Every Node for Itself: Fully Distributed Service Coordination

Stefan Schneider, Lars Dietrich Klenner, and Holger Karl
Paderborn University, Germany
{stschn, lklenner, hkarl}@mail.upb.de

***Abstract*—Modern services consist of modular, interconnected components, e.g., microservices forming a service mesh. To dynamically adjust to ever-changing service demands, service components have to be instantiated on nodes across the network. Incoming flows requesting a service then need to be routed through the deployed instances while considering node and link capacities. Ultimately, the goal is to maximize the successfully served flows and Quality of Service (QoS) through online service coordination. Current approaches for service coordination are usually centralized, assuming up-to-date global knowledge and making global decisions for all nodes in the network. Such global knowledge and centralized decisions are not realistic in practical large-scale networks.**

To solve this problem, we propose two algorithms for fully distributed service coordination. The proposed algorithms can be executed individually at each node in parallel and require only very limited global knowledge. We compare and evaluate both algorithms with a state-of-the-art centralized approach in extensive simulations on a large-scale, real-world network topology. Our results indicate that the two algorithms can compete with centralized approaches in terms of solution quality but require less global knowledge and are magnitudes faster (more than 100x).

I. Introduction

There is a rising demand of services consisting of multiple interconnected components, e.g., microservices in a service mesh or chained virtual network functions (VNFs) in network function virtualization (NFV) [1]. These services can scale flexibly by instantiating service components according to current demand. Such instances can run independently on any compute node in the network and process incoming flows.

The goal of service coordination is to ensure that flows requesting a service are processed successfully by traversing instances of all service components. Additionally, flows should complete with short end-to-end delay to ensure good Quality of Service (QoS). To this end, the requested services need to be scaled and their instances placed in the network, i.e., we have to decide how often and where to instantiate service components. Furthermore, incoming flows need to be routed from their ingress nodes through these deployed instances and finally to their egress nodes. In doing so, node and link capacities need to be respected and link delays should be considered.

Current approaches mostly coordinate services globally, i.e., they make centralized decisions for all flows and nodes and rely on global knowledge [2]–[5]. Since service demands of incoming flows and resource utilization can change frequently, up-to-date global knowledge is not realistic. In practice, up-to-date global knowledge would require prohibitive overhead. Besides, due to their time complexity, centralized decisions are often too slow for practical large-scale networks with many nodes and rapidly arriving or departing flows.

To address these issues, we propose two fully distributed approaches for online service coordination. These approaches can be executed individually by each node in the network, are simple and fast, and only require very limited knowledge. As there is no single coordinator that could fail or become disconnected, they are more robust to failures than existing centralized approaches. These properties make them ideal for practical large-scale networks with many flows. One algorithm is greedy in that nodes process incoming flows locally if possible and forward them along the shortest path to their egress nodes. The other algorithm leverages more available knowledge (e.g., utilization of neighboring nodes) to identify suitable nodes for processing incoming flows. Rather than processing all flows on the shortest path from ingress to egress, it distributes load to avoid congestion.

Overall, our contributions are:

- We formalize the problem of online service coordination in networks with limited node and link capacity.
- We propose two novel, fully distributed approaches for online service coordination. The algorithms can be executed individually by each node in the network, are simple and fast, and only require limited global knowledge.
- In our evaluation based on a large-scale, real-world network topology, we show that the proposed fully distributed algorithms can compete with a state-of-the-art centralized coordination approach but require less global knowledge and are magnitudes faster.
- We make our code publicly available on GitHub [6] to encourage reproduction of results and reuse.

II. Related work

There has been significant research interest in service coordination, e.g., in the context of edge or cloud computing [7], [8] and NFV [9]. Most authors propose centralized approaches that require global knowledge and make global decisions,

using optimization solvers and/or heuristic algorithms [2]–[5], [10]–[12]. For example, Luizelli et al. [12] propose an efficient heuristic for large-scale networks using a variable neighborhood search metaheuristic. Nevertheless, the authors still rely on centralized knowledge and coordinate services offline. These limitations make their heuristic unsuitable for practical large-scale networks with constantly fluctuating load, which require fast, online service coordination. In contrast, our fully distributed approaches do not have these limitations and are suitable for fast, online service coordination.

Virtual network embedding (VNE) is a well researched problem related to service coordination [13], where authors have proposed distributed approaches [14]–[18]. For example, Houidi et al. [15] split the network into different clusters. Similar to our approach, they process multiple incoming flows in parallel. Likewise, other authors [16]–[18] split the network into different clusters that are coordinated independently by separate coordinators. Still, each cluster is coordinated centrally by a single coordinator. In contrast, our approaches fully distribute service coordination to each node in the network without any centralized coordinating entity. This allows even more parallelization and faster processing of flows. It also requires less knowledge for service coordination and is more robust since there is no single coordinator that could fail. Moreover, the most common VNE variants only focus on placement of already scaled and chained services but disregard scaling and online flow scheduling [13]. We consider online service coordination including scaling, placement, and flow scheduling.

III. Problem Statement

We address the problem of coordinating services online over discrete time steps $t \in T$. The problem can be formalized as follows.

A. Problem Inputs

We consider a substrate network $G = (V, L)$ of distributed nodes connected by undirected links. Each node $v \in V$ has a generic compute capacity $\text{cap}_v \in \mathbb{R}_{\geq 0}$ (e.g., CPU). This could easily be extended to multiple different resource types. Each link $l = (v, v') \in L$ connects two nodes v and v' bidirectionally with a certain delay $d_l \in \mathbb{R}_{>0}$ and maximum data rate $\text{cap}_l \in \mathbb{R}_{>0}$ that is shared between both directions. Let $L_v \subseteq L$ be the subset of links leaving node v and $V_v \subseteq V$ the set of v's neighbors that can be reached following a link in L_v.

Incoming flows may request multiple different services. Each service $s \in S$ consists of a chain of components $C_s = \langle c_1, ..., c_{n_s} \rangle$. Each service component $c \in C_s$ can be instantiated multiple times at different nodes in the network. All instances of a component are identical and independent of each other. Service requests arrive in form of flows at geographically distributed ingress nodes in the network. Each flow $f = (s_f, c_f, v_f^{\text{in}}, v_f^{\text{eg}}, \lambda_f, t_f, \delta_f, m_f) \in F$ is defined by

- the service s_f it requests,
- its currently requested component $c_f \in C_s \cup \{\varnothing\}$ to keep track of the flow's current processing state (cf. network service header [19]),
- its ingress and egress node v_f^{in} and v_f^{eg},
- its requested data rate λ_f at this point, which may change when traversing components (e.g., a compression function),
- its time of arrival t_f, its duration δ_f,
- and optional metadata m_f, which can be used for coordination.

All nodes can be ingress or egress nodes. To successfully complete a flow, it has to traverse instances of all service components in order and then leave the network at the egress node. After traversing the last component c_{n_s}, the flow finished processing and needs to be routed to its egress node; this is the case once $c_f = \varnothing$. When an instance of component c processes a flow, the flow incurs a processing delay $d_c \in \mathbb{R}_{\geq 0}$, which may be fixed or randomly distributed. We do not consider queuing delays here, but they could be added to d_c in a similar fashion.

B. Decision Variables

In online service coordination, scaling and placement of service components as well as routing and processing of flows needs to be decided for each time step t. To this end, we introduce decision variables $x_{c,v}(t)$ and $y_{f,c,v}(t)$. Binary variable $x_{c,v}(t) \in \{0, 1\}$ denotes whether to place an instance of component c at node v at time t (placement). The same component can be placed at none, one, or multiple nodes (scaling). We assume a serverless computation environment where we only decide at which nodes to place a certain component c (inter-node). *Within* a node v (intra-node), we assume the operating system or a system like Kubernetes [20] to start and scale instances of c transparently if $x_{c,v}(t) = 1$. The system may internally start multiple instances of c within v, e.g., depending on whether v is a single machine or a cluster. We assume such intra-node scaling to be transparent and out of scope for our inter-node service coordination problem.

Rather than having fixed resource requirements, component instances require resources proportional to the total data rate of flows they are currently processing. The duration for completely processing a flow f at an instance of c depends on the component's delay d_c and the flow's duration δ_f, during which resources proportional to data rate λ_f are required.

In addition to scaling and placement, incoming flows need to be routed from their ingress, through the requested instances, to their egress node. Variable $y_{f,c,v}(t) \in V_v \cup \{v\}$ indicates how to route and process flow f requesting component c and arriving at node v. If $y_{f,c,v}(t) = v$, it means v processes f locally at its instance of c at time t. This is only possible if

an instance of c is placed at v and v has enough remaining resources to process f. If $y_{f,c,v}(t) = v' \in V_v$, it indicates that v does not process f locally at an instance of c but, instead, forwards f to its neighbor v' (routing). Forwarding f along link $l = (v, v') \in L_v$ is only possible if it does not exceed l's maximum data rate cap_l.

We denote the currently utilized resources of node v with $r_v \leq \text{cap}_v$ and of link l with $r_l \leq \text{cap}_l$. This utilization depends on the coordination decisions $y_{f,c,v}(t)$ and the flow data rate λ_f and duration δ_f.

C. Objectives

The goal of online service coordination is to set variables $x_{c,v}(t)$ and $y_{f,c,v}(t)$ such that incoming flows are processed successfully and with short end-to-end delay. We formalize this high-level goal as two objectives o_f and o_d:

$$\max o_f = \frac{|F_{\text{succ}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \tag{1}$$

$$\min o_d = \frac{1}{|F_{\text{succ}}|} \sum_{f \in F_{\text{succ}}} \sum_{c \in C_{s_f}} d_c + \sum_{\substack{(v,v')=l \in L, \\ c \in C_{s_f}, t \in T}} \mathbb{1}_{\{y_{f,c,v}(t)=v'\}} d_l \tag{2}$$

Maximizing objective o_f means to process as many flows successfully (F_{succ}) as possible, avoiding dropped flows (F_{drop}), thus maximizing the percentage of successful flows (eq. 1). To avoid dropping flows due to lack of available node or link capacities, load should be balanced across multiple nodes and links according to their capacities.

At the same time, the goal is to minimize o_d, which is the end-to-end delay averaged over all successful flows (eq. 2). The end-to-end delay of a flow f consists of two parts. First, the sum of processing delays d_c of components c whose instance f traversed. And second, the sum of link delays d_l that f experienced during routing. In eq. 2, $\mathbb{1}_{\{y_{f,c,v}(t)=v'\}}$ is an indicator variable that is 1 if f traversed link $l = (v, v')$ and 0 otherwise.

Note that the two objectives o_f and o_d may be conflicting. For example, distributing flows over more nodes and links to balance the load helps with processing more flows successfully (improves o_f) but also leads to longer paths and higher end-to-end delays (degrades o_d). In our algorithms, we approach this trade off by optimizing o_f and o_d in lexicographical order, i.e., prioritizing o_f but still trying to optimize o_d as far as possible.

IV. Fully Distributed Service Coordination

To solve the online service coordination problem of Sec. III, we propose two fully distributed algorithms. Both algorithms are executed independently in parallel on each node in the network. Nodes hence rapidly decide how to treat incoming flows. The algorithms require knowledge of the network topology including link delays, which we assume to be rather static and globally available. The algorithms, however, do not rely on full up-to-date, global knowledge of fast changing information such as the current utilization of all nodes and links in the network. Instead, GCASP (Sec. IV-A) does not require any further global information and SBC (Sec. IV-B) can be configured to leverage any useful information that is available.

Algorithm 1 GCASP Algorithm

```
1:  procedure GCASP(v, f)
2:      while r_v < cap_v and c_f ≠ ∅ do
3:          x_{v,c_f}(t) ← 1
4:          y_{f,c_f,v}(t) ← v
5:      if v = v_f^in then
6:          m_f.dst ← v_f^eg
7:          m_f.path ← shortest_path(v, m_f.dst, L)
8:      if v = m_f.dest and c_f ≠ ∅ then
9:          m_f.dst ← random_choice(V)
10:         m_f.path ← shortest_path(v, m_f.dst, L)
11:     if c_f = ∅ and m_f.dst ≠ v_f^eg then
12:         m_f.dst ← v_f^eg
13:         m_f.path ← shortest_path(v, m_f.dst, L)
14:     FORWARD_FLOW(v, f, L)
```

Algorithm 2 Flow forwarding using adaptive shortest paths

```
1:  procedure FORWARD_FLOW(v, f, L)
2:      v' ← m_f.path.pop()
3:      if r_(v,v') + λ_f ≤ cap_(v,v') then
4:          y_{f,c_f,v}(t) ← v'
5:      else
6:          L_free ← L \ {l ∈ L_v | r_l + λ_f > cap_l}
7:          m_f.path ← shortest_path(v, m_f.dst, L_free)
8:          y_{f,c_f,v}(t) ← m_f.path.pop()
```

A. Greedy Coordination with Adaptive Shortest Paths

The main design goals for Greedy Coordination with Adaptive Shortest Paths (GCASP) were to be simple, effective, fast, and frugal, in that it works without any global up-to-date information about node and link utilization. Instead, each node only knows the utilization of its own compute resources and outgoing links. Nodes process flows greedily and forward them along the shortest path to their egress nodes, minimizing end-to-end delay. To avoid dropping flows due to congestion, GCASP adjusts routing when a link is congested, i.e., sending a flow via the link would violate its capacity cap_l.

1) Algorithm: Alg. 1 shows the GCASP algorithm, which is run by a node v once a flow f starts arriving (passed as arguments in ln. 1). First, v locally processes as many components of f as possible (ln. 2–4). Specifically, v processes f at an (existing or newly started) instance of requested component c_f if there are enough available resources. After

processing, c_f points to the next requested component in C_{s_f}. Again, v processes f locally at an instance of c_f if it has sufficient resources. This local, greedy processing continues until either v's compute resources are fully utilized or after f traversed all requested service components ($c_f = \varnothing$).

In the former case, f continues processing in a similar fashion on the next node. In the latter case, f finished processing successfully and leaves the network once it reaches its egress v_f^{eg}. In both cases, v tries to forward f to neighbor v' on the shortest path to f's egress node. If v is f's ingress node, it calculates the shortest path to v_f^{eg} based on the static topology and weighted by link delays in L, e.g., using Dijkstra's algorithm (ln. 5–7). The destination and calculated path are saved in metadata m_f, which is used when forwarding the flow (ln. 14).

Forwarding is done in Alg. 2, which retrieves the next node v' on the computed path (ln. 2 in Alg. 2). If the link to v' still has enough available data rate, v sends f to v' (ln. 3–4). Otherwise, v recomputes the shortest path without using any of its currently congested outgoing links and sends f along the adapted path (ln. 5–8). Note that just excluding all congested links up front (at $t = 0$) does not suffice since link utilization changes dynamically over time according to flow arrival and the algorithm's decisions. In case all outgoing paths are congested, we assume that v cannot buffer the entire flow f and thus drops it.

If a flow arrives at its egress before being fully processed, it is sent to and processed at surrounding nodes and returns to the egress once it finished processing completely. To balance the load among the surrounding nodes, GCASP randomly chooses a new, temporary destination node, computes the shortest path, and sends f towards the new destination (ln. 8–10 in Alg. 1). As before, nodes on the path to the new destination process f greedily. Once f is fully processed, it is no longer sent towards its temporary destination but is immediately rerouted to its egress node (ln. 11–13). In doing so, remaining processing is done close to the egress node as far as possible, keeping the path delay low.

2) Complexity: The most common coordination decisions concern processing an incoming flow (ln. 2–4 in Alg. 1) and then forwarding it along its path (ln. 2–4 in Alg. 2). Both operations only take constant time, i.e., $O(1)$. Computation of the initial or adapted shortest path is more expensive but happens much more rarely. Using Dijkstra's algorithm, it takes $O(|V|^2)$ or $O(|L| + |V| \log |V|)$, depending on the implementation.

GCASP only relies on the static network topology for computing shortest paths, leading to a space complexity of $O(|V| + |L|)$. Except for shortest path calculation, all other decisions only take $O(1)$ space, e.g., based on a node's local resource utilization or a flow's attributes.

3) Example: Fig. 1 illustrates GCASP's adaptive routing for a flow f. In step 1, f is sent along the shortest path from its ingress towards its egress. On the path, intermediate nodes (omitted in the figure) process f greedily. At v_1, the outgoing link on the shortest path is congested and the path is recomputed (step 2). Here, f reaches its egress without being fully processed. Hence, GCASP randomly selects v_3 as new, temporary destination and sends f towards v_3 (step 3). Nodes on the path continue to process f greedily such that f is fully processed when reaching intermediate node v_2, which is still close to the egress. At that point, it is immediately rerouted to its egress node and leaves the network successfully (step 4).

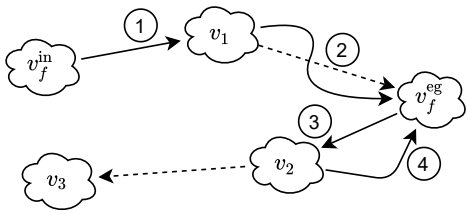


Fig. 1: Example illustration of GCASP's adaptive routing.

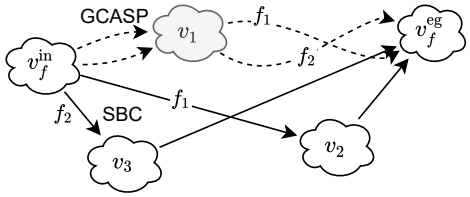


Fig. 2: Rather than sending all flows along the same shortest path, SBC avoids congestion by distributing flows across different nodes based on their calculated score.

B. Score-Based Coordination with Adaptive Shortest Paths

While full up-to-date global knowledge may be unrealistic, it is likely that some information is available through monitoring or beaconing. We designed the score-based coordination (SBC) algorithm to leverage such available information for its coordination decisions. In particular, SBC calculates node scores based on the available information that indicate each node's suitability to process a given flow. Rather than sending all flows to their egress nodes along the same shortest path, SBC tries to distribute flows among different paths and actively selects promising nodes for processing. SBC's score calculation can be configured to use information that is available, which depends on the network and monitoring setup.

1) Motivating Example: Fig. 2 illustrates the benefit of SBC over GCASP in an example. GCASP greedily sends all flows with the same ingress and egress node (here, f_1, f_2) along the same shortest path (dashed lines). All flows compete for node and link resources on the path which may lead to congestion, overloaded nodes (e.g., v_1), and dropped flows. Instead, SBC uses available knowledge to calculate a score and select the most suitable destination nodes (with the highest score) for

Algorithm 3 SBC Algorithm

```
1: procedure SBC(v, f)
2:     if v = v_f^in then
3:         SET_DEST(v, f, V, L)
4:     if v = m_f.dst then
5:         if r_v < cap_v and c_f ≠ ∅ then
6:             x_{v,c_f}(t) ← 1
7:             y_{f,c_f,v}(t) ← v
8:         SET_DEST(v, f, V, L)
9:     FORWARD_FLOW(v, f, L)
10: procedure SET_DEST(v, f, V, L)
11:     if c_f = ∅ then
12:         m_f.dst ← v_f^eg
13:     else
14:         V_{v,f} ← candidates(v, f, V, L)
15:         for all v' ∈ V_{v,f} do
16:             g(v') ← Σ_{i∈[1,|A|]} w_i a_i(v')
17:         m_f.dst ← arg max_{v'}[g(v')]
18:     m_f.path ← shortest_path(v, m_f.dst, L)
```

processing each flow. Here, nodes v_2, v_3 receive the highest score for f_1, f_2, respectively. Processing the flows there avoids congestion and dropped flows.

2) Algorithm: Alg. 3 shows the SBC algorithm, which is called when a new flow f starts arriving at a node v. If v is f's ingress node, v computes the first destination node for processing f (ln. 2–3). The calculation of the destination node based on available information is the core procedure of SBC (ln. 10–18). If f is already fully processed ($c_f = \varnothing$), its egress node is set as destination (ln. 11–12). Once f reaches its egress fully processed, it successfully leaves the network. Otherwise, SBC chooses a new suitable destination node from a set of candidate nodes based on their calculated score (ln. 15–16). Specifically, for each candidate node v', it considers a set $\mathcal{A} = \{a_1, ..., a_k\}$ of k weighted (by w_i) node attributes a_i to calculate an overall node score $g(v')$ (ln. 15–16). Attributes $\mathcal{A}$ and candidate nodes $V_{v,f}$ can be selected depending on the available information in the network, e.g., $V_{v,f}$ could be all nodes V or just neighbors V_v (ln. 14). Attributes a_i should be scaled to $[0, 1]$ for comparability and reflect available metrics of interest that help asses a node's suitability for processing. Examples are the shortest path delay to the egress via v', the current utilization of v', the amount of successfully processed flows at v', etc. Ultimately, the candidate v' with the highest score $g(v')$ is chosen as next destination and the shortest path is calculated, e.g., using Dijkstra's algorithm (ln. 17–18).

Similar to GCASP, SBC then forwards f along the computed path (ln. 9) using Alg. 2. Rather than processing f greedily on the path, f is processed at its selected destination node if it has enough available resources (ln. 4–7). Afterwards, a new destination is selected (ln. 8).

3) Complexity: Like GCASP, SBC makes fast forwarding and processing decisions in $O(1)$. Only adapting the shortest paths due to congestion and setting a new destination require more time. The latter requires $O(|V_{v,f}|\phi)$ where ϕ is the time complexity of calculating $g(v)$, which depends on the complexity of computing the chosen node attributes. If all attributes are simple measurements that can be retrieved in $O(1)$, it results in $\phi = O(k) = O(1)$ and complexity $O(|V_{v,f}|)$ for setting a new destination. Calculation of the shortest path can be done in $O(|V|^2)$ or $O(|L| + |V| \log |V|)$.

In addition to the static network topology, SBC also relies on k attributes and the computed score for all candidate nodes $V_{v,f}$. Hence, SBC's space complexity is $O(|V| + |L| + k|V_{v,f}|) = O(|V| + |L| + |V_{v,f}|)$.

4) Configuration: SBC's configurability allows numerous different variants of candidate node selection and attributes $\mathcal{A}$. For the variant we use in our evaluation, we set candidate nodes $V_{v,f}$ to be all nodes other than current node v with sufficient remaining resources for processing f. The score of each candidate node v' is calculated based on $k = 3$ equally weighted ($w_i = 1$) attributes based on 1) the shortest path delay from current node v to v' and from there to the egress node, 2) the number of dropped flows at v' so far, 3) the total data rate of flows currently being processed or forwarded at v'.

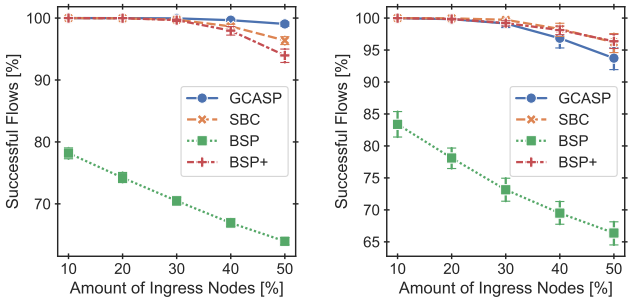
V. Evaluation

A. Evaluation Setup

We evaluate our two proposed algorithms, GCASP and SBC, using extensive simulations on the real-world DFN network topology [21] with 58 nodes and 87 links. We set link capacities to $\text{cap}_l = 50$ and use either homogeneous node capacities ($\text{cap}_v = 10$) or randomly picked heterogeneous node capacities ($\text{cap}_v \in \{0, 10, 50\}$), depending on the evaluation scenario.

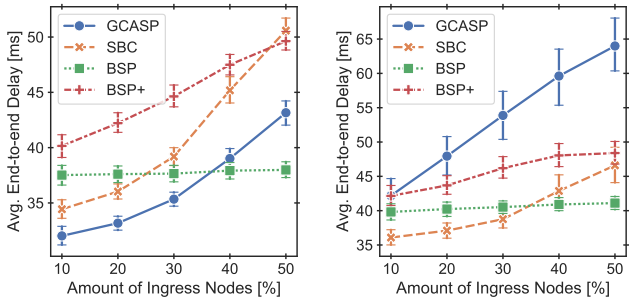
Furthermore, we consider three services $s_1 = \langle c_{\text{IDS}}, c_{\text{proxy}}, c_{\text{web}} \rangle$, $s_2 = \langle c_{\text{proxy}}, c_{\text{IDS}}, c_{\text{web}} \rangle$, and $s_3 = \langle c_{\text{FW}}, c_{\text{DPI}} \rangle$. Instances of each service component require resources linear in the currently processed load. Furthermore, they incur a per-flow processing delay that is normally distributed with $\mathcal{N}(5\,\text{ms}, 1\,\text{ms})$, where values are cut off at 0 ms to prevent negative delays. Flows arriving at the network's ingress nodes request one of the three services chosen uniformly at random. For each ingress, flow inter-arrival times vary randomly between 1, 2, 5, and 10 time steps, flow duration $\delta_f \in \{1, 2\}$, and flow data rate $\lambda_f \in \{1, 2, 4, 6\}$. The duration per experiment is $|T| = 1000$ time steps.

We compare GCASP and SBC with a state-of-the-art centralized coordination algorithm, BSP, from our previous work [5]. We call BSP once for each new flow that enters the network to compute where to process and how to route the flow. In doing so, BSP requires global knowledge of the



(a) Homogeneous node cap. (b) Heterogeneous node cap.

Fig. 3: Comparison of successful flows.



(a) Homogeneous node cap. (b) Heterogeneous node cap.

Fig. 4: Comparison of end-to-end delay.

currently available node and link resources. Even with up-to-date global knowledge at the time of flow arrival, BSP may choose a currently free node for processing that is later fully utilized by other flows when the scheduled flow arrives there. Hence, flows may be dropped when they arrive for processing at an over-utilized node. Therefore, we also consider a variant, BSP+, where we call BSP again to recompute flow processing and routing when a flow cannot be processed at an over-utilized node.

All experiments were repeated 50 times with different random seeds. Our figures show the mean and 95% confidence interval of these repetitions. For running the evaluation, we used machines with Intel Xeon W-2145 CPU and 32 GB RAM. For reproducibility, we publish the code of both our proposed fully distributed algorithms and all evaluation results on GitHub [6]. Similarly, BSP is publicly available [22].

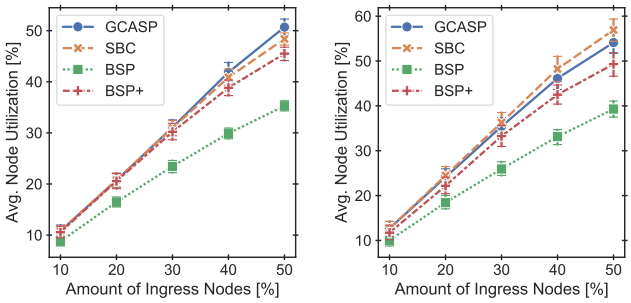
B. Service Coordination Quality

First, we compare the achieved solution quality of our proposed fully distributed service coordination algorithms (GCASP and SBC) and the centralized approach (BSP and BSP+). As evaluation parameter, we vary the ingress node percentage, i.e., the percentage of nodes in the network at which flows arrive. With an increasing ingress node percentage, more nodes are randomly selected as ingress nodes, leading to increasing overall load. The percentage of egress nodes is fixed to 30%, to which flows are assigned uniformly at random. We also vary between homogeneous vs. heterogeneous node capacities. As metrics to evaluate the service coordination quality, we consider the percentage of successfully processed flows o_f and their average end-to-end delay o_d at the end of each experiment as defined in Sec. III-C. We also consider resource utilization of both nodes and links since service coordination algorithms should use resources economically.

1) Successful Flows: Fig. 3 shows the percentage of successful flows achieved by the different algorithms. The percentage of successful flows decreases with increasing load as the network becomes more congested and some flows cannot be processed or forwarded. BSP drops comparatively many flows as it only makes coordination decisions once per flow when flows enter the network, based on information that may be outdated soon after. Like many centralized service coordination approaches, BSP is designed to focus on few long-running flows rather than many, rapidly arriving, sequential, and partially overlapping flows. BSP+, which recalculates flow processing and routing whenever a flow would otherwise be dropped due to lack of node capacities, performs much better and achieves close to 100% successful flows.

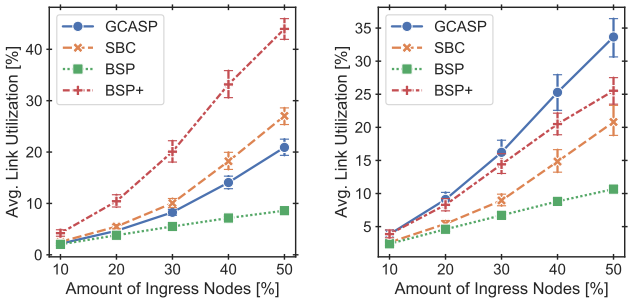
Nevertheless, the two fully distributed algorithms process similar or even more flows successfully than BSP+. Clearly, the many fast and individual per-node decisions of these fully distributed algorithms work well and process close to 100% of flows successfully. In the network with homogeneous node capacities (Fig. 3a), GCASP even outperforms all other algorithms, even though it uses no global knowledge except for the static network topology. This is because, with homogeneous node capacities, there are likely enough compute capacities on the shortest path from ingress to egress and around the egress node. In contrast, in the case of heterogeneous node capacities (Fig. 3b), SBC is slightly better than GCASP and on par with BSP+ because it leverages available knowledge to actively select nodes with sufficient processing capacities. Still, GCASP processes more than 90% of flows successfully even under high load.

2) End-to-end Delay: Fig. 4 shows the avg. end-to-end delay of successfully processed flows. While BSP drops an increasing percentage of flows with increasing load, it ensures relatively low and constant end-to-end delay for the remaining successful flows. For the other algorithms, avg. end-to-end delay increases with increasing load as more flows are rerouted due to congestion, leading to longer paths. Again, our fully distributed algorithms can compete with and even outperform the centralized BSP and BSP+ approaches. The figure also confirms that the greedy GCASP is best in the homogeneous case (Fig. 4a). In the heterogeneous case (Fig. 4b), SBC is



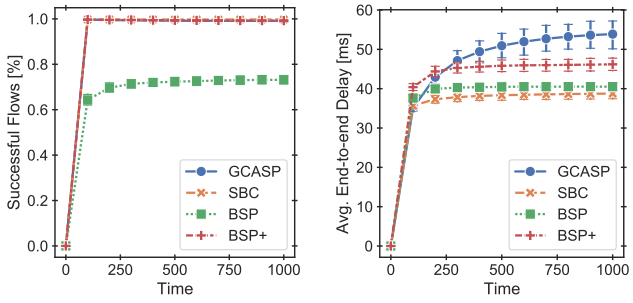
(a) Homogeneous node cap. (b) Heterogeneous node cap.

Fig. 5: Comparison of node resource utilization.



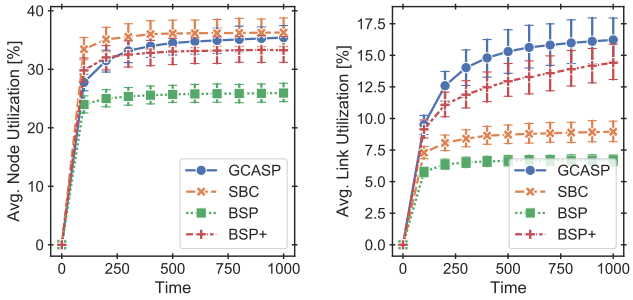
(a) Homogeneous node cap. (b) Heterogeneous node cap.

Fig. 6: Comparison of link resource utilization.



(a) Successful flows (b) Avg. end-to-end delay

Fig. 7: Percentage of successful flows and avg. end-to-end delay over time.



(a) Avg. node utilization (b) Avg. link utilization

Fig. 8: Avg. node and link resource utilization over time.

better again and achieves lower end-to-end delay.

3) Resource Utilization: Fig. 5 shows the node resource utilization averaged over all nodes. The node utilization correlates with the percentage of successful flows in Fig. 3 as processing more flows completely (i.e., successfully) requires more node resources. Accordingly, BSP requires less node resources than the other algorithms because it drops more flows. The two distributed algorithms utilize similar percentages of node resources as the centralized BSP+ approach.

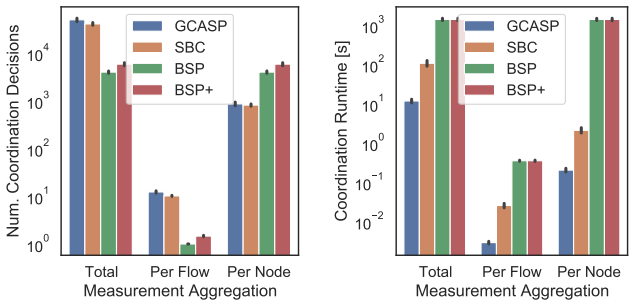
Fig. 6 shows the link resource utilization averaged over all links. Again, by dropping more flows, BSP reduces the overall load in the network and thus also the link utilization compared to the other algorithms. More interestingly, the link utilization also reflects how efficiently the algorithms route traffic. Longer detours lead to more traversed links and higher link utilization. For homogeneous traffic (Fig. 6a), both distributed algorithms utilize less than 30% link resources and far less than BSP+, which reroutes traffic whenever flows could not be processed. As discussed before, GCASP performs worse for heterogeneous traffic (Fig. 6b) and routes flows along detours to complete processing. Nevertheless, the link utilization of GCASP is not much higher than of BSP+. SBC uses available information to route traffic more effectively and outperforms BSP+ in terms of link utilization.

C. Service Coordination Stability

The results in Sec. V-B show the quality metrics at the end of each experiment, i.e., after $|T| = 1000$ time steps. Here, we evaluate the service coordination stability over these 1000 time steps during which flows keep arriving randomly as described in Sec. V-A. We consider the DFN network with heterogeneous node capacities and 30% ingress nodes.

Fig. 7 shows that both the percentage of successful flows (Fig. 7a) and the avg. end-to-end delay (Fig. 7b) are very stable over time for all algorithms. The initial jump from $t = 0$ to $t = 100$ is simply because all metrics are initialized to 0. While GCASP performs worse on heterogeneous than on homogeneous node capacities, its achieved end-to-end delay still only increases slightly in the beginning but then converges and stabilizes.

Fig. 8 shows the avg. node and link utilization over time. Again, the utilization is fairly stable for all algorithms. Only for BSP+ the link utilization slowly grows over time as more and more flows are being rerouted due to congestion. Overall,



(a) Coordination Decisions (b) Coordination Runtime

Fig. 9: Coordination decisions and their runtime (log. scale).

the results indicate that the two distributed algorithms not only achieve competitive solution quality but also converge early and maintain stable service coordination over time.

D. Service Coordination Scalability

In addition to ensuring high and stable service quality, service coordination should be fast and scalable to handle rapidly arriving flows in practical scenarios. To evaluate coordination scalability, we compare the number of coordination decisions (i.e., algorithm calls) and algorithm runtime of the different approaches. Again, we consider the DFN network with heterogeneous node capacities and 30% ingress nodes. Fig. 9 shows the algorithms' number of coordination decisions on a logarithmic scale. Our proposed fully distributed algorithms (GCASP and SBC) make coordination decisions individually at each node for each incoming flow. In contrast, BSP and BSP+ are only called once when a new flow enters the network and, in case of BSP+, when a flow cannot be processed due to lack of node resources. Hence, our fully distributed algorithms make far more coordination decisions in total and per flow than the centralized approaches (Fig. 9a). We assume that BSP and BSP+ make all decisions at a single centralized node (at least logically). On the other hand, GCASP and SBC distribute coordination over all nodes in the network such that they still make significantly fewer coordination decisions *per node*.

The runtime for these coordination decisions is even more important than the number of decisions. Due to their simplicity, the runtime (per flow, per node, and in total) of our fully distributed algorithms is much lower than of BSP and BSP+ (Fig. 9b). While we do not consider this coordination time as part of a flow's end-to-end delay in Sec. V-B and V-C, it would still impact overall delay and service quality in practice.

We also tested GCASP and SBC successfully on the large GTS CE network [21] with 149 nodes and 193 links. Compared to the DFN network, the algorithms' total decisions and runtimes were higher, but the numbers per flow and per node were comparable to the ones in the DFN network. Due to the prohibitive long runtimes of BSP and BSP+, we do not present a full performance evaluation of the GTS CE network. Nevertheless, our results on the two networks indicate that our proposed, fully distributed algorithms scale well to practical, large-scale networks.

VI. Conclusion

Our proposed fully distributed algorithms coordinate services well and achieve comparable service quality to a state-of-the-art centralized coordination algorithm. At the same time, they require less or no global network information, are faster, can be massively parallelized, and are more robust to failures. Hence, in contrast to centralized approaches, such fully distributed algorithms are useful in practical large-scale networks with rapidly arriving flows. We therefore believe that our proposed algorithms can significantly improve service coordination and resulting QoS in practice.

In future work, we plan to investigate robustness and fault tolerance in fully distributed service coordination as well as hybrid approaches, where some coordination decisions are made centrally and others in a distributed manner.

Acknowledgments

This work has received funding from the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the German Federal Ministry of Education and Research through Software Campus grant 01IS17046 (RealVNF).

References

[1] Cisco, "Cisco annual internet report (2018–2023)," 2020, online: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (accessed March 27, 2020.

[2] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[3] S. Ahvar, H. P. Phyu, S. M. Buddhacharya, E. Ahvar, N. Crespi, and R. Glitho, "CCVP: Cost-efficient centrality-based vnf placement and chaining algorithm for network service provisioning," in *IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–9.

[4] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1562–1576, 2018.

[5] S. Dräxler, S. Schneider, and H. Karl, "Scaling and placing bidirectional services with stateful virtual and physical network functions," in *IEEE Conference on Network Softwarization (NetSoft)*, 2018.

[6] L. Klenner and S. Schneider, "Fully Distributed Service Coordination GitHub Repository," https://github.com/CN-UPB/distributed-coordination, 2020.

[7] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.

[8] Z. Á. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–34, 2015.

[9] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[10] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *IEEE International Conference on Network and Service Management*, 2014, pp. 418–423.
[11] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Computer Communications*, vol. 102, pp. 1–16, 2017.
[12] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, pp. 67–77, 2017.
[13] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.
[14] M. Feng, J. Liao, S. Qing, T. Li, and J. Wang, "COVE: Co-operative virtual network embedding for network virtualization," *Journal of Network and Systems Management*, vol. 26, no. 1, pp. 79–107, 2018.
[15] I. Houidi, W. Louati, and D. Zeghlache, "A distributed virtual network mapping algorithm," in *IEEE International Conference on Communications*. IEEE, 2008, pp. 5634–5640.
[16] X. Shi, X. Wen, Y. Sun, L. Li, and W. Ma, "A novel distributed VNet mapping algorithm," in *IEEE International Conference on Communication Technology*. IEEE, 2012, pp. 311–316.
[17] A. Song, W.-N. Chen, T. Gu, H. Yuan, S. Kwong, and J. Zhang, "Distributed virtual network embedding system with historical archives and set-based particle swarm optimization," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.
[18] M. T. Beck, A. Fischer, H. de Meer, J. F. Botero, and X. Hesselbach, "A distributed, parallel, and generic virtual network embedding framework," in *IEEE International Conference on Communications (ICC)*. IEEE, 2013, pp. 3471–3475.
[19] P. Quinn, U. Elzur, and C. Pignataro, "Network service header (NSH)," IETF, Internet Request for Comments RFC 8300, 2018.
[20] Cloud Native Computing Foundation, "Kubernetes: Production-grade container orchestration," https://kubernetes.io/ (accessed Jan 31, 2020), 2020.
[21] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.
[22] S. Schneider, "BSP GitHub Repository," https://github.com/CN-UPB/B-JointSP, 2020.