicasts up to $dstNo(s) = 20$. The expected value of those probability distributions are $E(X) = 1.49998$, $E(X) = 3.55523$, and $E(X) = \frac{(20+1)}{2} = 10.5$. The frame size is chosen randomly per stream out of (125, 375, 625, 1500) in Bytes. The stream period is chosen randomly per stream out of (250, 500, 1000, 2000) in µs. Each scenario has 30 time steps i where the solver generates a traffic plan. We initially add 100 streams and then for every following time step, we remove $|\text{RemS}(p_i)| = 25$ random streams and request to add $|\text{ReqS}(p_i)| = 50$ new streams.

We run the program in parallel with up to 6 threads. To parallelize the Threading Building Blocks library (intel oneAPI TBB [dev23]) is used.

The linear path routing method specifies the algorithm used to find $n_{linPath}$ different paths to one destination. As stated in Section 6.4.2 we consider the Dijkstra Overlap algorithm. We chose $n_{linPath} = 6$ for the maximum number of created linear candidate paths.

The path-tree routing method specifies the algorithm used to find n_{path} different paths to all destinations of a stream. This correlates to the different path-tree construction methods we introduced. We vary the maximum number of created candidate path-trees n_{path} to see if it makes a difference for the schedule. We choose the possible parameters out of the range from two to six.

Falk et al. [FGD+22] provided us with an updated version of the code that contains multiple conflict graph building options. We use their random and classic versions. The classic version works as described in [FGD+22] where the configurations are chosen one candidate path after the other with the same phase and after all these candidate routes are chosen shifting the phase multiple time steps forward. In contrast to that, the random version chooses the next phase at random to avoid similar overlap cases. In contrast to Falk et al. [FGD+22], we use a conflict graph that is static and therefore the number of configurations per stream does not expand over time. This means that we generate all configurations when we add a stream to the conflict graph. The expansion of the configurations was removed to keep it more deterministic.

For the heuristic of the solver, we consider the GFH heuristic of [FGD+22] with the modification of the additional tiebreaker with the $dstNo(s)$. We separately make a comparison evaluation to the aggressive GFH alternative that uses the $dstNo(s)$ as first decision.

The optimal maximum number of configurations per stream n_{cps} is evaluated as well. For n_{cps} we used (12,24,36,48,60,72,84,96,108,120) configurations per stream in the conflict graph.

8.2 Evaluation Metrics

In this section we present the six main metrics we use for our empirical evaluation. These metrics are called *total runtime per time step*, *number of edges in the conflict graph*, *amount of rejected streams*, *number of destinations of scheduled streams*, *deviation between scheduled and requested number of destinations per stream*, and *overlap*. The evaluation results are plotted and assessed using all of these metrics. All plots showing line plots display the mean of the metric unless stated otherwise. The lines are extended with an error band containing the confidence interval of 95%.

The *total runtime per time step* describes how long the program needs to provide the next traffic plan p_i in the time step i. This time includes the time it needs to add new streams, extend the conflict graph with configurations of the new streams, solve the independent colorful set problem, remove rejected streams, collect data for the logger, and some clean up. The runtime per time step is calculated by summing up all these logged times per time step. The runtime per time step is displayed in seconds. We evaluate this metric to see if the adapted planner program is still fast enough. A low runtime per time step is the desired outcome.

Another interesting metric is the size of the conflict graph the planer program generates. Especially the *number of edges in the conflict graph* is interesting for us because this metric correlates to the difficulty to find an independent colorful set. The number of edges in the conflict graph is directly logged for each time step. We want the number of edges in the conflict graph to stay as low as possible.

The planner tries to integrate all streams, but that is not always possible. Therefore, it is very important to look at the *amount of rejected streams* $|\text{RejS}(p_i)|$ in each time step i. Ideally this number would stay at zero. This is the case if all streams can be scheduled in the traffic plan. In the worst case all streams would end up being rejected. Then the amount of rejected streams matches the amount of requested streams $|\text{RejS}(p_i)| = |\text{ReqS}(p_{i-1})|$.

Since we route and schedule multicasts with different amounts of destinations it is also interesting to see the average number of destinations the scheduled streams contain. The amount of scheduled destinations $|\text{ActiveDst}(p_i)|$ is calculated by our logger at runtime. The average *number of destinations of scheduled streams* is calculated out of the amount of scheduled destinations and the amount of scheduled streams $E' = \frac{|\text{ActiveDst}(p_i)|}{|\text{ActiveS}(p_i)|}$. This metric E' represents the expected value of $dstNo(s)$ for a scheduled stream in the traffic plan. A high E' value means that the multicasts with multiple destinations are part of the traffic plan.

This metric can also be displayed relative to the expected destination amount per stream. Then we can see the *deviation between scheduled and requested number of destinations per stream*. We calculate this destination per stream deviation with $\Delta E = E' - E(X)$. Ideally the solver can handle streams independently of the destination amount. Then this metric would stay at zero. If streams with a higher destination amount are rejected more frequently, this metric would become negative. The metric is positive if streams with fewer destinations than the expected amount of destinations per stream are rejected more often.

Since we especially care for the routes of the streams we also want to take a look at those. Therefore, we calculate the *overlap* between the candidate path-trees with $\frac{2 \cdot |\text{overlapping links(path-tree1,path-tree2)}|}{|\text{path-tree1}|+|\text{path-tree2}|}$ and save the minimum, maximum, and median overlap for each stream. For each time step we log the median over all streams in the traffic plan of the median overlap of the candidate routes. This provides us with a rough estimation about the general difference of the candidate path-trees. The metric is designed to be interpreted as a percentage between zero and one. If we have zero then the routes do not overlap and if the metric is one they are all the same route.

8.3 Results for Different Path-tree Construction Methods

In this section, we compare the different path-tree construction method heuristics of the candidate path selection methods, the modified Dijkstra Overlap methods, and the intermediate node distance methods to each other. Afterwards, we take a look at the different selection order approaches and the linear path integration. In Table 8.3 we provide the evaluation parameters for test 01 to 03. Test 01 evaluates the candidate path selection methods, test 02 the modified Dijkstra Overlap methods, and test 03 the intermediate node distance methods.

parameter name	Test 01	Test 02	Test 03
network size n	81		
network topologies	all		
scenario type	some_m		
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6, $G(\frac{1}{4})$ until 6, $U(6)$, $RW(4)$		
number of time steps	30		
initial streams	100		
$\lvert\text{ReqS}(p_i)\rvert$	50		
$\lvert\text{RemS}(p_i)\rvert$	25		
number of used threads	6		
linear path routing method	Dijkstra Overlap		
$n_{linPath}$	6		
candidate paths	all versions	—	—
modified Dijkstra Overlap	—	all versions	—
intermediate nodes	—	—	all versions
path-tree destination order	all versions		
path-tree integration type	end, rewrite, naive		
n_{path}	4		
conflict graph builder	random and classic		
solver heuristic	GFH-dst-tiebreaker		
conflict graph expansion	static		
n_{cps}	36		

Table 8.3: Test parameter table for construction methods

8.3.1 Comparison of Candidate Path Construction Method Heuristics

Now, we compare the heuristics of the candidate path selection method with test 01. Figure 8.3 depicts the corresponding results.

Figure 8.3a displays the total runtime per step. The *index based* approach performs the worst with a runtime per step of 1.02 s after 30 time steps. The *similar start base* approach has a lower runtime with up to 0.96 s per time step. The *similar start* heuristics do not show any significant runtime improvements. For the *least links base* approach we can report that the runtime is slightly below the *similar start base* approach (0.92 s after 30 time steps). The *least links* heuristic and the *similar start* heuristics make no considerable difference in runtime because the paths are too short.

Next, we present Figure 8.3b with the amount of edges in the conflict graph. The *index based* approach contains the most with up to 2,771,055 edges. The *similar start base* approach is in the middle again with 2,672,000 edges and the *least links base* approach performs the best with 2,531,121 edges after 30 time steps.

The amount of rejected streams is shown in Figure 8.3c below. Consistently the most are rejected in the *index based* approach with up to 14.63 streams. With the *similar start base* approach, we reject up to 13.49 streams. This is 0.52 less than in the *least links base* approach that rejects 14.01 streams after 30 time steps. The handling of fewer links in the *least links end* heuristic leads to bad linear path being selected and therefore a higher rejection of 14.46 after 30 time steps. The *similar start depth cut* heuristic does not reach the low rejection result of the other *similar start* heuristics. With 14.24 rejected streams it presents similar results as the *least links base* approach.

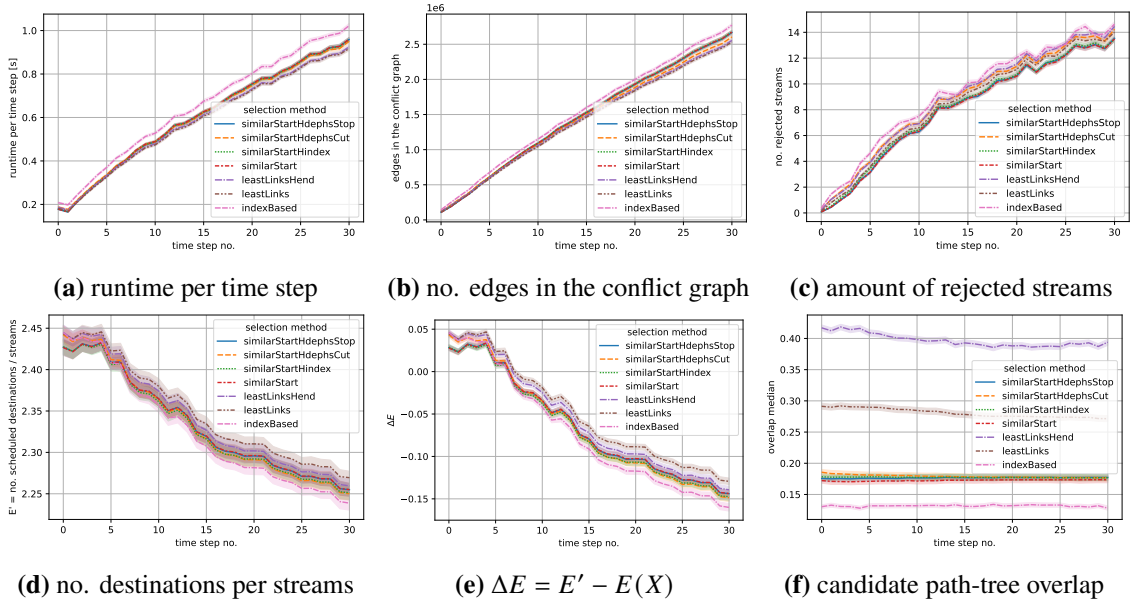
Figure 8.3d displays the number of destinations per stream. The *index based* approach starts with a higher destination amount per stream compared to the *similar start base* approach. It starts with 2.44 destinations in time step zero but then drops significantly faster down to 2.38 in time step seven. By the end, it reached 2.24 after 30 time steps. The *similar start* approaches end up in the middle between the *index based* approach and the *least links* approaches. They start off with 2.43 and manage to limit the reduction so that they only drop down to about 2.26 after 30 time steps. The *least links base* approach has the best results and ends up with 2.27 destinations per stream after 30 time steps. Since the *least links base* approach manages to keep the amount of destinations per stream very high the selection regarding which streams to reject seems to be more balanced compared to the other approaches.

Regarding the path-tree overlap Figure 8.3f depicts the differences between the candidate path selection method heuristics. The *index based* approach has despite the bad rejection only 0.13 as overlap. This is caused by the Dijkstra Overlap that produces linear candidate path with small overlap. Since the *index based* approach does not use one candidate path twice, the low overlap is transferred to the path-trees. The *similar start* approaches manage to keep the overlap around 0.17 quite small. The highest overlap in the *least links base* approach ends up with 0.27 and 0.39 for the *least links* heuristic.

The *index based* approach performs the worst over the majority of the metrics. These results can be explained with the lack of control over the resulting candidate path-trees. In the *index based* approach, the path-trees are not built to use few links. This causes the path-trees to be very broad. The additional links cause more spatial collisions and therefore more time is needed to check for temporal separation. Between these broad path-trees there also occur more temporal collisions. If we have more spatial and temporal collisions, it becomes harder to accept all streams.

The *similar start* approaches manage to construct stable path-trees which fit neatly between other streams because the starts of the path-trees behave like one linear path for quite a while. Unfortunately, this characteristic is not as scalable regarding the destination amount. The streams that need to be rejected are more often streams with higher destination counts compared to the *least links base* approach. For the *similar start* heuristics, we conclude that the path-trees are too short to lead to any significant differences compared to the *similar start base* approach.

We conclude that the *similar start* and *least links* approaches perform better than the *index based* approach. Additionally, we see that the *similar start* and *least links* approach heuristics are worse than the base approaches themselves.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.3: Candidate path selection path-tree construction methods test 01

8.3.2 Comparison of Modified Dijkstra Overlap Construction Method Heuristics

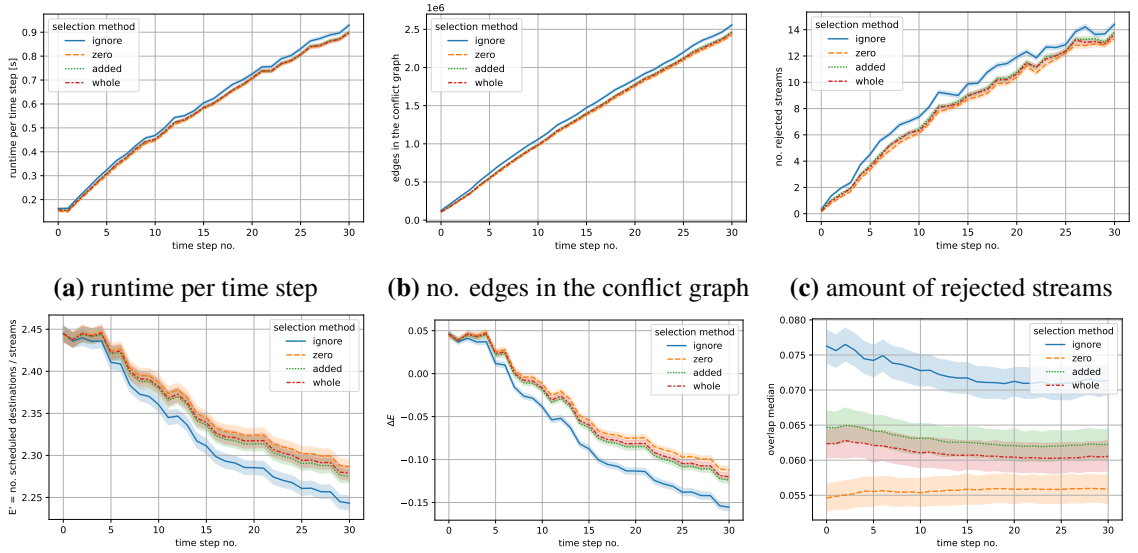
Next are the modified Dijkstra Overlap approaches from test 02. Figure 8.4 displays the corresponding results.

We start with Figure 8.4a that shows the total runtime per time step. The *ignore* heuristic takes slightly longer than the other heuristics with 0.929 s after 30 time steps. The *zero* heuristic has a runtime of up to 0.897 s while the *whole* heuristic has almost the same runtime with 0.898 s and the *added* heuristic also takes 0.902 s per time step.

The second Figure 8.4b shows how many edges the heuristics have in the conflict graph. The modified Dijkstra Overlap *ignore* heuristic shows again the highest result with up to 2,556,807 edges. With 2,465,058 edges the *added* heuristic comes next and the *whole* heuristic needs 2,454,004 edges. The *zero* heuristic performs the best with 2,424,653 edges after 30 time steps.

Figure 8.4c shows the amount of rejected streams. The *ignore* heuristic rejects the most streams with 6.76 rejected streams in time step eight, 11.92 streams in time step 20, and finally 14.42 streams after 30 time steps. The difference of the *added*, and *whole* heuristics in the temporal link weight is negligible small that it makes no difference. They both manage to settle in the middle with 13.80 and 13.64 streams being rejected after 30 time steps. However, setting the link weight of the path-tree that is currently under construction to zero maximizes the impact of the temporal link weight and leads to 13.32 rejected streams after 30 time steps.

Figure 8.4d displays the number of destinations per stream. Here, the *ignore* heuristic does not perform good either. With 2.243 destinations per stream after 30 time steps the *ignore* heuristic is unable to hold the multicasts in the traffic plan as well as the other heuristics. The modified Dijkstra Overlap *zero* heuristic performs the best with 2.287 destinations per stream after 30 time steps. For



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.4: Modified Dijkstra Overlap path-tree construction methods test 02

the *added* and *whole* heuristic we see that they are not far off from the *zero* heuristic. The difference of the *added* heuristic is small with 0.011 destinations per stream and for the *whole* heuristic we end up with 2.279 destinations per stream after 30 time steps.

Next, we look at Figure 8.3f that contains the results regarding the candidate path-tree overlap. All modified Dijkstra Overlap approaches have with under 0.1 a very low overlap. For the *ignore* heuristic the overlap is between 0.077 and 0.071 which is better than the candidate path selection methods (cf. Section 8.3.1) but still worse than the other modified Dijkstra Overlap heuristics. The *added* heuristic manages to reach an overlap between 0.065 and 0.062. The *whole* heuristic manages to reach an overlap between 0.063 and 0.060. The lowest and therefore the best overlap is shown by the *zero* heuristic with an overlap between 0.056 and 0.055.

The *ignore* heuristic performs not as good as the other modified Dijkstra Overlap heuristics because it does not consider the benefit of routing the linear path together. The *ignore* heuristic has unique candidate path-trees and therefore a low overlap. On the other hand each path-tree is not as slim as it could be. This leads to more links with spatial overlap resulting in more rejected streams and a lower number of destinations per stream.

The *whole* heuristic is better than the *added* heuristic since the temporal link weight is reduced for more links every time. Essentially, that makes the links in the path-tree that is currently constructed cheaper to traverse and therefore the end result of the path-tree contains fewer links overall. Setting the link weight of the path-tree that is currently under construction to zero maximizes the impact of the temporal link weight. However, having just few links in the candidate path-trees would generally lead to a higher overlap, but this is mitigated in the modified Dijkstra Overlap by the link weight increase.

We conclude that the *zero* heuristic performs the best among the modified Dijkstra Overlap heuristics. It has the tendency to reject the fewest streams while also resulting in the most destinations per stream.

8.3.3 Comparison of Intermediate Node Distance Construction Method Heuristics

The next comparison is test 03 for the intermediate node distance method. Figure 8.5 illustrates the evaluation results.

The intermediate node distance heuristic that creates a *spanning tree* instead of calculating the distance separately to all intermediate nodes shows in Figure 8.5a a better runtime of 0.95 s after 30 time steps. The *normal* approach has a longer runtime per time step with 0.99 s after 30 time steps. The runtime speedup of the *spanning tree* heuristic is not caused by choosing an approximate option or to end the search early but from letting the Dijkstra linear path search run until it has seen all intermediate nodes instead of running it multiple times. This assures that the resulting path-trees are the same as in the *normal* approach.

The number of edges in the conflict graph shown in Figure 8.5b is about the same for both heuristics. It reaches 2,487,983 edges for the *spanning tree* heuristic.

The number of rejected streams and the amount of destinations per scheduled stream is almost exactly the same for both heuristics. These are 13.50 rejected streams after 30 time steps for the *normal* approach and 13.55 streams for the *spanning tree* heuristic. The amount of destinations per stream with 2.29 after 30 time steps in Figure 8.5d is the same. Because the *spanning tree* heuristic has the same path-trees as the *normal* approach the number of rejected streams is not influenced.

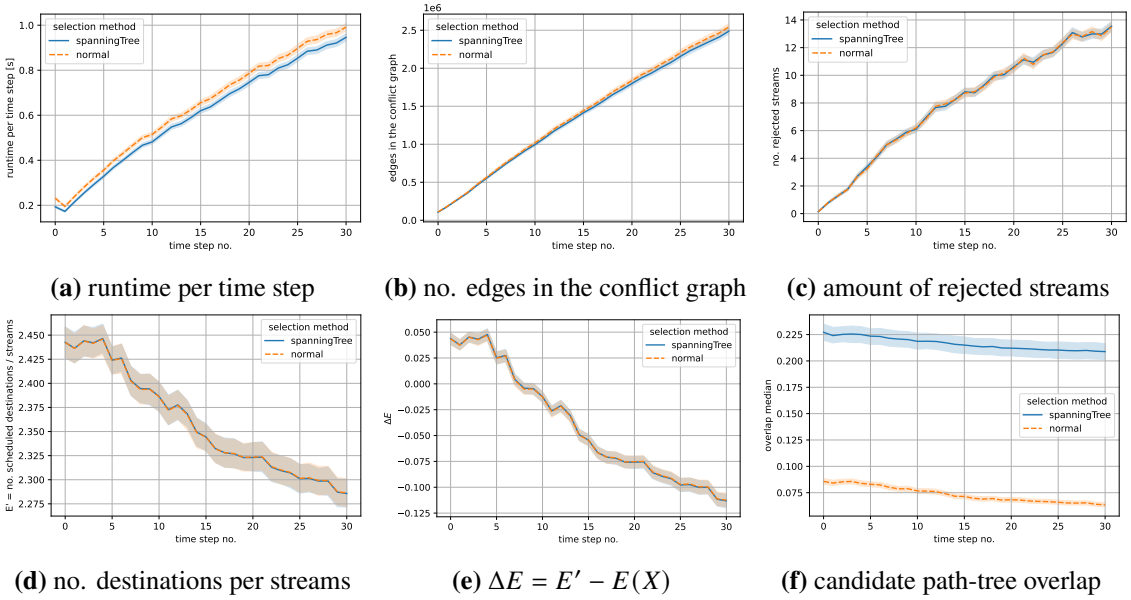
The difference in the overlap in Figure 8.5f for the *spanning tree* heuristic occurs specifically because of the combination with the random per path tree destination order. The overlap of the *normal* approach has a higher overlap than the modified Dijkstra Overlap but is better than the candidate path selection methods. It stays around 0.06 after 30 time steps.

We conclude that the *spanning tree* heuristic has a better runtime and rejects the same amount of streams even though it produces higher overlaps between the candidate path-trees compared to the *normal* approach.

8.3.4 Comparison of Destination Order Approaches

Now, we take a look at the different destination order approaches. We plot the evaluation results from test 01, 02, and 03 separated regarding the destination order.

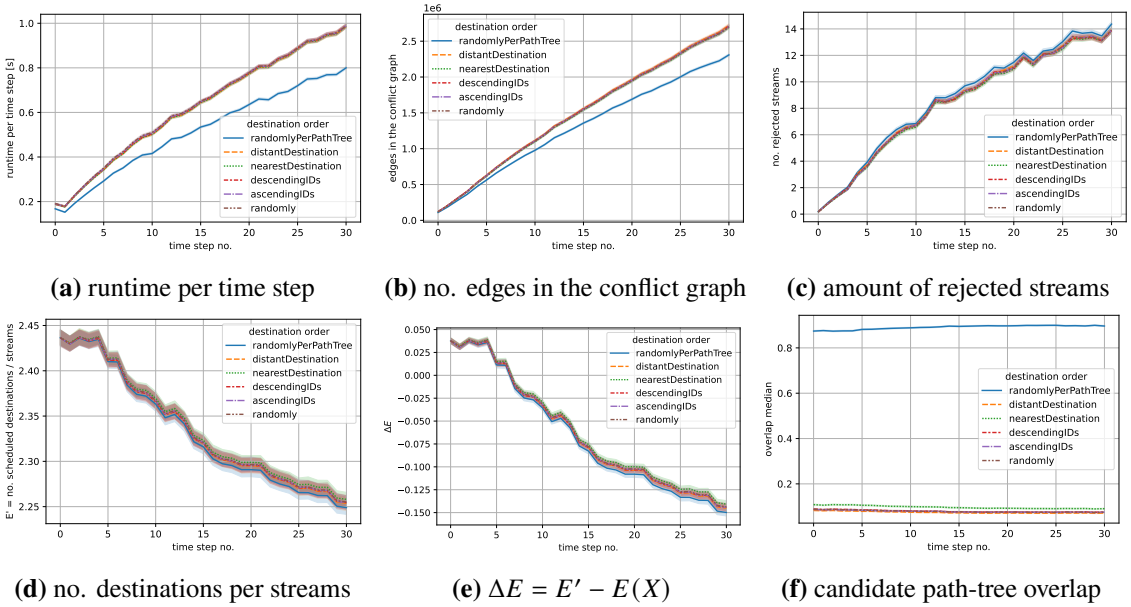
First, we look at Figure 8.6a for test 01 that displays the total runtime per step. Here in the candidate path selection method cases, all ordering approaches need up to 0.99 s except *randomly per path-tree* that needs noticeably less with 0.80 s after 30 time steps. Similar results can be seen in Figure 8.6b with the edges in the conflict graph where the *randomly per path-tree* approach contains 2,308,474 edges after 30 time steps. The *nearest destination* needs up to 2,694,643 edges while the *distant destination* approach produces 2,720,037 edges after 30 time steps. The number of rejected streams in Figure 8.6c shows no difference depending on the ordering mechanism, except again for the *randomly per path-tree* approach. Here, 14.36 streams are rejected while all the other approaches reject 13.90 streams after 30 time steps. The approach where near destinations are handled first has a tendency to entail more destinations per stream. Figure 8.6d presents the *nearest destination* approach with 2.26 destinations per stream after 30 time steps. The *randomly per path-tree* approach rejects more streams with many destinations and ends up at 2.25 destinations per stream. These



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.5: Intermediate node distance path-tree construction methods test 03



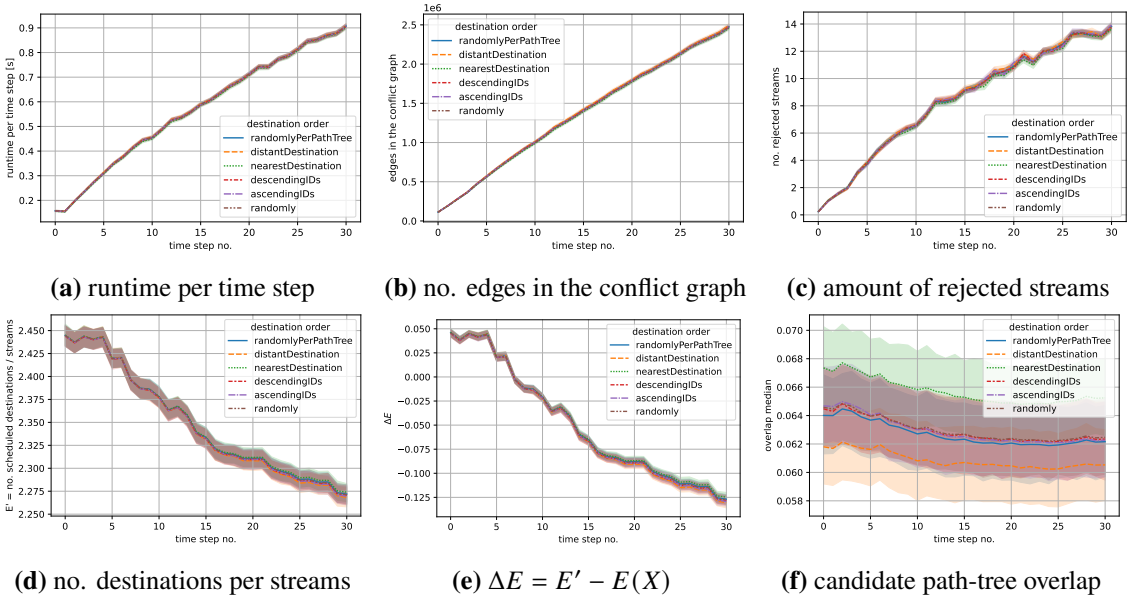
(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.6: Candidate path selection destination order approaches test 01

numbers are so close to each other that the difference is within the variance. In Figure 8.6f we can see that the *randomly per path-tree* produces similar candidate path-trees since its overlap of 0.88 is very high while in contrast the other ordering approaches manage to stay at a low 0.09 overlap.

In the modified Dijkstra Overlap methods, there is no difference between all ordering approaches, neither in runtime, number of edges in the conflict graph, amount of destinations per stream, nor candidate path-tree overlap. In Figure 8.7 even the *randomly per path-tree* looks the same. The amount of rejected streams shows a tendency towards ordering the destinations from near to far.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.7: Modified Dijkstra Overlap destination order approaches test 02

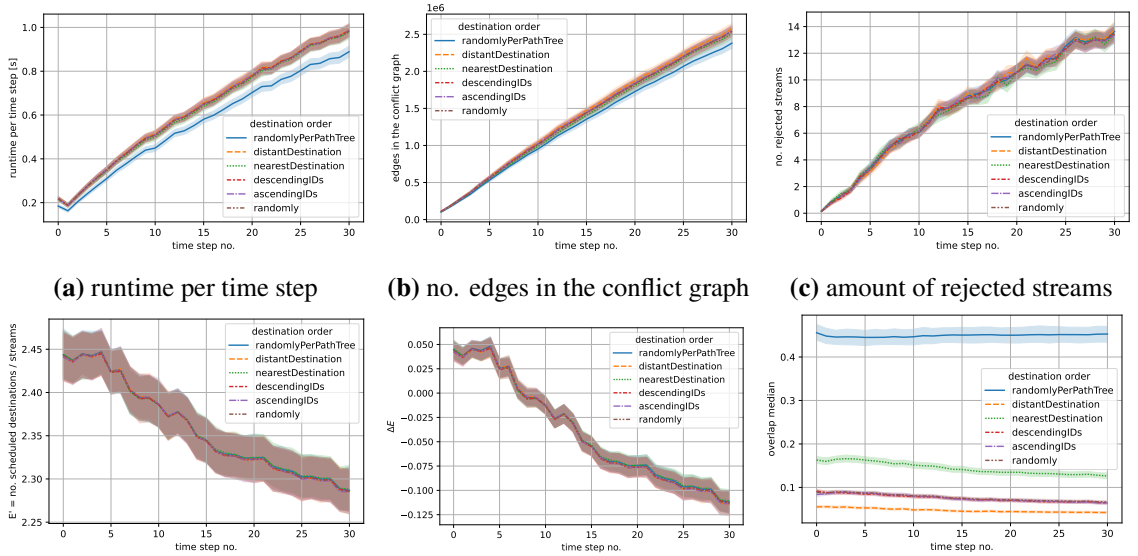
For the intermediate node distance methods the *randomly per path-tree* ordering approach is 0.10 s faster than the rest. Figure 8.8a displays a runtime of 0.98 s after 30 time steps for the other ordering approaches. This difference is also represented in Figure 8.8b where the *randomly per path-tree* has 160,192 fewer edges than the other ordering approaches. With 13.49 the amount of rejected streams and with 2.29 the destination number per stream are the same for all approaches after 30 time steps. Regarding the candidate path-tree overlap in Figure 8.8f, we see that the *randomly per path-tree* approach has more similar path-trees with 0.45. The *nearest destination* order approach is significantly lower with 0.14 followed by other approaches with 0.07. The lowest is the *distant destination* approach with around 0.05.

Generally, we can say that the destination order has an impact on the candidate path-tree shape. This can be concluded from the different overlap values. In contrast, the destination order has no impact on how good these path-trees are for scheduling. This leads to very similar amounts of rejected streams. If a deterministic approach is desired, selecting the destinations with increasing distance is the recommended solution (*nearest destination*). Otherwise, the *randomly per path-tree* approach is a good choice. It results in similar numbers of rejected streams while reducing runtime in some cases.

8.3.5 Comparison of Linear Path Integration Approaches

This section compares the three integration approaches *naive*, *end*, and *rewrite* by plotting the results of test 01 and 02 separated according to the integration approach. Test 03 for the intermediate node distance linear path selection is not needed since it always uses the *end* integration approach.

The integration approaches show differences if we use the linear path selection methods. Figure 8.9 displays test 01. Mainly, the *naive* integration approach stands out to be not as good as the other two. It has a slightly higher runtime of up to 0.97 s while the *rewrite* and *end* approaches need



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

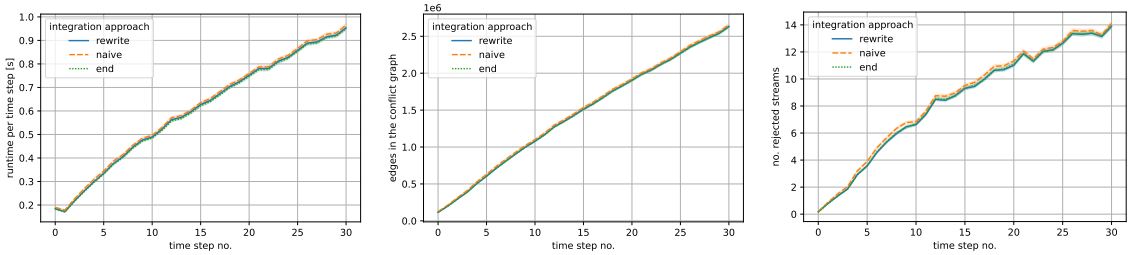
Figure 8.8: Intermediate node distance destination order approaches test 03

0.95 s after 30 time steps. With 2,655,973 edges in the conflict graph and 14.11 rejected streams after 30 time steps the *naive* integration has a tendency to be higher. The amount of destinations per stream suffers under the *naive* integration approach with a difference of -0.15 after 30 time steps. Figure 8.9f shows that the overlaps are all very close together with slightly overlapping confidence intervals. The *naive* approach is the highest with 0.224 after time step zero while the *end* approach with 0.219 has the least overlap between the candidate path-trees.

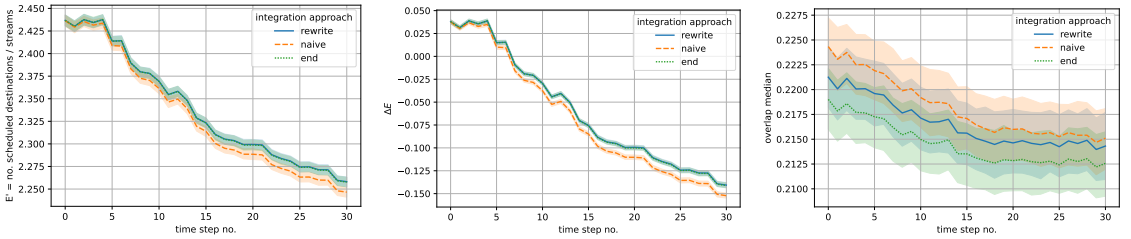
These results can be explained by the fact that the path-trees that use the *naive* integration contain more links. Even though these links are unnecessary and result in invalid path-trees they still have to be considered during the spatial and temporal collision calculations. This problem becomes worse the more destinations a path-tree contains. More collisions in the unnecessary links lead to preferred rejection of multicast streams with many destinations. This increases the deviation of the expected number of destinations per stream. Regarding the *rewrite* and *end* approaches it does not matter how the linear path-trees are integrated as long as they are valid path-trees. This implies that the optimizations of the routes the *rewrite* approach offers are so small that it is irrelevant for the whole process.

If we examine the integration approaches combined with the modified Dijkstra Overlap in Figure 8.10 we see no difference in any metric. This means that modified Dijkstra Overlap only suggests linear path that are already valid.

As a result, we can finally drop the *naive* integration approach since the other two have better runtime and destinations per stream results. Overall the *rewrite* and *end* approach seem to make no difference. The *end* approach is the most elegant and simple algorithm while still maintaining to avoid invalid path-trees.

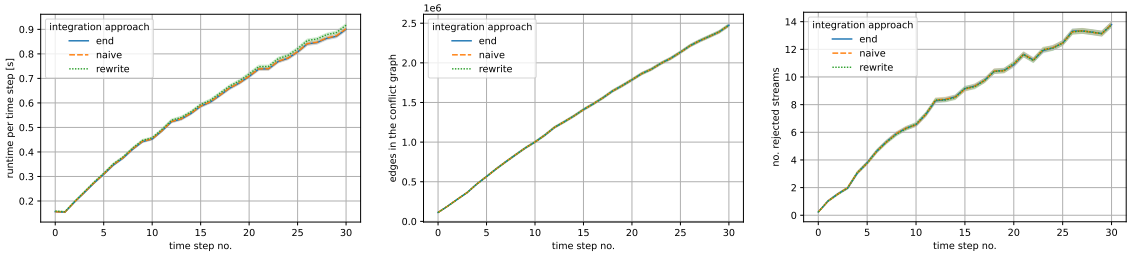


(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

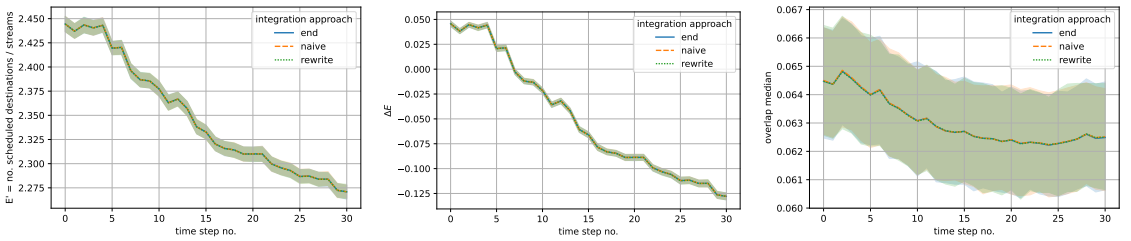


(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.9: Candidate path selection path integration approaches test 01



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams



(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.10: Modified Dijkstra Overlap path integration approaches test 02

8.3.6 Selection of Best Approaches

Generally, we recognize that the differences between the path-tree construction methods are not as broad as expected. We will now state a selection of the best combinations that we further analyze. The rest of the methods paired up with the different destination order and integration approaches will no longer be present in the following evaluations.

The destination selection order is continued with the *randomly per path-tree* and the *nearest destination* approaches. We select the *randomly per path-tree* approach since it is the fastest and the *nearest destination* approach since it is the best out of the deterministic approaches. For the candidate path selection methods, we just further analyze the *least links base* and *similar start base* approaches. All the heuristics have a negligible impact on runtime and rejection. For this reason, it is sufficient to continue the evaluation with the base approaches and not include the heuristics. Regarding the modified Dijkstra Overlap, we continue with the *zero* approach because it has the best rejection results. For the intermediate node distance, we just keep the *spanning tree* approach since it is faster than the *normal* approach while rejecting the same amount of streams. We choose the *end* integration approach over the *rewrite* approach since it is closer to the integration in the intermediate node distance construction approach. This brings us down to eight path-tree construction method combinations. We will now recompare these combinations against each other using test 04 with the parameters that are described in Table 8.4 and select the best overall combination.

This construction method comparison shows us in Figure 8.11 that they are all very similar. The different methods are even so close that they almost everywhere have overlapping confidence intervals. One exception from that is the *similar start* method that has a few more edges in the conflict graph and a higher deviation from the expected amount of destinations per stream than the *spanning tree* method. The other exception presented in Figure 8.11f is the candidate path-tree overlap. The modified Dijkstra Overlap *zero* approach has by far the lowest overlap.

Each of these methods is a suitable choice to continue the evaluation because they are all similarly good. To keep the rest of the evaluation manageable, we select just one method. We choose the intermediate node distance *spanning tree* method since it showed the most promising results in our pre-evaluations.

8.4 Results for Different Planner Input Parameters

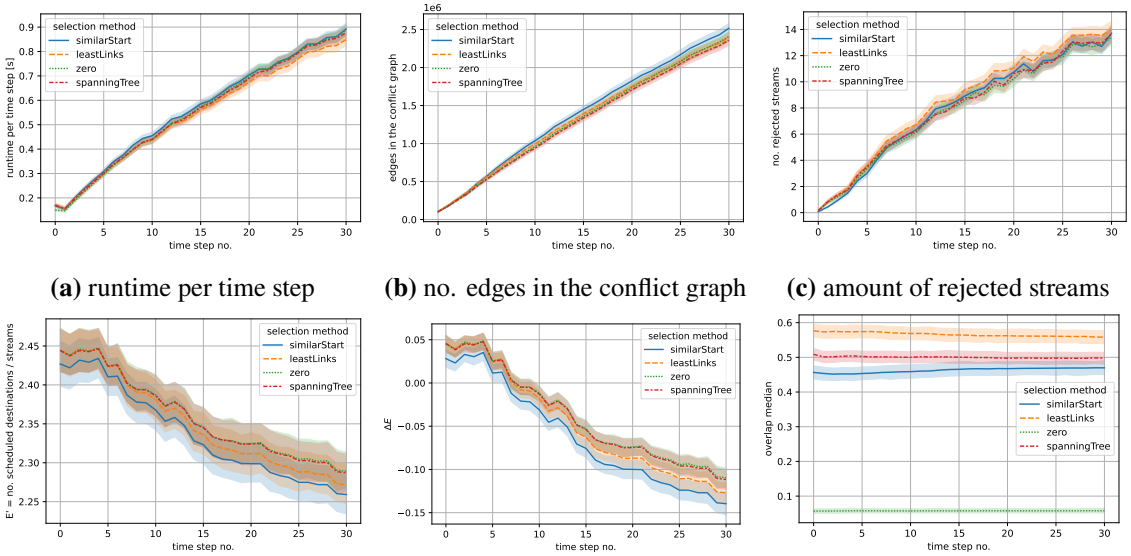
In the following section we evaluate the influence of the configurations per stream n_{cps}, the amount of candidate path-trees n_{path}, and the conflict graph builders. Table 8.5 provides the overview of the used parameters for those three tests. Test 05 and 06 keep either n_{cps} or n_{path} constant while varying the other parameter. Test 07 compares the random and classic conflict graph builders.

8.4.1 Comparison of Configurations per Stream Parameter

Figure 8.12 shows the behavior of different n_{cps} values based on test 05. With increasing amount of n_{cps} the calculation of the next traffic plan takes longer, because there are more configurations that need to be checked for spatial and temporal conflicts. This runtime per step is 0.41 s after

parameter name	Test 04
network size n	81
network topologies	all
scenario type	some_m
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6, $G(\frac{1}{4})$ until 6, $U(6)$, $RW(4)$
number of time steps	30
initial streams	100
$\lvert \text{ReqS}(p_i) \rvert$	50
$\lvert \text{RemS}(p_i) \rvert$	25
number of used threads	6
linear path routing method	Dijkstra Overlap
$n_{linPath}$	6
candidate paths	least links, similarStart
modified Dijkstra Overlap	zero
intermediate nodes	spanningTree
path-tree destination order	randomly per path-tree, nearest destination
path-tree integration type	end
n_{path}	4
conflict graph builder	random and classic
solver heuristic	GFH-dst-tiebreaker
conflict graph expansion	static
n_{cps}	36

Table 8.4: Test parameter table for the best construction methods



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.11: Best construction methods test 04

parameter name	Test 05	Test 06	Test 07
network size n	81		
network topologies	all		
scenario type	some_m		
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6 and 20, $G(\frac{1}{4})$ until 6 and 20, $U(6)$, $U(20)$, $RW(4)$		
number of time steps	30		
initial streams	100		
$\lvert\text{ReqS}(p_i)\rvert$	50		
$\lvert\text{RemS}(p_i)\rvert$	25		
number of used threads	6		
linear path routing method	Dijkstra Overlap		
$n_{linPath}$	6		
candidate paths	—		
modified Djikstra Overlap	—		
intermediate nodes	spanning tree		
path-tree destination order	nearest destination and randomly per path-tree		
path-tree integration type	end		
n_{path}	4	2,3,4,5,6	4
conflict graph builder	random and classic		random, classic
solver heuristic	GFH-dst-tiebreaker		
conflict graph expansion	static		
n_{cps}	12,24,36,48,60,72,84,96,108,120	84	84

Table 8.5: Test parameter table for n_{cps}, n_{path}, and builder

30 time steps for 12, 0.98 s for 36, and 1.91 s for 60 configurations per stream. As illustrated in Figure 8.12a it further extends to 3.26 s after 30 time steps for 84 configurations and 6.13 s for 120 configurations.

Figure 8.12b presents the rising numbers of edges in the conflict graph. The amount of configurations per stream directly correlates with the number of nodes in the conflict graph, and therefore we also have more edges in the conflict graph. The range is quite large from 2,541,704 edges after 30 time steps with 36 configurations over 16,560,596 edges with 84 configurations to 35,157,841 edges after 30 time steps for 120 configurations.

For the evaluated n_{cps} results, the difference in the amount of edges in the conflict graph increases the higher the n_{cps} becomes. This means that the number of edges is causally dependent on n_{cps} and increases faster than a linear function. Because the total runtime depends on the spatial and temporal overlap calculation, it also increases more than linearly for a change in n_{cps}.

Next in Figure 8.12c, we look at the number of rejected streams. With 12 configurations the solver rejects 21.64 streams after 30 time steps. A rejection of only 17.57 streams after 30 time steps with only 24 more configurations is already an improvement. 120 configurations perform the best with only 11.52 rejected streams after 30 time steps. With 84 configurations we are already really close to the best results with only 12.29 rejected streams after 30 time steps.

If there are more configurations per stream and a fixed number of candidate path-trees, then the number of potential phases per stream scales proportional to n_{cps}. This increase in phases per stream helps, if a spatial overlap between two streams occurs, to find configurations where they do not conflict temporally with each other. If we can find configurations that are temporally separated, we can admit the streams instead of rejecting them. This means that increasing the n_{cps} helps to avoid rejection because of more temporal separated configurations. The higher n_{cps} is the less the number of rejected streams decreases.

If there are fewer streams that are rejected then the deviation of the expected destinations per stream is reduced. For low n_{cps} Figure 8.12d shows that the number of destinations per stream is rapidly decreasing. With $n_{cps} = 36$ the metric stays at 3.50 destinations per stream after 30 time steps. Increasing the n_{cps} even more leads to smaller differences with each step leading towards 3.65 destinations per stream after 30 time steps.

With the additional information of the destinations per stream plot we can say that the lower amount of rejected streams benefits especially path-trees that contain many destinations as well as path-trees with many links.

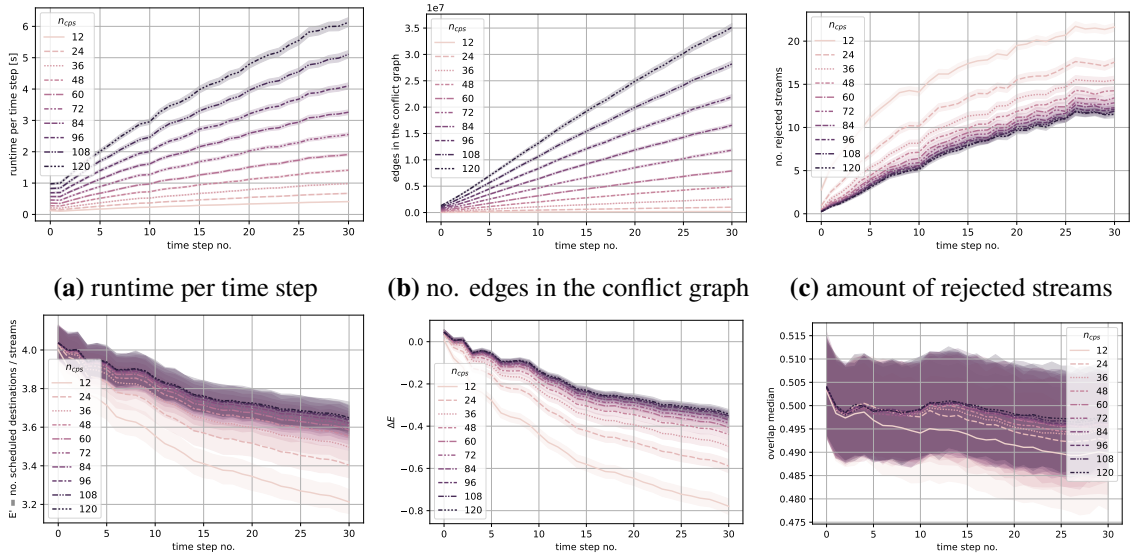
Obviously, the overlap of the candidate path-trees is independent of how many n_{cps} we consider. In Figure 8.12f the confidence intervals are overlapping for all n_{cps}. This makes all overlaps the same.

We can conclude that similar to the unicasts in [FGD+22] also the multicasts benefit from a higher n_{cps}. The only drawback is the significantly longer runtime it needs to compute the temporal and spatial overlap of these extra configurations. This increase in runtime is growing the larger n_{cps} becomes while the improvement for the number of rejected streams is getting smaller. Therefore, it is important to find a good balance between short runtimes and few rejected streams.

8.4.2 Comparison of Candidate Path-trees Parameter

Figure 8.13 displays the evaluation results of test 06 regarding different numbers of candidate path-trees.

In Figure 8.13a we can see that the more candidate routes we want to have the longer the runtime becomes. For 2 routes the runtime is quite short with 2.59 s after 30 time steps. It increases to 3.26 s after 30 time steps for 4 n_{path} and takes 4.25 s after 30 time steps for 6 n_{path}. The increase in runtime is caused by the additional routes that need to be calculated. This seems to be slightly more than a linear increase.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

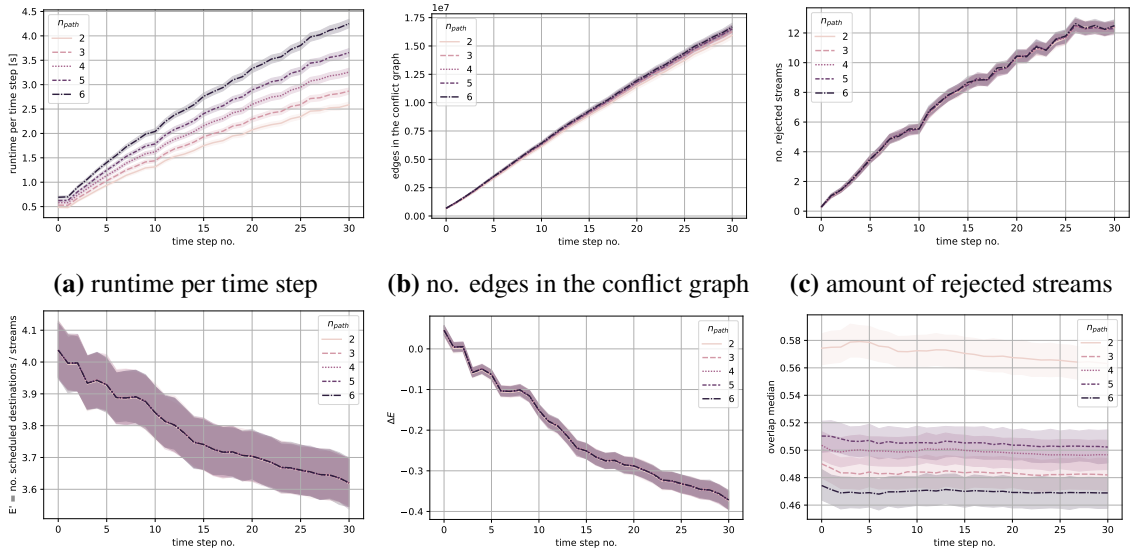
Figure 8.12: Comparison n_{cps} test 05

The number of candidate path-trees has no relevant effect on the edge amount in the conflict graph, as Figure 8.13b shows. In Figure 8.13c we can see that there is no relevant difference in the amount of rejected streams depending on the n_{path} since the confidence intervals are overlapping. Also, the amount of scheduled destinations per stream in Figure 8.13d and Figure 8.13e are exactly the same independent of n_{path}.

For the rejection, we can only speculate that most links causing the spatial and temporal collisions are essential parts that appear in all candidate path-trees. This matches with the fact that the scheduled streams have the same amount of destinations independent of the n_{path}. For streams that are rejected with 2 candidate path-trees, it is likely that with 5 candidate path-trees the same streams or at least streams with the same amount of destinations are rejected.

However, the overlap of the candidate path-trees in Figure 8.13f differ from each other. For 2 candidate path-trees the overlap ranges from 0.579 to 0.563. With an n_{path} of 3, 4 and 5 the overlap is between 0.510 and 0.482. The lowest overlap of 0.474 to 0.469 is reached with 6 candidate path-trees.

The overlap is changing for different n_{path} since the extra path-tree has a different overlap to the other path-trees than the other path-trees between each other. This causes the median candidate overlap to shift a little in the direction of the overlap the extra path-tree has. We see that the streams that are rejected have an effect on the overlap because the overlap is measured based on the streams of the traffic plan. Here, we show that the overlap does not necessary have an effect on the amount of rejected streams. This is because the rejection is based on the links that are in all candidate path-trees. It makes the number of rejected streams and the overlap just correlated in one direction and not the other way around.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.13: Comparison n_{path} test 06

One thing to keep in mind is, that test 06 varies the n_{path} while keeping n_{cps} fixed. This has the side effect that for each candidate path-tree route there are fewer configurations with different phases. Therefore, the benefit of having more candidate path-trees is countered with less variation of the phase for each route.

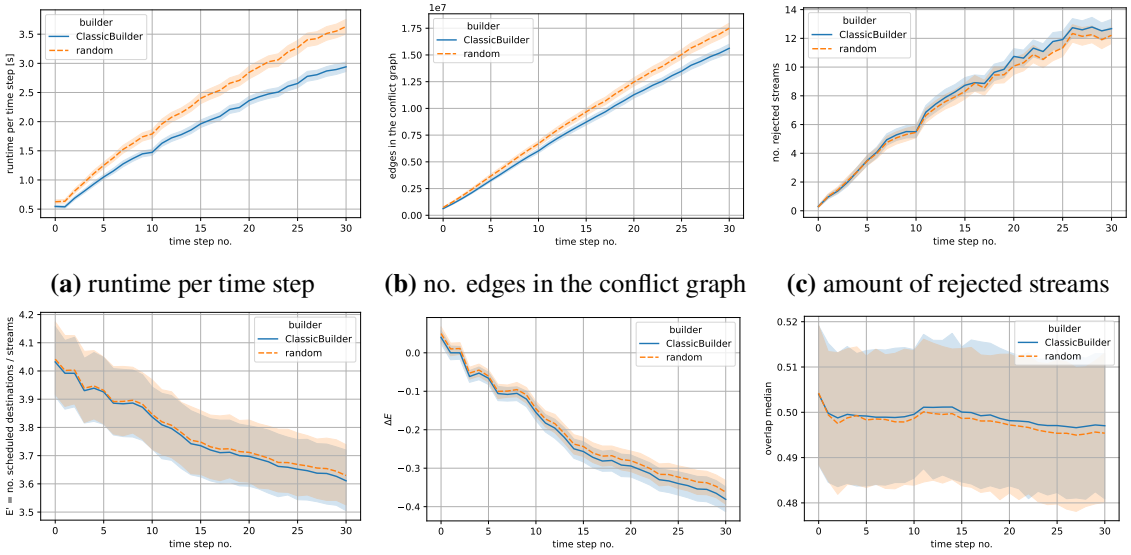
Overall we conclude that there is not much benefit of having many candidate routes. This lack of influence can be explained with the fact that if a stream is rejected, it is due to a link that is needed in all path-trees.

8.4.3 Comparison of the Conflict Graph Builder

Figure 8.14 shows the evaluation results of the classic and the random conflict graph builder comparison. Figure 8.14a reveals that the random builder is slightly slower with up to 3.63 s compared to the 2.94 s per time step for the classic builder. Both conflict graph builder reject similar streams and therefore have the same amount of destinations per stream. The surprisingly low difference in the number of rejected streams is explainable by the high $n_{cps} = 84$. The higher the n_{cps} is the smaller the differences in the amount of rejected streams between the random and the classic builders become.

8.5 Results of GFH Alternatives

In this section we want to evaluate what difference it makes if we modify the GFH solver heuristic to first care about the destination amount. For this evaluation the test 08 uses the configurations of Table 8.6 and Figure 8.15 displays the results.



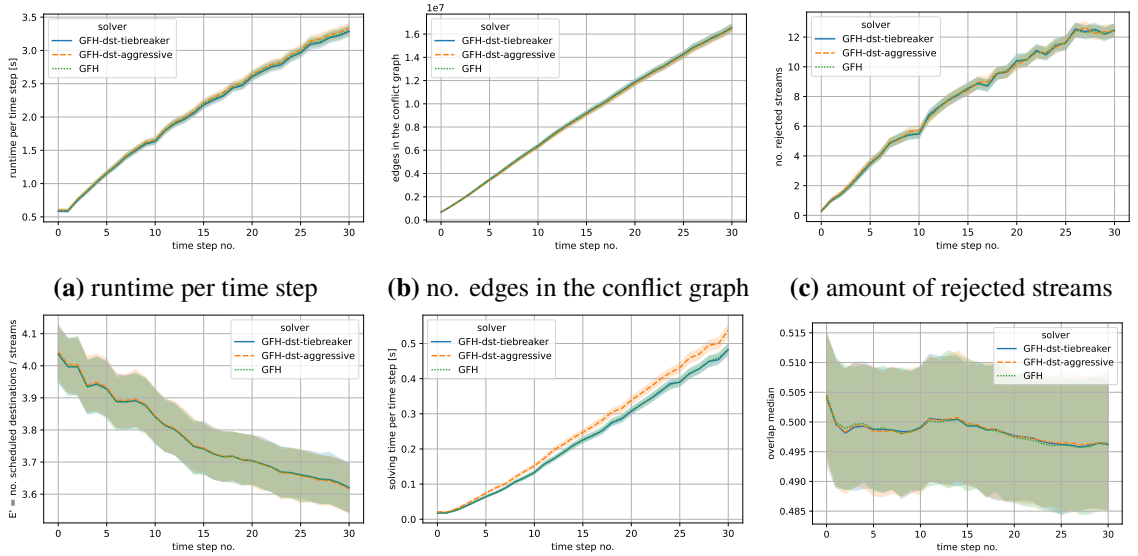
(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.14: Conflict graph builder test 07

parameter name	Test 08
network size n	81
network topologies	all
scenario type	some_m
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6 and 20, $G(\frac{1}{4})$ until 6 and 20, $U(6)$, $U(20)$, $RW(4)$
number of time steps	30
initial streams	100
$\lvert \text{ReqS}(p_i) \rvert$	50
$\lvert \text{RemS}(p_i) \rvert$	25
number of used threads	6
linear path routing method	Dijkstra Overlap
$n_{linPath}$	6
candidate paths	—
modified Djikstra Overlap	—
intermediate nodes	spanning tree
path-tree destination order	nearest destination and randomly per path-tree
path-tree integration type	rewrite and end
n_{path}	4
conflict graph builder	random and classic
solver heuristic	GFH, GFH-dst-tiebreaker, GFH-dst-aggressive
conflict graph expansion	static
n_{cps}	84

Table 8.6: Test parameter table for GFH alternatives



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** solving time per time step **(f)** candidate path-tree overlap

Figure 8.15: Heap ordering approaches test 08

The total runtime per step is the same for our GFH destination tiebreaker approach and the GFH aggressive alternative, as well as the original GFH approach. Figure 8.15a presents the total runtime per step and Figure 8.15e displays only the runtime of the solver per step. It leads to a total runtime of 3.30 s after 30 time steps. The aggressive approach actually has a higher solving time of 0.54 s compared to the destination tiebreaker approach that even has the same solving time than the original GFH of 0.48 s after 30 time steps.

This difference is negligible and becomes unnoticeable in the total runtime since the total runtime is mostly defined through the spatial and temporal overlap calculations as well as the candidate path-tree finding algorithm.

The number of edges in the conflict graph and the amount of overlap of the candidate path-trees is not influenced by the different approaches and therefore is the same for all of them. Regarding the amount of rejected streams, all three approaches have similar results. They all end up at 12.45 rejected streams with the same amount of destinations per stream of 3.62 after 30 time steps.

This means that adding the destination tiebreaker is not used that often, and because of that, it has no negative effects but also no positive benefits. The aggressive approach, which sorts regarding the destination amount first, results in a similar order, than what GFH comes up on its own. The aggressive solver takes a little longer, but the sorting decision is more independent of the conflict graph.

In summary, we can say that all GFH solvers work equally well. The aggressive solver alternative takes slightly longer and is therefore not selected. The modified GFH solver with destination tiebreaker performs the same as the original GFH. The destination tiebreaker solver is still the right one for us since it acts more like our optimization function than the original GFH solver in regard to handling multicasts.

parameter name	Test 09
network size n	81
network topologies	all
scenario type	some_m
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6 and 20, $G(\frac{1}{4})$ until 6 and 20, $U(6)$, $U(20)$, $RW(4)$
number of time steps	0
initial streams	350, 850, 1350
$\lvert\text{ReqS}(p_i)\rvert$	0
$\lvert\text{RemS}(p_i)\rvert$	0
number of used threads	6
linear path routing method	Dijkstra Overlap
$n_{linPath}$	6
candidate paths	—
modified Djikstra Overlap	—
intermediate nodes	spanning tree
path-tree destination order	nearest destination and randomly per path-tree
path-tree integration type	end
n_{path}	4
conflict graph builder	random and classic
solver heuristic	GFH-dst-tiebreaker
conflict graph expansion	static
n_{cps}	84

Table 8.7: Test parameter table for static scenarios

8.6 Results of Static Scenarios

This section is about solving the joint routing and scheduling statically. For this offline setting we evaluate scenarios that try to schedule all streams in one time step. Table 8.7 lists the parameters for test 09.

In Figure 8.16a the total runtime is displayed. For static scenarios, the total runtime per step metric is the same as the total runtime the program is running, since we have only the initial time step. The total runtime increases with the number of streams that are requested to be scheduled. For 350 streams it takes 4.26 s. With 850 streams the runtime increases to 24.98 s. The longest runtime is 65.17 s for 1350 streams. The increase in runtime from 350 streams to 850 streams is comparable to the time steps 10 and 30 of the dynamic scheduling. In the static approach we have an increase in the runtime of 20.72 s while the increase in the dynamic approach stays below 2.00 s per time step.

This increase in runtime is caused by the higher amount of streams that needs to be scheduled for the same traffic plan. The more streams are scheduled by the solver the more configurations have to be calculated and checked for spatial and temporal overlaps. The dynamic approach has a lower increase in the runtime per step since the already calculated configurations and spatial and temporal

overlap results are reused. While the static approach has to recalculate all for every stream the dynamic approach saves time by just calculating the configuration and overlap for the newly added streams.

It has to be noted that cumulating the runtimes per step in the dynamic scenario has a higher cumulated runtime for the same amount of streams than the total runtime in the static scenario. This suggests that if we just need one traffic plan it is better to use the static version. But if we have to regularly calculate a new traffic plan then the dynamic version becomes much better because then just the runtime until the next traffic plan is relevant.

Next, we look at Figure 8.16b that displays the amount of edges in the conflict graph. This metric also increases with the amount of streams. 350 streams lead to 4,559,130 edges while the scenarios with 850 streams contain 27,037,380 edges in the conflict graph. For 1350 streams we end up with 68,287,539 edges.

The increase in the number of edges in the conflict graph comes from additional streams that we want to schedule. The behavior seems to be less than but close to a quadratic increase between these three evaluation results.

Figure 8.16c presents the number of rejected streams. Scenarios with 350 streams do not reject any streams in our evaluation. The median of the 850 scenarios is similarly low at 8.00 rejected streams, but this can go up to 600.00 rejected streams. In the 1350 stream scenarios the number of rejected streams is higher with 198.50 rejected streams.

The more streams are in ActiveS(p_i) the harder it is to fit an extra stream into the traffic plan. As a result the number of rejected streams increases if overall more streams are scheduled. If the total amount of requested streams is low almost everything fits in the traffic plan. But if there are many requested streams the number of rejected streams increases quickly because the network can not increase the capacity of streams that it can handle.

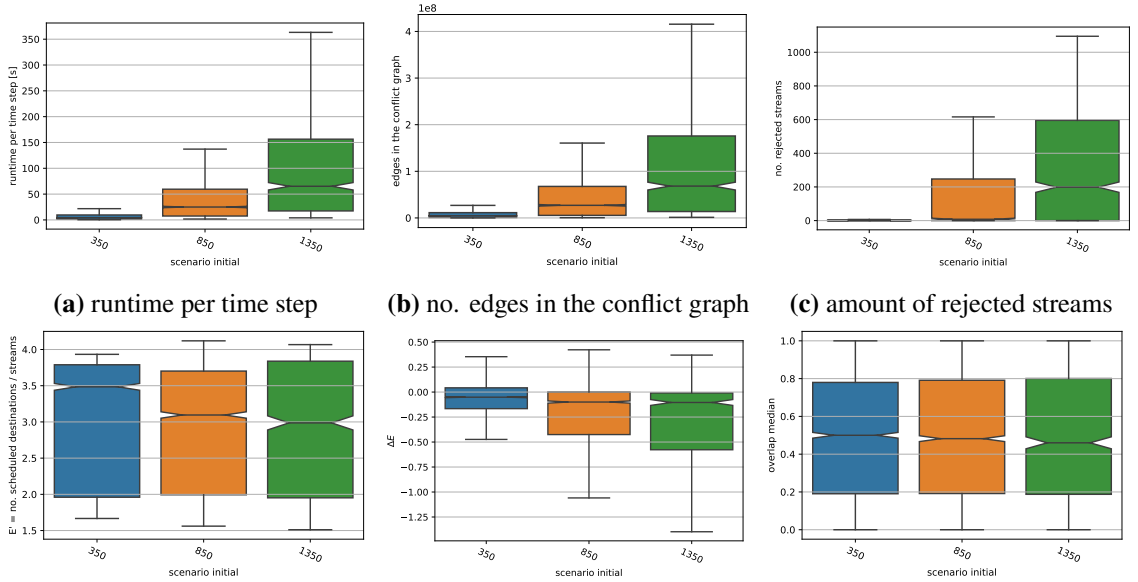
As visualized in Figure 8.16d the median amount of destinations per stream is at 3.48 for 350 streams and at 2.99 for 850 and 1350 streams. The deviation of the expected amount of destinations per stream is low with -0.05 for 350 streams and -0.10 for 850 streams. Figure 8.16e displays a similar deviation of -0.10 destinations per stream in the 1350 stream scenarios.

The overlap of the candidate path-trees in Figure 8.16f is around 0.48 in all static scenarios. This is expected since the candidate path-tree overlap is dependent on how the path-trees are routed and not how many streams we have in our traffic plan. We do just calculate the median over all streams.

In conclusion, we can say that the dynamic joint routing and scheduling solver computes the next traffic plan faster than the static version. The planner program can schedule 850 streams in 24.98 s with just 8.00 rejected streams.

8.7 Results of Domain Dependent Parameters

Up to this point, we have evaluated different scenarios and networks and aggregated them to have an independent solution. In this section we want to emphasize that even though we generally evaluate independent of domain specific parameters they still have an impact on the evaluation result and



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.16: Static scenarios test 09

should not be ignored. Table 8.8 presents the parameters for test 10 and test 11. In test 10 we look at the influence of the network topology and in test 11 we evaluate the impact of the multicast destination probability distribution.

8.7.1 Comparison of Network Topologies

Figure 8.17 displays the results of test 10 regarding the different network topologies grid, ring (n, k), Erdős-Rényi, and Waxman.

First, we look at Figure 8.17a for the runtime after 30 time steps. The ring (81,1) has a very long runtime with 6.73 s per time step. The next highest are the two grid topologies with 4.96 s. Below that the other rings have a runtime of 4.30 s. The Waxman81,1 is significantly faster with only 2.35 s per time step. The Erdős-Rényi topologies are very fast with 1.81 s and 1.42 s. The best runtime belongs to the Waxman81,5 topology with the most network links. It just needs 0.51 s per time step. A higher diameter and lower amount of links in the network topology lead to a much longer runtime.

Figure 8.17b displays the amount of edges in the conflict graph. The 9x9 grid and the ring (81,1) topology have the most edges with 25,220,490 and 25,573,264 after 30 time steps followed by the ring (81,3) and ring (81,4) that have up to 23,130,022 edges. Below that the ring (81,2) and the 27x3 grid have 20,223,547 edges after 30 time steps. With a lower result the Waxman81,1 has only 14,917,589 edges after 30 time steps. The Erdős-Rényi topologies have a very small conflict graph with up to 11,359,778 edges after 30 time steps. The smallest conflict graph belongs to the Waxman81,5 topology with the most network links. It has just 2,707,315 edges in the conflict graph after 30 time steps. A higher diameter means that most path-trees are longer and more links are candidates for spacial conflicts. Because of these extra link conflicts the number of edges in

parameter name	Test 10	Test 11
network size n		81
network topologies	all	all
scenario type		some_m
$P(X = dstNo(s))$	all	$G(\frac{1}{2})$ until 6 and 20, $G(\frac{1}{4})$ until 6 and 20, $U(6)$, $U(20)$, $RW(4)$
number of time steps		30
initial streams		100
$\lvert \text{ReqS}(p_i) \rvert$		50
$\lvert \text{RemS}(p_i) \rvert$		25
number of used threads		6
linear path routing method		Dijkstra Overlap
$n_{linPath}$		6
candidate paths		—
modified Djikstra Overlap		—
intermediate nodes		spanning tree
path-tree destination order		nearest destination and randomly per path-tree
path-tree integration type		end
n_{path}		4
conflict graph builder		random and classic
solver heuristic		GFH-dst-tiebreaker
conflict graph expansion		static
n_{cps}		84

Table 8.8: Test parameter table for topology and $P(X = dstNo(s))$

the conflict graph is generally much higher for such topologies. The ability to have alternative routes causes fewer edges in the conflict graph. This is especially seen with the ring and Waxman topologies.

Next, we show Figure 8.17c for the number of rejected streams. The ring (81,1), ring (81,2), and 27x3 grid reject the most streams with 27.42 rejected streams after 30 time steps. The ring (81,3) is already an improvement. It only rejects up to 21.09 streams. This is further improved in the ring (81,4) topology that rejects 11.31 streams after 30 time steps. The 9x9 grid rejects almost just half as much as the 27x3 grid. For the 9x9 grid there are 16.25 streams rejected after 30 time steps. All Waxman and Erdős-Rényi topologies yield good results. Even after 30 time steps they reject less than 5 streams.

In topologies like the Waxman and Erdős-Rényi with a small diameter, only a few streams are rejected, even with a small conflict graph. For topologies with high diameters, like the grid, more flexibility in the route reduces the number of rejected streams. The rejection of more than 20 streams has the effect that the number of edges in the conflict graph is increasing noticeably slower than when a topology rejects fewer streams. This effect is caused by the lower number of configurations due to fewer scheduled streams. This can be seen in the ring (81,2) and the 27x3 grid.

We now look at Figure 8.17d for the amount of destinations per stream and Figure 8.17e for the deviation. The destinations per stream decrease similar to the increase of the rejected streams. The Waxman and Erdős-Rényi topologies deviate less than -0.17 from the expected amount of destinations per stream since they accept almost all streams. This results in between 4.04 and 3.82 destinations per stream. The ring (81,1) has the lowest amount of destinations per stream with 2.89 after 30 time steps. But even this topology deviates less than -1.10 destination from the ideal number of destinations per stream. The 9x9 grid has with 3.46 a very good amount of destinations per stream after 30 time steps.

Figure 8.17f plots the candidate path-tree overlap separated according to the topology. The Waxman and Erdős-Rényi topologies have a low overlap of around 0.35 to 0.43 while the grid and ring topologies have an overlap of 0.52 to 0.65.

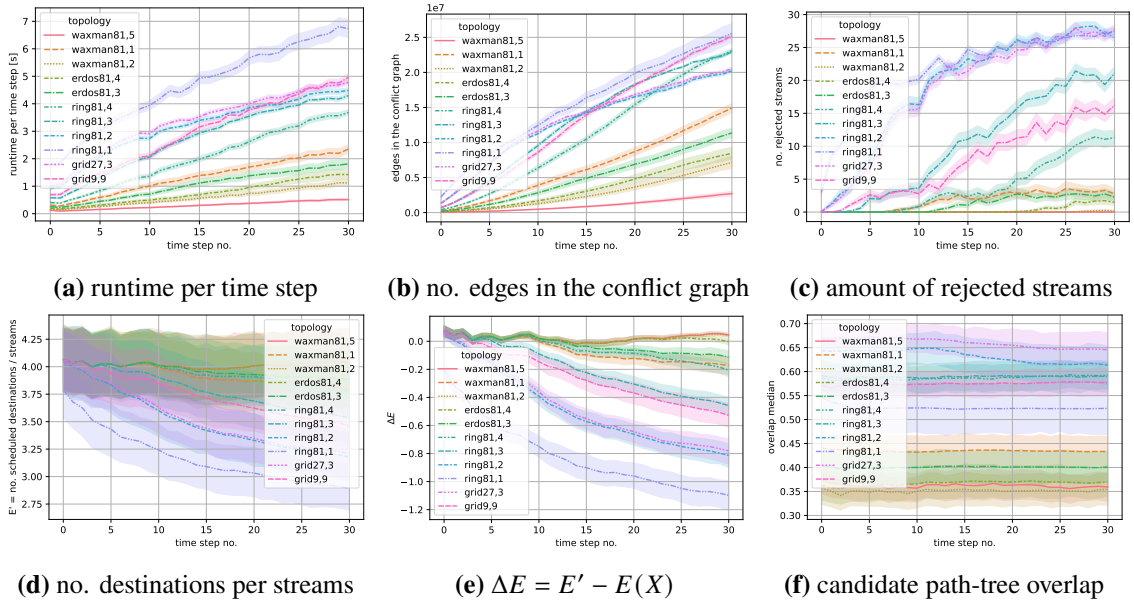
Grid networks are generally performing poorly in all metrics. A reason for this is the low number of edges in the topologies, so there is no way to skip portions of the network. Besides this, the 9x9 grid, which still has a long runtime, can mitigate some of its drawbacks with its symmetry, which allows for many alternative routes and leads to a relatively low number of rejected streams. The ring networks are improving the more links are used. This means that ring topologies with higher k benefit from the shorter alternative routes and therefore bridges can be skipped in comparison to a ring (n,1). The Erdős-Rényi and Waxman topologies are holding up quite well. We can say that graphs containing more links are better since there are more options for routing to begin with. Having many links also leads to a lower network diameter. If the diameter is smaller, the path-tree needs fewer hops to reach all destinations. As a consequence, there are fewer links that need to be considered which can cause spatial and temporal collisions. This reduces the runtime per step, the amount of edges in the conflict graph, and the number of rejected streams all at once.

Overall, we can say that the more links the topology has and the lower the diameter is the better the planner program performs. The Erdős-Rényi and Waxman network topologies perform better than grid and ring (81,k).

8.7.2 Comparison of Destination Probability Distributions

Figure 8.18 presents the results of test 11 regarding the different multicast destination probability distributions. We evaluate with the uniform probability distributions $U(6)$ and $U(20)$. For the geometric probability distributions $G(\frac{1}{2})$ and $G(\frac{1}{4})$ we have variants until six and 20 destinations. Another distribution we use is $RW(4) = \frac{1}{10} * (5 - dstNo(s))$ for $dstNo(s) \leq 4$ else zero.

The by far longest runtime in Figure 8.18a results from $U(20)$. It has a runtime of up to 4.89 s per step. For $U(6)$ it takes 3.27 s after 30 time steps. $G(\frac{1}{2})$ and $G(\frac{1}{4})$ until 20 destinations take similarly long with 3.22 s and 3.18 s per step after 30 time steps. $G(\frac{1}{4})$ until 6 destinations has about the same runtime with up to 2.97 s per step. $G(\frac{1}{2})$ until 6 destinations and $RW(4)$ are the fastest with 2.69 s per step after 30 time steps. In Figure 8.18b we see that after 30 time steps $U(20)$ ends up with 20,669,582 edges while the other destination probability distributions end up with lower values at about 14,631,259 to 17,230,300 edges.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

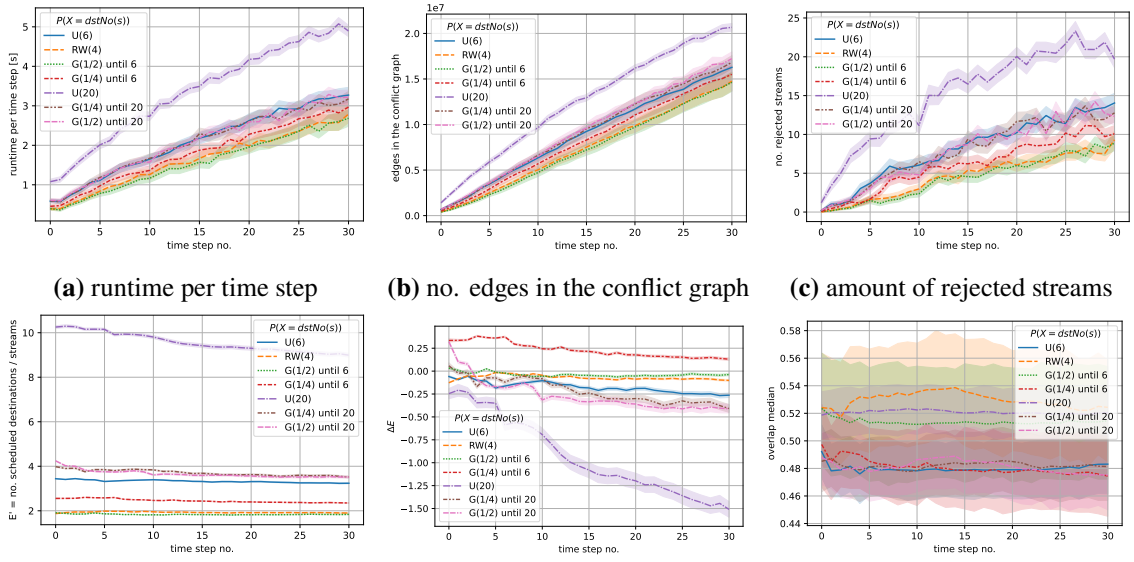
Figure 8.17: Used network topologies test 10

The $U(20)$ and $U(6)$ probability distributions have significantly more multicasts with many destinations and therefore more links in the path-tree. These larger path-trees result in more edges in the conflict graph and a longer runtime compared to the geometric probability distributions. For the $G(\frac{1}{2})$ and $G(\frac{1}{4})$ probability distributions the range of multicasts up to 20 destinations makes the runtime a bit longer than if we allow just multicasts up to 6 destinations.

Next, Figure 8.18c shows the number of rejected streams for the different destination probability distributions after 30 time steps. $U(20)$ rejects 19.70 streams. This is much higher than what the other distributions reject. $U(6)$ rejects 14.05 streams. $G(\frac{1}{2})$, and $G(\frac{1}{4})$ until 20 destinations reject 12.78 streams. $G(\frac{1}{4})$ until 6 destinations rejects 10.08 streams. The least streams are rejected by $G(\frac{1}{2})$ until 6 destinations and $RW(4)$. They reject 8.88 streams.

In Figure 8.18e we can see that all destination probability distributions stay around their accepted amount of destinations per stream except $U(20)$ that deviates up to -1.51 destinations per stream. Because of that the number of destinations per stream in Figure 8.18d are approximately equal to the expected values.

The uniform probability distributions $U(6)$ and $U(20)$ require a lot of overhead in runtime, while still rejecting many streams. For these probability distributions multicast with many destinations are more likely and therefore need to schedule a lot more path-trees with many links that cover a larger portion of the total network. For the geometric probability distribution $G(p)$ a higher p leads to a longer runtime and more rejected streams. Here the same reasoning can be applied since a higher p in $G(p)$ also signals less unicasts and more multicasts with many destinations. The probability distribution from the related work [STP+20] displays a similar result than $G(\frac{1}{2})$ until 6 destinations.



(a) runtime per time step **(b)** no. edges in the conflict graph **(c)** amount of rejected streams

(d) no. destinations per streams **(e)** $\Delta E = E' - E(X)$ **(f)** candidate path-tree overlap

Figure 8.18: Destination probability distributions test 11

In summary, we can conclude that a schedule is harder to find if multicasts with a high amount of destinations become more likely. This leads to a trade-off between multicasts with many destinations in the probability distribution function and runtime as well as the number of rejected streams. The number of destinations per stream does not deviate far away from the expected value even with many multicasts that contain a lot of destinations.

8.8 Results of Multiple Unicasts Compared to Multicast

In this section we evaluate the overhead and scalability of multicasts compared to unicasts. Therefore, we compare only unicasts and multiple unicasts with all multicast and a combination of multicast and unicast. Table 8.9 lists the chosen parameter for this test 12.

The only unicast scenarios (*only_u*) represent the case that multicasting is not needed. Here, we reach only one destination with each stream. The multiple unicast scenarios (*multiple_u*) represent the case that multicasting is implemented on a higher layer. For the data-link layer, a multicast would be mapped to sending unicast frames to each destination. In our case this is realized with a set of unicast stream requests. We want to point out that there is no mechanism in place that makes sure that the set is accepted or rejected as a whole. It can happen that some unicast streams for the multicast are accepted while the others are rejected. So with multiple unicasts we could never be sure that all destinations of a multicast are reached. The only multicast scenarios (*only_m*) represent the case that actual layer two multicast streams are requested. Here, we reach all destinations by sending one frame. In these scenarios we have two and more destinations for every stream. The some multicast scenarios (*some_m*) are supposed to represent more realistic scenarios where multicast and unicast stream requests are possible. Here, we have streams that reach one destination and streams that reach multiple destinations.

parameter name	Test 12
network size n	81
network topologies	all
scenario type	only_u, multiple_u, only_m, some_m
$P(X = dstNo(s))$	$G(\frac{1}{2})$ until 6 and 20, $G(\frac{1}{4})$ until 6 and 20, $U(6)$, $U(20)$, $RW(4)$
number of time steps	30
initial streams	100
$\lvert\text{ReqS}(p_i)\rvert$	50
$\lvert\text{RemS}(p_i)\rvert$	25
number of used threads	6
linear path routing method	Dijkstra Overlap
$n_{linPath}$	6
candidate paths	—
modified Djikstra Overlap	—
intermediate nodes	spanning tree
path-tree destination order	nearest destination and randomly per path-tree
path-tree integration type	end
n_{path}	4
conflict graph builder	random and classic
solver heuristic	GFH-dst-tiebreaker
conflict graph expansion	static
n_{cps}	84

Table 8.9: Test parameter table for unicast to multicast comparison

The first Figure 8.19a shows the runtime of the different scenarios. The *multiple_u* scenarios need a significant longer time than the other scenarios. After 10 time steps it already takes 16.37 s and after 30 time steps it takes 32.51 s per time step. The *only_m* scenarios schedule the requests much faster than the *multiple_u* scenarios. We need up to 3.76 s per time step for *only_m* scenarios after 30 time steps. The *some_m* scenarios take up to 3.26 s per time step. For *only_u* scenarios we have the shortest runtime with 1.96 s after 30 time steps.

The *only_m* scenarios are closer to the *only_u* scenarios than the *multiple_u* scenarios. This means that the extra destinations we have for our multicasts do not add too much to the total runtime. Additionally, if we want to estimate the total runtime per step we can just do that based on the number of streams and do not need to specifically consider the destinations. As expected the *some_m* scenarios are between the *only_m* and *only_u* scenarios.

Second, let us look at the edges in the conflict graph. Figure 8.19b displays that the *multiple_u* scenarios needs 73,563,354 edges after 30 time steps. This is more than four times as many as the *only_m* scenarios with 17,517,612 edges and the *some_m* scenarios with 16,536,779 edges. For *only_u* scenarios 12,045,043 edges are sufficient after 30 time steps.

The planner program has n_{cps} times the amount of streams nodes in the conflict graph. A larger conflict graph also means more spatial and temporal overlap checks. This leads to the higher runtime of the *multiple_u* scenarios. If the spatial or temporal overlap occurs at the start of the path-tree in *only_m* scenarios just one edge must be included in the conflict graph while the *multiple_u* scenarios add several edges. This is necessary since *multiple_u* scenarios look at the linear path-trees completely independent of each other. This characteristic explains the high amount of edges in the *multiple_u* scenarios.

Next, we look at the results in Figure 8.19c regarding the amount of rejected streams. Additionally, we present the amount of rejected destinations in Figure 8.19f since the amount of destinations for the same stream differs between the scenarios. The *multiple_u* scenarios reject the most streams (73.18 after 30 time steps) while the *only_m* scenarios reject less with only 15.67 rejected streams after 30 time steps. On the other hand, the *only_m* scenarios reject 17.24 more destinations than the *multiple_u* scenarios which reject 90.42 destinations after 30 time steps. The *only_u* scenarios reject 4.77 streams and the same number of destinations.

Figure 8.19e presents the number of scheduled destinations in the traffic plan. The *only_u* scenarios manage to include 885.15 destinations in the schedule of the traffic plan. We can see that the *multiple_u* and the *only_m* scenarios both manage to schedule much more destinations with around 3,361.51 and 3,077.59 destinations after 30 time steps.

The *only_m* scenarios reject fewer streams but more destinations than the *multiple_u* scenarios since the rejected streams are multicast streams that reach more than one destination. The amount of rejected destinations is lower for the *multiple_u* scenarios since it can accept some of the streams that are part of a multicast.

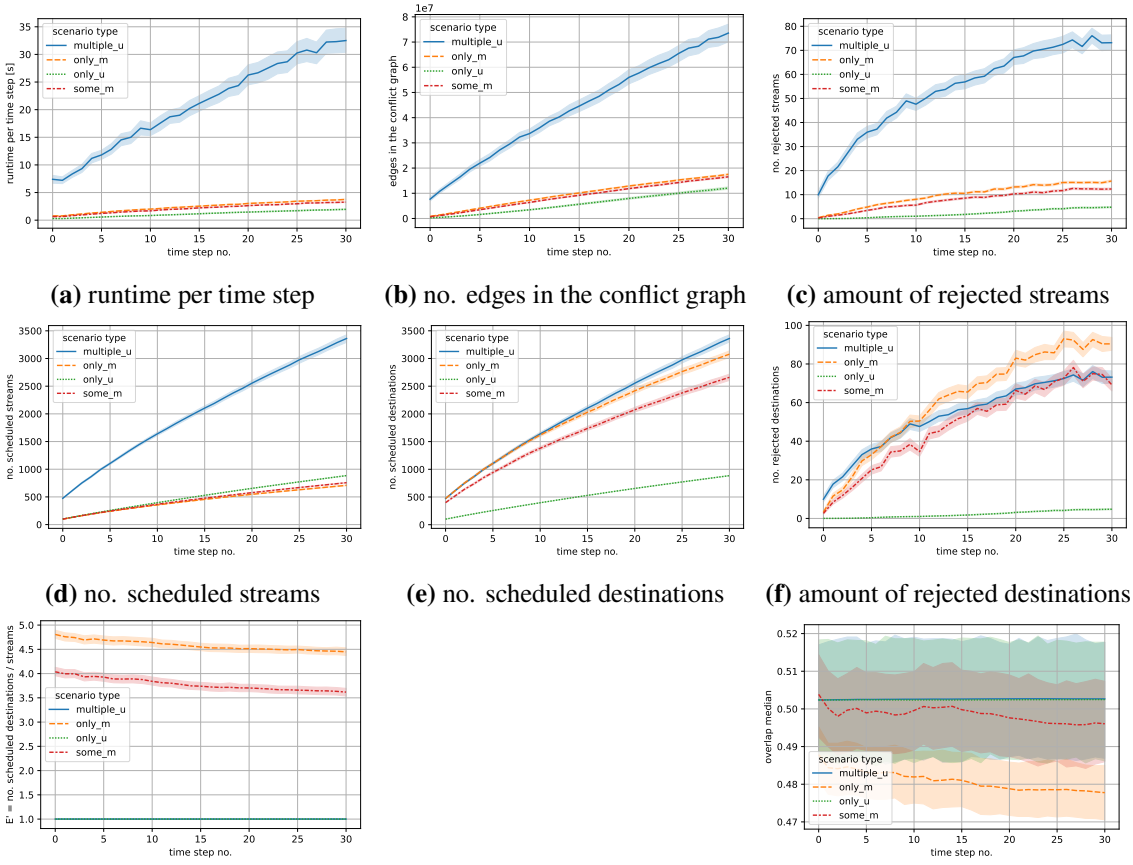
Figure 8.19g displays the number of destinations per stream. The *multiple_u* and *only_u* scenarios are constant at one destination per stream. The *only_m* scenarios have around 4.45 destinations per stream. The *some_m* scenarios start at 4.04 destinations per stream but fall down to 3.62 destinations per stream after 30 time steps.

The *multiple_u* scenarios can fit their set of unicast streams partially into the traffic plan and therefore is capable of scheduling streams that lead to the same amount of scheduled destinations compared to the *only_m* scenarios. Overall we conclude that it is highly beneficial for the runtime and the conflict graph to use *only_m* scenarios or the more realistic *some_m* scenarios. For this reason, multicasts should be directly implemented as multicasts on the data-link layer.

8.9 Discussion

In this section, we want to summarize and discuss the results of all evaluations we made.

The *least links* and *similar start* methods are the best of the candidate path selection methods. Both are faster than the *index based* method. The *similar start* method has the best number of rejected streams, while the *least links* method has the better number of destinations per stream. Our other candidate path selection heuristics do not bring any benefit due to the short path-trees. The leading result of the modified Dijkstra Overlap methods is achieved by the *zero* heuristic that maximizes the temporal link weight effect for the overall lowest candidate path-tree overlap. For the intermediate node distance, the *spanning tree* heuristic performs the best because the runtime per step is low.



(a) runtime per time step

(b) no. edges in the conflict graph

(c) amount of rejected streams

(d) no. scheduled streams

(e) no. scheduled destinations

(f) amount of rejected destinations

(g) no. destinations per streams

(h) candidate path-tree overlap

Figure 8.19: Unicast to multicast comparison test 12

Regarding the ordering approaches, the *randomly per path-tree* approach has the fastest runtime. As deterministic ordering, we recommend taking the *nearest destination* approach. For the linear path integration, the *end* approach should be used.

We see that a high n_{cps} helps a lot to reduce the number of rejected streams while causing a significantly longer runtime. On the other hand, having multiple candidate path-trees does not reduce the amount of rejected streams. The multicast streams are mainly rejected because of links that are necessary for all candidate path-trees.

The different GFH adaptations for multicast which we examine show similar results compared to the original GFH but have the advantage to consider the optimization function for multicasts. We are able to show that also for multicast the dynamic planner program has an advantage in the runtime to calculate the next traffic plan for large amounts of streams.

It is necessary to mention that a topology with a low diameter and many links for alternative routes helps to create different configurations which can be processed quickly. With the multicast destination probability distributions, we show that the solver admits fewer streams the more multicasts with high amounts of destinations are requested. If the number of destinations for all multicasts is too high, a broadcast approach should be considered. Using multicast has a significantly better runtime than multiple unicasts and manages to schedule the same amount of destinations.

The amount of links used by all scheduled streams has an influence on different metrics like the runtime and number of edges in the conflict graph. The amount of links is higher for multicasts in comparison to unicasts, but it manages to reduce the number of links in comparison to multiple unicast stream sets. At several points during our evaluation we argue that the runtime is strongly influenced by the amount of links that have to be checked for spatial and temporal overlaps to construct the conflict graph. The construction overhead of conflict graph-based approaches is among other things also described by Geppert et al. [GDBR23].

It is difficult to compare the various related works with our results because they use different network sizes, scenarios and hardware. Nevertheless, we compare them with reservations. In contrast to Schweissguth et al. [STP+20] that need up to 3.45x the only unicasts runtime for multicasts our approach schedules multicasts in less than twice the time of *only_u* scenarios. Additionally, we schedule noticeable more streams with less runtime. Yu and Gu [YG20] reach a maximum of 450 scheduled streams for networks with 24 switches in 83 minutes. We evaluate networks containing 81 switches and thereby we are able to statically schedule over 1000 streams in under two minutes. In our dynamic evaluation we schedule 708.70 streams after 30 time steps without reaching the limit of the solver where every new requested stream is rejected. The approach of Santos et al. [SSN19] takes about two hours to schedule streams to 40 destinations which is slower and more limited in the number of destinations they could schedule. Besides that we are more flexible in how many hops our path-trees take. Our approach has a similar runtime after zero time steps for 100 streams as the approach of Pahlevan et al. [PTO19].

9 Conclusion and Outlook

The last chapter contains a short summary and conclusion of this paper. Afterwards, we share an outlook on potential future work. Furthermore, we include some considerations for a different spatial and temporal overlap calculation.

Conclusion

In this thesis, we look at the joint routing and scheduling problem. We consider multicast time-triggered streams for the dynamic conflict graph-based solver. The traffic plan is calculated with an independent colorful set. To fit multicast, the stream routes are extended to path-trees. We describe how to generate the candidate path-trees with the intermediate node distance and other algorithms. The intermediate node distance method constructs path-trees by selecting for each destination a direct linear path between the already existing intermediate nodes and the destination.

Based on the path-tree construction method evaluation, we choose the intermediate node distance method for further investigations. These additional evaluations show that higher numbers of configurations allow more scheduling options per stream and result in less rejected streams. The multicast streams are mainly rejected because of links that are necessary in all candidate path-trees. The more multicasts with high amounts of destinations are requested the more streams do not fit in the traffic plan. Multicast streams cannot be admitted for the next constructed traffic plan if the links are spatially or temporally overlapping with other streams.

The spatial and temporal overlap calculation is the major contributor to the runtime. This runtime scales according to the increasing number of links that are compared for potential overlaps. Using multicast has a significantly better runtime than multiple unicasts. We confirm that our data-link layer multicasts have a small runtime and conflict graph overhead compared to single unicasts. This is achieved by a low number of additional links per destination in the path-trees.

We can statically route and schedule 842 multicast streams in 25 seconds for networks with 81 nodes. Our dynamic intermediate node distance algorithm ends up with 708.70 scheduled multicast streams that lead to 3,077.59 destinations after 30 time steps. Traffic plan updates are calculated in less than 3.76 s. Our multicast streams in the traffic plan contain 4.45 destinations per stream. We also look at a more realistic scenario containing multicast and unicast streams with an average of 3.62 destinations per stream. For this scenario we need up to 3.26 s to generate the next traffic plan that contains 757.26 streams.

Outlook

There are a variety of topics to work on in the future.

One way to directly extend the work of this paper would be to continue with further evaluations. We evaluated the intermediate node distance method in detail. Such a detailed evaluation can be done as well for the modified Dijkstra Overlap with the zero approach, the candidate path construction method with the similar start base, and the least links base approach. The expected outcome is a similar result compared to the intermediate node distance method.

We evaluated that a high number of candidate path-trees does not improve the schedulability of streams. Therefore, we propose to drop the KSP like approach and instead just find two completely different routes. For the first route, some shortest spanning tree algorithm would be interesting, and for the second route, a path-tree that has a minimal overlap with the spanning tree would act as a fallback option. This second path-tree could for example also focus on minimizing the total load on each link.

If we fill a network with streams until we reach the total capacity, there will be sources sending different streams. Routing each stream produces side information in Dijkstra's algorithm, which we currently just throw away. An alternative could be to store a shortest spanning tree for every source and reuse that tree for each stream that starts at this node. To make this scalable for large networks, we would recommend starting Dijkstra's algorithm as soon as the first stream of a source is requested. Instead of just remembering the path-tree to the destinations of the requested stream, the entire state of the algorithm is frozen and stored. The next stream request can directly read the path-tree out of the frozen algorithm. If a stream contains a destination that is not reachable at this point, Dijkstra's algorithm could continue at the point it left off. This routing approach would produce path-trees similar to broadcast routing approaches, and streams would just select what portion of the path-tree they need to use. This would be a good idea if each sender deals with many streams, and having multiple candidate path-trees brings no benefits.

When a destination is added to a stream, we currently discard the stream and request a new stream with the extra destination. This is a straight forward solution. However, it might be beneficial to make single destinations subscribe to streams like in PubSub systems. This would make sense if it could bypass routing and scheduling the entire stream again.

The domain of wireless networks was completely ignored so far. It would be interesting to see if the solver could be applied to static wireless networks as well. The other way around, it would also be interesting if the solver could work with dynamic network topologies. Currently, if we change the network, we have to shut down the solver and restart a new one with the changed topology.

Spatial and Temporal Overlap Calculation Considerations

Since we build on top of the work of Falk et al. [FGD+22], we also have some thoughts on how to improve the spatial and temporal overlap calculations.

We want to suggest them to try out if a hierarchy of graphs makes the solver more scalable and enables them to use larger networks. They would need an abstracted version of the network that hides simple subnetworks in some nodes. This abstract network is hard to calculate, but the calculation

needs to be done just once. Then they could calculate the spatial and temporal separation on the higher hierarchy first. The assumption is that in the higher hierarchy there are fewer links to check against each other, and therefore it is more scalable. Only if a collision occurs, we have to check spatial and temporal separation in the original network. This check could be minimized to only include the subnetwork the collision occurs in. We got this idea from Yu and Gu [YG20], where they prune and group links in the network first before they schedule streams. This concept would pay off if it is unlikely for a collision to happen in the abstract network. This is especially the case if they deal with large networks containing streams that are in completely different regions of the network.

We were not able to find a path-tree spatial overlap algorithm with a lower runtime complexity compared to the current code. However, we found that the temporal separation algorithm could potentially be made more scalable by using interval trees [SI87], a binary search, or traversing linear through both sorted lists. Currently, the respective loop iterating over all combinations of overlapping links is made fast by parallelizing the checks.

Another way of thinking would be to calculate the spatial and temporal overlap in a lazy manner. The idea is to change the spatial separation graphs to graphs that contain a positive edge if they are separated, a negative edge if they have an overlap, and no edge between configurations if the overlap has not yet been checked. This should be combined with saving the overlapping links instead of recalculating the overlap during the temporal separation. With such a graph, we are able to only calculate overlaps of configurations we are interested in instead of checking every combination during the integration. In contrast, this would require changes regarding the solver heuristic.

Bibliography

[14] "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture". In: *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)* (2014), pp. 1–74. DOI: 10.1109/IEEESTD.2014.6847097 (cit. on p. 19).

[16] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic". In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), pp. 1–57. DOI: 10.1109/IEEESTD.2016.8613095 (cit. on p. 17).

[17] "IEEE Standard for Local and Metropolitan Area Networks:Overview and Architecture–Amendment 2: Local Medium Access Control (MAC) Address Usage". In: *IEEE Std 802c-2017 (Amendment to IEEE Std 802-2014 as amended by IEEE Std 802d-2017)* (2017), pp. 1–26. DOI: 10.1109/IEEESTD.2017.8016709 (cit. on p. 19).

[88] *Distance Vector Multicast Routing Protocol*. RFC 1075. Nov. 1988. DOI: 10.17487/RFC1075. URL: https://www.rfc-editor.org/info/rfc1075 (cit. on p. 19).

[BU18] S. Brooks, E. Uludag. *Time-sensitive networking: From theory to implementation in industrial automation*. Tech. rep. 2018 (cit. on p. 24).

[CCCL20] B. Y. Chen, X.-W. Chen, H.-P. Chen, W. Lam. "Efficient algorithm for finding K shortest paths based on re-optimization technique". In: *Transportation Research Part E Logistics and Transportation Review* 133 (Jan. 2020), p. 101819. DOI: 10.1016/j.tre.2019.11.013 (cit. on p. 25).

[CCCL21] B. Y. Chen, X.-W. Chen, H.-P. Chen, W. Lam. "A fast algorithm for finding K shortest paths using generalized spur path reuse technique". In: *Transactions in GIS* 25 (Feb. 2021), pp. 516–533. DOI: 10.1111/tgis.12699 (cit. on p. 25).

[dev23] intel tbb developers. *Intel oneAPI Threading Building Blocks*. oneTBB. Version 2021.9.0. May 26, 2023. URL: https://github.com/oneapi-src/oneTBB (visited on 05/26/2023) (cit. on p. 56).

[DN16] F. Dürr, 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* (2016) (cit. on pp. 18, 27, 29).

[Epp94] D. Eppstein. "Finding the k shortest paths". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 154–165 (cit. on p. 25).

[FDR20] J. Falk, F. Dürr, 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. DOI: 10.1109/RTAS48715.2020.00-12 (cit. on pp. 23, 27).

[FGD+22] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows in the Real-Time Data Plane”. In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1807–1825. DOI: 10.1109/TNSM.2022.3150664 (cit. on pp. 16, 21, 23, 24, 27, 33, 51, 54, 56, 70, 88).

[GDBR23] H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. *Just a Second – Scheduling Thousands of Time-Triggered Streams in Large-Scale Networks*. 2023. arXiv: 2306.07710 [cs.NI] (cit. on pp. 25, 42, 85).

[HMS07] J. Hershberger, M. Maxel, S. Suri. “Finding the k Shortest Simple Paths: A New Algorithm and Its Implementation”. In: *ACM Trans. Algorithms* 3.4 (Nov. 2007), 45–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290682. URL: https://doi.org/10.1145/1290672.1290682 (cit. on p. 25).

[HSS08] A. A. Hagberg, D. A. Schult, P. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: 2008 (cit. on p. 54).

[KSSO19] S. Kotachi, T. Sato, R. Shinkuma, E. Oki. “Multicast Routing Model to Minimize Number of Flow Entries in Software-Defined Network”. In: *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2019, pp. 1–6. DOI: 10.23919/APNOMS.2019.8893074 (cit. on p. 24).

[Mar22] R. Marks. “Link-Layer Software-Defined Addressing using Block Address Registration and Claiming (BARC)”. In: (Feb. 2022). DOI: 10.36227/techrxiv.19105118.v1. URL: https://www.techrxiv.org/articles/preprint/Link-Layer_Software-Defined_Addressing_using_Block_Address_Registration_and_Claiming_BARC_/19105118 (cit. on p. 24).

[MB08] L. de Moura, N. Bjørner. “Z3: an efficient SMT solver”. In: vol. 4963. Apr. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 24).

[Moy94] J. Moy. *Multicast Extensions to OSPF*. RFC 1584. Mar. 1994. DOI: 10.17487/RFC1584. URL: https://www.rfc-editor.org/info/rfc1584 (cit. on p. 19).

[NAS05] J. Nicholas, A. Adams, W. Siadak. *Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)*. RFC 3973. Jan. 2005. DOI: 10.17487/RFC3973. URL: https://www.rfc-editor.org/info/rfc3973 (cit. on p. 24).

[PTO19] M. Pahlevan, N. Tabassam, R. Obermaisser. “Heuristic List Scheduler for Time Triggered Traffic in Time Sensitive Networks”. In: *SIGBED Rev.* 16.1 (Feb. 2019), pp. 15–20. DOI: 10.1145/3314206.3314208. URL: https://doi.org/10.1145/3314206.3314208 (cit. on pp. 24, 85).

[QP03] E. de Queirós Vieira Martins, M. M. B. Pascoal. “A new implementation of Yen’s ranking loopless paths algorithm”. In: *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1 (2003), pp. 121–133 (cit. on p. 25).

[SCS18] R. Serna Oliver, S. S. Craciunas, 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. DOI: 10.1109/RTAS.2018.00008 (cit. on p. 27).

[SDT+17] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, 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*. RTNS '17. Grenoble, France: Association for Computing Machinery, 2017, pp. 8–17. ISBN: 9781450352864. DOI: 10.1145/3139258.3139289. URL: https://doi.org/10.1145/3139258.3139289 (cit. on p. 23).

[SI87] M. Sibuya, Y. Itoh. "Random sequential bisection and its associated binary tree". In: *Annals of the Institute of Statistical Mathematics* 39 (1987), pp. 69–84 (cit. on p. 89).

[SSN19] A. C. T. d. Santos, B. Schneider, V. Nigam. "TSNSCHED: Automated Schedule Generation for Time Sensitive Networking". In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 69–77. DOI: 10.23919/FMCAD.2019.8894249 (cit. on pp. 24, 85).

[STP+20] E. Schweissguth, D. Timmermann, H. Parzyjegla, P. Danielis, G. Mühl. "ILP-Based Routing and Scheduling of Multicast Realtime Traffic in Time-Sensitive Networks". In: *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2020, pp. 1–11. DOI: 10.1109/RTCSA50079.2020.9203662 (cit. on pp. 16, 23, 55, 80, 85).

[THM+08] D. Traskov, M. Heindlmaier, M. Medard, R. Koetter, D. S. Lun. "Scheduling for Network Coded Multicast: A Conflict Graph Formulation". In: *2008 IEEE Globecom Workshops*. 2008, pp. 1–5. DOI: 10.1109/GLOCOMW.2008.ECP.96 (cit. on p. 23).

[Yen71] J. Y. Yen. "Finding the K Shortest Loopless Paths in a Network". In: *Management Science* 17 (1971), pp. 712–716 (cit. on p. 25).

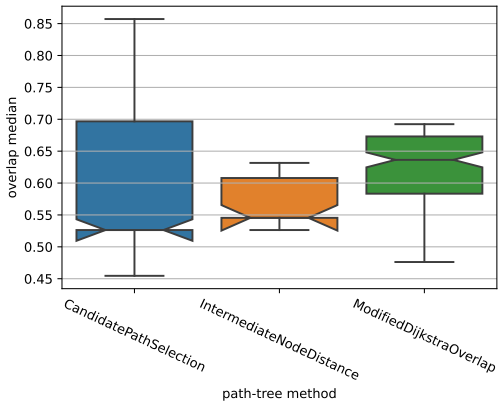
[YG20] Q. Yu, M. Gu. "Adaptive Group Routing and Scheduling in Multicast Time-Sensitive Networks". In: *IEEE Access* 8 (2020), pp. 37855–37865. DOI: 10.1109/ACCESS.2020.2974580 (cit. on pp. 16, 24, 85, 89).

[ZSQ23] X. Zhang, D. Simsek, M. D. L. Quintana. "Candidate Path Routing Investigation forTime-triggered Scheduling Problems". In: (May 30, 2023) (cit. on p. 25).

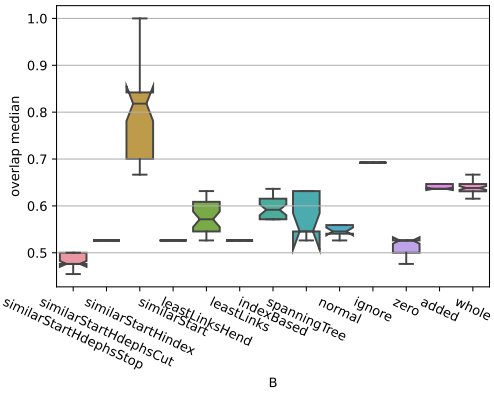
All links were last followed on July 01, 2023.

A Theoretical Example Candidate Path-trees

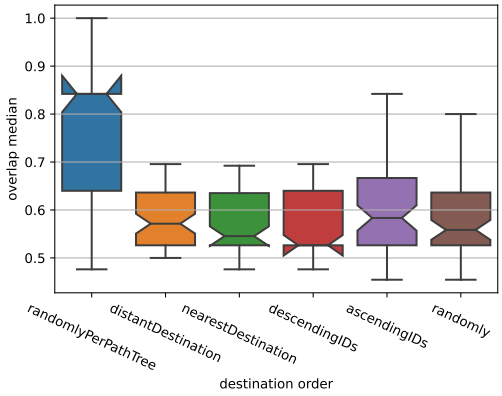
We separately evaluate a scenario for the candidate path-tree construction example in Chapter 6 with the corresponding network. On this specific network topology, we use a scenario with one time step. Here, just one multicast stream from the source on the bottom reaching all three destinations at the top is requested. Figure A.1 presents the overlaps of this example. Figure A.2 shows the candidate path-trees the planer program returns. Be aware, that this network is too small to see the same overlap results that arise from larger networks.



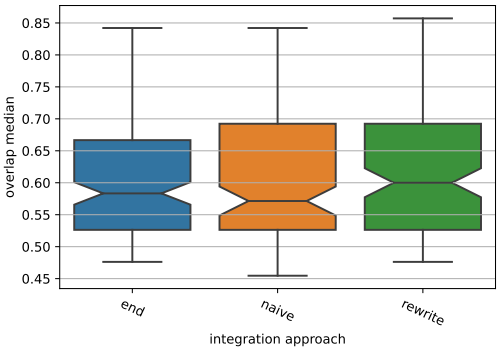
(a) General construction method



(b) Detailed construction method

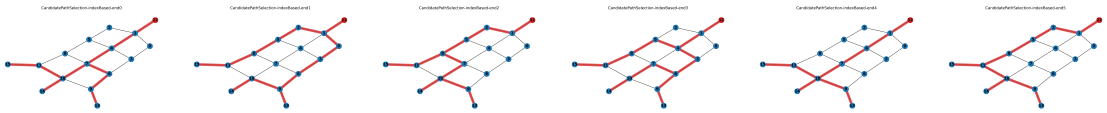


(c) Destination order method

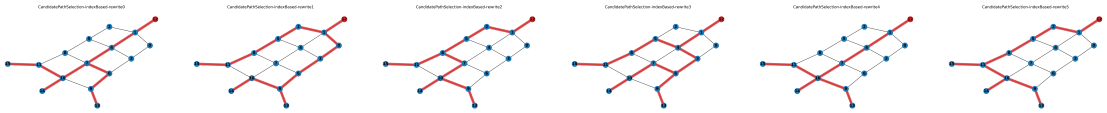


(d) Integration approaches

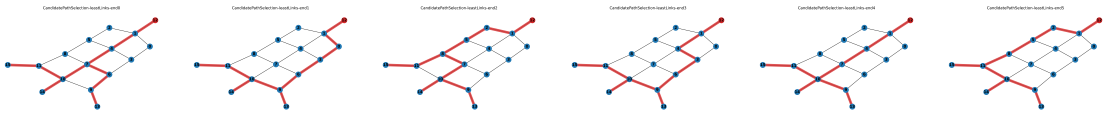
Figure A.1: Overlap results of theoretical example



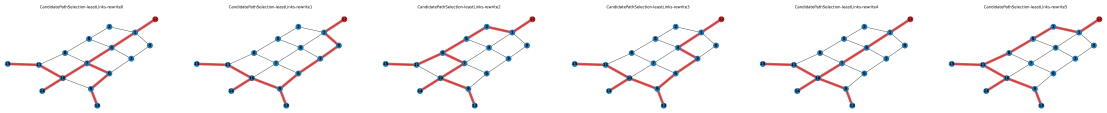
(a) Candidate path-trees of the candidate paths selection index based method with end integration



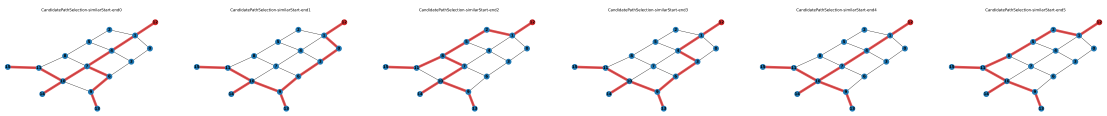
(b) Candidate path-trees of the candidate paths selection index based method with rewrite integration



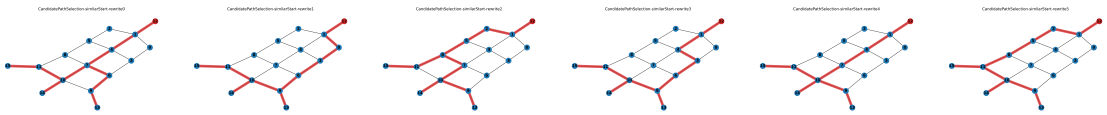
(c) Candidate path-trees of the candidate paths selection least links method with end integration



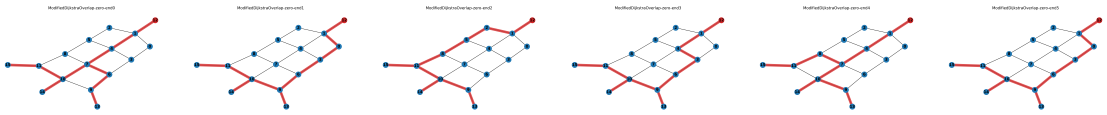
(d) Candidate path-trees of the candidate paths selection least links method with rewrite integration



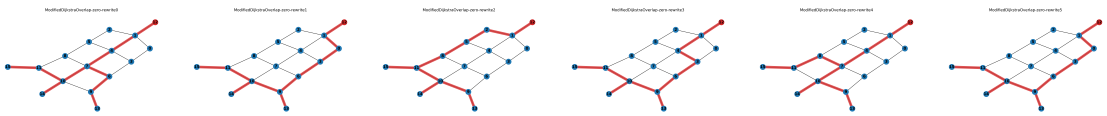
(e) Candidate path-trees of the candidate paths selection similar start method with end integration



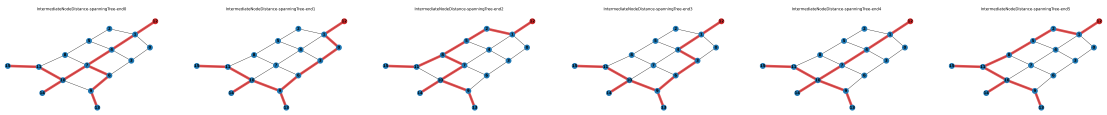
(f) Candidate path-trees of the candidate paths selection similar start method with rewrite integration



(g) Candidate path-trees of the modified Dijkstra Overlap zero method with end integration



(h) Candidate path-trees of the modified Dijkstra Overlap zero method rewrite integration



(i) Candidate path-trees of the intermediate node distance spanning tree method

Figure A.2: Candidate path-tree results of theoretical example

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature