Flexible Reassignment of Flow Processing-aware Controllers in Future Wireless Networks

Sébastien Auroux  
University of Paderborn  
Paderborn, Germany  
E-mail: sebastien.auroux@upb.de

Holger Karl  
University of Paderborn  
Paderborn, Germany  
E-mail: holger.karl@upb.de

Abstract—The growing number of network mechanisms results in a large amount of data flow processing to be performed in future wireless access networks. Determining suitable locations for data processing in a network is important for a good network performance but also a non-trivial task as data processing imposes many requirements on network resources. Further, data flows and network load generally change over time, possibly very quickly, so that a static, one-time placement is not an adequate solution. Hence, a framework that determines these locations should also flexibly reassign them over time.

We address this need by a flexible flow processing-aware controller placement framework (FlexFCPF), which places and flexibly reassigns controller devices that are able to perform both network control (conventional SDN) and data flow processing for an efficient management of future wireless networks. We have developed and implemented a fast heuristic framework and we prove its efficiency by providing evaluation results of FlexFCPF's performance in a dynamic network simulation.

I. INTRODUCTION

Modern cellular networks deployed by mobile network operators are generally very dense and heterogeneous to cope with the consistently growing traffic demands in mobile networks [1], [2]. These networks impose several challenges, such as a large amount of signaling overhead, inter-cell interference or high energy consumption. In recent work, Software-Defined Networking (SDN) with distributed controllers is often found to be a promising approach for this task [3], [4].

However, future networks need a vast range of network mechanisms [5]. All of these mechanisms induce several data flows in the network and thus there is considerable data flow processing work to be done in future wireless networks besides the mere control tasks associated with SDN controllers. Because of this, we have focused our research on controller devices that are able to perform both conventional SDN network control and data flow processing to efficiently manage future wireless networks.

But finding a good placement for a given time only solves the problem temporarily. As the load of a mobile network can change very quickly, the performance of a static one-time placement will generally become worse over time. Especially in modern dense and crowded networks, many data flows appear and expire every second. The obvious solution might be to compute a new controller placement from scratch when the performance of the current placement decreases, but such a placement is expected to result in a lot of different control assignments compared to the previous placement, causing much reconfiguration overhead. In addition, even with a fast heuristic algorithm, the runtime for a completely new placement is likely too high for frequently changing network states.

As a result, we consider flexible controller reassignment based on the current controller placement. Our flexible flow processing-aware controller placement framework (FlexFCPF) is able to place and flexibly modify a controller architecture in a cellular network while meeting the requirements on latency, data rate and processing capacity imposed by both data flow processing and network control. In our evaluation we demonstrate the superiority of our flexible reassignment approach compared to from-scratch recalculation.

First, the paper gives an overview of previous and related work in Section II. Next, we elaborate the problem statement in Section III, followed by details of the FlexFCPF heuristic in Section IV. Finally, we provide evaluation results of FlexFCPF obtained from a dynamic network simulation in Section V and conclude our work in Section VI.

II. RELATED WORK AND BACKGROUND

In our previous work [6], [7] we have already presented flow processing-aware controller placement approaches with optimization models as well as a heuristic algorithm, providing good results compared with the optimization model. However, our previous work has only considered static snapshots of a cellular network, neglecting the dynamic nature of a real-world cellular network. With the present work, we rectify this shortcoming by providing an extended framework that is able to deal with dynamic network operation (Section IV). Further, our implementation and heuristic methods have been massively improved in terms of both runtime and solution quality compared to the approach from [7].

As for related work, many researchers have addressed SDN controller placement. Bari et al. [8] propose an SDN controller placement framework with periodic controller reassignment to optimize the average flow set-up time in a network and present a heuristic algorithm based on simulated annealing with promising results. But periodic reassignment cannot handle rapidly changing network load, in contrast to reassignment dynamically triggered by these changes. The authors of [4] propose an elastic distributed controller architecture in which
the number of controllers is dynamically adapted according to the network load. Still, neither work includes data flow processing and the resulting constraints demanded from the network elements.

Other related work deals with placement of virtual machines (VMs) and Virtual Network Functions (VNFs). Gember et al. [9] introduce a network-aware orchestration layer which collocates related VMs with dense communication patterns. While their framework provides good results, they do not support tight control loops in their cloud scenario. The authors of [10] propose a model to describe chains of VNFs in operator networks and an optimization model to place them based on requirements of the operator. However, this work does not include the network control aspect.

In total, the present research on SDN controller placement neglects data rate and data processing aspects, while the research on VM and VNF placement disregards control assignments and (partially) latency aspects, which we all combine in one controller placement framework.

### III. PROBLEM STATEMENT

FlexFCPF has been developed in the context of the FP7 project CROWD [11] and currently uses the project’s two-layer controller architecture, consisting of CROWD Local Controllers (CLCs) and CROWD Regional Controllers (CRCs). The CLCs operate on a local scope and execute fast, short time scale mechanisms. The CRCs operate on a more global scope and coordinate the CLCs to compensate for sub-optimal choices which may be owing to the myopic view of the local controllers. CLCs and CRCs are seen as applications that can be installed on network equipment that fulfills the necessary hardware requirements, i.e. sufficient memory and processing capacity.

FlexFCPF uses the input parameters summarized in Table I. As in our previous work [6], [7] we look at a given network as a graph \( G = (V, E) \) with nodes \( V \), e.g. LTE eNBs, WiFi APs and switches, and unidirectional backhaul links \( E \). Each link \((u, v) \in E\) has a maximum data rate of \( b_{cap}(u, v) \) bit/s and a latency of \( l_{cap}(u, v) \) seconds. The potential controllers, i.e. nodes from \( V \) that fulfill the necessary hardware requirements for serving as a controller, are denoted by \( C \subseteq V \) and each \( c \in C \) has a processing power of \( p_{node}(c) \) FLOPS.

To establish a fully functional network control layer, we aim to create a complete control structure, i.e. each node \( v \in V \) is controlled by at least one CLC (a node can be controlled by more than one CLC if this is beneficial for network performance) and each CLC is required to be coordinated by exactly one CRC. Being the CLC of a node or the CRC of a CLC requires a processing capacity of \( p_{CLC} \) or \( p_{CRC} \) FLOPS, respectively. Also, the routing path between a node and its CLC needs to have a round trip latency of at most \( l_{CLC} \) seconds and a data rate of at least \( b_{CLC} \) bit/s. Similarly, the round trip latency \( l_{CRC} \) seconds and the data rate \( b_{CRC} \) bit/s are required for the routing path from a CLC to its CRC. For the scope of this work, we assume that all nodes of the network are connected to the operator’s core network by a backbone infrastructure, e.g. optical metro rings as deployed in metropolitan areas [12]. For now, we assume this infrastructure to provide unlimited data rate and zero latency.

We further look at a set \( F \) of data flows originating at certain nodes from \( V \) and demanding processing at a CLC. The connection matrix \( W \in \{0, 1\}^{|F| \times |C|} \) represents the relationship between data flows and network nodes. A data flow \( f \in F \) arrives at time \( t_{arr}(f) \) with a duration of \( l_{dur}(f) \) seconds and can be satisfied by at most one CLC. A CLC \( c \in C \) is said to satisfy a data flow \( f \in F \) if and only if controls all nodes \( v \) with \( W_{f} = 1 \) and if the routing paths from all these nodes to \( c \) provide sufficient resources to allocate the requested data rate of \( b_{flow}(f) \) bit/s and the requested round trip latency of \( l_{flow}(f) \) seconds, considering the required processing capacity of \( p_{flow}(f) \) FLOPS for processing \( f \) at \( c \). We consider the full round trip for all data flows to incorporate both upload and download traffic. The scheduler for data processing at a CLC is assumed to use equal share between all controlled nodes and all satisfied data flows. To guarantee that no constraint for network control and data processing is violated, FlexFCPF determines the necessary routing paths and verifies that all conditions are met at any time.

The main objective of FlexFCPF is to determine a valid solution, i.e. a complete control structure, with a maximum amount of data flows being satisfied by a CLC. While we would like to see all data flows satisfied at any time, this might not always be possible because of a combination of too many data flows and too little network resources. In this case, we see an incomplete control structure, i.e. not all nodes are correctly controlled, as more critical for the network than a couple of yet not processed data flows.

### IV. FLEXFCPF HEURISTIC IMPLEMENTATION

In this section we elaborate on the implementation of FlexFCPF. First, we focus on the initial controller placement functionality of FlexFCPF in Section IV-A before elaborating on the flexible reassignment abilities in Section IV-B. All procedures use the parameters from Table I. To provide an initial overview, Figure 1 summarizes the key aspects of FlexFCPF in a flow chart.
A. Initial controller placement

When confronted with a network without an existing controller placement, FlexFCPF executes the CPGREEDY procedure shown in Algorithm 1. CPGREEDY successively adds CLCs to the network by calling FINDCLC and first focuses on having a CLC assigned to each network node. Whenever all nodes are controlled by at least one CLC and if there are still potential controllers available, CPGREEDY switches its strategy and fully focuses on data flow satisfaction. The procedure ends if all data flows are satisfied, if no more controllers are available or if no new flows could be satisfied by adding an additional CLC, e.g. if all remaining data flows cannot be satisfied with the network’s remaining resources. Right before terminating, the CLEANUPCLCCONTROLS procedure removes all CLC-to-node assignments that eventually did not serve any purpose, i.e. the CLC does not satisfy any data flow passing through the node and the node is assigned to more than one CLC.

Algorithm 1 CPGREEDY()

```plaintext
option = “neighbors”
while |uncontrolled nodes| > 0 do
    FINDCLC(option)
    if |CLCs| = |C| then
        break // no more potential controllers available
    if |CLCs| < |C| then
        option = “flows”
    while |unsatisfied flows| > 0 do
        lastsat = |satisfied flows|
        FINDCLC(option)
        if |CLCs| = |C| or lastsat = |satisfied flows| then
            break
CLEANUPCLCCONTROLS()
```

The FINDCLC procedure presented in Algorithm 2 determines which potential controller is going to be added as a CLC next, according to the option set by CPGREEDY, acquiring the best candidates via GETCLCCANDIDATES. But before adding a candidate as a CLC, FINDCRC checks if a CRC can be found for it, as a CLC that cannot be assigned to a CRC would violate the integrity of our two-layer controller architecture. FINDCRC first checks the already available CRCs, starting with the closest one. If no existing CRC can be assigned to a candidate, a new CRC is added to the network considering all potential controllers that are not yet CRCs. For the first CRC in the network, we seek the most centralized potential CRC in order to benefit all future CRC-to-CLC assignments, otherwise we go by the distance to the current candidate. If a CRC can be found, the candidate is confirmed as a new CLC and assigned to the CRC. Finally, Algorithm 4 assigns nodes and data flows to the new CLC.

Algorithm 2 FINDCLC(option)

```plaintext
candidates = GETCLCCANDIDATES(option)
for v in candidates do
    c = FINDCRC(v)
    if c is not None then
        ADDNEWCLC(v)
        break // next CLC determined
```

The critical CLC candidate selection is shown in Algorithm 3. GETCLCCANDIDATES considers all potential controllers that are not yet used as a CLC and if possible also excludes any CRC to leave them with resources for controlling future CLCs. To determine the best CLC candidates, GETCLCCANDIDATES uses the option neighbors or flows, depending on what has been specified by CPGREEDY. In addition, the more specific options isolated nodes or isolated flows are used in case the metrics used by neighbors or flows do not provide a sufficient result for sorting the candidates.

Algorithm 3 GETCLCCANDIDATES(option)

```plaintext
candidates = C - CLCs
if candidates - CRCs = ∅ then
    candidates = candidates - CRCs // avoid CRCs if possible
if option = “neighbors” then
    sort candidates by highest amount of uncontrolled nodes in \{v\} ∪ neighbors(v)
    if best value = 0 then // no node with uncontrolled neighbors
        candidates = GETCLCCANDIDATES(“isolated nodes”)
else if option = “isolated nodes” then
    sort candidates by shortest distance to an uncontrolled node
else if option = “flows” then
    sort candidates by highest amount of unsatisfied flows passing through v
    if best value = 0 then // no node with unsatisfied flows
        candidates = GETCLCCANDIDATES(“isolated flows”)
else if option = “isolated flows” then
    sort candidates by shortest distance to a node with an unsatisfied flow passing through
```

As stated before, ADDNEWCLC (Algorithm 4) is responsible for assigning nodes and data flows to a new CLC v. To this end, it calculates the shortest paths from v to all other nodes in the network and these, depending on whether a network already has a complete control structure or not. After the prioritization is done, nodes are assigned to be controlled by v. The CHECKCLCCONTROL procedure verifies that no
Algorithm 4 ADDNEWCLC(v)

```plaintext
paths = {SHORTESTPATH(source=v, target=i) for i in nodes} potential flows = {}
if |uncontrolled nodes| > 0 then // no valid solution yet
    sort paths by paths to uncontrolled nodes first, path length next
else // valid solution already found, now focus on flow satisfaction
    sort paths by paths to nodes with unsatisfied flows first, path length next
while (paths > 0 or |potential flows| > 0) and |
    ((uncontrolled nodes) > 0 or |unsatisfied flows| > 0) do
    if |potential flows| > 0 and (uncontrolled nodes) = 0 or |
        new nodes controlled| ≥ VCRatio then
        f = GETMOSTDEMANDINGFLOW(potential flows)
        CLCs = \{CLCs\}
        if CHECKFLOWSAT(v, f) = True then
            ADDFLOW(v, f)
            potential flows.remove(f)
        else
            p = GETNEXTPATH(paths)
            if CHECKCLCCONTROL(p) = True then
                ADDCLCCONTROL(p)
            UPDATEPOTENTIALFLOWS(v)
        break // no more resources left at CLC v
```

model constraint is violated in the process. When a node is assigned to v, UPDATEPOTENTIALFLOWS updates the list of potential flows for v, i.e. the flows that are only passing through nodes that v already controls and thus can be satisfied by v if sufficient resources are available. But if there is a network with many data flows, each CLC might have a lot of potential flows after controlling only a few nodes, run out of resources by satisfying them quickly and a complete control structure would probably not be obtained. Hence, data flows may only be satisfied by a CLC once it controls at least VCRatio = VCLCs nodes that had previously been uncontrolled or if there are no more uncontrolled nodes in the network.

Once v is allowed to satisfy data flows, we successively try to assign the most demanding, in terms of requested processing capacity, data flow to v. Again, all constraints are checked using the CHECKFLOWSAT procedure before confirming that v henceforth satisfies a certain data flow. In any case, the data flow is then removed from the list of potential flows for v, as it is either now satisfied by v or as it cannot be satisfied by v. ADDNEWCLC terminates when no more nodes and no more data flows can be assigned to v.

B. Flexible reassignment

For the scope of this paper, we consider the following triggers for changing the network configuration:

1) Incoming data flows
2) Low load situations

1) Satisfying incoming data flows: Each time that a new data flow appears in the network, FlexFCPF at first tries to assign it to one of the network’s CLCs by calling the BROWSECLCSFORFLOW procedure (Algorithm 5). Unlike the ADDNEWCLC procedure (Algorithm 4), where a CLC is assigned to as many data flows as possible from its nearby nodes, this procedure specifically tries to assign a certain data flow to the closest possible CLC. If BROWSECLCSFORFLOW fails to assign an incoming data flow, FlexFCPF launches the CPGREEDY procedure which will in this case result in executing FINDCLC(flows) (see Algorithm 1) to add a new CLC that will satisfy the new data flow.

2) Low load detection: While it is obvious when it is necessary to add an additional CLC, answering the question when it is possible to remove a CLC while still satisfying all data flows is much more complex. To do this, we define a notion of load for a CLC:

\[
\text{CLCload}(c) = \max_{f \in \text{Sat}(c)} \left( \frac{\text{RTT}(f)}{l_{\text{flow}}(f)} \right) \max_{v \in \text{Contr}(c)} \frac{\text{RTT}(v)}{l_{\text{CLC}}}
\]

where Sat(c) is the set of data flows satisfied by c, Contr(c) is the set of nodes controlled by c and RTT(·) is the round trip time for a CLC-to-flow or CLC-to-node assignment.

Recall: because of the assumed equal share scheduling for data processing (see Section III), the round trip time increases for all nodes and flows assigned to a CLC as soon as a new node or flow is assigned to it. As FlexFCPF ensures that the latency constraints for an existing assignment are never violated, it holds that \(\text{CLCload}(c) \in (0, 1] \forall c \in \text{CLCs}\).

Algorithm 5 BROWSECLCSFORFLOW(f)

```plaintext
CLCs* = {CLCs controlling all nodes f is passing through}
for v in CLCs* do
    if CHECKFLOWSAT(v, f) = True then
        return ADDFLOWSAT(v, f)
CLCs + = CLCs - CLCs*
sort CLCs+ by average distance to all nodes f is passing through
for v in CLCs do
    if CHECKCLCCONTROL(p) = True then
        ADDCLCCONTROL(p)
    else // v cannot satisfy f
        if all nodes assigned and CHECKFLOWSAT(v, f) = True then
            return ADDFLOWSAT(v, f)
        else // v cannot satisfy f, remove useless control assignments
            for p in paths do
                REMOVECLCCONTROL(p)
```

Algorithm 6 LOWLOAD()

```plaintext
tmpCLCs = |CLCs|
load = \left\{ \sum_{c \in \text{CLCs}} \text{CLCload}(c) \right\} // surely required CLCs
while |CLCs| > load do
    REMOVECLC(GETCLCWITHLEASTLOAD())
if |CLCs| < tmpCLCs then
    BROWSECLCS()
if |uncontrolled nodes| + |unsatisfied flows| > 0 then
    CGREEDY()
else // LoadLimit = LoadLimit - 0.05
    LoadLimit = 0.9
```

To detect low load situations, FlexFCPF monitors the average CLC load in the network and if it stays below a
threshold value $\text{LoadLimit} = 0.9$ for 10 seconds or more, FlexFCPF executes the LOWLOAD procedure (Algorithm 6), which removes all not necessarily required CLCs and then executes BROWSECCLCs, which browses through all current CLCs and tries to assign uncontrolled nodes and unsatisfied data flows to them. As this procedure works very similar to the ADDNEWCLC procedure from Section IV-A we skip a detailed representation. If afterwards there are still uncontrolled nodes or unsatisfied data flows, CPGREEDY is launched to correct this as in Section IV-B. In case that there are not less CLCs than before, the low load correction is seen as a failure and $\text{LoadLimit}$ is decreased by 0.05 for the next round.

V. EVALUATION

In this section we present the results from a dynamic network simulation to evaluate FlexFCPF. All calculations are executed in single-threaded mode on Intel Xeon X5650 CPUs running at 2.67 GHz.

A. Evaluation Scenario

We have simulated the network operation with FlexFCPF over the course of 48 simulated hours. The four evaluation networks have either 36 or 100 nodes and backhaul links with mesh or tree topology. The nodes are placed on a regular grid with a mean inter-BS distance of $s = 1000$ m, shifted in both x and y directions by normally distributed random variables with zero mean and a standard deviation of $\frac{s}{\sqrt{2}}$. For the mesh topologies, two nodes are connected if their distance is less than or equal to $1.5 \cdot s$, which has produced fully connected but not unrealistically dense topologies. For the tree topologies, we first define the most central network node as connected and then recursively connect the unconnected node being closest to the network’s center to its closest connected and more central node. Each node becomes a potential controller with processing capacity $p_{\text{node}} = 200$ GFLOPS with a probability of $P_C = 0.6$. Each link’s capacity is 2.5 Gbit/s and the latency is determined by the link’s length multiplied by 1.45 and divided by the speed of light, modeling an optical backhaul network.

Incoming data flows are generated using a nonstationary Poisson process with $\lambda = |V| \cdot \text{loadlevel}(t)$, simulated using the thinning method [13]. Our daily load curve loadlevel($t$) (Figure 2), with $t$ being the current time in seconds, is derived from [14]. A flow’s duration is set using an exponential variable with parameter 0.02 resulting in an expected flow duration of 50 seconds and thus an approximated expected number of data flows in the network at time $t$ of $50 \cdot |V| \cdot \text{loadlevel}(t)$. To assign data flows to nodes, we use the GreenTouch connectivity model [15] to connect a flow to up to three nodes with the best connectivity until a certain SINR threshold is reached, assuming mobile user equipment as origin and/or destination for this evaluation. A flow belongs to one of three types, as shown in Table II (data extrapolated from [2]). At last, the requested processing capacity by data flow $f$ is determined by

$$p_{\text{flow}}(f) = 4 \cdot b_{\text{flow}}(f) \cdot \sum_{v \in V} W_{f,v} \text{ FLOPS} \text{ bit/s}$$

At last, the CLC and CRC parameters from Table I are chosen as follows: $b_{\text{CLC}} = b_{\text{CRC}} = 10$ Kbit/s, $l_{\text{CLC}} = 1$ ms, $l_{\text{CRC}} = 10$ ms, $p_{\text{CLC}} = p_{\text{CRC}} = 100$ KFLOPS.

<table>
<thead>
<tr>
<th>type</th>
<th>probability</th>
<th>$b_{\text{flow}}$</th>
<th>$l_{\text{flow}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>voice</td>
<td>0.5</td>
<td>500 Kbit/s</td>
<td>5 ms</td>
</tr>
<tr>
<td>video</td>
<td>0.4</td>
<td>1 to 4 Mbit/s</td>
<td>10 ms</td>
</tr>
<tr>
<td>data</td>
<td>0.1</td>
<td>1 to 20 Mbit/s</td>
<td>50 ms</td>
</tr>
</tbody>
</table>

All simulations were launched at $t = -3600$ seconds to start the 48 hour network monitoring at $t = 0$ in a running state, skipping an initial transient. To generate the data for our evaluation, we have extracted the data from FlexFCPF each time the set of CLCs is modified in the course of reassignment. At these points, we further perform an initial controller placement on an empty copy of the current network for comparison.

B. Simulation results

This section summarizes the results obtained from the simulations described in Section V-A. Throughout all simulations, FlexFCPF has solved all networks with only one CRC and satisfied all data flows apart from the tree100 topology where a few data flows were not satisfied and more than one CRC had to be used during high load periods due to the sparse backhaul tree topology compared to the larger network size.

Figure 3 summarizes the most important data from our simulation runs. Figure 3a displays the average number of CLCs used. We see that the flexible reassignment is fully competitive with the from-scratch comparison considering the number of used CLCs, even though it is technically disadvantaged as it has to build up on an already existing controller placement. Next, Figures 3b to 3d give an impression of the reconfiguration overhead caused by the newly calculated assignments. Figure 3b shows the average amount of newly added CLCs compared to the CLCs of the previous assignment. Similarly, Figure 3c and Figure 3d depict the average number of new CLC-to-node and CLC-to-flow assignments. As can be seen, the flexible reassignment clearly outperforms the from-scratch initial placement for all these metrics. Hence, the flexible reassignment saves a large amount of reconfiguration overhead caused by establishing new assignments in the network.

Figure 3e shows the average CLC control ratio in the networks, i.e. the average amount of CLCs a node is assigned to. We recognize that the reassignment’s strategy of finding a CLC for a specific flow (see Section IV-B) leads to nodes being assigned to multiple CLCs, as generally not always the same
CLCs are going to have available resources to satisfy data flows incoming at a certain node. Meanwhile, the initial placement provides a smaller CLC control ratio as its global view allows assigning as many nearby nodes and its passing through data flows as possible to a CLC. The average CLC path length shown in Figure 3f follows this observation. Clearly, we have to accept longer routing paths with the flexible reallocation when no nearby CLC is able to satisfy an incoming data flow. In total, the flexible reallocation has to devote more resources for the network control compared to the initial placement, taking away resources for processing data flows in theory. But as seen earlier, the flexible reallocation still performs equally well for data flow satisfaction and for the number of CLCs used. We can explain this by observing that the resources required for the network control are significantly smaller than the resources required by the data processing.

Finally, Figure 3g gives an overview of the algorithm runtimes and we see that the flexible reallocation clearly beats the initial controller placement. Especially in crowded networks, we see a small runtime as a significant advantage.

VI. CONCLUSION

In our previous work [6], [7] we have introduced the idea of flow processing-aware controller placement and presented a fast heuristic algorithm that opened the way to real-world deployment. Now we have taken this idea further by providing a significantly improved version of our algorithm from [7] with the new ability of performing flexible reallocation during network operation. As our evaluation shows, the flexible reallocation performs almost equally well compared to a new controller placement from scratch while saving a considerable amount of reconfiguration overhead and runtime, thereby invalidating that efficient network reconfiguration can be done by simply calculating a new controller placement from scratch.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318115 (CROWD).

REFERENCES