UbiFlow: Mobility Management in Urban-scale Software Defined IoT

Di Wu[†], Dmitri I. Arkhipov[‡], Eskindir Asmare[†], Zhijing Qin[‡], Julie A. McCann[†]
[†]Department of Computing, Imperial College London, UK
[‡]Department of Computer Science, University of California, Irvine, USA

***Abstract*—The growing of Internet of Things (IoT) devices has resulted in a number of urban-scale deployments of IoT multinetworks, where heterogeneous wireless communication solutions coexist. Managing the multinetworks for mobile IoT access is a key challenge. Software-defined networking (SDN) is emerging as a promising paradigm for quick configuration of network devices, but its application in multinetworks with frequent IoT access is not well studied. In this paper we present UbiFlow, the first software-defined IoT system for ubiquitous flow control and mobility management in multinetworks. UbiFlow adopts distributed controllers to divide urban-scale SDN into different geographic partitions. A distributed hashing based overlay structure is proposed to maintain network scalability and consistency. Based on this UbiFlow overlay structure, relevant issues pertaining to mobility management such as scalable control, fault tolerance, and load balancing have been carefully examined and studied. The UbiFlow controller differentiates flow scheduling based on the per-device requirement and whole-partition capability. Therefore, it can present a network status view and optimized selection of access points in multinetworks to satisfy IoT flow requests, while guaranteeing network performance in each partition. Simulation and realistic testbed experiments confirm that UbiFlow can successfully achieve scalable mobility management and robust flow scheduling in IoT multinetworks.**

I. Introduction

Recent developments in wireless communications and embedded systems have resulted in consumer devices becoming highly ubiquitous creating a strong interest in the Internet of Things (IoT) as part of smart city solutions. Real world urban IoT applications are expected to be heterogeneous, due to various access networks and connectivity capabilities, resulting in geographically wide-scale *multinetworks* [1] where there is a coexistence of multiple wireless communication solutions (*e.g.* WiFi, Bluetooth, Cellular). Given the heterogeneity of IoT multinetworks, it is challenging to coordinate and optimize the use of the heterogeneous resources in mobile environments.

A. Motivations

Software Defined Networking (SDN) [2] is a relatively new paradigm for communication networks which separates the control plane (that makes decisions about how traffic is managed) from the data plane (actual mechanisms for forwarding traffic to the desired destinations); where control is handled by the SDN *controller*. This decoupling abstracts low-level network functionalities into higher level services, therefore allowing quick and flexible configuration for flow-based routing and enabling rescheduling over the network components. SDN is particularly useful when networks have to be adapted to ever changing traffic volumes with different demands. It is for this reason that we believe that SDN is a good approach to solving the resource management and access control issues in urban-scale IoT multinetworks.

This work was partially supported by the Intel Collaborative Research Institute for Sustainable Connected Cities (ICRI Cities).

OpenFlow [2] is the most prominent approach which implements the SDN concept, where the controller takes charge of all the functions in control plane, while the OpenFlow switch retains only the basic data forwarding functions. In the OpenFlow centralized control model, all routes are determined by the controller taking a global *view* of the network status. However, the request processing capability of a single controller is limited; for example NOX [3] can process about 30K requests per second. In fact, large-scale network environments (*e.g.* IoT applications in smart cities) have the potential to provide vast amounts of data flows; according to the report from Cisco, by 2016, there will be over 10 billion mobile-connected IoT devices and the monthly global mobile data traffic will surpass 10 exabytes [4]. With the increasing scale of IoT deployments, centralized controllers will have serious implications for scalability and reliability. Hence the next logical step is to build a distributed control plane with multiple physical controllers, which can provide the scalability and reliability yet preserves the simplicity of the control function.

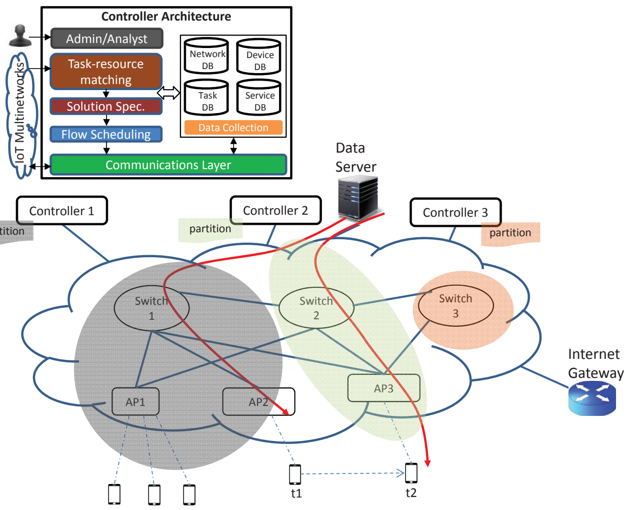


Fig. 1. UbiFlow system architecture.

Fig. 1 presents a software defined IoT system with the support of distributed controllers and partially connected OpenFlow switches in multinetworks with heterogeneous *access points*. However, the current implementations of SDN technologies are still far from addressing the heterogeneous and dynamic needs of ubiquitous IoT applications, especially in mobile environments. The popular use of SDN technologies today is in Data Center Networks (DCNs) [5], [6], where the focus is on the optimization of network behaviours (*e.g.*, bandwidth consumption) where nodes are linked via fast interconnections within a data center. In contrast, as shown in Fig. 1, in the urban-scale IoT multinetwork setting, state information is gathered from devices distributed over a more loosely coupled ubiquitous network.

Therefore, the main issues related to the application of software defined IoT are: (1) The operation of a distributed control plane requiring scalable control combined with consistent management to coordinate multiple controllers and switches for message exchange, while providing data replication and maintaining flow scheduling. This is especially challenging given IoT devices roam frequently in urban environments and each controller needs a network view about the mobility of these IoT devices to manage their spatio-temporal access requests and collaborate with other controllers for adaptive handover and dynamic flow scheduling over multinetworks. Where component failure or traffic congestion occurs, distributed controllers are required to be fault tolerant and able to load balance. (2) Unlike the DCN situation, link and node capabilities in IoT multinetworks are highly heterogeneous and application requirements are correspondingly different. This implies that single objective optimization techniques of typical DCN flow scheduling are not directly applicable in IoT multinetworks. Controllers should schedule the access point to transmit IoT flows based on specific per-device service requirements, while providing network traffic balance through the interactions between controllers and devices. (3) The performance metrics of interest in IoT multinetworks go beyond bandwidth consumption. With more heterogeneous and time-sensitive traffics; unlike DCNs, whose network requirements primarily focus on utilization and throughput, IoT multinetworks' metrics are delay, jitter, packet loss, and throughput.

B. Summary of Prior Work

Two popular approaches have been used in scalable SDN management. One is to design a distributed SDN architecture, such as Onix [7], where the network view is distributed among multiple controller instances. The alternative approach is to offload the partial workload of controllers to switches, as DevoFlow [5]. This approach can improve scalability to some extent, however the switch hardware is required to be modified. Nevertheless, all of the above scalable techniques are designed specifically for DCN, not designed for IoT multinetworks. More recently, SDN techniques are being applied to heterogeneous wireless networks, differently from traditional flow and access scheduling schemes for specific networks [8], [9]. OpenRadio [10] suggests the idea of decoupling the control plane from the data plane to support ease of migration for users from one type of network to another, in the PHY and MAC layers. The flow scheduling between WiFi and Bluetooth networks when video data is streamed has been prototyped in MINA [11], using centralized controller. OpenFlow based vertical handover is also discussed and implemented in the GENI testbed [12]. These wireless SDN solutions provide the necessary building blocks for managing IoT multinetworks, but they are not sufficient. Two important functions absent in these wireless SDN solutions are mobility management and distributed control.

C. Our Approach

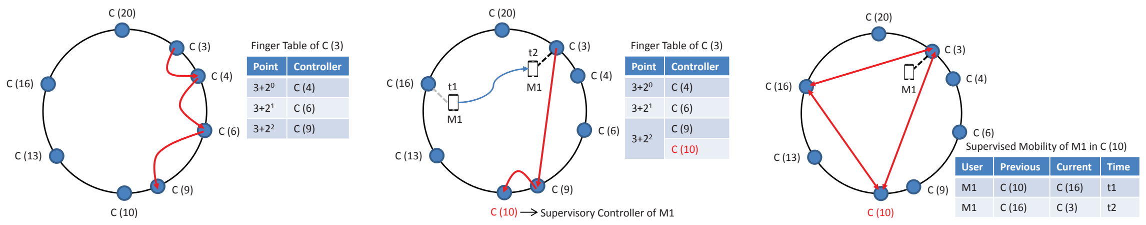
In this paper, we present UbiFlow, the first software-defined IoT system for ubiquitous flow control and mobility management in urban heterogeneous networks. To achieve light-weight processing in IoT devices, in UbiFlow all jobs related to mobility management, handover optimization, access point selection, and flow scheduling are executed by the coordination of distributed controllers. Specifically, UbiFlow adopts multiple controllers to divide an urban-scale SDN into different geographic *partitions* to achieve distributed control of IoT flows. A distributed hashing based overlay structure is proposed to maintain network scalability and consistency. Based on this UbiFlow overlay structure, relevant issues in mobility management such as scalable control, fault tolerance, and load balancing have been carefully examined and studied. The UbiFlow controller differentiates flow scheduling based on per-device equirements as well as whole-partition capabilities. Therefore, it can present a network status view for the optimized selection of access points in multinetworks to satisfy flow requests, while guaranteeing the network performance in each partition. The key contributions of UbiFlow are:

- A novel overlay structure to achieve mobility management and fault tolerance in software-defined IoT.
- An optimal assignment algorithm for the controller to match the best available access points to IoT devices, with network status analysis and flow requests as inputs.
- A load balancing scheme for distributed controllers by analysing the variations in flow traffic characteristics.

The rest of this paper is organized as follows. Section II gives a system overview. Section III addresses mobility management. Section IV addresses flow scheduling. Section V evaluates the performance. Section VI concludes our paper.

II. System Overview

The system architecture of UbiFlow is illustrated in Fig. 1, where the data server, controllers, switches, access points and IoT devices act as its core components. Multiple controllers are deployed to divide the network into several partitions, which represent different geographical areas. All IoT devices in a single partition associate with different types of access points (*e.g.* WiFi, WiMAX, Cellular), which are connected to local switches to request various types of data flow (*e.g.* text, audio, video) from the corresponding data server. Information pertaining to service requests and flow transmissions



Finger Table of C (3)

Point	Controller
$3+2^0$	C (4)
$3+2^1$	C (6)
$3+2^2$	C (9)

Finger Table of C (3)

Point	Controller
$3+2^0$	C (4)
$3+2^1$	C (6)
$3+2^2$	C (9)
	C (10)

Supervised Mobility of M1 in C (10)

User	Previous	Current	Time
M1	C (10)	C (16)	t1
M1	C (16)	C (3)	t2

(a) Overlay (b) Mobility (c) Handover

Fig. 2. Mobility management in UbiFlow overlay network.

can be analyzed and administrated by the partition-dependent controller. For urban-scale SDN, mobile IoT devices roam across different partitions at different times. Newly joining and leaving IoT devices are also recorded in the local controller to indicate user density and resource usage. Therefore, each controller has a partitioned view of its local network status.

The architecture of UbiFlow controller is also illustrated in Fig. 1. The data collection component collects network/device information from the IoT multinetwork environment and stores it in databases. This information is then utilized by the layered components in the controller. The task-resource matching component maps the task request (*e.g.* flow requirements) onto the existing resources (*e.g.* available access points) in the multinetwork. Once candidate resources are selected, the solution specification component adds more network characteristics and constraints (*e.g.* partition view) to filter resources. Finally, the flow scheduling component takes these requirements and schedules flows that satisfy them.

In UbiFlow architecture, switches from different partitions are partially interconnected, so that the network information recorded in different controllers can be exchanged through these connected switches to achieve network consistency and robust maintenance. In addition, connected switches can also facilitate the inter-controller flow migration over the IoT multinetwork for load balancing purpose. In general, there are two types of IoT flows in the context of a distributed SDN as shown in Fig. 1. The first one is the IoT flow between the data server and the IoT device. This is scheduled through intra-partition communication, with the assistance of a local access point, switch and controller. The second one is the IoT flow between IoT devices located with different partitions. This type of IoT flow needs to be scheduled through inter-partition communication. Utilizing the connected switches, controllers can coordinate to direct the flow initiated from one partition to a different access point in another partition. Note that IoT device to IoT device multi-hop wireless communication (*e.g.* ZigBee, WiFiDirect) also exists in the UbiFlow system. If this happens in the same partition and the last hop is directed to an access point, then it can be classified as the first type of IoT flow. Otherwise, if it is purely multi-hop wireless communication without SDN support, it does not belong to the discussion of this paper, since we focus on using SDN to improve the network performance.

Given the UbiFlow architecture, we will discuss its mobility management and flow scheduling in the following sections.

III. Mobility Management in UbiFlow

When IoT devices roam from one partition to another, a consistent scheme to coordinate controllers is required for the mobility management of IoT devices. UbiFlow presents an overlay structure based mobility solution, as shown in Fig.2. We will illustrate its key functions in the following sections.

A. Overlay Structure

Two types of IDs are used in the mobility management:

- *Mobile ID*: the ID of a mobile IoT device (*e.g.* IP v6 address or MAC address);
- *Controller ID*: the ID of a controller in distributed SDN.

To provide scalable and efficient mobility management, UbiFlow maintains a controller network based on structured overlays (*e.g.* Chord DHT [13]), where a *consistent hashing* [13] is maintained based on an ordered ring overlay, as shown in Fig.2 (a). In the consistent hashing framework, distributed controllers are configured as overlay nodes with unique integer identifiers in the range of $[0, 2^{m-1}]$. Each controller ID can be represented by m bits. The consistent hashing also matches each mobile ID with an m-bit integer as a "key" using a base hash function h, such as SHA-1 [13]; therefore key = h (Mobile ID). The key can be later used for the lookup of controllers, as explained in Section III-B.

Each controller $C(n)$ with ID n maintains a routing table, namely the "finger table", to achieve scalable key lookup in this overlay structure. Each finger table has up to k entries. The ith entry in the table indicates the closest controller to the corresponding point, where the controller ID $\geq (n + 2^{i-1})$. A query for a given key is forwarded to the nearest node that most immediately precedes the key, among the k entries at the controller. Finger tables are used for the case where there is no controller with the exact ID as the key value. In that case, we designate the closest successor of the key as the expected controller. For example, in Fig.2 (a), we represent the controller with ID n as $C(n)$, and there are 3 entries in the finger table of $C(3)$. The 3rd entry of the finger table points the successor of the key $(3 + 2^2)$, which is $C(9)$.

Theorem 1: In an N-controller overlay network based on consistent hashing, the lookup cost to find a successor is bounded by $O(\log N)$.

Proof: The lookup cost in an N-controller overlay network indicates the number of nodes that must be contacted to find a successor. The above theorem has been proved in the Chord structure [13], also based on consistent hashing. ■

B. Mobile Handover

In the SDN based mobility management, we achieve efficient handover through the coordination between controllers. Specifically, we classify SDN controllers as two types:

- *Associated Controller*: the current controller that the mobile IoT device is associated with;
- *Supervisory Controller*: the controller that is assigned to a newly joined IoT device as its initially associated controller. Note that each supervisory controller also functions as an associated controller but with additional information to record the mobility behaviour of its supervised IoT devices. The updated mobile information of an IoT device could be collected through information exchange with its current associated controller, following our UbiFlow architecture as described in Section II

When a new IoT device joins the distributed SDN network, as a bootstrapping step, it will be assigned to a supervisory controller as its initially associated controller, based on the hash result of its mobile ID. In the mobility scenario, for each IoT device with its mobile ID as the original value, the UbiFlow overlay structure can hash the mobile ID to get an integer key, and use this to localize its supervisory controller. Both the controller ID and the hashed key of the mobile user are required to be placed in the same ID space ranging $[0, 2^{m-1}]$. Specifically, to localize the supervisory controller, we follow the rule to assign a hashed key to the controller that has the closest ID, namely the immediate successor of the key.

Since every controller can use consistent hashing to localize an IoT device's supervisory controller, the supervisory controller is used in UbiFlow to record the previous and current associated controllers of the mobile IoT device. Using this scheme for distributed SDN, the new associated controller can localize the previous associated controller of the IoT device. As shown in Fig.2 (b), mobile IoT device 1, denoted as $M1$, was previously associated with controller $C(16)$ at time $t1$, and its supervisory controller is $C(10)$. When $M1$ moves to the geographical partition of $C(3)$ at time $t2$, $C(3)$ needs to localize its previous associated controller and reroute flows to its current partition. To do this, $C(3)$ first tries to localize the supervisory controller according to the hashed key of $M1$. Based on the finger table, $C(3)$ can forward the lookup request to the furthest controller $C(9)$ that is closer to the supervisory controller. Then $C(9)$ can help to localize $C(10)$ as the expected supervisory controller. As shown in Fig.2 (c), once $C(3)$ localizes $C(10)$ from the traceback route, as $C(10) \rightarrow C(9) \rightarrow C(3)$, it can later directly communicate with $C(10)$ to learn previous associated controller of $M1$ is $C(16)$ at $t1$. After this, $C(3)$ can directly communicate with $C(16)$ to fetch the previous session between $M1$ and $C(16)$ and reroute flows to current partitions. As for $C(10)$, it will also update the current associated controller of $M1$ to be $C(3)$ at $t2$, and notify $C(16)$ to end the previous session for $M1$.

Note that as a mobile IoT device in the urban scenario, $M1$ may leave the partition of $C(3)$ and re-enter again with unpredictable mobility, therefore in the UbiFlow overlay structure, there is an extra entry in the finger table of each controller to label all the supervisory controllers of its current and previously associated IoT devices with a TTL (time-to-live). Hence, when previously associated IoT device enters its partition again, the controller does not need to initiate another multi-hop request to localize the supervisory controller. Instead, the controller can localize the supervisory controller using fast lookup in its finger table, which will further save the communication cost and improve the efficiency of handover.

Proposition 1: The mobile lookup cost to find the previous associated controller for an IoT device in UbiFlow could be either $O(\log N) + 1$ or $O(2)$.

Proof: Theorem 1 has proven that the usually lookup cost in a consistent hashing is bounded to $O(\log N)$. Since supervisory controller records previous association of supervised devices, the normal mobile lookup cost to find the previous associated controller for an IoT device is $O(\log N) + 1$, by localizing supervisory controller first and then requesting local lookup in the supervisory controller. If the supervisory controller was found before and has been recorded in the local finger table as the additional information, the lookup cost for the corresponding mobile device is then just a local lookup as $O(2)$, with one step to reach the supervisory controller and one step to request local lookup in the supervisory controller. ■

C. Scalable Control

To achieve scalable mobility management by multiple controllers in distributed SDN, we focus on the Join and Leave operations of controllers, as follows:

- *Join*: When a new controller with ID n joins an existing SDN, it first identifies its successor by performing a lookup in the SDN according to its ID. Once it localizes the successor, it selects the successor's keys that the new controller is responsible for. After this, the new controller sets its predecessor to its successor's former predecessor, and sets its successor's predecessor to itself. Meanwhile, an initial finger table will be built in the joined controller by performing lookup points $(n + 2^i - 1)$, for i=1, 2, $\ldots k$, where k is the number of finger print entries.
- *Leave*: When a controller with ID n wants to leave an existing SDN, it first moves all keys that the controller is responsible for to its successor. After this, it sets its successor's predecessor to its predecessor, and sets its predecessor's successor to its successor. For consistency purposes, before the controller leaves the distributed network, the SDN related control information (*e.g.* network status and flow status) in the controller will be copied to its successor by default, and other controllers can later update their finger tables by replacing controller n with its successor in the corresponding entry. If the controller also performs as the supervisory controller for some IoT devices, its successor will be also designated as the new supervisory controller for these IoT devices, and it records the existing mobility information from the leaving controller.

D. Fault Tolerance

To handle failure in the distributed SDN, we tackle failures of different components in UbiFlow. As for controller level failure, we adopt data replication to achieve robust control. That is, we copy the data from local controller n to its r live successors in UbiFlow overlay, by searching key $(n+2^{i-1})$, for i=1, 2, ... r. The r successors also update these replications periodically. So, if the local controller fails, we can find a new successor that still can provide the control service.

As for finger-table level failure, we adaptively choose alternate paths while routing. If a finger does not respond, we take previous fingers in the local table, or finger-table replicas from one of the r successors. The local finger table also performs self-check to refresh all fingers by periodically looking-up the key $(n+2^{i-1})$ for a random finger entry i. The periodic cost is $O(\log N)$ per controller due to the finger refresh.

As for access-point failure, we designate an associated controller to detect the failure and redirect flows going through failed access points to others in its partition. We address the optimal assignment of access point to IoT device in Section IV.

IV. Flow Scheduling in UbiFlow

Given flow requirements from an IoT device, the UbiFlow controller needs to find an access point that can both satisfy the flow request of the IoT device and guarantee optimal network performance of the whole partition. The relevant design to achieve robust flow scheduling is described in this section.

A. Network Calculus based Partition View

The UbiFlow controller of each partition needs a partition view via obtaining current network status within this partition for flow scheduling. To guarantee the performance of software defined IoT flow scheduling with various flow requirements, UbiFlow controller employs Network Calculus [14] to describe the arrival traffic pattern($A(t)$), served traffic pattern($S(t)$), and departure traffic pattern($D(t)$) on a network node during the time interval [0,t) as the partition view in its partition. We assume that each node has a constant bandwidth capacity (transmission rate) R and can provide a service curve $S(t) = R[t-T]^+$, where $[x]^+ = \max\{0, x\}$, and T is the transmission delay, which is the time between the first bit of the packet received and the last bit of the packet sent by this node. T depends on R, the length of this packet, and the amount of data currently in the node queue. We can use min-plus convolution on arrival and service curves, to generate a departure curve: as $D(t) = A(t) \otimes S(t)$, which means: $D(t) \geq \inf_{s\leq t}(A(s) + S(t-s))$.

If there are multiple flows going through a node, all flows share the same transmission service. Here each intermediate node is assumed to have a FIFO scheduler–packets are served in the sequence as they arrived. Flow i will have a leftover service curve: $S_i = \dfrac{\theta^i}{\sum_{j\neq i}\theta^j}R[t-T]^+$, where R is the transmission rate, and θ is the weight of each flow; In a multi-hop path, the departure curve of the current hop is the arrival curve of the next hop, and a combination service curve along the path $S(t)$ can be obtained by iteratively adding each node's service curve using the associative operation in min-plus convolution, as: $S(t) = S_1 \otimes S_2 \otimes \ldots \otimes S_n$.

In order to provide a better partition view of the traffic, UbiFlow models the traffic as a set of disjoint points (each point represents a packet) in Network Calculus. It assumes that the profile of each flow (*e.g.*, packet length and sending time) is known at each sender, and each packet is served by the service curve $S(t)$ with a constant packet rate R and a delay T. Upon packet arrival, we check the current queue state in terms of the number of packets in the queue and their lengths. The delay T is the time needed for sending out all packets that are already in the queue. Hence the total delay of a packet consists of two parts: one is T and the other is the transmission (service) time of the packet itself. In this way, we can get an approximate end-to-end delay for each packet. In our verification, we examine three QoS parameters: delay, throughput, and jitter. For each flow, we use the profile points at the sender side to plot the curve. We can get the arrival curve $D(t)$ of flow i at the destination node by the modified Network Calculus model, and then we compare it with flow i's initial arrival curve A(t). Each point (packet) will have a delay and a recorded arrival time at the destination node. The average delay, average jitter, and total throughput for each flow can be calculated by UbiFlow controller accordingly. Therefore, the modified Network Calculus model can be used by UbiFlow controller to obtain the partition view.

B. IoT Multinetworks Matching

After obtaining the partition view of current network status from the network calculus model, the UbiFlow controller can manage handover between heterogeneous access networks by assigning newly joined mobile IoT devices to the best access point, based on the current multinetwork capacity in the controlled partition, the supported radio access technologies and the types of services the mobile devices are requesting. We formulate the assignment of a set of newly joined mobile IoT devices $\mathbb{MD}$ to a set of access points $\mathbb{AP}$ as a generalized assignment problem (GAP). Each access point j is characterized by a residual bandwidth capacity function $B(j)$, and each mobile device i is characterized by a bandwidth demand function $d(i,j)$ that describes the bandwidth demand of device i when assigned to access point j. A utility function $u(i,j)$ measures the benefit obtained by the system as a result of assigning a mobile device i to access point j. The assignment problem is formulated as:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j\in\mathbb{AP}}\sum_{i\in\mathbb{MD}} u(i,j)x(i,j) \\
\text{subject to} \quad & \sum_{i\in\mathbb{MD}} d(i,j)x(i,j) \leq B(j), \forall j \in \mathbb{AP} \\
& \sum_{j\in\mathbb{AP}} x(i,j) \leq 1, \forall i \in \mathbb{MD} \\
& x(i,j) = 0 \ or \ 1, \forall i \in \mathbb{MD}, \forall j \in \mathbb{AP}
\end{aligned}$$

where $x(i,j) = 1$ if device i is assigned to access point j or 0 otherwise.

As the set of access points and especially the set of mobile devices changes dynamically, the assignment is done in an adaptive time-window based manner. The assignment is performed at the end of each window using (1) the capacity, demand and utility functions evaluated at that time, (2) the set of newly joined mobile devices within the window of time, and (3) the set of active access points at that time. Algorithm 1 is an adaptation of the GAP approximation in [15] combined with a greedy heuristics for the Knapsack problem that sorts items based on their utility-to-demand ratio and tries to pack as much high-ratio items as possible. It takes the sets of access points and devices, a matrix of utilities and demands, and a vector of capacities as an input. It starts by checking the residual capacity (capacity at the end of the time window) of the set of access points ($\mathbb{AP}$) against the demand vector ($D_*[ap]$) of mobile devices with respect to that access point type, and creating a feasible set of access points ($\mathbb{AP}_f$) by selecting those that can at least satisfy the minimum demand. It then initialises the assignment vector (X) and iteratively computes the assignment as follows. For each access point, it creates a utility vector from the utility matrix using either the original utility value or the difference in utility depending on whether the mobile device is assigned to an access point in the previous iteration. The utility-to-demand ratio is then computed using the utility vector, and the set of mobile devices ($\mathbb{MD}$) is sorted in non-decreasing order based on this ratio. Using this ratio and a greedy Knapsack packing scheme, the mobile devices are assigned to the current access point. This is repeated until all access points are exhausted. The vector X is the result of the assignment where $X[md]$ indicates that mobile device md is assigned to access point $X[md]$ if $X[md]$ is not -1.

The UbiFlow controller determines each mobile device's compatibility (*i.e.* support for the radio access technology used by the access point) with access points and requirement with respect to quality of service such as bandwidth demand (d) and the maximum tolerable latency (l_t) based on the types of services the device is trying to access. The demands of IoT devices can be obtained during their request processes, and the partition status can be derived from the network calculus model, as described in Section IV-A. If there is compatibility between a device and an access point, the degree of satisfaction of a mobile device, if assigned to the access point, with respect to these requirements is modeled by utility functions namely u_d and u_l respectively. In addition, a utility function u_a that measures the load (*i.e.* the number of mobile devices) on an access point is used in order to take the degree of distribution of load into consideration so that one capable access point will not be overloaded. Given an access point with latency l and N mobile devices already assigned to it, the utility functions are:

$$u_d(i,j) = log\left(1 + \frac{B(j)}{d(i,j)}\right), d(i,j) > 0$$

$$u_l(i,j) = log\left(1 + \frac{l_t}{l}\right), l > 0$$

$$u_a(j) = log\left(1 + \frac{|\mathbb{MD}|}{N + |\mathbb{MD}|}\right), \mathbb{MD} \neq \emptyset$$

Using these utility functions, the controller computes the utilities for each potential assignment, normalizes them and then computes the total system utility using predefined positive weights that capture the significance of each type of utility as:

$$u(i,j) = \begin{cases} w_d\hat{u}_d(i,j) + w_l\hat{u}_l(i,j) + w_a u_a(j) & \text{compatible} \\ 0 & \text{otherwise} \end{cases}$$

where $\hat{u}_d$ and $\hat{u}_l$ are the normalized utilities and $w_d + w_l + w_a = 1$. It then performs assignments that would maximize the overall system utility.

Algorithm 1 Mobile Device to Access Point Assignment

```
Input: $\mathbb{AP}$, $\mathbb{MD}$, U, D, $C$
Output: $X$
 1: for $ap \in \mathbb{AP}$ do
 2:     if $C[ap] \geq min\{D_*[ap]\}$ then
 3:         $Add\ ap\ to\ \mathbb{AP}_f$
 4:     end if
 5: end for
 6: for $r = 1$ to $|\mathbb{MD}|$ do
 7:     $X[r] = -1$
 8: end for
 9: for $ap \in \mathbb{AP}_f$ do
10:     for $md \in \mathbb{MD}$ do
11:         if $X[md] == -1$ then
12:             $U_{ap}[md] \leftarrow U[md][ap]$
13:         else
14:             $U_{ap}[md] \leftarrow U[md][ap] - U[md][X[md]]$
15:         end if
16:         Compute utility-to-demand ratio vector: $R_{ap}[md] = \frac{U_{ap}[md]}{D[md][ap]}$
17:     end for
18:     Sort $\mathbb{MD}$ such that $R_{ap}[md]$ is in non-increasing order
19:     $b = min\{q \in \{1, ..., |\mathbb{MD}|\} : \sum_{r=1}^{q} D_{ap}[r] > C[ap]\}$
20:     for $q = 1$ to $b - 1$ do
21:         $X[md] = ap$
22:     end for
23: end for
```

The algorithm has a time complexity of $O(|\mathbb{AP}||\mathbb{MD}|\log(\mathbb{MD}))$ when an $|\mathbb{MD}|\log(\mathbb{MD})$ algorithm is used to sort the utility-to-demand ratio vector. The proof directly follows from [15].

C. Load Balancing

One key limitation of existing SDN systems is that the mapping between a switch and a controller is statically configured, making it difficult for the control plane to adapt to temporal and spatial traffic load variations. As load imbalance occurs, it is desirable to migrate a switch from a heavily-loaded controller to a lightly-loaded one. Following our architecture as illustrated in Fig.1, UbiFlow consists of a cluster of autonomous controllers that coordinate amongst themselves to provide a consistent control logic for the entire network. We can design a robust load balancing scheme based on the UbiFlow architecture to dynamically shift the load across switches and controllers.

Given a controller n, if new flow requests, collected from local IoT devices, cause traffic imbalance (*e.g.* over maximum capacity, longer process delay) controller n needs to switch the flow to a lightly-loaded controller. However, the usual linear balancing scheme that relays the flow request to one of its r successors is not robust enough in the mobile SDN scenario, because the r successors have locally loaded flows and these may be heavily-loaded as well. Furthermore, the fault tolerant scheme presented in Section III-D will generate redundant data in the r successors, so additional flow requests from other partitions tend to cluster the requests of the flows into contiguous runs, which may even overlap in our circular overlay structure. In addition, because of the importance of the supervisory controller, if the supervisory controller is heavily-loaded and cannot accept other newly joined IoT devices, we also need a scheme to mitigate the traffic flow on this supervisory controller by directing the flows for new IoT devices to other controllers as a backup supervisory controller. Meanwhile, we need a consistent scheme for other controllers to be able to localize these backup supervisory controllers.

To avoid the linear clustering of heavily-loaded controllers and guarantee system consistency in the UbiFlow overlay, we use double hashing to balance a large number of flow requests and distribute them fairly in the overlay structure. Specifically, different from the hash function h used in the finger key search, we choose a secondary hash function, h' for collision handling. If h maps some finger key k to a controller $C[i]$, with $i = h(k)$, that is already heavily-loaded, then we iteratively try the controllers $C[(i+f(j)) \mod P]$ next, for $j = 1, 2, 3, \ldots$, where $f(j) = jh'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \mod q)$, for some prime number $q < P$. Also, P should be a prime number.

Note that, for supervisory controller, the heavy load status may cause local failure, but its mobility records for IoT devices are important for the mobility management. When UbiFlow observes the load imbalance in a supervisory controller, it also uses the double-hashing scheme to copy the mobility information to other controllers as a backup. By this way, UbiFlow can effectively protect the mobility information, in case consecutive failures happen and the redundancy scheme in Section III-D fails.

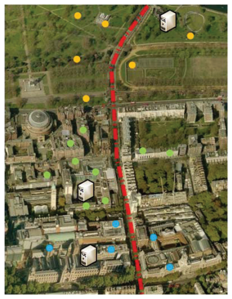
V. Performance Evaluations

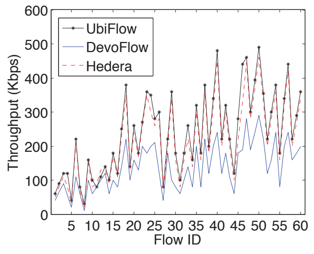
We have implemented a prototype of UbiFlow, and evaluated its performance on flow scheduling and mobility management by both simulation and real testbed experiments.

A. Simulation Results

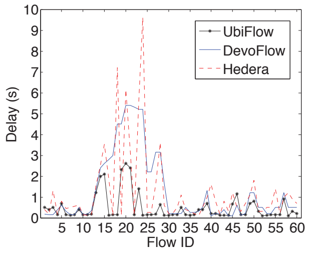
Recently, OMNeT++ [16] has incorporated the OpenFlow module for SDN simulation. However, its controller only supports the wired data center networks and lacks mobility management. We have extended the functions of the SDN controller with UbiFlow framework, and used it for the evaluation of mobility management in heterogeneous IoT multinetworks.

To verify the performance of UbiFlow in urban scenario, our simulation is based on a popular South Kensington area in the city of London, which consists of several parks, universities, and museums, as shown in Fig. 3 (a). This area is usually crowded by high density of tourists, students and workers, with large number of IoT devices and various types of flow requests. Therefore, in our first set of evaluation, three controllers have been deployed in park partition, university partition and museum partition, respectively, for flow scheduling and mobility management. The backbone topology consists of 3 data servers (each of the three data servers provides either file sharing, audio, or video streaming services), 3 switches (each switch has a 1Gbps Ethernet link to one server; each controller directly controls one switch), and 20 access points (each access point has one 100Mbps Ethernet link to every switch). There are three types of access points: WiMAX, WiFi and Femtocell, with data rates 30Mbps, 10Mbps, and 2Mbps respectively. Each IoT device has three network interfaces to directly connect with corresponding access points, and at each time instance only one interface can be used.

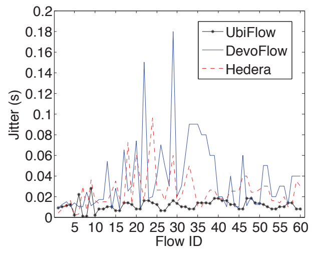

(a) Urban Scenario



(b) End-to-End Throughput



(c) End-to-End Delay



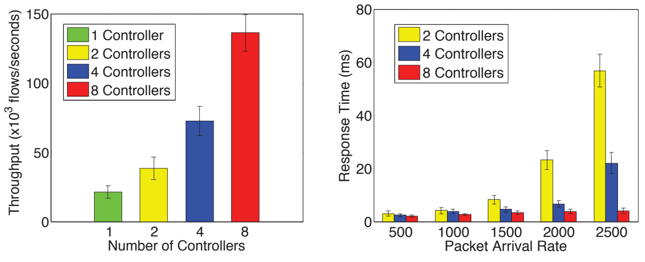
(d) End-to-End Jitter

Fig. 3. Mobile flow scheduling in UbiFlow.

1) Handover in UbiFlow: In our first set of simulation, as shown in Fig. 3 (a), there are 5 access points (orange dots) in the park partition, 9 access points (green dots) in the university partition, and 6 access points (blue dots) in the museum partition. Some of these access points are already under heavy traffic load, and others still have enough capacity. Assume there are 60 IoT devices sending new flow requests at a time, and they are moving along the red path. 10 of them request file sharing services, 20 of them request audio services, and 30 of them request video streaming services. In our evaluation, file sharing flows are modeled by sending Constant Bit Rate with packet length uniformly distributed in [100, 1000] bytes with period T, the latter uniformly distributed in [0.01, 0.1] seconds. Audio and video streaming flows are from real traffic traces [17], [18]. For practical applications, the file sharing service requires large throughput, the audio service requires low delay, while the video streaming service requires low jitter. We evaluate our UbiFlow scheduling and compare it with other two common scheduling algorithms used in SDN world: DevoFlow [5] and Hedera [6]. The former tries to accommodate as many flows as possible into a single link to maximize the link utilization. Instead, the latter assigns flows into a link so that the total amount of the flows are proportional to the capacity of the link.

As shown in Fig. 3, we have totally 60 flows (each of 60 end devices has one flow): flows 1-10 are file sharing, flows 11-30 are audio, and flows 31-60 are video streaming. Fig. 3 (b) shows the comparison of flow throughput. For file sharing flows, UbiFlow outperforms DevoFlow by an average of 67.21%, while it has an average of 15.91% throughput increase if compared with Hedera. The reason is that in wireless links when link utilization exceeds a threshold, the packet drop rate increases dramatically. The load balancing scheme in UbiFlow uses the controller to schedule flows according to the utilization status of each access point; therefore it can achieve comparably fair allocation of flow traffic to decrease packet drop rate. Fig. 3 (c) shows that for audio flows, our proposed algorithm can improve the end-to-end delay performance by 72.99% and 66.79%, compared to DevoFlow and Hedera respectively. Audio flows have bursty traffic patterns; it might not have big data volume, but if two flows are scheduled with similar bursty patterns in the same link, a large delay occurs. Due to the traffic-aware dynamic flow scheduling scheme, UbiFlow can schedule flows both by the consideration of partition load and device requirement; therefore it can reduce the impact of flow interference. Fig. 3 (d) shows that video streaming flows have an average 69.59% and 49.72% less jitter with UbiFlow than DevoFlow and Hedera. Because of the holistic solution in flow scheduling and mobility management, distributed controllers in UbiFlow can provide more stable video flow for IoT devices.

2) Scalability in UbiFlow: In the implementation of OpenFlow, the Packet-In message is a way for the OpenFlow switch to send a captured packet to the controller. A flow arrival resulting in sending a Packet-In message to the controller. In the second set of simulation, we use Packet-In message to evaluate the scalability of flow scheduling by UbiFlow.



(a) Flow Throughput (b) Flow Delay

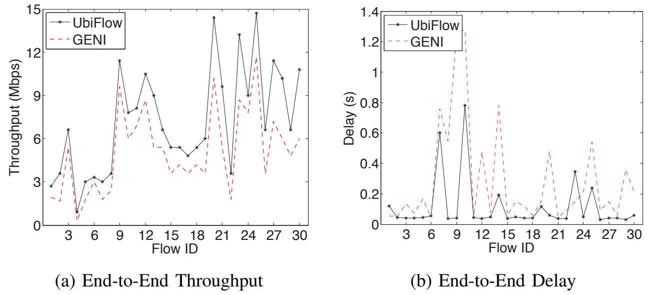
Fig. 4. Scalability in UbiFlow.

For better scalability evaluation, we add more controllers in the above urban scenario. In addition, for every controller in its partition, the controller is directly connected with 3 to 5 switches, and controls 20 to 50 access points with various heterogeneous interfaces. For each controller, we send 10000 consecutive Packet-In messages to it and plot the throughput of UbiFlow with varying number of controllers, as shown in Fig. 4 (a). We observe that adding controller nodes increases the throughput almost linearly. This is because in the architecture of UbiFlow, as shown in Fig. 1, each controller mainly controls the traffic flows in its own partition. However, if there is an imbalance in one controller, other controllers with light-weight traffic also can help to migrate the flows to their partitions by physically partial connected switch and the UbiFlow overlay structure. To further illustrate the scalability of UbiFlow, we also plot the response time behaviour for Packet-In messages with changing flow arrival rate, as shown in Fig. 4 (b). We repeat the experiment while changing the number of controller nodes. As expected, we observe that response time increases marginally up to a certain point. Once the packet generation rate exceeds the capacity of the processor, queuing causes response time to shoot up. This point is reached at a higher packet-generation rate when UbiFlow has more number of nodes.

B. Testbed Experiment Results

In our real testbed experiments, we use ORBIT [19] as the wireless network testbed to evaluate UbiFlow. ORBIT is composed of 400 radio nodes, where a number of experimental "sandboxes" can be accessed via its management framework. Available sandboxes include WiFi, WiMAX, USRP2, etc. ORBIT supports Floodlight [20] based OpenFlow controller to switch access between the WiFi and WiMAX interfaces, and uses Open vSwitch (OVS) [21] to allow a network interface to be OpenFlow-enabled.

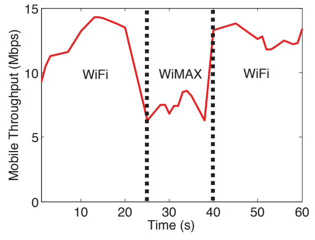
We choose an ORBIT sandbox with 1 WiMAX node and 7 WiFi nodes in our experiments. We are aware that real mobile access pattern of IoT devices in urban scenario does not follow the random waypoint model. Actually, the urban-scale access of multinetworks is more like event or motivation driven behaviour. To better evaluate UbiFlow in this kind of mobile scenario, we collected a campus-wide mobile trace driven by class events, and use it in our evaluation. Specifically, the trace is collected during a period (10 minutes) between two consecutive classes around a lecture building. During that period, some students leave the building after previous class, some students come to the building for incoming class, and some students still stay in the building. Therefore, the wireless access of their IoT devices can be classified as "leaving", "joining", and "staying". We match the 8 OpenFlow-enabled ORBIT nodes as corresponding access points in the building, and use two Floodlight based OpenFlow controllers to scheduling different service requests from more than 300 IoT devices during that period, according to the mobile trace file.



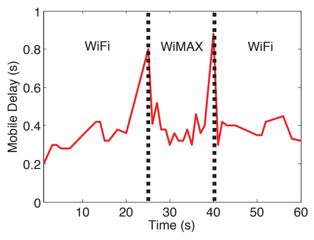
(a) End-to-End Throughput (b) End-to-End Delay

Fig. 5. Mobile flow scheduling in real testbed.

We compare UbiFlow with an OpenFlow-based handover scheme proposed by GENI [22] (namely GENI). The GENI handover [12] is a vanilla implementation of SDN in wireless environment, without ubiquitous flow scheduling and mobility management. As shown in Fig. 5, we select 30 flows from the hundreds of active IoT devices, where flows 1-5 are file sharing, flows 6-15 are audio, and flows 16-30 are video streaming. The performance shows the similar results as previous simulation results with various flow types. Generally, UbiFlow outperforms GENI handover both on end-to-end throughput and delay evaluation. For the 30 flows, UbiFlow can achieve an average throughput as 7.24 Mbps, while GENI only can provide 5.09 Mbps; UbiFlow improves the average throughput performance by 42.24%. The average delay in UbiFlow is around 0.11 s, while the delay in GENI is 0.29 s; UbiFlow reduces the average delay by 62.07%. In comparison with GENI, UbiFlow adopts dynamic flow scheduling scheme from the views of partition and device aspects, therefore can achieve better assignment of access points to satisfy different flow requirements of IoT devices. In addition, the overlay structure based load balancing can effectively allocate flows in UbiFlow, by the coordination of controllers and switches. It also can help to improve the throughput and reduce the delay.



(a) Mobile Throughput



(b) Mobile Delay

Fig. 6. Mobility management in real testbed.

To test the mobility management of UbiFlow in real testbed, we choose one mobile device and evaluate the change of its multinetwork access in a period of one minute, while associating with different types of access points. The performance of throughput and delay of its access is shown in Fig. 5 (a) and (b) respectively. As we can see, since there is only one WiMAX node in our testbed, and it is crowded by other mobile users, the throughput provided by WiMAX is much lower than WiFi nodes. According to this situation, the SDN controller only assigns the mobile device to access WiMAX when there is no available WiFi access points providing higher data rate. Once the controller finds a WiFi access point with better capacity and the mobile device sends flow request in its range, it will assign the mobile device to access the WiFi node. In mobile scenario, we notice that the average flow transmission delay for this mobile device is below 0.4s, which presents stable performance of our mobility management, considering there are hundreds of active IoT devices and only 8 working access points. Mobile delay only increases obviously when UbiFlow runs handover steps to assign new access point to the mobile device, which happens at the 25th second and 40th second of this period. Usually, when a mobile device requests an access point, it will initially send the request to the controller, and then controller sends the assignment decision back. This process results in the extra delay for message exchange and computation, which cannot be avoided if we use the controller to match access points with mobile devices. However, in these special cases, UbiFlow still can achieve a handover delay less than 0.9 seconds, therefore shows satisfactory results.

VI. Conclusion

UbiFlow is a software-defined IoT system for efficient flow control and mobility management in urban multinetworks. In addition to flow scheduling, it shifts mobility management, handover optimization, and access point selection functions from the relatively resource constrained IoT devices to more capable distributed controllers. The distributed controllers are organized in a scalable and fault tolerant manner. The system was evaluated through simulation and testbed.

References

[1] Z. Qin, L. Iannario, C. Giannelli, P. Bellavista, G. Denker, and N. Venkatasubramanian, "Mina: A reflective middleware for managing dynamic multinetwork environment," in *IEEE/IFIP NOMS*, 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 2, pp. 69–74, 2008.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[4] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011-2016 ," Tech. Rep., 2012.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011, pp. 254–265.

[6] M. Al-Fares, S. Radhakrishnanl, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *USENIX NSDI*, 2010.

[7] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *USENIX OSDI*, 2010, pp. 351–364.

[8] D. Wu, L. Bao, and C. H. Liu, "Scalable Channel Allocation and Access Scheduling for Wireless Internet-of-Things," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3596–3604, 2013.

[9] D. Wu, S. Yang, L. Bao, and C. H. Liu, "Joint multi-radio multi-channel assignment, scheduling, and routing in wireless mesh networks," *Wireless Networks*, vol. 20, no. 1, pp. 11–24, 2014.

[10] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "Openradio: a programmable wireless dataplane," in *ACM HotSDN*, 2012, pp. 109–114.

[11] Z. Qin, G. Denker, C. Giannelli, P. Bellavista, and N. Venkatasubramanian, "A software defined networking architecture for the internet-of-things," in *IEEE/IFIP NOMS*, 2014, pp. 1–9.

[12] R. Izard, A. Hodges, J. Liu, J. Martin, K. Wang, and K. Xu, "An OpenFlowTestbed for the Evaluation of Vertical Handover Decision Algorithms in Heterogeneous Wireless Networks," in *Proc. of the 9th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, 2014.

[13] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[14] J.-Y. L. Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[15] R. Cohen, L. Katzir, and D. Raz, "An efficient approximation for the Generalized Assignment Problem," *Information Processing Letters*, vol. 100, no. 4, pp. 162–166, 2006.

[16] OMNeT++, http://www.omnetpp.org.

[17] Skype tele audio trace files, http://tstat.polito.it/traces-skype.shtml.

[18] Video streaming trace files, http://trace.eas.asu.edu/TRACE/ltvt.html.

[19] ORBIT, https://www.orbit-lab.org.

[20] Project Floodlight, http://www.projectfloodlight.org/floodlight.

[21] Open vSwitch, http://openvswitch.org.

[22] GENI, http://www.geni.net.