Hybrid Computation Offloading: Mixing Peers and Superior Nodes

CS 6250 Milestone 5: Final Report

Ahmed Saeed, Deep Ghosh, Tarun Mangla, Kapil Agarwal, and Ahmed GHARBI

1 Introduction

Mobile cloud computing is becoming more and more the standard computational model for mobile devices as the expectations of mobile users increase. Using the cloud to help mobile devices perform more complex tasks fast (e.g. natural language processing, finding commuting routes, etc). Relying completely on clouds assumes cheap, low-delay, and persistent communication lines between the mobile device and cloud. However, this is not the case as in many scenarios only intermittent cheap connection is available (e.g. walking around GT campus only a few places have good WiFi coverage). Recently, mobile device clouds or cyber foraging became of interest to the research community as means to overcome this drawback and generally use nearby computational resources (e.g. peer mobile devices[2], nearby desktops or cloudlets[1], etc). The new proposed systems focus on using available nearby computational resources as an alternative for the cloud.

Our goal is to develop the implementation of an offloading system that bridges the gap between scenarios where only peers are available and scenarios where a full cloud is available. In particular, we are interested in a scenario where the mobile device would know: 1) its own location, 2) commuting route, 3) the location of WiFi access points along that route, and 4) the location and processing power of mobile devices it will meet on that route. Thus, the problem of interest is choosing a combination of resources (i.e. one or more computational resource) that would help it finish a certain task optimally.

1.1 Scope of the project

Given the scope and time frame of the course, we focus on obtaining the context information and scheduling the tasks accordingly. We developed several components of this system:

- Context awareness for peers capabilities and mobility patterns **(implemented on Android)**.
- Communication between the peers and the initiator assuming all nodes are already on the same network and a functional routing protocol exists**(implemented on laptops)**.
- Task scheduling to optimize the distribution of tasks over the peers based on the context information and tasks characteristics using a greedy heuristic **(implemented as a simulation module)**.
- Task processing which runs on the peers to execute the tasks that the initiator sends that can adapt to generic tasks **(implemented on laptops)**.

1.2 System Use Cases

The proposed system use case is presented in Figure 1. A mobile device, *initiator*, walks out of the coverage of a WiFi to an area that is not covered towards another covered area (e.g. GTWifi and the person is walking between Klaus to Clough Building). Hence, no access to the cloud is available in between the building. However, other students that the initiator encounters have mobile devices that are barely used especially while commuting between buildings.

During that period, those idle devices form a mobile device cloud. One of the devices acts as an access point that the initiator can connect to and broadcast a request for profiles of available

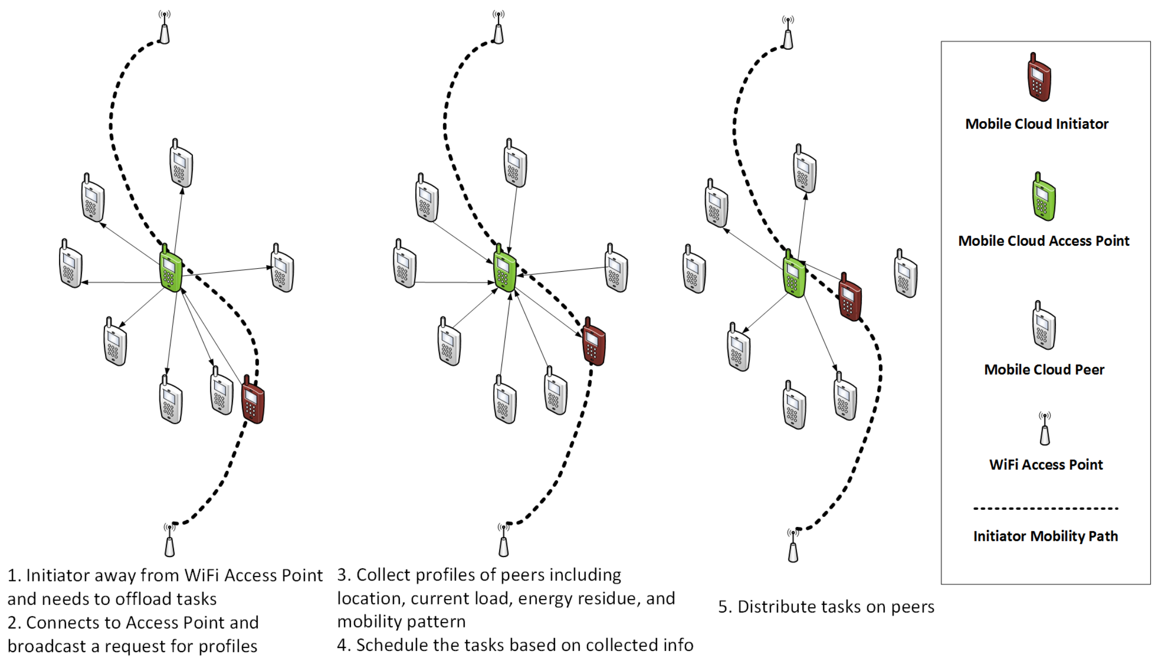
1. Initiator away from WiFi Access Point and needs to offload tasks
2. Connects to Access Point and broadcast a request for profiles
3. Collect profiles of peers including location, current load, energy residue, and mobility pattern
4. Schedule the tasks based on collected info
5. Distribute tasks on peers
Mobile Cloud Initiator
Mobile Cloud Access Point
Mobile Cloud Peer
WiFi Access Point
Initiator Mobility Path

Figure 1: System use cases.

devices. Those profiles can include mobility pattern, energy residue, and computational capabilities. The device then performs the scheduling of tasks based on how long is left to reach the next WiFi coverage area and the available resources in the mobile device cloud. Finally, based on the scheduling policy, the initiator distributes the tasks and data to its peers and collects the results before moving out of the coverage of the mobile device cloud.

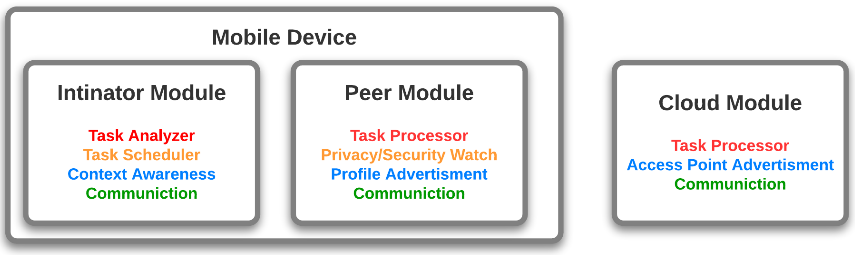

Figure 2: Initial System Design.

Building this system requires developing several components as shown in Figure 2. These components are responsible for connecting the devices, detecting and exchanging profile information (i.e. context awareness), analysis and scheduling of tasks of interest, and remote execution of tasks and collection of computation results. As part of this project, we demonstrate the communication protocol and offloading mechanism in action. We also developed an Android application to collect the required context information. Finally, we performed several simulations to validate the correctness of the proposed scheduling algorithm.

1.3 Report organization

This report presents the vision of all the components of the system while giving special attention to the components we plan to implement. The following sections describe the different components (as shown in Figure 2): communication, context awareness, tasks scheduling, and tasks processing. We consider task analysis and security and privacy to be outside the scope of this project.

2 Communication

Devices are expected to explore their environment to search for peers to collect context information. Each device should reply with its context information, so that the initiator can optimize its task dispatching to the different devices. We developed a simple protocol to allow the initiator to perform that simple task of context information collection.

2.1 Communication Protocol

The initiator sends a broadcast "Hello" messages. All devices that receive the "Hello" message reply with their context information (refer to section 3 for the details of the context information). The initiator waits for 2 seconds for all peers to reply[1]. We also specify a "maximum number of peers" parameter as the maximum number of replies that it waits for, even if the 2 seconds timeout has not been reached. UDP is used for broadcast messages and the context replies. The collected context information are all grouped in a Linked List that is then passed to the optimizer. The optimizer decides the dispatch schedule of tasks and their corresponding input data.

Once a schedule is reached, code and data are sent to each of the peers based on the assigned schedule. Task code is sent as a byte code file that is compiled based on the type of the peer. In other words, we assume that it's the responsibility of the initiator to provide compiled task code that is compatible with each of the peers. The data is sent as a serializable generic object using Java's `ObjectOutputStream` (Refer to Section 5 for more details on task processing). Both code and data are sent using TCP. After processing is done, answers are sent as the same generic serializable object.

To be able to track dispatched tasks, each data object is tagged with *task id*, *dispatch time*, and *dispatcher id*. The peer is expected to copy these information into the answer object it returns to the initiator. This information enables the initiator detect the actual performance of each peer. It also allows it keep track of the tasks and the answers it gets because answers will arrive out of order and each peer can be assigned more than one task.

2.2 Communication Implementation

Although our implementation could be extended to ad-hoc networks, we assume that one of the nodes acts as an access point. All peers search for and connects to that access point by default. We assume that this is done automatically. We use `Connectify` application on a laptop to create that access point.

A peer runs a server thread that is responsible for receiving broadcast messages and replying to them with its context information. The server listens on UDP port 1236. Each peer also runs a server to receive task code and data. It expects the data to be sent using `ObjectOutputStream`, followed by the code file as a stream of bytes. Both the code and the data are inserted in a queue for the task processor thread to handle.

The initiator broadcasts a message by sending a datagram to the broadcast address of the network (e.g. in our local network for experiments the broadcast address is 192.168.10.255, more generally, the address in the format of xxx.xxx.xxx.255). After gaining the dispatching schedule it sends the code and data to each node separately. It also runs a thread for receiving answer data and add them in a queue for final processing (e.g. command line output).

3 Context Management

The main purpose of our project is to provide the best scheme to offload tasks from a mobile initiator to neighbor mobile peers and to the cloud in order to realise the best possible performance. For that, the initiator needs to collect context information from the connected peers, to be able to estimate the load to be assigned to each of them. Our objective in the context management part is to provide the following context information:

[1]We assume that this is done only once at the beginning but it can also be done on demand to update the collected context information.

1. Unique ID
2. Network Bandwidth
3. CPU clock cycles that a peer is ready to share with the initiator.
4. Battery life left
5. Initial time of contact between initiator and peer.
6. Predicted final time of contact between initiator and peer within the next 5 mins.

3.1 Mobility Prediction

A key component in our application is predicting the user mobility pattern. To best ascertain the user location we have gone with a second order time varying Kalman filter. The Kalman filter has been designed based on the position and velocity measurements. The acceleration, for the time being, have been assumed to be constant. The equations used to design the filters are as follows:

Prediction Equations:

$$x(t) = Ax(t-1) + Bu(t)$$

where $x(t)$ is the predicted position at time t, A is $[[1,t],[0,1]]$ for a time t, B is $[t^2/2, t]$, $x(t-1)$ is the measured previous state, and $u(t)$ is the change in measured value per unit time.

$$P(t) = AP(t-1)A^T + E(x)$$

where $P(t)$ is the predicted covariance at time t, A^T is the transpose of A (A is same as defined for previous equation), and $E(x)$ is the covariance of the measurement noise(error in measurement).

Equations:

$$K(t) = P(t)C^T(CP(t)C^T + E(z))^-1$$

where $K(t)$ is the Kalman gain at time t, C is $[1,0]$, and $E(z)$ is the covariance of the measurement confidence.

$$x(t) = x(t) + K(t)(z(t) - Cx(t))$$

where $z(t)$ is the actual measurement data received from the sensor.

$$P(t) = (I - K(t)C)P(t)$$

where $P(t)$ is the updated covariance at time t based on Kalman gain. The equations marked as prediction equations will be used to predict future location for a user based on time. The equations marked as update equations will be executed in a recursive manner to update the Kalman gain and covariance based on the measurements received.

3.2 Implementation

- Unique ID: We need to create a unique random number to assign it to every mobile peer. We are using the Java function Settings.secure#android_id and to fix this unique value for each instance of context manager created.
- Network Bandwidth: We need to know the bandwidth available in the link between the initiator and the peer.the context management class is using the Wifi manager class of java to capture the relevant information.

- CPU clock cycles: We need to get the CPU usage percentage of the mobile and its CPU capacity in number of cycles in order to compute the number of cycles that the peer is able to share. For that were using the Linux command Top.

- Battery life left: we will use a Java class that can provide how much energy is left in the mobile battery. The battery life for the time left is supposed infinite, If time permits, we will add the corresponding Java class and we will include the battery life left to the data exchanged in order to be considered in the scheduler.

- Initial contact time and predicted final contact time: The measurements required for updating the Kalman filter are being captured using LocationManager class in Java.Once an initiator requests for context information from a peer, the peer will predict the future locations for the peer for each minute in a 5 minute period. The predicted locations will then be compared to the predicted locations for the initiator for the same time period to find if the peer is closing the distance to the initiator or moving away or moving alongside the initiator. Using the predicted location information we will be providing the initial and final contact point between the initiator and peer.If time permits, we will also include the acceleration information in the Kalman filter design. For capturing the data for acceleration we will be using the accelerometer.

We are implementing a feature to store the historical context information of a user in a file on the mobile in case the application is shut down. On start, the application will look for the saved historical data. If any historical data is found the Kalman gain and the related covariance will be updated based on the historical data.

Assumptions

We are supposing that the acceleration in the Kalman filter is constant, that the battery life is infinite and that the peer mobile devices connected to the initiator are collaborative and non malicious.

Result Evaluation

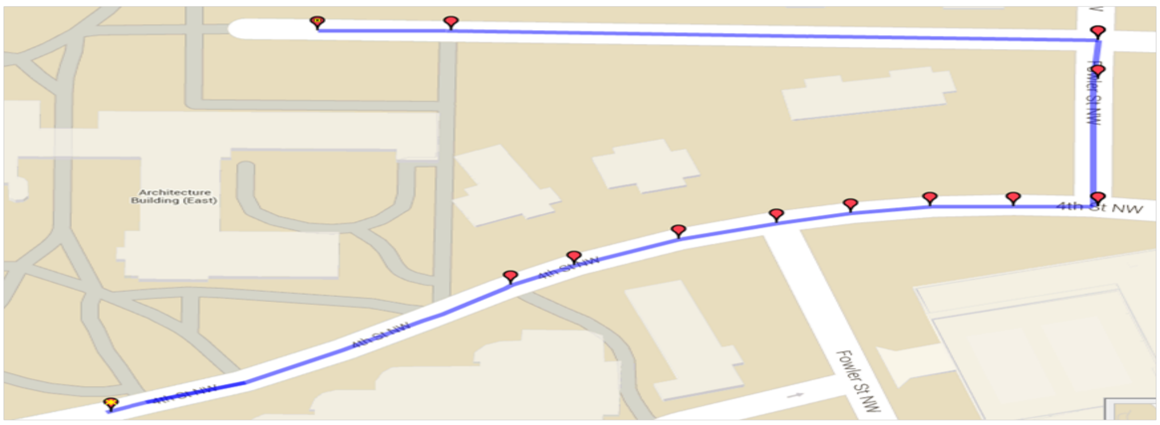

Figure 3: User input data (i.e. walking path).

On implementation of the Kalman Filter, we designed a Test module in Java to run the test in the android emulator of Eclipse. We generated a text file containing the route information from Clough to Klaus using a tool called Tyre. The text file contained the latitude and longitude of every way point on the route separated by a comma. The Test module was designed to read the text file from the emulator sandbox on the start of the application. The route information was then provided to the Kalman Filter as measurement input every 1 minute. We designed the Test module to print out predicted data by the Kalman filter every 30 seconds into the gap of the input data. First few inputs were ignored when representing the map as the Kalman filter was still adjusting at that time. Both maps have been generated with only the way points that were considered during testing.

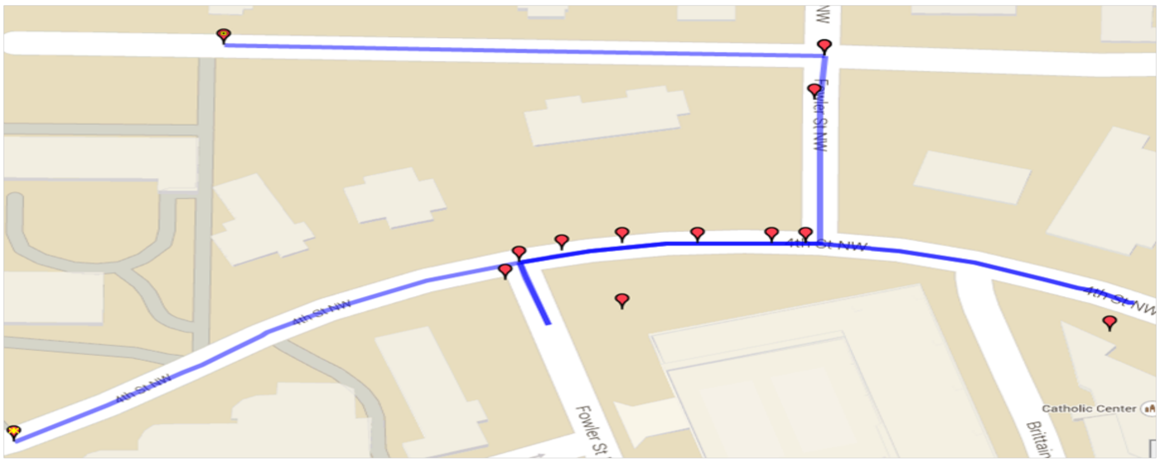

Figure 4: Kalman filter output.

Future Scope

The estimates used in the Kalman Filter have been adjusted manually based on test cases that were executed. Further improvements to the estimates can be achieved by using Monte Carlo Simulation. This will help in improving the prediction of the users mobility, which in turn will improve the efficiency of the computation offloading algorithm.

4 Task Scheduling

The main aim of the task scheduler is to decide whether to perform the task on the local device, wait to offload it to the cloud or offload it to nearby peers. This scheduling decision is made based on the collected context information which specifies the available resources and the constraints specified by the user (e.g. delay, cost, or energy). In our implementation, we have optimized the computation time of the task. The scheduling problem is essentially a decision problem and we have used mainly two approaches to solve it, one, by formulating it as an ILP problem, two, by using a greedy heuristic. It should be noted that we make few assumptions while solving the scheduling problem. We assume that there is only 1 initiator, there is an initial set of N tasks and all tasks have equal priority and can be done in parallel. In the next two paragraphs we will explain each approach in detail.

ILP

An initiator device contacts the access point to get the list of nearby peers to which tasks can be offloaded. The access point provides the list of peers along with several characteristics like time of initial contact, time of last contact, network bandwidth, bit rate, computational power of the node, etc. The initiator then decides which tasks it should offload to the peers, which tasks it can run itself and which tasks should it stall so as to wait for cloud access such that the time to complete all the tasks should be the earliest. For solving this decision problem, we have formulated an integer linear programming (ILP) model.

$$min(max(FT(local), \overset{n}{\underset{k=1}{\forall}} FT(k), FT(cloud)))$$
$$FT(local) = t_{initContactLocal} + \alpha(i)LC(i)$$
$$FT(k) = t_{initContactPeer}(k) + \beta(i,k)PC(i,k)$$
$$FT(cloud) = t_{initContactCloud} + \gamma(i)CC(i)$$
$$\alpha(i), \beta(i,k), \gamma(i) \in \{0,1\}$$
$$\alpha(i) + \sum_{k=1}^{numPeers} \beta(i,k) + \gamma(i) = 1, \forall i$$
$$FT(cloud) \leq t_{finContactCloud}, FT(k) \leq t_{finContactPeer}(k), FT(local) \leq t_{finContactLocal}$$
$$LC(i) = \frac{\text{Cycle\_Reqd}(i)}{LocalCompSpeed}$$
$$PC(i,k) = \frac{\text{Cycle\_Reqd}(i)}{PeerCompSpeed(k)} + \delta_{peer}(k) + \frac{d_{input}(i) + d_{output}(i)}{r_{peer}(k)}$$
$$CC(i) = \frac{\text{Cycle\_Reqd}(i)}{CloudCompSpeed} + \delta_{cloud} + \frac{d_{input}(i) + d_{output}(i)}{r_{cloud}}$$

ILP explanation

Let us consider that there are m tasks and n peers. Then the objective function is to minimize the finish time of jobs on each computational resource.

$$min(max(FT(local), \overset{n}{\underset{k=1}{\forall}} FT(k), FT(cloud)))$$

FT is finish time of all the jobs assigned to that computational resource which could be a peer, local device or cloud which can be expressed as:

$$FT(local) = t_{initContactLocal} + \alpha(i)LC(i)$$
$$FT(k) = t_{initContactPeer}(k) + \beta(i,k)PC(i,k)$$
$$FT(cloud) = t_{initContactCloud} + \gamma(i)CC(i)$$

$\alpha(i)$, $\beta(i)$, $\gamma(i)$ are indicator functions and take value 1 if the task i is performed on that device

$$\alpha(i), \beta(i,k), \gamma(i) \in \{0,1\}$$

Since the task is performed only on one device, hence the indicator variable should satisfy the constraint.

$$\alpha(i) + \sum_{k=1}^{n} \beta(i,k) + \gamma(i) = 1, \forall i$$

Moreover, the finish time of all the offloaded tasks on a device should be less than the final contact time with the device.

$$FT(cloud) \leq t_{finContactCloud}, FT(k) \leq t_{finContactPeer}(k), FT(local) \leq t_{finContactLocal}$$

Note that the final contact time of the local and cloud is assumed to be ∞ in our scenario.
$LC(i)$ is the computation time required to do the task i on the initiator and calculated using

$$LC(i) = \frac{\text{Cycle\_Reqd}(i)}{LocalCompSpeed}$$

```
Data: resources, taskList
Result: Hash mapping resource id with list of tasks to be executed on it
size = resources.size;
curDevFinishTime[size];
newDevFinishTime[size];
for $i \leftarrow 1$ to $size$ do
    $curDevFinishTime(i) \longleftarrow resources(i).firstContactTime$;
    Init offloadedTasks as empty hash map;
end
for $t \in taskList$ do
    $curMinFinTime \longleftarrow \infty$;
    $curMinPos = -1$;
    for $i \leftarrow 1$ to $size$ do
        $newDevFinishTime(i) \longleftarrow$
        $calculateTaskTime(resources(i), t) + curDevFinishTime(i)$;
        for $i \leftarrow 1$ to $size$ do
            if $newDevFinishTime(i) < resources(i).lastContactTime$ and
            $newDevFinishTime(i) < curMinFinTime$ then
                $curMinFinTime = newDevFinishTime(i)$;
                $curMinPos = i$;
            end
        end
    end
    curDevFinishTime(curMinPos) = newDevFinishTime(curMinPos);
    $key = resources(curMinPos).peerId$;
    $value = t.taskid$;
    $offloadedTasks.add(key, value)$;
end
return $offloadedTasks$;
```

Algorithm 1: Greedy task scheduler

Similarly, the computation time of task i on peer k and cloud can be calculated by adding the overhead due to network latency and device latency. The expression for the computation time are:

$$PC(i,k) = \frac{\text{Cycle\_Reqd}(i)}{PeerCompSpeed(k)} + \delta_{peer}(k) + \frac{d_{input}(i) + d_{output}(i)}{r_{peer}(k)}$$

$$CC(i) = \frac{\text{Cycle\_Reqd}(i)}{CloudCompSpeed} + \delta_{cloud} + \frac{d_{input}(i) + d_{output}(i)}{r_{cloud}}$$

Greedy Algorithm

Since solving ILP is an NP-hard problem, hence we apply a greedy approach to solve the scheduling problem as it gives solution in polynomial time. This scheduling problem is very similar to the knapsack problem. The greedy algorithm function proceeds by assigning each task to either local, peer or cloud in an iterative manner. For each task (sorted by priority), it calculates the minimum finish time if it were to be done locally, on the peer and and on the cloud and assigns the task to the corresponding device.

Evaluation

We performed simulations to evaluate the performance of our scheduling algorithm. The experiment setup constituted of an initiator, n peers and a cloud. For the sake of simplicity, the computation power and network bandwidth of each peer is kept equal. Also, the departure time of a peer is linearly related to its *id*. The computation power of cloud is $10x$ times the resource computation power. The time when the cloud first becomes available is known and in our experiments it is proportional to the total finish time of the task on the initiator itself. The set of tasks consists of identical tasks requiring equal computation time and network transfer time which is 20% of the computation time. This might be an overestimate as the tasks that we will like to offload in a real scenario might require less network resources but are more computationally intensive. We also assume a startup latency on the peers and cloud and this is more for cloud than the peers.

The first experiment aimed to quantify the benefit of using both cloud and peers instead of using only one of them. We ran a simulation where in we find out the finish time of the set of tasks under different scenarios as we vary the number of tasks. The different scenarios are: executing all the tasks locally, offloading only to peers, offloading only to cloud and offloading to both peers and cloud. It is clear from the Figure 5, the total finish time of the tasks get minimized if we utilize the computation power of peers as well as cloud.

In the next set of experiments we evaluated our scheduling algorithm (referred as smart greedy) against two other heuristics, Random and Naive Greedy. The random algorithm as the name suggests chooses a valid computation resources randomly. A resource is considered valid if it can compute the task and transfer the output before its departure time. Our second heuristic, which we call naive greedy algorithm, first sorts the peer in ascending order of their departure time. It then assigns the task to the first valid peer. The idea is to utilize the peer computation power as greedily as possible without taking into consideration the finish time of the task on that peer. Figure 6 shows the finish time when we vary the number of peers for the three scheduling algorithms. It is evident from the figure that the smart greedy algorithm outperforms both random and naive greedy algorithm. In the next experiment, we varied the number of tasks and kept the number of peers to be constant i.e. 5. It is clear from Figure 7 that finish time is least in smart greedy followed by naive greedy and random algorithm is the worst. Note that the straight lines are due to the fact that the tasks are identical.

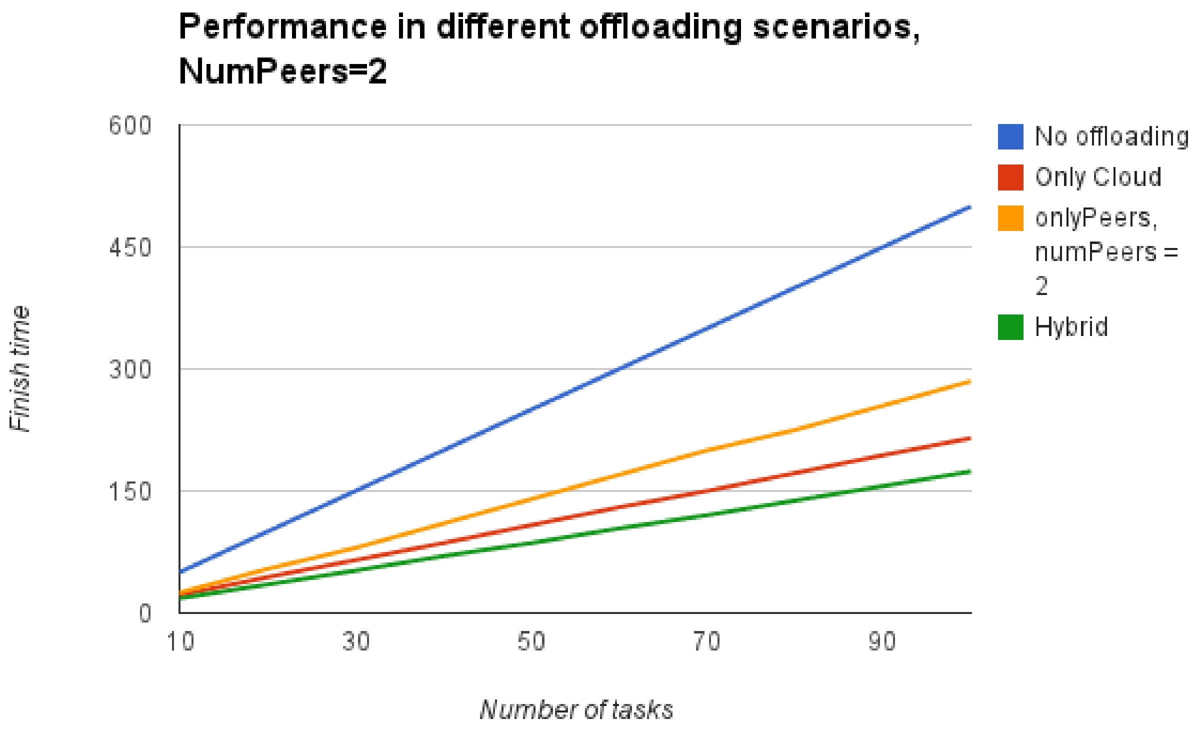

Figure 5: Task finish time for different offloading scenarios

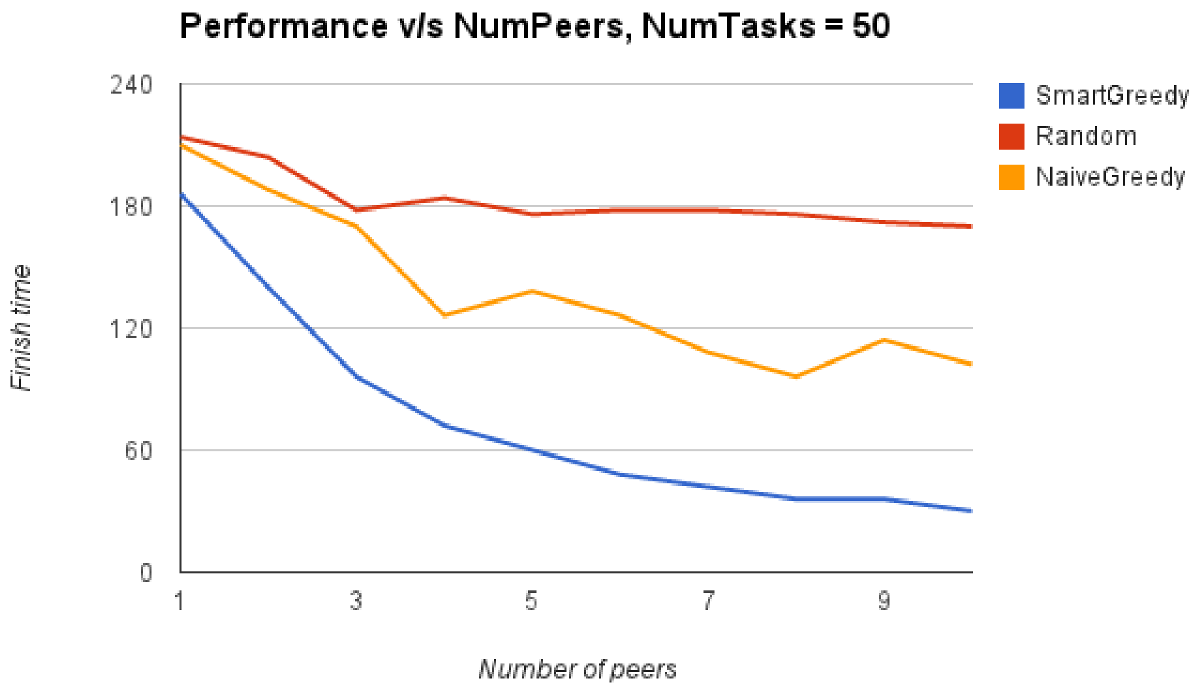

Figure 6: Finish time of tasks as the number of peers change

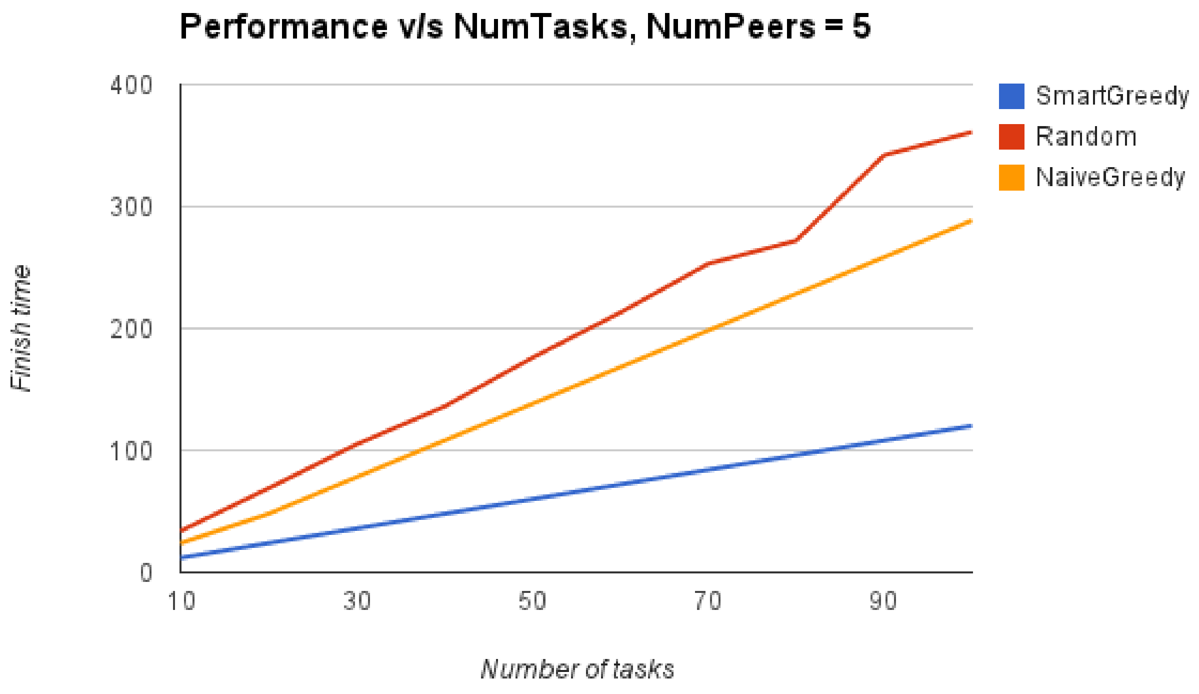

Figure 7: Finish time of tasks as the number of tasks vary

5 Task Processor and Demo Scenarios

The task processor has a very simple responsibility of checking the queue that is populated by the communication modules. All our implementation is on Java, hence, it's easiest to run the code sent by the initiator using reflection. For the rest of this section, we first talk about the details of the implementation then talk about the current state of our demo scenario.

5.1 Task Processor Implementation

The initiator sends compiled byte code to the peer. Our current implementation is only on laptops, hence, normal `.class` files are usable to be transferred and then executed using reflection at the the peer. A peer expects the code it receives to extend an abstract class that has an `initialize()` and a `run()` methods. The reflected instance is cast as the base class. The processor thread checks the queue populated by the communication module every second if the queue is empty and sequentially processes tasks once the queue is populated.

The received data is expected to be a generic data class that has different fields that can represent almost all types of input (i.e. an array of integers, an array of doubles, and an array of strings). We expect that the offloaded code knows how to understand the generic data object.

5.2 Demo Scenario

Figure 8: Intel Edison.

Our current demo is composed of two laptop that act as a peer and one of them acts as the access point. The initiator is an Intel Edison board (Figure 8) that connects to the access point. The current demo doesn't use the optimization modules. It also uses one dummy task (i.e. adding two number) that it sends to all peers. Each task is assigned a random ID to differentiate between tasks and keep track of them. The peers return the answer to the initiator and the initiator displays the result.

It should be noted that the number of peers is arbitrary as we can run the code on more nodes with no edits at all.

6 Link to the code

`https://github.com/kapiliitr/HybridComputationOffloading`

References

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

[2] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth*

ACM international symposium on Mobile Ad Hoc Networking and Computing, pages 145–154. ACM, 2012.