Locality-Aware Load Sharing in Mobile Cloud Computing

Albert Jonathan
University of Minnesota
Minneapolis, MN
albert@cs.umn.edu

Abhishek Chandra
University of Minnesota
Minneapolis, MN
chandra@cs.umn.edu

Jon Weissman
University of Minnesota
Minneapolis, MN
jon@cs.umn.edu

ABSTRACT

The past few years have seen a growing number of mobile and sensor applications that rely on Cloud support. The role of the Cloud is to allow these resource-limited devices to offload and execute some of their compute-intensive tasks in the Cloud for energy saving and/or faster processing. However, such offloading to the Cloud may result in high network overhead which is not suitable for many mobile/sensor applications that require low latency. So, people have looked at an alternative Cloud design whose resources are located at the edge of the Internet, called Edge Cloud. Although the use of Edge Cloud can mitigate the offloading overhead, the computational power and network bandwidth of Edge Cloud's resources are typically much more limited compared to the centralized Cloud and hence are more sensitive to workload variation (e.g., due to CPU or I/O contention). In this paper, we propose a locality-aware load sharing technique that allows edge resources to share their workload in order to maintain the low latency requirement of Mobile-Cloud applications. Specifically, we study how to determine which edge nodes should be used to share the workload with and how much of the workload should be shared to each node. Our experiments show that our locality-aware load sharing technique is able to maintain low average end-to-end latency of mobile applications with low latency variation, while achieving good utilization of resources in the presence of a dynamic workload.

KEYWORDS

Mobile Cloud Computing, Edge Cloud, Load Sharing

ACM Reference Format:
Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2017. Locality-Aware Load Sharing in Mobile Cloud Computing. In *Proceedings of UCC '17: 10th International Conference on Utility and Cloud Computing (UCC '17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3147213.3147228

1 INTRODUCTION

The past few years have seen a growing number of devices that are connected to the Internet. The type of devices varies from statically installed public sensors such as smart traffic light systems and weather forecasting sensors to privately owned mobile devices [1] such as smart phones, smart watches, wearable health sensors, etc. [1, 3, 36, 39]. This rapid growth is predicted to continue increasing as Cisco estimates that there will be approximately 50 billion devices that are connected to the Internet by 2020 [10].

Today's mobile devices are still facing challenges due to their limited resources such as CPUs, storage, and battery power. Yet, these resources will be unable to satisfy most of today's mobile applications that require low latency to the users while they constantly produce or consume data [9, 21, 30]. Recent research has looked at the opportunity of integrating Cloud Computing platforms that provide much more powerful computation and/or storage resources to assist many mobile applications. Furthermore, people have looked at an alternative Cloud design consisting of resources that are located at the edge of the Internet, called *Edge Clouds* [4, 18, 31, 35, 40, 43]. The main benefit of using an Edge Cloud is to mitigate the overhead of computational offloading since they are closer to the users [2], thus reducing the processing time and/or saving energy consumption on the devices.

Although there have been a number of works that provide various computational offloading techniques in the context of mobile-edge computing [2, 8, 12], there are few works that have looked at *which* edge nodes the computational should be offloaded to. The problem of node selection for computational offloading is interesting for a couple reasons: First, the type of edge resources are typically highly heterogeneous in terms of both their computational power and latency to end-users. The resources vary from edge servers provided by Internet service providers (ISPs) to network access points and they provide varying network latency and bandwidth. Second, the dynamic nature of workload makes the node selection problem more challenging since resources in Edge Cloud are typically much more limited compared to the centralized Cloud's resources. Although the mobility aspect of end-users may add additional dynamic to the workload, we argue that the user's mobility is much less time sensitive compared to the changes in workload. For example, trending topic spreads much faster compared to human's mobility.

In this paper, we propose a locality-aware load sharing technique in the context of Mobile-Edge Computing (MEC). Specifically, we study the problem of *which* nodes should be selected for load sharing and *how much* workload should be shared to each of the nodes while considering the heterogeneity aspect of the edge nodes' resources. Our goal is to maintain the low latency requirement of common mobile applications that are continuously producing and consuming data in the case of runtime dynamics that may cause a contention in a node's network resources.

Our system constitutes different layers of Cloud resources ranging from distant resource-rich Cloud servers to edge resources with

[1] In this paper, we refer to any type of resource-limited devices as mobile devices

[2] The closeness is measured in term of network latency rather than actual physical distance.

UCC '17, December 5–8, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-5149-2/17/12…$15.00
https://doi.org/10.1145/3147213.3147228

less computational resources that are closer to the end-users. It provides a location-based edge node discovery mechanism that allows users to find the closest available nodes. The edge nodes in our system are aware of the availability of their *neighboring nodes*. A node is considered as a *neighbor* to another node if they are located close to each other. This neighborhood information is maintained by each node and is used for workload sharing in the case of high workload. To prevent sharing workload with a neighbor that has already had high workload, each node periodically shares its load information to all of its neighbors. This neighbor-awareness is useful to guarantee low overhead in sharing a workload. This locality-aware load sharing mechanism allows load sharing with little overhead and thus is able to maintain the low latency requirement of mobile applications in the case of workload dynamics.

We evaluate our system and techniques using a real geo-distributed Edge Cloud platform deployed on PlanetLab [7] testbed. Our experiments are based on a sample of real Twitter trace from December 2015 which consists of approximately 4 million tweets/day. We show that our locality-aware load sharing technique is able to better satisfy the application's latency goal even when there is an increase in the workload. We also show that our load sharing technique results in up to 1.5X and 3X lower latency for 95th percentile latency compared to an Edge Cloud system that does not consider load sharing and the centralized Cloud platform respectively.

2 BACKGROUND

Heterogeneous Edge Resources. Nodes in an Edge Cloud are located at the edge of the Internet and hence are closer to the end-users. They generally provide less computaional power but lower latency to the end-users compared to the Data Center's nodes. Edge nodes are also typically more heterogeneous in terms of their computational hardwares as well as network connectivity: cloudlets, servers provided by ISPs, home servers, to access points that use Wi-Fi, bluetooth, etc. To handle this heterogeneity, some works have proposes a common interface and mechanism for mobile devices by providing a virtualization layer that encapsulates a mobile-application execution inside virtualized machines (VMs) or containers [19, 26, 31]. This mechanism allows variety of applications from possibly different devices to run concurrently in isolation.

Computational Offloading in Mobile-Edge Computing. In the context of Mobile-Edge Computing (MEC), the main purpose of the Cloud is to support mobile devices by allowing them to offload their data processing to the Cloud's nodes for better performance and/or saving energy consumption [8, 24, 25, 27]. For example, a compute-intensive object/image analysis on a mobile device can be processed on one of the Cloud's nodes that is equipped with GPUs, leaving only the final image rendering to be processed on the mobile device itself. Throughout the paper, we assume that the decision on which parts of application programs that should be offloaded to the Cloud have already been made externally.

Low Latency Requirement of Mobile Applications. Many emerging mobile applications are latency sensitive since they are interactive applications [1, 6, 9]. For example, interactive collaborative mobile games require continuous image processing and augmented

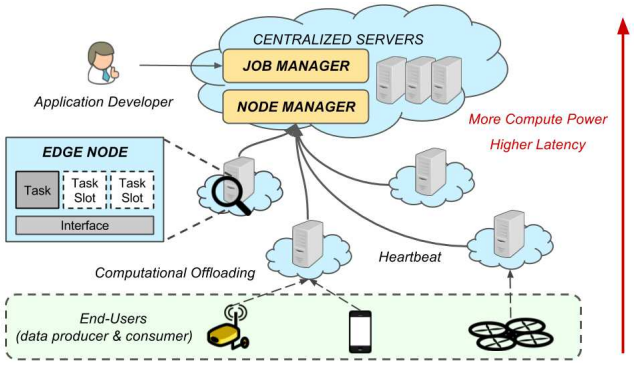


Figure 1: Edge Cloud System Model

reality applications on wearable devices require very low latency for better user experiences [15, 25, 42]. Many applications also require data aggregation from mobile devices. For example, a real time event detection in a social network application that detects earthquake needs to aggregate a vast number of microblogs that are originated from a specific area and detect the trend. These applications require low latency while continuously producing and consuming data. In a MEC environment, workload may change frequently due to the nature of hotspots. Nevertheless, the Cloud should satisfy each application's desired goal regardless of the runtime dynamics.

Location Property of Mobile Applications. Geographic or location information has become an important factor for many mobile/sensor applications. There are many applications that use and rely on the users' locations in providing their services. For example, numerous recent augmented reality games rely on their users' locations. Another example includes map-based applications and environmental monitoring that use sensor-equipped mobile devices [22, 25, 33]. These applications share a common property of continuous data production/consumption and they need the support from Clouds' resources for processing their data efficiently. Furthermore, some of them rely on information aggregation from other nearby users' activities such as real-time traffic detection that needs to aggregate information from nearby drivers to detect whether a certain road is congested and multi-player mobile games that need to detect the availability of other players for matchmaking.

3 EDGE CLOUD MODEL & IMPLEMENTATION

In this section, we discuss the Edge Cloud system that we consider throughout this paper. Figure 1 shows the system model. It consists of 1) a set of components that are hosted in a centralized reliable server: *Node Manager* and *Job Manager*; and 2) a set of edge nodes that are distributed across geographic locations [18].

3.1 System Components

• **End-Users and Applications.** The end-users are any type of resource-limited devices that rely on an Edge Cloud's supports by offloading their computational to the Cloud's nodes. In this paper, we mainly focus on a class of applications that requires low latency while continuously producing and consuming data. We consider a

task as an instance of an application that is running on a node and a task may consume one or more inputs from possibly different input providers/end-users.

• **Edge Nodes.** The edge nodes are computational resources that are geographically distributed. They provide a common interface that allows end-users to offload their data to be processed on the nodes.

• **Node Manager.** The Node Manager is responsible for monitoring the availability of all nodes in the system. It uses a *heartbeat* mechanism to detect node failures. Nodes that do not respond to the heartbeat message in a timely manner will be considered unavailable. Each node information that is monitored by the Node Manager is periodically shared to the *Job Manager*.

• **Job Manager.** The Job Manager provides an interface for application developers [3] to submit their applications to the system. It is responsible for scheduling, deploying, and managing all tasks that are running on the nodes.

3.2 Implementation

Computational Offloading. All tasks that are running on our edge nodes are encapsulated inside a virtualized layer and thus are independent of the applications. This computational offloading mechanism is done using the following steps: 1) The end-user sends its data to one or more edge nodes through the nodes' interface layer (this is done in the background). This interface is implemented using a generic socket layer and so is independent of the type of the applications. We assume that the application program itself that is running on the node has already been deployed to the nodes. In a real deployment, this decision depends on each application's area of interests such as the environment or location. For example, in a city traffic monitoring, the program should only be pushed to the nodes that are located in that city. We consider this deployment issue to be orthogonal to our work. 2) The edge node processes the data as an input to a task that is running on a *task slot*, which is an abstraction of computational resources on which a task can be deployed. So, the number of task slots of a particular node corresponds to the number of tasks that can be run concurrently on the node. We use the number of CPUs as a metric to determine the number of slots of a particular node. 3) Once the processing is complete, the node sends the result back to the users. Inactive tasks that are deployed on the node can be terminated using a *least-recently-used* policy.

Node Monitoring. To monitor the availability of each node, the Node Manager periodically sends a heartbeat message to every node. Each node includes some additional information to its heartbeat response: 1) The node's location, which can also be estimated using a geo-location service, 2) The number of available task slots, and 3) The current load information of every task that is currently running on the node. The location information of a node is used to determine *which* node an end-user should be associated with to minimize the connectivity overhead between them (will be discussed later Section 5). The location information is also used as a metric to determine the neighborhood information that is used for locality-aware load sharing (will be discussed in Section 6). The number of available task slots and the load factor are used to determine the resource availability of the nodes and its current load respectively. The Node Manager periodically gathers all of this information and forward it to the Job Manager which will use this information for task management and scheduling.

[3] An application developer is not part of the runtime entities. Its only task is to deploy an application to the system (e.g., traffic monitor organization, social network analyzer, etc.).

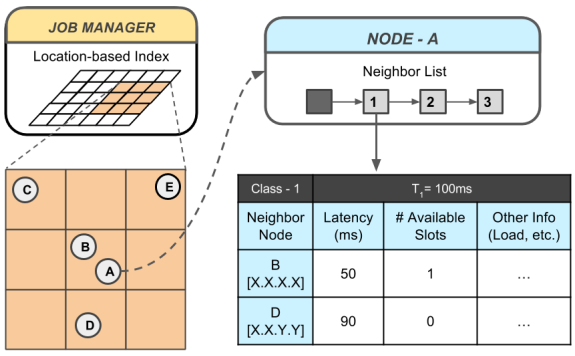


Figure 2: Neighbor Index Structure

Locating an Edge Node. When an end-user tries to discover an edge node for offloading her computation, she will need to query the availability of the nodes in the system. When the Job Manager receives this query, it will return a set of possible edge nodes where the user can connect to. Once the user receives this response, she can connect to any of the nodes without going to the Job Manager again and any subsequent requests can be directly sent to the nodes. The decision on *which* nodes the Job Manager should return will be discussed in Section 5.

4 LOCALITY-AWARENESS

Having discussed the system model, we will now define the locality-awareness property that we have implemented in our system. This locality-awareness is used to intelligently associate end-users to one of the edge nodes in the system (Section 5) and to handle runtime dynamics through load sharing (Section 6).

4.1 Node Neighborhood

The Job Manager periodically gets an update about the nodes' availability along with their resource information from the Node Manager. It stores this information in a global location-based index structure (shown in Figure 2). The main purpose of this index structure is to quickly find and map a node to one of the index's cells based on the node's location. This index is also used to cluster nodes based on their locations to determine the neighborhood of nodes. Nodes that are located close to each other will be mapped to the same index cell. Nodes that lie within the same index cell or in adjacent cells to a node will be considered as its neighbors, since intuitively they are close to each other.

Although an actual geographic distance between two end-points may not guarantee a low latency between them in a wide-area setting, large values of geographic distance between two end-points have a strong tendency of having circuitous routing as studied by

previous works [28, 32]. Furthermore, a node's IP-address can be used to estimate its location in the network and the linearized distances between two end-points have been shown to have a strong correlation to end-to-end delay.

The size of the index cell also determines the size of the neighborhood and it should be configurable depending on how close a node should be considered a neighbor. If the size of the cell was set too big, e.g., $1000x1000km^2$, this would not give a meaningful filtering result since nodes that are very far from each other would still be considered as neighbors. On the other extreme, limiting the size of each cell too small, e.g., $100x100m^2$ would filter too many nodes that in reality have low inter-node latency. We evaluate the effect of the cell size later in the experimental section. The location-based index can also be implemented using a different index structure that provides a more sophisticated partitioning scheme that partition the location in a gradual manner such as an R-tree.

4.2 Neighbor-Aware Edge Nodes

The neighboring information that is maintained by the Job Manager is shared to all the nodes. Thus, each node in the system is aware of the availability of its neighbors. Whenever a node joins (leaves) the system, the Job Manager will add (remove) the node information from its index structure and propagate this update to all nodes within the neighborhood. When an edge node initially joins the system, it will use the neighboring information and measure the estimated network latency to each of its neighbors. Once the node gets the estimated latency to its neighbors, the node will use this information to further filter and classify its neighbors by constructing a hierarchical data structure. This hierarchical data structure defines the priority of the neighbors (shown in Figure 2). Each level i in the hierarchy consists of all neighbor nodes whose latency to the node, L, is within a latency threshold: $(i-1)T < L \leq iT$ where $i \geq 1$ and $i = 1$ is the top most level. This latency threshold T is set as a system parameter depending on the sensitivity required to the latency. For example, a neighbor node will be classified as a top-class neighbor if the latency between them is less than $T = 100ms$. On the other hand a neighbor node that has significantly high latency $L > 500ms$ can be ignored even if it is located in an adjacent cell.

This neighboring hierarchy is constructed and maintained by each node, meaning that any particular node may be classified to different levels by different nodes. For example, a node A may be classified as a top-level neighbor by node B since they are close to each other, but is considered as a third-level neighbor by node C since they are far away from each other. Nodes within the same class can be considered to have similar latency. The main reason of classifying nodes into different classes rather than simply sorting the nodes based on their latency is that it is less susceptible to latency variance. Thus, every node in the system is aware of the availability of its neighboring nodes and the latency to its neighbors.

To maintain an accurate and up-to-date latency between nodes, a node will have to frequently monitor the latency to all of its neighbors and update any of the latency information that has a large change. Although this fine grained monitoring will result in a high accuracy, in a large-scale environment, this may incur a high monitoring overhead. Furthermore, if the nodes in the system are relatively static (regardless of the mobility of the end users), the network latency between nodes should be relatively stable. We rely on an estimation to determine the latency between nodes. Initially, each node will try to get an estimate of the network latency to each of its neighbors by sending multiple sizes of data to get an estimate latency for each data size. We assume that each individual data that is sent between nodes can be mapped to the latency prediction mapping. We believe this assumption is reasonable since each individual offloading request/data for most mobile applications is typically small (e.g., image update, sensor reading, etc.) [3, 13, 15] and thus its latency can be predicted within a small error margin. Although the size of each individual update is small, the challenge comes from the rate which may constrain the network availability of each node.

The location-based index that is maintained by the Job Manager determines the initial range of neighboring nodes that need to be monitored by each node. This may have a drawback of having false positive and false negative nodes during the pruning step which correspond to ignoring nodes that are located outside the neighboring bounding box with low inter-node latency and including nodes within the bounding box that have high inter-node latency respectively. However, this early pruning gives the benefit of removing the majority of high latency nodes which is highly beneficial in a real Mobile-Edge Computing environment consisting of a large scale of edge nodes.

Our Edge Cloud design pushes most of the decision making to the edges rather than relying on the global decision made by the Job Manager. This is made possible since each edge node itself has a complete knowledge of its neighbors. For example, in the case of workload burst, an edge node may determine which of its neighbors can be used to share its workload based on the latency information between them which can be used as a projection of latency that determines the computational hand-off overhead. Although the localized decision may be sub-optimal compared to a global decision, it gives us the benefit of allowing decisions to be made quicker and preventing potential bottlenecks in the centralized server.

5 EDGE NODE DISCOVERY

In this section, we discuss the node discovery mechanism to find any edge nodes that are located close to end-users. Most Mobile-Edge Computing applications rely on discovering nodes within the end-user's network coverage area. This means that an end-user can only discover edge nodes that are within the user's area or within the same network range (connected by local area network or within access points that are only a few hops away). This is a common approach for discovering nearby nodes especially for most sensor/IoT devices that use a broadcast discovery mechanism to find nearby edge nodes. Although this mechanism guarantees that the nodes that are found have little overhead to the users, this mechanism greatly limits the range of possible edge nodes that the users can utilize, leading to several disadvantages. First, when there are no available nodes within a close network range, nodes that are a few hops away with available resources may not be discovered at all, and hence the users are unable to utilize the nodes.

Second, an edge node that is within the discovery range may have a high processing load or high network I/O loads (e.g., due to a hotspot). Thus, offloading to an already-overloaded node may even degrade the overall performance to the users. Hence, a user should be exposed to more node selection options.

Our edge node discovery mechanism relies on the global node availability provided by the Job Manager. It allows end-users to find a wide range of edge nodes even if there are no available nodes within the users' network range. Note that our technique does not eliminate the node discovery mechanism that allow end-users to find nodes that are located within a close network range. Instead, it can be used as an additional mechanism if the previous mechanism could not find any nearby edge nodes. The Job Manager uses user's location information to find any nodes that are located close to her by using the location-based index discussed in the previous section. Although the list of nodes that are returned by the Job Manager may include some nodes that do not have very low latency to the user, the list guarantees the closest available nodes in the system.

Once a user has been associated with a specific node, the node will share its neighboring node information to the user. This mechanism is used to handle potential failures of the associated node. When the associated node fails, its users can quickly find other nearby nodes from the node's neighboring list. In this case, the latency to a neighboring node is used as a projection to the latency between the user to the neighbor node. This results in a short distance from the neighbor node to the user since the distance between the failed node and its neighbors is small. If there was no available node in the neighboring list, the user would have to query the Job Manager again following the same steps as for initial node discovery mechanism.

6 LOCALITY-AWARE LOAD SHARING

As the number of resource-limited devices that are connected to the Internet increases rapidly along with the number of compute-intensive applications running on these devices, more and more applications would rely on Cloud's support for processing their data. This may cause a dynamic and skewed workload distribution which results in a particular set of edge nodes being heavily used and potentially becoming a bottleneck while leaving other nodes idle. If the system is unable to detect such a behavior, the use of the Edge Cloud for computational offloading may worsen the overall performance compared to leaving the computation on the devices themselves even with some of them have limited computing power. Although edge nodes have relatively more powerful computational capability compared to most mobile/sensor devices, these nodes typically have much more limited resources (e.g., CPU power, memory capacity, etc.) and network bandwidth compared to Data Center's nodes. So, these nodes are more prone to becoming overloaded compared to nodes in a centralized Cloud. Without the capability of sharing or balancing their workload, the use of edge nodes can potentially hurt the application desired goals. Thus, there is a need for Edge Cloud systems to handle such workload dynamics to maintain the low end-to-end latency requirement of mobile applications.

A common technique to handle workload dynamics in distributed systems is to dynamically share some of the workload to other nodes that are relatively idle. The load sharing in the context of Mobile-Edge Computing (MEC) can be performed in two steps: 1) By handing-off some of the tasks to other nodes that are lightly loaded, and 2) By redirecting some of the end-users to other nodes if the high load is caused by a large volume of end-users connecting to a hotspot node [11, 14, 34, 37]. The latter case may require a task to be handed-off first from the overloaded node to the new nodes before redirecting its users to the new nodes (shown in Figure **??**). The mechanism for seamlessly migrating a task from one node to another has been intensively studied in the context of handling user's mobility in MEC environment [12, 14, 34]. Similar mechanism can be used to handle task migration. However, the load sharing problem is still left with the question on *when* to share the workload and *which* nodes the workload should be shared with.

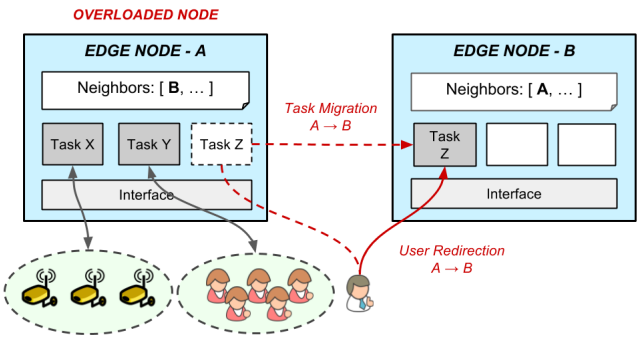


Figure 3: Load Sharing Mechanism

One possible metric to determine whether a node is overloaded is by monitoring if there is a slowdown of any of its tasks. This metric, however, only detects whether a node is overloaded and cannot be used to determine whether the node should share its workload since the latter requires a knowledge of the load information of other nodes as well. The problem of load sharing in MEC becomes more challenging since the edge nodes are highly distributed and may be connected by heterogeneous WAN with limited network bandwidth, unlike the intra-Data Center network. So, the system should carefully consider *which* nodes (if any) the workload should be shared with. A poor decision in selecting nodes for load sharing may worsen the overall performance to the end-users. For example, it may not be desirable to offload a task to another node that has already been heavily loaded or had its network congested, or to a node that incur too much network latency because it is far away from the user. This may result in an increase in the overall end-to-end latency caused by resource contention as in the former case, and high task migration or latency overhead in the latter case.

We propose a locality-aware load sharing technique with the goal of achieving applications latency goals in the case of dynamic workload. We achieve this goal by selectively choosing nodes where an overloaded node should share its workload with in order to prevent the possible issues that are discussed above. Our technique considers the nodes' resource availability, their current load, as well as the overhead of migrating the task and redirecting the users. All of this information can be obtain from the neighboring information that is maintained by each node.

```
Input  : neighbor-list N = ⟨N_0, ...N_z⟩, latency-goal L_app
Output : n ∈ N
for index level i : 0 → z do
    if iT < L_app then
        for n ∈ N_i do
            if n not busy and has slot then
                return n;
            end
        end
    else
        cannot find node, return;
    end
end
```

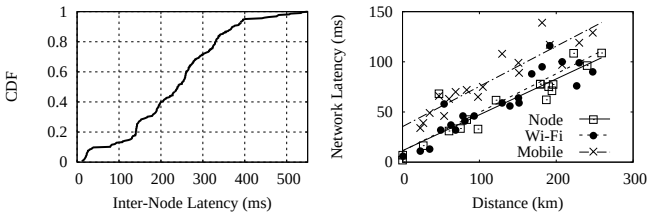
Algorithm 1: Node Selection for Load Sharing

To determine whether a node needs to share its workload, we rely on a per-application's desired-latency goalL_{app} . The goal of the system is to keep the end-to-end latency below the application's goal. To achieve this, each node needs to monitor the end-to-end latency information of each of its tasks which includes the time needed to process the offloaded data as well as the time taken to send the data between the user and the node. Each node periodically monitors the average latency $\bar{L}$ of a given task within the last time window t (e.g., $t = 30seconds$). If $\bar{L} < L_{app}$, the node does not need to share its workload. Otherwise, the node will start to locate one of its neighbors to share its workload with. To determine *which* node to share the workload with, it traverses its neighboring hierarchy starting from the top level as shown in Algorithm 1 and finds any nodes that have an available task slot and are not overloaded. Neighbor nodes that are located within the same level will be considered to have similar latency to the node and thus the node may rely on other factors such as their computation power, current loads, etc.

Whenever a node decides to share its workload, we limit the number of additional neighbor node to share by 1 every t. If $\bar{L} > L_{app}$ even after the node has shared its workload, the node will locate an additional neighboring node to further reduce its workload. Thus, the amount of sharing increases gradually every t. The value of t itself provides a trade-off between the responsiveness to workload change and the waste of computation resources respectively. We implement such policy to prevent a node from hoarding its neighbors' resources in a short time period, resulting in an exhaustion of resources for other tasks and unpredictability of loads. Gradually acquiring resources will also prevent interference of already running tasks on the neighboring node due to a sudden increase in the number of additional shared tasks. This approach also performs well in practice since workload changes usually increase/decrease in a gradual manner within a few seconds.

Once a neighboring node is selected, the node needs to determine the amount of workload that should be shared with its neighbor based on the workload characteristic over the last t window. To determine this, the node will find the ratio of its workload to share to its neighbors by solving the following equation:

$$Min(Max((r_{self} * \bar{L}), ..., (r_i * \bar{L}_i))), where \sum r = 1 \qquad (1)$$



(a) Inter-node latency.

(b) Correlation between mobile nodes and mobile phone networks.

Figure 4: Edge Node Deployment

where r_{self} and r_i are the ratios of the workload to be run on the original node and neighbor node i respectively. L and L_i are the estimated latency of accessing the original node and the neighboring node which is obtained from the neighboring information. We estimate the latency between end-user to the neighboring node using a projection of the inter-node latency. The Max property in the function is used since the completion time is typically determined by the completion time of processing the last record. The optimization problem is then to minimize the overall time.

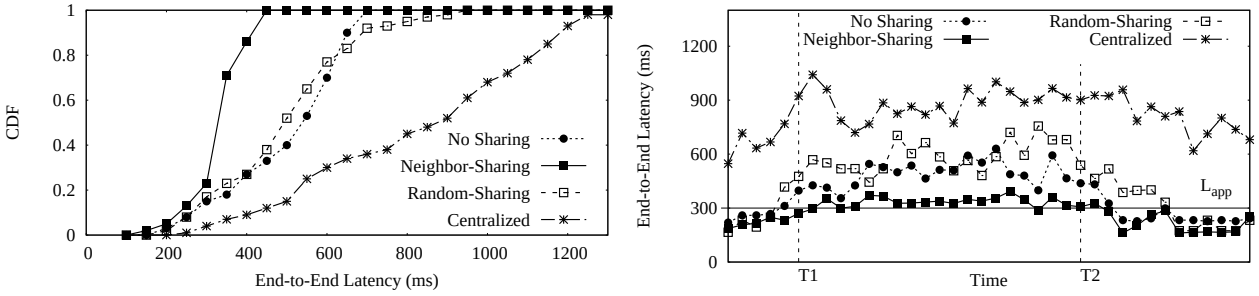
These load sharing ratios for each node will be used for workload in the next t interval. If $\bar{L}$ falls below λL_{app}, the node will gradually reduce the number of neighbors it used to share the workload with. The reason of adding a λ factor is to prevent a fluctuation in latency. For example, if we set $\lambda = 1$ and $\bar{L} = 290ms < L_{app} = 300ms$ after the node shares 40% of its workload to another node and decides that it will stop sharing the workload since $\bar{L} < L_{app}$, this may increases $\bar{L}$ again in the next t window. In this case, the latency of the application will fluctuate for every t interval. We set $\lambda = 0.8$ in our deployment since it works well with the dynamism of the workload we consider.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup and Methodology

Edge Cloud System Setup. We evaluated our Edge Cloud system using 20 physical PlanetLab [7] nodes (with up to 3 virtual nodes in each location) that are geographically distributed across the U.S. (ranging from U.S. West to U.S. East). We did not consider nodes that are located in the same physical machine as neighbors to each other since doing so will completely eliminate the latency to the neighboring nodes which is unrealistic in a real deployment. Both the Job Manager and the Node Manager were deployed on a reliable centralized server located in Minnesota. This centralized server did not participate in any task processing and it was not a bottleneck in our deployment since its only purpose was for node monitoring and task scheduling. Figure 4(a) shows the inter-node latency in our deployment. We can see that sending data to a distant node may incur up to 40X higher latency compared to sending the data to a nearby node. This shows the importance of node selection policy for workload sharing.

We simulated the end-users using a daemon program running on 10 different PlanetLab nodes and 10 other PlanetLab nodes as the workload generator where some of them are actually the same



(a) End-to-end latency distribution during a workload dynamic.

(b) Meeting latency goal during a workload dynamic.

Figure 5: Benefit of Locality-Aware Load Sharing

nodes. These nodes are not part of the edge nodes that are contributing as compute nodes. We call these emulated users as *mobile nodes*. We validated our methodology of simulating the mobile users using mobile nodes (Figure 4(b)). We compared the latency of accessing the edge nodes using mobile nodes with the latency of accessing edge nodes using mobile phones with Wi-Fi and mobile network (*Node*, *Wi-Fi*, and *Mobile* respectively). We see that they have a strong correlation.

The location-based index of the Job Manager is implemented using a grid index with 144 cells (unless explicitly specified) which partitions the geographic U.S. map into equal size of cells based on the coordinates. Nodes that lie within the same cell or one of the adjacent cells are considered as neighbors. For the neighboring list that is maintained by each node, we set the threshold of each level i to $T_i = i * 100ms$ with a maximum of 3 levels. Thus, neighboring nodes that are $> 300ms$ away will not be considered for workload sharing. We also set the evaluation time window t to 10 seconds. So, each node will evaluate whether to share its workload based on the workload characteristic in the past $t = 10$ seconds.

Workload Trace and Applications. We used a real three-day Twitter sample trace as our workload that contains approximately 4 million tweets per day. This trace was obtained from December 2015 and it includes dynamic workload over time from real mobile users. We sped up the tweet rate by 24X since the trace only shows a fraction of the real Twitter workload. This is done to better see the effect of overloaded nodes in a real deployment. The tweet data sources are partitioned and deployed to the nodes based on each tweet's location coordinates. So, the skewness of workload and the location of hotspots are naturally included in the workload.

We use a location-based top-k popular topic as our application that is executed on the edge nodes. This application aggregates all the tweets within a specific region and returns the top k words. The data processing on the edge nodes itself incurs only a small fraction of the end-to-end latency which is the time taken from the time the end-user sends the data to the edge node to the time the user obtains the result. We observe that the major fraction of the end-to-end latency is incurred by the wide-area network latency for offloading the data. This is reasonable for many Mobile-Cloud applications since most of the task processing can typically be processed with low processing time.

System Comparison. We compare our neighbor-aware load sharing technique (*Neighbor Sharing*) with 1) The use of an Edge Cloud that does not consider load sharing (*No Sharing*), 2) The use of an Edge Cloud that use load sharing with random node selection to determine which nodes a workload should be shared with (*Random Sharing*), and 3) The use of a centralized Cloud (*Centralized*) as our baselines. All of the Edge Cloud usages were deployed using the same node deployment while in the centralized case, we deployed more nodes/resources in a single location to simulate a higher computational power available in a centralized Cloud.

In the case of Random-sharing, an overloaded node would randomly choose any available nodes ignoring the latency overhead between them that is used as a projection to the end-user and the current workload of the other nodes. On the other hand, the overloaded nodes in the Neighbor-sharing would only consider sharing their workload to nodes that are located close to the overloaded nodes and the nodes are only selected if they are not overloaded. The nodes in the centralized Cloud itself are deployed using PlanetLab nodes. Furthermore, for the centralized result, we deployed the Centralized server in different number of locations for each iteration: U.S. West, U.S. Mid-West, and U.S. East.

7.2 Benefit of Locality-aware Load Sharing

In the following set of experiments we evaluate our locality-aware load sharing technique in the case of dynamic workload. We set the application's latency goal L_{app} to 300 ms and gradually scale up/down the workload by up to 14X. Figure 5 shows the performance impact caused by the workload dynamic. We make a few observations from the result. First, we can see that the use of Edge Cloud significantly outperforms the use of centralized Cloud in all of the cases as shown in Figure 5(a) which shows the latency CDF of all approaches. We also observe that the increase in workload did not have significant slow down in the computation time performed on the node itself. Rather, the latency incurred by the data offloading to an edge node is the dominant factor due to the contention of the limited wide-area network bandwidth. Thus, although the centralized Cloud deployment has more processing power (lower computation time), it still suffers from the high overhead between the mobile nodes to the centralized nodes.

Secondly, we observe that the node selection decision for load sharing is critical. We can see from Figure 5(a) that the Random-sharing performs worse compared to the Neighbor-sharing. The

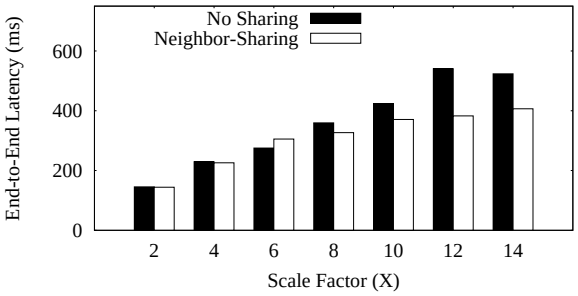


Figure 6: Increase in Latency Over Scaled Workload

reason is that in the Random-sharing approach, an overloaded node might select one of its neighbors that are far away to the users or neighbors that have already had high load. Comparing the No-sharing and the Random-sharing approaches, we can also see that the latter performs better than the former approach below the 75th percentile but has a longer tail later in the distribution. This shows that randomly selecting nodes to share a workload may result in selecting nodes with very high overhead and may suffer at a higher percentile.

Thirdly, our neighbor-aware load sharing is able to maintain a close gap to the application-specified latency goal that was set to 300 ms in this case. Figure 5(b) shows a snapshot of the trace during a dynamic workload where the workload increased from T_1 to T_2 and decreased afterward. Before T_1, all Edge Cloud approaches performed comparably. However, as the workload increased ($T_1 < T < T_2$), some of the edge nodes became overloaded and started to result in a higher end-to-end latency. Approaches that consider load sharing (Random-sharing and Neighbor-sharing) started to look for other nodes for sharing their workload.

We can see that the Neighbor-sharing is able to maintain a close gap to the application latency goal (within 20% increase in latency). If there were no neighboring nodes that could provide low overhead, the overloaded node would handle the tasks by itself. However, if there were any nearby nodes with low network latency overhead, it would start sharing the workload to one of the neighboring nodes. When the workload decreased at T_2 and the end-to-end latency dropped below the threshold, all policies that used load sharing mechanism started to decrease the number of shares and eventually stopped sharing their workload. These results show that the use of Edge Cloud is not sufficient to satisfy application's latency goal especially in the case of dynamic workload.

We also observe the scalability of our locality-aware load sharing approach by scaling up the load with different scale factors. Figure 6 shows the impact to end-to-end latency due to an increase in the workload. We can see that even with the Neighbor-sharing approach, the latency may still increase beyond the end-to-end latency-desired goal. The main reason to this is that some of the edge nodes in our deployment did not have any neighbors where they can share their workload with. Thus, they caused a slowdown to the overall time. However, it increases in a much more graceful way compared to the No-sharing approach. We also see that the number of tasks that satisfied the latency goal is much higher than the No-sharing approach.

7.3 Impact of Neighbor Distance

In this experiment we study the effect of setting the minimum distance between nodes which we call as *neighbor distance* as a parameter to determine whether a node should be considered as a neighbor. At one extreme, the neighbor distance may cover the entire area which make every node consider all other nodes as its neighbors. In this case, nodes that are very far away may still be selected for workload sharing since they are considered as its neighbor. This is effectively similar to the random node selection approach with an additional consideration of not selecting busy nodes. At the other extreme, the index cell size may be limited to have a very small coverage which makes a node only consider another node as its neighbor if they are very close to each other.

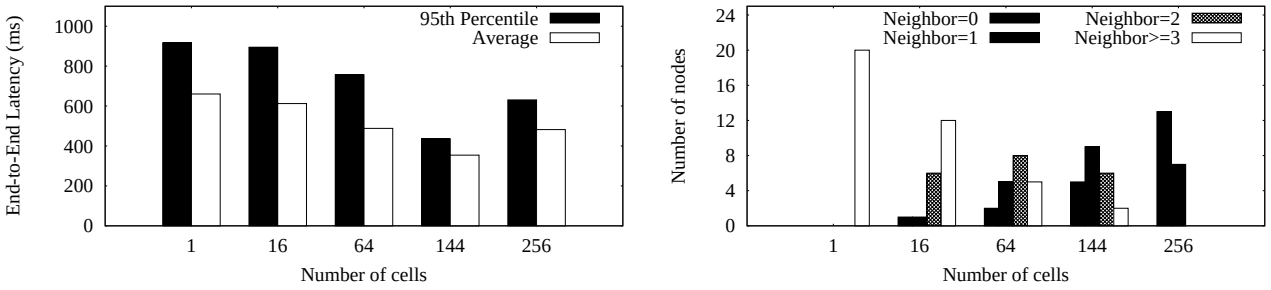
Figure 7 shows the impact of varying the number of grid cells to the end-to-end latency. The larger the number of the grid cells is, the smaller the neighbor distance is and vise versa. We can see that the performance improved as we increased the number of cells but later decreased as the number of cells was set too high (number of cells = 256). The reason behind the improvement in the early increasing number of cells is because the neighboring list only included nodes that were actually close to each other. However, as we limited the cell size too small, more and more nodes were not able to find any neighbors and hence were unable to share their workload. This pattern will converge as the number of neighbor nodes for each node reaches 0.

Figure 7(b) explains this phenomenon. With cell size equals to 1, all of the nodes were considered as neighbors to every node even if some of them are very far away (one in U.S. West and the other one is in U.S. East). As the number of cells increases (the neighbor distance/cell size decreases), less number of nodes were considered as neighbors but these neighbor nodes were actually close to the node itself.

7.4 Handling Node Failure

In this experiment we show the benefit of neighbor-awareness in the case where the node that has been associated with an end-user fails. When the node fails, the end-user can quickly re-associate herself to one of the node's neighbors without requesting for a new node from the centralized server. This mechanism is made possible since this neighbor node information is shared to the user.

In this experiment, we did not add any other variations to the workload. Figure 8 shows the impact of adding failure to nodes in our system. We randomly terminated any of edge nodes that were supporting any end-users starting from time T_1. We can see that by having a knowledge of the availability of the alternative nodes, the end-user can quickly re-associate herself to one of the failed node's neighbors. There is a small increase in the latency due to the timeout mechanism that was used to detect node failure and re-association time to the new node. On the other hand, if there was no information of the availability of the neighboring nodes, the end-user would have to connect to the centralized server. In either case, when the failed node returns at time T_2, the user could be redirected back to the recovered node or kept being supported by the covering node depending on whether the current end-to-end latency met the application-desired latency goal. If the latency has already met the latency goal, there is no reason for re-associating



(a) Effect of different cell sizes to the overall end-to-end latency.

(b) Neighboring node distribution over different cell sizes

Figure 7: Effect of Different Cell Sizes to the Performance

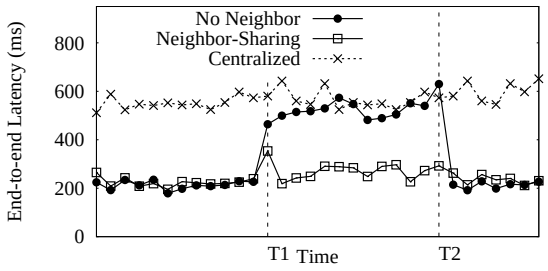


Figure 8: Maintaining Low Latency During Node Failure

back to the recovered node. We also plot the latency of the centralized Cloud to show that the use of Edge Cloud still outperform the centralized Cloud even in the case of node failure. This shows that using an Edge Cloud platform is more suitable for the Mobile-Cloud Computing applications that we consider compared to the use of centralized Cloud even if the Edge Cloud is more susceptible to failures.

8 RELATED WORK

There are a number of projects that have looked at the opportunity of utilizing Cloud resources to support mobile/sensor devices. Most of them focus on the oportunity of reducing the processing time or saving energy consumption or both. [2, 8, 16, 27, 31, 40]. Others [13, 17, 20, 41] have also looked at mobile data management and utilizing Cloud's storage services for managing mobile users' data by using Cloud's resources as caches or consistency management. Although many of these works are relevant, most of these systems do not consider the runtime dynamics which are common in Mobile-Edge Computing environment. Our work is different from most of the existing works in that we focus on handling runtime dynamics where some of the edge nodes may become a bottleneck due to changes in workload. Furthermore, we study the problem of selecting *which* nodes should be selected for load sharing and *how much* of the workload should be shared if a node decides to share its workload. Existing mechanism for computational offloading and policies that determine which parts of the computation should be offloaded can be applied to our work.

Load balancing is a common technique that has been extensively studied in the area of distributed systems, Cloud Computing, network routing, and peer-to-peer systems [5, 23, 29, 38]. Most of the existing techniques rely on the tasks scheduling decision that determines where to schedule new tasks on the system to balance the workload. Our work, however, is different from them in that we focus on the Mobile-Edge Computing environment where the nodes are interconnected by wide-area network. In our context, the load sharing is done using a user redirection and task migration rather than tasks scheduling. Furthermore, our main goal is not to get a balance workload in the system. Instead, we try to achieve each application's desired latency goals during workload dynamics.

9 CONCLUSION AND FUTURE WORK

In this paper, we study the problem of load sharing to handle runtime dynamics in a Mobile-Edge Computing (MEC) environment. Our motivation is based on the dynamic property of the workload in MEC along with the low latency requirement for many of today's mobile/IoT applications. The Edge Cloud platform that has been proposed to provide computational offloading support for mobile applications faces additional challenges in handling workload dynamics since the nodes in Edge Clouds are typically connected by WAN with high network latency and limited bandwidth. We propose a locality-aware load sharing technique that allows edge nodes to share their workload to other nodes to meet the low latency requirement of the mobile applications in the case of workload increases. Our load sharing technique allows nodes to 1) Intelligently determines whether to share their workload to other nodes, 2) Selectively chooses which nodes the workload should be shared with, and 3) Determines how much of the workload should be shared. Our experimental results based on a real Twitter's trace show that our locality-aware load sharing technique is able to keep the overall latency of mobile applications close to the applications' desired goals as well as better utilize resources even in the case of dynamic workload.

In the future, we would like to consider the overhead of task migration between nodes in addition to the data transfer overhead and incorporate the cost to the load sharing decision. Furthermore, we would also like to consider different classes of mobile applications whose execution time itself may be the dominant part of the computational offloading overhead. In this case, the system should consider this variety of applications and may make the load sharing decision differently. Lastly, we would also like to integrate the

energy consumption consideration on the mobile devices and allows the device itself to intelligently determine whether to use a Cloud's resources during a high workload condition.

10 ACKNOWLEDGMENT

The authors would like to acknowledge grant NSF CSR-1162405 and CNS-1619254 that supported this research.

REFERENCES

[1] Suman Banerjee and Dapeng Oliver Wu. 2013. Final report from the NSF Workshop on Future Directions in Wireless Networking. (2013).

[2] Marco V Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. 2013. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *INFOCOM, 2013 Proceedings IEEE*. IEEE, 1285–1293.

[3] David Barrett. 2013. One surveillance camera for every 11 people in Britain, says CCTV survey. *The Telegraph* 10 (2013).

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.

[5] Valeria Cardellini, Michele Colajanni, and Philip S Yu. 1999. Dynamic load balancing on web-server systems. *IEEE Internet computing* 3, 3 (1999), 28–39.

[6] Kyungmin Lee David Chu, Eduardo Cuervo, Johannes Kopf, Sergey Grizan, Alec Wolman, and Jason Flinn. [n. d.]. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Cloud Gaming. ([n. d.]).

[7] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. 2003. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review* 33, 3 (2003), 3–12.

[8] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 49–62.

[9] Stephen R Ellis, Katerina Mania, Bernard D Adelstein, and Michael I Hill. 2004. Generalizeability of latency detection in a variety of virtual environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 48. SAGE Publications Sage CA: Los Angeles, CA, 2632–2636.

[10] Dave Evans. 2011. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper* 1 (2011), 1–11.

[11] Chaima Ghribi, Makhlouf Hadji, and Djamal Zeghlache. 2013. Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 671–678.

[12] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently.. In *OSDI*, Vol. 12. 93–106.

[13] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. 2014. Bolt: Data Management for Connected Homes.. In *NSDI*. 243–256.

[14] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2015. *Adaptive vm handoff across cloudlets*. Technical Report. Technical Report CMU-CS-15-113, CMU School of Computer Science.

[15] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 68–81.

[16] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. 2015. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 9–16.

[17] Wassim Itani, Ayman Kayssi, and Ali Chehab. 2010. Energy-efficient incremental integrity for securing storage in mobile cloud computing. In *Energy Aware Computing (ICEAC), 2010 International Conference on*. IEEE, 1–2.

[18] Albert Jonathan, Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2017. Nebula: Distributed Edge Cloud for Data Intensive Computing. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[19] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2011. Cloud4Home–Enhancing Data Services with@ Home Clouds. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. IEEE, 539–548.

[20] Johannes Kolb, William Myott, Thao Nguyen, Aniruddha Chandra, and Jon Weissman. 2014. Exploiting User Interest in Data-Driven Cloud-Based Mobile Optimization. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*. IEEE, 228–235.

[21] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Alec Wolman, Yury Degtyarev, Sergey Grizan, and Jason Flinn. 2015. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. *GetMobile: Mobile Computing and Communications* 19, 3 (2015), 14–17.

[22] Min-Joong Lee and Chin-Wan Chung. 2011. A user similarity calculation based on the location for social network services. In *Database Systems for Advanced Applications*. Springer, 38–52.

[23] Lei Lei, Zhangdui Zhong, Kan Zheng, Jiadi Chen, and Hanlin Meng. 2013. Challenges on wireless heterogeneous networks for mobile cloud computing. *IEEE Wireless Communications* 20, 3 (2013), 34–44.

[24] Dawei Li, Theodoros Salonidis, Nirmit V Desai, and Mooi Choo Chuah. 2016. DeepCham: Collaborative Edge-Mediated Adaptive Deep Learning for Mobile Object Recognition. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 64–76.

[25] Christian Licoppe and Yoriko Inada. 2006. Emergent uses of a multiplayer location-aware mobile game: The interactional consequences of mediated encounters. *Mobilities* 1, 1 (2006), 39–61.

[26] Peng Liu, Dale Willis, and Suman Banerjee. 2016. ParaDrop: Enabling Lightweight Multi-tenancy at the NetworkâĂŹs Extreme Edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 1–13.

[27] Emiliano Miluzzo, Ramón Cáceres, and Yih-Farn Chen. 2012. Vision: mClouds-computing on clouds of mobile devices. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 9–14.

[28] Venkata N Padmanabhan and Lakshminarayanan Subramanian. 2001. An investigation of geographic mapping techniques for Internet hosts. In *ACM SIGCOMM Computer Communication Review*, Vol. 31. ACM, 173–185.

[29] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. 2003. Load balancing in structured P2P systems. *Peer-to-Peer Systems II* (2003), 68–79.

[30] Mahadev Satyanarayanan. 1996. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1–7.

[31] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE* 8, 4 (2009), 14–23.

[32] Lakshminarayanan Subramanian, Venkata N Padmanabhan, and Randy H Katz. 2002. Geographic Properties of Internet Routing.. In *USENIX Annual Technical Conference, General Track*. 243–259.

[33] Pratap Tokekar, Deepak Bhadauria, Andrew Studenski, and Volkan Isler. 2010. A robotic system for monitoring carp in Minnesota lakes. *Journal of Field Robotics* 27, 6 (2010), 779–789.

[34] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees De Laat, Joe Mambretti, Inder Monga, Bas Van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. 2006. Seamless live migration of virtual machines over the MAN/WAN. *Future Generation Computer Systems* 22, 8 (2006), 901–907.

[35] Luis M Vaquero and Luis Rodero-Merino. 2014. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review* 44, 5 (2014), 27–32.

[36] George Vellidis, Michael Tucker, Calvin Perry, Craig Kvien, and C Bednarz. 2008. A real-time wireless smart sensor array for scheduling irrigation. *Computers and electronics in agriculture* 61, 1 (2008), 44–50.

[37] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung. 2015. Dynamic service migration in mobile edge-clouds. In *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 1–9.

[38] Xianglin Wei, Jianhua Fan, Ziyi Lu, and Ke Ding. 2013. Application scheduling in mobile cloud computing with load balancing. *Journal of Applied Mathematics* 2013 (2013).

[39] W Wen. 2008. A dynamic and automatic traffic light control expert system for solving the road congestion problem. *Expert Systems with Applications* 34, 4 (2008), 2370–2381.

[40] Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. 2015. The cloud is not enough: saving iot from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[41] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M Levy. 2016. Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications.. In *OSDI*. 723–738.

[42] Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 426–438.

[43] Jiang Zhu, Douglas S Chan, Mythili Suryanarayana Prabhu, Prem Natarajan, Hao Hu, and Flavio Bonomi. 2013. Improving web sites performance using edge servers in fog computing architecture. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*. IEEE, 320–323.