Workload Management for Dynamic Mobile Device Clusters in Edge Femtoclouds

Karim Habak
Georgia Institute of Technology
karim.habak@cc.gatech.edu

Mostafa Ammar
Georgia Institute of Technology
ammar@cc.gatech.edu

Ellen W. Zegura
Georgia Institute of Technology
ewz@cc.gatech.edu

Khaled A. Harras
Carnegie Mellon University
kharras@cs.cmu.edu

ABSTRACT

Edge computing offers an alternative to centralized, in-the-cloud compute services. Among the potential advantages of edge computing are lower latency that improves responsiveness, reduced wide-area network congestion, and possibly greater privacy by keeping data more local. In our previous work on Femtoclouds, we proposed taking advantage of clusters of devices that tend to be co-located in places such as public transit, classrooms or coffee shops. These clusters can perform computations for jobs generated from within or outside of the cluster. In this paper, we address the full requirements of workload management in Femtoclouds. These functions enable a Femtocloud to provide a service to job initiators that is similar to that provided by a centralized cloud service. We develop a system architecture that relies on the cloud to efficiently control and manage a Femtocloud. Within this architecture, we develop adaptive workload management mechanisms and algorithms to manage resources and effectively mask churn. We implement a prototype of our Femtocloud system on Android devices and utilize it to evaluate the overall system performance. We use simulation to isolate and study the impact of our workload management mechanisms and test the system at scale. Our prototype and simulation results demonstrate the efficiency of the Femtocloud workload management mechanisms especially in situations with potentially high churn. For instance, our mechanisms can reduce the average job completion time by up to 26% compared to similar mechanisms used in traditional cloud computing systems when used in situations that suggest high churn.

CCS CONCEPTS

•**Human-centered computing** → **Mobile computing;** *Ubiquitous computing;*

KEYWORDS

Edge Computing, Mobile Cloud Computing, Mobile Computing, Computational Offloading, Cloud Computing

ACM Reference format:
Karim Habak, Mostafa Ammar, Ellen W. Zegura, and Khaled A. Harras. 2017. Workload Management for Dynamic Mobile Device Clusters in Edge Femtoclouds. In *Proceedings of SEC '17, San Jose / Silicon Valley, CA, USA, October 12–14, 2017,* 14 pages.
DOI: 10.1145/3132211.3134455

1 INTRODUCTION

Edge computing offers an alternative to centralized, in-the-cloud compute services. Among the potential advantages of edge computing are lower latency that improves responsiveness, reduced wide-area network congestion, and possibly greater privacy by keeping data more local. Early efforts proposing edge computing include the vision of cyberforaging by Balan et al. [3], the Cloudlet work of Satyanarayanan et al. [28] and Cisco's Fog computing [4].

Recent trends in edge computing, including our own work on Femtoclouds [18], have argued for leveraging the resources of underutilized mobile edge devices. In Femtoclouds, we proposed taking advantage of clusters of devices that tend to be co-located in places such as public transit, classrooms or coffee shops. These clusters can perform computations for jobs generated from within or outside of the cluster. We observed that these devices could operate as a Cloudlet (i.e., an edge compute resource) when managed from a nearby controller, especially if the settings have semantics that suggest membership stability (e.g., students mostly arrive in class on time and stay until the end). In addition, the Femtocloud system overcomes some of the deployment barriers faced by cloudlets due to the staggering cost of deploying cloudlet-servers that cover the edge.

Our earlier work did not, however, address the full requirements for the *workload management* functions required within a Femtocloud. At a high level, these functions enable a Femtocloud to provide a service to *job originators* that is comparable to that provided by a centralized cloud service, namely the submission of jobs for completion in a timely and reliable manner[1]. Further, because the Femtocloud comprises mobile device *helpers*, the system must provide an interface for these devices to opt in and out. Under the covers, the system must manage a continually changing pool of helpers, assigning and moving computational work in response to job demands and device churn.

[1]Some cloud services provide dedicated servers. That is clearly not possible with mobile device clusters, and instead we focus on a comparable job processing service.

We begin with the observation that selective use of the cloud for control and management allows a Femtocloud to retain deployment advantages while significantly increasing the opportunity to provide a stable interface to both job initiators and willing helpers. In particular, a *controller* in the cloud can serve as the persistent and well-known contact point to receive jobs for processing, to receive helper requests to join, and to monitor helpers during job execution.

A cloud-based controller is a good starting point towards a stable service, but a collection of additional workload management functions are needed to further overcome and mask the effects of device churn. These functions bear some similarities to those used in traditional computation services (e.g., admission control, job/task assignment), however they depart from tradition by making primary the assumption that compute resources are highly dynamic.

These functions are required to enable the system to receive compute jobs from *job originators* and enable mobile devices called *helpers* to process them in a reliable and scalable manner. The system should also be able to handle a large class of jobs including those that consist of multiple interdependent *tasks*, where the dependency can be modeled with a directed acyclic graph (DAG). Mobile devices should be able to opt in the system to act as helpers and share their resources at any point in time regardless of their location. The system has to efficiently handle churn of helpers since devices may leave at any point in time due to user mobility, resource limitations, lack of connectivity and/or energy constraints.

In this paper we develop a system architecture that relies on the cloud to efficiently control and manage a Femtocloud. Within this architecture, we develop adaptive workload management mechanisms and algorithms to manage resources and effectively mask churn. These mechanisms include: (1) An efficient admission control mechanism designed to maintain system stability and avoid helper overload. (2) A task assignment algorithm that incorporates and adapts to critical path scheduling to suit the highly-dynamic and heterogeneous environment inherent in mobile device clusters. (3) A checkpointing mechanism to avoid losing the results of critical tasks after being completed due to helper churn. (4) A helper queue management algorithm that ensures fair resource allocation between the jobs that share the same helper.

We implement a prototype of our Femtocloud system on Android devices and utilize it to evaluate the overall system performance. We also use simulation to isolate and study the impact of each of our workload management mechanisms, and test the system at scale. Our prototype results demonstrate the efficiency of the Femtocloud workload management mechanisms specially in situations with potentially high churn. In particular, when the helper churn is relatively high, the Femtocloud workload management mechanisms workload management can reduce the average job completion time by up to 26% compared to the CPOP scheduling algorithm [35] which is used in traditional cloud computing systems. In larger scale experiments, our simulations showed that our admission control mechanism maintains the stability of the system regardless of the job arrival rate. In addition, using our checkpointing mechanism further reduces the average job completion time achieved by our task assignment mechanism by up to 31%.

Our contributions can be summarized as follows:

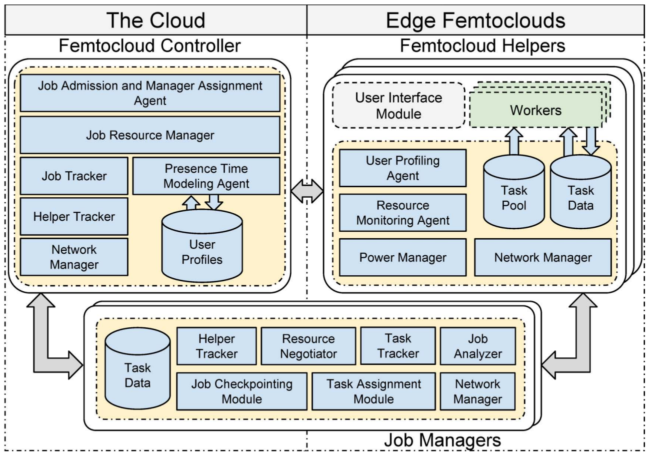


Figure 1: System architecture for edge Femtoclouds.

- A hybrid edge-cloud architecture that utilizes the cloud for management and to provide a stable service interface while using the edge for low latency computation.
- A set of workload management mechanisms that enable an edge computing service comprised of mobile devices with churn to serve DAG structured jobs.
- A prototype implementation that we use to evaluate the performance of the Femtocloud system. We also use simulations to assess the efficiency of each of our workload management mechanisms, independently.
- A pilot study to identify suitable incentive mechanisms to encourage users to opt in a Femtocloud system and share their mobile compute resources.

The rest of the paper is organized as follows. Section 2 describes the details of the Femtocloud system architecture. Section 3 develops the details of the workload management functions within the Femtocloud architecture. Section 4 show results that assess the efficiency of each of our workload management mechanism independently. Section 5 presents details of our prototype implementation and the performance evaluation of the whole system in a small-scale scenario. Section 6 addresses the challenges associated with large scale deployment. In particular, it discusses how to provide users with incentives to share their compute resources and surveys mechanisms to protect the Femtocloud architecture against malicious helpers and/or job originators. Finally, Section 7 summarizes related work and the paper is concluded in Section 8.

2 SYSTEM ARCHITECTURE

In this section, we provide an overview of the Femtocloud system and its main architectural components depicted in Figure 1. It consists of three main components: a *Femtocloud controller* running on the cloud that manages the available compute resources and provides the initial interface to job originators and helpers; a set of mobile devices running a *helper* client application and sharing portions of their compute resources; a set of *job managers* each of which is responsible for managing only a single job, taking over the interaction with the originator and the helpers for a certain job after the controller has accepted the job. Hence, job managers provide functional separation between the work to accept a job and the work to manage a job to completion. For scalability, they allow the work of run-time job management to be distributed. A job manager may be located in the cloud or, if appropriate, at a helper willing to take on more than just job computation. Furthermore, if a job manager can be located near its job originator and/or its helper set, rather than in the cloud, there may be performance advantages.

We assume that each job is represented by a directed acyclic graph (DAG) where each node in the graph is a *task* and a directed edge indicates a completion dependency. Each task node is labeled with an estimate of the computational requirement of the task; each edge is labeled with an estimate of the communication requirement to send the task results to downstream tasks that depend on them. This job model covers a wide range of applications, though clearly not all possible jobs. In particular, jobs whose task structure and requirements change at run-time do not fit this model.

We now provide the details of the three main architectural components (depicted in Figure 1).

2.1 Femtocloud Controller

The Femtocloud controller provides a registration and management interface for users who have a job to execute or helper resources to share. To maintain system reliability, the Femtocloud controller is deployed on a commercial cloud (e.g., Amazon EC2 or Windows Azure). Figure 1 illustrates the main components of the Femtocloud controller. To support helper management, the controller periodically collects information about helper status, user profiles, and resource availability through the **helper tracker** function. It uses the gathered information to model individual device churn using the **presence time modeling agent**. These models are later shared with the job managers to be used for task assignment and job management purposes. The controller also collects information about a helper's network connectivity and use it in deciding which helpers to use for which jobs. As a simple example, a job that requires a large amount data to be moved between between dependent tasks is ideally scheduled on helpers that are close to one another (in network terms).

In addition to helper monitoring, the controller is also responsible for job admission and resource allocation. When a new job arrives, the **job admission and manager assignment agent** first decides whether or not the job can be accepted and executed by the Femtocloud cluster using the algorithm presented in Section 3.1. If accepted, the controller spawns a job manager in the cloud to handle the recently accepted job. The controller may then migrate the job manager to the task originator or one of the helpers to enhance communication efficiency and increase responsiveness. It periodically communicates with the job manager to collect job progress updates through the **job tracker**. It also handles the job manager's resource allocation requests using the **job resource manager**.

2.2 Femtocloud Helpers

A Femtocloud helper is a client application responsible for executing assigned computation tasks. A user first needs to configure his/her personalized resource sharing policy and privacy requirements via the **user interface module**. These policies influence the behavior of the **user profiling agent**, by dictating what is allowed and not allowed with respect to monitoring user mobility and device usage patterns. User profiles are generated and shared with the Femtocloud controller based on user-defined privacy constraints. Additionally, the helper monitors the CPU and memory usage using the **resource monitoring agent**, and implements the **power manager** functionality which tracks the available battery level on the device. All this information is periodically shared with the controller and any job managers to which the helper has been assigned.

To support task assignment that is cognizant of network performance, each helper implements a **network manager** that monitors the available network interfaces to estimate the bandwidth and the round trip time (RTT) while communicating with other entities (controller, job originators, and other helpers).

Each helper has a pool of tasks assigned to it by job managers. It also has a set of worker modules each of which is in charge of a single shared CPU resource (e.g., core). Once a CPU resource becomes available, the worker in charge of that resource picks up a task from this pool using the algorithm described in Section 3.3 in order to provide fairness among different jobs. Once the task completes execution, the worker writes the task's results and state information to the task data storage.

2.3 Job Managers

A job manager is responsible for handling only one job and dealing with the job's originator. It starts as a service at the controller and can then be migrated to one of the helpers or the job originator to enhance communication efficiency and increase responsiveness while handling originator requests and/or managing helpers. Once started, a job manager analyzes the task dependency graph of its associated job to determine its critical portions and overall resource requirements using the **job analyzer**. Based on this information, the **resource negotiator** contacts the controller seeking resources as needed. We describe the details of the resource allocation and negotiation process in Section 3.2.2. Once a set of helpers are assigned to the job manager by the controller, the job manager periodically collects information about the available resources at these helpers through the **helper tracker**. It also collects information about the helper network connectivity using the **network manager**.

The job manager's main function is to quickly complete the associated job and return the results to its originator. To achieve this goal, the **task assignment module** uses the available information about tasks and helpers to assign tasks accordingly. We present the details of the task assignment mechanism in Section 3.2.3. It, further, tracks the progress of the assigned tasks using the **task tracker**. To mitigate the effect of churn and avoid re-executing tasks upon a departure of a helper, the **job checkpointing module** selectively backs up the results of a subset of the completed tasks on a selected set of helpers as well as the cloud using the algorithm described in Section 3.4.

3 WORKLOAD MANAGEMENT

A brief scenario would best explain the breakdown and interaction between each of the mechanisms described in subsections 3.1 through 3.4. We begin with a Femtocloud controller running in

Table 1: Job Parameters

Symbol	Description
c_T	Compute (processing) requirement of task T
o_T	The size of the output of task T
e_T	The size of the executable code coupled with the external data needed by task i
d_{Tk}	Determines whether task T requires the output of task k to start executing (1) or not (0)
f_T	Determines whether the helpers have finished executing task T (1) or not (0)

the cloud and actively managing a group of helpers sharing some compute resources and running tasks assigned to them by a set of active job managers. When a new job is created at a job originator, this originator contacts the Femtocloud controller inquiring whether or not it can help execute this new job. The controller initially relies on our admission control mechanism presented in subsection 3.1 to decide whether to accept or reject this job. If accepted, the controller instantiates a new job manager to assign job tasks to various helpers as described in subsection 3.2. This new job manager analyzes the task dependency graph of the job and identifies its critical sections, divides the job into stages to mitigate churn (i.e., impact of helper/resource departure), requests resources on-demand, and distributes tasks across available helpers accordingly. Since a single helper can be assigned to multiple job managers, we use the algorithms described in subsection 3.3 to achieve a fair compute resource distribution across jobs on a given helper. Finally, once a helper finishes executing a task, the job manager may decide to replicate the results of the task on multiple helpers and/or in the cloud to avoid the need for re-executing these tasks to further mitigate helper churn. This checkpointing mechanism is described in subsection 3.4.

All our mechanisms and algorithms rely on the structure of the job DAG and on estimates of job computation and data requirements. The set of parameters representing these requirements is shown in Table 1. Note that the DAG structure of jobs has been thoroughly studied and is widely used [29]. In addition, we only need estimates for computation and data parameters to guide our decision making. These estimates can be obtained using techniques such as those outlined in [6]. The effect of errors in these estimates is evaluated in Section 4.

3.1 Job Admission Control

A Femtocloud strives to minimize the job completion time and enhance its users' quality of experience under the constraint of not overwhelming the helpers. Therefore, it is critical for the Femtocloud controller not to accept incoming jobs that will overwhelm the helpers and are beyond the system capacity. There are many options for the admission control policy, and our aim in this paper is not to explore them in detail. Instead, we use a simple job admission policy in which the controller accepts new jobs based on progress towards completion of the jobs already in the system. Specifically, if the total relative work remaining on existing jobs is above a threshold, the new job is rejected. An appropriate value for the threshold can be learned and adapted over time using measurements of completion time for jobs, average level of work parallelization for jobs, number of helpers in the system, and average helper utilization.

To apply our admission control policy, only the job progress information is periodically reported from active job managers to the controller. The current relative progress of a job is calculated at the job manager using the following equation:

$$\rho = \frac{\sum_T f_T c_T}{\sum_T c_T}$$

where ρ is the relative job progress, $\sum_T f_T c_T$ is the computational load of the fully executed tasks, and $\sum_T c_T$ is the job's computational load.

Although this method values progress on short and long jobs equally with respect to the admission decisions, the size of the job is implicitly taken into account since shorter jobs will progress faster than longer ones. For instance, if all the arriving jobs are relatively short, it is expected that they will progress faster and thus more jobs will be admitted over time. On the other hand, if all the jobs are relatively long, less number of jobs will be admitted over time due to the slow progress of the jobs.

3.2 Single Job Task Assignment

In this section, we develop a task assignment algorithm that takes a single job, represented by a DAG, and assigns its tasks to helpers such that the job completion time is minimized. The algorithm consists of two key mechanisms for (1) helper allocation and (2) task assignment. The helper allocation mechanism, presented in section 3.2.2, assigns the job a set of helpers that matches its requirements. The task assignment mechanism, presented in section 3.2.3, decides which task is to be executed by which helper. In the next section we deal with issues of fairness between jobs that share the same helpers.

Two conflicting considerations must be balanced in the assignment algorithm. The first is that helpers are ephemeral, thus suggesting a conservative approach to assigning tasks to a given helper, in case it leaves the system before completion[2]. The second is that there may be significant communication costs with transferring the output of one task to those downstream, thus suggesting that tasks with data dependencies should be scheduled on the same helper. On the other hand, the fact that different jobs may have different bottlenecks suggests that the task assignment algorithm should take the task requirements and its location in the DAG of the job into account while making any assignment decision. Finally, the churn in helpers suggests that overall the algorithm should schedule tasks in batches, retaining the ability to adapt as the job executes.

3.2.1 Critical Path and Stages. At a high level, tasks in the job dependency graph form execution paths, each of which consists of a set of tasks that has to be sequentially executed. The path with the highest total computation requirement is referred to as the *critical path*. As tasks complete, the remaining computation load in each path may change. Therefore, we use a notion of the *current critical*

[2]We later discuss the use of checkpointing to help with this issue. Checkpointing has costs, however, so reducing the need for checkpointing is important.

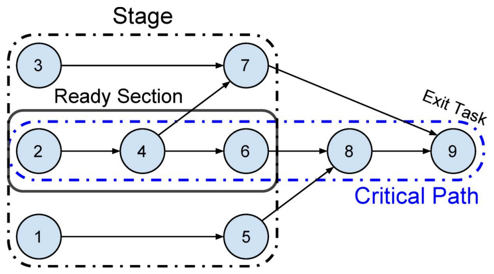


Figure 2: Critical path, ready section, and stage illustrative example. For simplicity, all tasks are assumed to have equal computational demand

path, i.e., the path of tasks in the job dependency graph with the highest total remaining computation requirement.

As shown in Figure 2, we refer to the last task of the job that marks its completion as *the exit task*. Formally, the computational requirements of the current critical path to the exit task T_{exit}, ($C_{cp}(T_{\text{exit}})$) can be calculated using the following formula:

$$C_{cp}(T) = \begin{cases} c_T + \text{argmax}_k \left[d_{Tk} C_{cp}(k) \right] & \text{if } f_T = 0 \\ 0 & \text{if } f_T \neq 1 \end{cases}$$

where $C_{cp}(T)$ is the compute requirements of the critical path leading to task T including the task itself, c_T is the compute requirements of task T, f_T determines whether task T has already been completely executed (1) or not (0), and d_{Tk} determines whether task T requires the output of task k to start running (1) or not (0).

A subset of the tasks in the current critical path has no unfulfilled incoming data dependencies and are ready to be executed once assigned to a helper. We refer to this set of tasks as the *ready section*, which we prioritize when we schedule tasks. Figure 2 shows an example that illustrates the difference between the critical path and the ready section for a job. Progress on the critical path will be important to scheduling on-critical-path tasks. In particular, we will keep track of the *relative critical path progress* from executing a ready section:

$$\mathcal{P} = \frac{\sum_{T \in S_{\text{ready}}} c_T}{C_{cp}(T_{\text{final}})}$$

where $\mathcal{P}$ is the relative critical path progress and S_{ready} is the ready section's set of tasks.

In the absence of helper/resource churn, as in most classical cloud environments, it is clear how to assign tasks to resources once the critical path is identified. The critical path tasks can be assigned first and then all the non-critical path tasks can be assigned to the resource that is going to finish them fastest [35]. The assignment decision, however, is not as straightforward in a Femtocloud due to potentially high churn. For instance, rushing into assigning and executing tasks that do not belong to the critical path before their results are actually needed may lead to result loss (if the helper leaves) and the need to re-execute.

Lemma 3.1. *For every non-critical path in the job dependency graph, the path's computational progress will not affect the job completion time if the relative progress of the path, at any point in time, is at least equal to the critical path's relative progress.*

Based on Lemma 3.1, we highlight that an efficient job assignment strategy should (1) strive to achieve high progress in the critical path, and (2) maintain the relative progress of any non-critical path to at least be equal to the relative progress of the critical path. In addition, we argue that the assignment strategy should also avoid achieving relatively high progress in any non-critical path to mitigate the risks associated with the churn of helpers.

To achieve these goals, we define a **stage** as the smallest set of tasks that include (1) the current critical path's ready section, and (2) all the tasks, belonging to non-critical paths, that are needed to keep the relative progress of these paths greater than or equal to the relative progress of the critical path. Figure 2 shows an example that illustrate our main concepts. The main objective of our task assignment mechanism thus can be reduced to minimizing the time needed to finish the execution of all the tasks that belong to the current stage.

3.2.2 *Helper Allocation.*

The controller is responsible for managing the complete pool of helpers and for allocating them to job managers dynamically. A job manager requests helpers from the controller using one stage lookahead, to avoid delay when the next stage is ready to start. The job manager estimates its need for helper capacity in the next stage by estimating the expected stage-level parallelism using the following equation:

$$E_p = \left\lceil \frac{\sum_{T \in S_{\text{stage}}} c_T}{\sum_{T \in S_{\text{ready}}} c_T} \right\rceil$$

where E_p is the expected stage-level parallelism ($E_p \geq 1$), S_{stage} is the set of tasks in the target stage, S_{ready} is the set of tasks in the critical path ready section of that target stage, and c_T is the compute requirement of task T. In example illustrated in Figure 2, the expected stage-level parallelism equals 2.33 and the target number of helpers equals 3 (the ceiling of E_p) .

The job manager compares the target number of helpers for the next stage to the current set of helpers and decides to request/release helpers accordingly. If the manager decides to release helpers, it releases the ones with the least expected compute resource availability (the allocated compute capacity multiplied by the expected presence time). Otherwise, it sends a request to the controller with (1) the target number of additional helpers, (2) the target helper's compute resource availability of the ready section compute requirements ($\sum_{T \in S_{\text{ready}}} c_T$).

Once the controller receives a resource allocation request from a job manager, it checks the availability of helpers that satisfy the requested compute requirement based on their (1) expected presence time, (2) shared compute capacity, and (3) other-job commitments. Helpers that satisfy the compute requirement with the minimum average latency while communicating with the originator, job manager, and already assigned helpers are selected and assigned to the job. If the number of helpers that satisfy the requested compute requirement does not fulfill the job manager's request, the controller will assign all of available helpers, if any, to the job manager to enable it to start tasks as soon as possible. It will also maintain the job manager's request and assign additional helpers as they join the Femtocloud.

Algorithm 1 Task Assignment

```
1: procedure AssignTasks({T}, {H})        ▷ T is of a task type. H is of a helper type
2:     selectedH ← null; lowestRisk ← +∞;
3:     for H in {H} do
4:         compTime[H] ← H.completionTime({T});
5:         churnProb[H] ← H.churnProbability(compTime[H]);
6:         if lowestRisk > churnProb[H] then
7:             selectedH ← H;
8:             lowestRisk ← churnProb[H]
9:         end if
10:    end for
11:    SortedH ← sorted({H}, churnProb)        ▷ Risk based sorting
12:    for H in {SortedH} do
13:        if compTime[H] > compTime[selectedH] then
14:            continue;        ▷ high risk, high compute time helper
15:        end if
16:        AddedRisk ← $\frac{churnProb[H]-churnProb[selectedH]}{churnProb[H]}$;
17:        Gain ← $\frac{compTime[selectedH]-compTime[H]}{compTime[selectedH]}$;
18:        if Gain > AddedRisk then
19:            selectedH ← H;
20:        end if
21:    end for
22:    return selectedH
23: end procedure
```

3.2.3 Risk-Controlled Task Assignment. Once a stage is ready to be started and its needed resources are allocated, the task assignment process begins. To assign the stage tasks to their helpers, we adopt a path-based assignment policy to minimize the overhead of moving data between helpers. We implement a path selection mechanism to pick paths from the current stage to be directly assigned to their helpers. This mechanism iterates on all the paths of unassigned tasks with fulfilled dependencies and selects the one with the highest total compute requirements to be assigned first. This mechanism is repeated until all the tasks in the stage are assigned to helpers.

When a path of tasks is ready to be assigned to a helper, we use Algorithm 1 to select the helpers to which the tasks should be assigned. In this algorithm, we first estimate the amount of time needed to finish all the tasks in the path on each helper based on (1) the amount of time needed to send each task coupled with its input data to the helper, and (2) the execution time of all the tasks based on the helper capacity. Let us use T_{ch} to denote the path completion time of the path on helper h. We also use $\mathcal{T}_{ph}$ to denote the total presence time of helper h measured from its arrival time. Based on the estimated completion time and the helper's churn model, we estimate our risk factor, called *churn probability* $P_h(\mathcal{T}_{ph} < (T_{ch} + S_h)|\mathcal{T}_{ph} > S_h)$, which is the probability that the helper will opt out of the system prior to completing the tasks. Note that S_h denotes the helper's time in the system up until this moment. The churn probability is computed using a model of the distribution of the time a helper spends in the system, also called *presence time*. This distribution can be learned using the User Profiling functions in the helper client application and/or learned on a system-wide basis by the controller.

Once the churn probability and the completion time are calculated for all the helpers, we sort the helpers according to their churn probability and the helper with the lowest churn probability is selected. Then, we iterate on the sorted list of helpers using a risk-controlled mechanism, commonly used in economics [30], to compare the gain of switching to this helper as the reduction ratio in the task completion time to the relative addition in the risk. In particular we use the following equations to calculate the gain and the risk:

$$G = \frac{T_{ch^*} - T_{ch}}{T_{ch}}$$
$$R = \frac{F(h, T_{ch}) - F(h^*, T_{ch^*})}{F(h, T_{ch})}$$

where G is the relative gain of using using h over the selected helper h^*, $F(h, T_{ch})$ is a function that calculate the churn probability of h^{th} helper ($F(h, T_{ch}) = P_h(\mathcal{T}_{ph} < [T_{ch} + S_h]|\mathcal{T}_{ph} > S_h)$) and R is the relative added risk introduced by using helper h over the selected one. If the gain exceeds the risk, it switches to the helper with the higher gain.

3.3 Multi-Job Helper Queue Management

Since the same helper can be assigned to multiple job managers and can execute tasks that belong to different jobs, determining the appropriate order in which the helper executes these tasks is important. A helper should be (1) predictable, allowing the job managers to make correct decisions, and (2) optimized to enhance the performance of the jobs that it contributes to.

3.3.1 Fair queuing based task pick up. To enhance predictability, we implement a fair queuing based task pick up mechanism, described in Algorithm 2. Each helper maintains multiple execution queues, each of which is associated with only one job. Each of these queues is associated with a credit counter used to insure fairness. The process starts when a helper becomes associated with a new job manager. In this case, the helper creates a new queue for the tasks that belong to the new job and assigns it zero credit. When a worker thread becomes available at the helper, it invokes the "executeTask" function to pick up a task from these queues and executes it. To insure fairness, this function will pick up the first task in the queue that has the highest credit, executes it, decreases the credit of the queue by the amount of time taken to finish the task. To avoid credit drifts, it adjusts the queue credits maintaining the same relative difference with every pick up decision.

3.3.2 Deadline-based Optimization. As we described in Section 3.2.1, tasks differ in their urgency level depending on whether they belong to the critical path or not. Even within the same stage different paths may significantly differ in terms of their task compute requirements and their computation time on the helper to which they are assigned. Therefore, their task execution urgency may significantly vary. For instance less urgent tasks on a high capacity helper may tolerate delays without affecting their stage completion time. Such delay tolerance can be utilized to execute more urgent tasks and enhance the overall system performance. To utilize this fact, we rely on the job managers to set a starting deadline for each task while assigning it and extend the fair queuing

Algorithm 2 Helper Queue Management

```
procedure EXECUTETASK({Q})                          ▷ Q is a task queue type.
    adjustCredit({Q});
    selectedQ ← selectQueue({Q});
    execute(selectedQ.popHead());
    selectedQ.credit ← selectedQ.credit - ExecTime;
end procedure
procedure SELECTQUEUE({Q})                          ▷ Q is a task queue type.
    sortedQ ← sorted({Q});                   ▷ descending sorting on credit.
    earliestDeadline ← +∞; timeBuffer ← +∞;
    selectedQ ← null;
    for Q in sortedQ do
        if not Q.isEmpty() then
            continue;
        end if
        if Q.head.startingDeadline() < earliestDeadline and Q.head.exTime() < timeBuffer then
            selectedQ ← Q;
            earliestDeadline ← Q.head.startingDeadline();
            timeBuffer ← min(timeBuffer - Q.head.exTime(), earliestDeadline - now);
        end if
    end for
    return selectedQ
end procedure
procedure ADJUSTCREDIT({Q})                         ▷ Q is a task queue type.
    maxCredit ← −∞ ;
    for Q in {Q} do
        if not Q.isEmpty() then
            maxCredit ← max(maxFreq, Q.credit);
        end if
    end for
    for Q in {Q} do
        Q.credit ← min(0, Q.credit - maxCredit);
    end for
end procedure
```

based task pick up mechanism to take these deadlines into account while picking up tasks for execution.

To assign a starting deadline for a task, the job manager implements a two phase mechanism. First, while assigning the first path of a stage (the critical path's ready section) to a helper, we estimate the target stage completion time which is equal to the estimated path completion time. The second phase is activated while assigning all the remaining paths. In this phase, while assigning a path, tasks are assigned starting with deadlines derived from the target stage completion time and the helper's shared capacity.

At the helper, we implement an early pick up mechanism, as shown in Algorithm 2, where urgent tasks from queues with low credit can be executed before tasks from ones with higher credit **if and only if** they will not interfere with their deadline requirements.

3.4 Task Checkpointing

Section 3.2 presents two key approaches to mitigate the impact of helper churn prior to fully executing tasks. First, using the concept of execution stages avoids executing tasks and getting their results too early compared to when they are needed. Second, the risk-controlled task assignment minimizes the risk of helper departure prior completing the assigned tasks. Once the task results become available, however, it is essential to preserve them till they are used in order to further mitigate the effect of churn. Therefore, we implement a checkpointing mechanism with which the job manager may replicate the results of a selected set of finished tasks on a set of helpers and/or in the cloud. The main objective of this replication is to avoid losing the results of finished tasks.

Our checkpointing mechanism runs periodically (every 15 seconds in our implementation) and determines for a finished task whether new replicas need to be added, the current state needs to be kept as is, or the results need to be deleted since they are no longer needed. To describe our mechanism, let's use r_{Th} to indicate whether the results of task T exist on helper h (1) or not (0).

The process starts with estimating the probability of losing the results of each of the completed tasks in a specific period of time X using the following equation:

$$P(\text{loss-time}(T) < X) = \prod_{h=0}^{m} [1 - r_{Th}(1 - F(h, X))]$$

where loss-time(T) is the time needed to lose all the replicas of task T, and $F(h, X)$ is a function that calculate the churn probability of h^{th} helper ($F(h, X) = P_h(\mathcal{T}_{ph} < [X + S_h] | \mathcal{T}_{ph} > S_h)$) during a period X given the helper's prior stay of S_h. Note that if a replica of the results of task T exist on the h^{th} helper ($r_{Th} = 1$), $[1 - r_{Th}(1 - F(h, X))] = F(h, X)$. However, if the helper does not have a replica of the task results ($r_{Th} = 0$), $[1 - r_{Th}(1 - F(h, X))] = 1$. We set X to be equal to twice the time needed to re-execute the checkpointing mechanism ($X = 30$ seconds in our implementation).

A task is considered *well-maintained* if its loss-time probability is less than a *reliability threshold ($\mathcal{K}$)*, which is set based on the environment and target level of reliability. To decide if tasks have to be replicated to reach a well-maintained status, we iterate over the not-well-maintained tasks in order of their loss-time probability and calculate the amount of computation (E_T) needed to reconstruct their results from the set of well-maintained tasks. We compare E_T to a linear function of the result size of the task (a_T) and decide accordingly whether the task needs to be replicated or not. Note that the coefficients of this function are environment dependent and based on the available bandwidth between helpers and their average shared capacities.

Once all tasks that require replications are marked, we re-iterate on them in order to determine the ones no-longer needed, relative to the current set of well-maintained tasks. We then issue replication requests for the ones that require additional replication followed by a delete request for the results of the tasks that are no longer needed to free up storage at the helpers. To replicate a task, we iterate over helpers in descending order of their churn probability and assign a replica to a helper if and only if it has available storage. This process is repeated until the loss-time probability becomes lower than the reliability threshold $\mathcal{K}$.

4 MECHANISM EVALUATION

In this section, we evaluate the performance of Femtocloud and assess the efficiency of its workload management mechanisms. We use simulations to isolate the true impact of each mechanism individually. We simulate different scenarios and environments,

Table 2: Experimental tasks' characteristics.

Task Type	Computation	Output
Lightweight tasks	10 MFLOPs	0.2 MBytes
Medium tasks	30 MFLOPs	2 MBytes
Compute Intensive tasks	100 MFLOPs	0.5 MBytes
Data Generating tasks	20 MFLOPS	20 MBytes

analyze the impact of using various management mechanisms, and study the effect of different parameters on the performance of the Femtocloud system. We start by describing our experimental setup (Section 4.1) followed by representative simulation results (Section 4.2).

4.1 Experimental Setup

We start by describing the experimental job model followed by the summary of the characteristics of the set of mobile devices used in our experiments. We then summarize our metrics and parameters followed by presenting our baselines.

4.1.1 Job models. In our experiments, we use a set of synthesized jobs to evaluate the performance of our workload management mechanisms. To construct the task dependency graphs for these jobs, we use a set of models that represent a large variety of application and programming paradigms. We focus on the following four representative job models:

- **Pipeline Job Model:** All the tasks in the job form a single path and must be sequentially executed. This model represents a wide range of single threaded jobs and/or applications.
- **Parallel Path Model:** Tasks form p parallel paths with very limited inter-path dependencies (set at a 0.1 probability). This job model represents multi-threaded applications with minimum inter-thread dependency and synchronization.
- **General Parallel Path Model:** Tasks form p parallel paths with more significant inter-path dependencies (set at 0.4 probability). This job model reflects general multi-threaded applications.
- **Pyramid Job Model:** Tasks form a tree structure where the task dependency direction goes from leaf-nodes towards the root. This model encompasses a wide range of map-reduce jobs.

To construct a job that follows one of these models, we first generate a fixed number of tasks. Each of the generated tasks falls in one of four categories: (1) lightweight tasks, (2) medium tasks, (3) compute intensive tasks, and (4) data generating tasks. Based on the selected category, we pick the computational requirements of the task and the output size from a normal distribution with the mean values listed in Table 2. Once all the tasks are built, we set the dependency edges between them using a pseudo-random process based on the target job model

4.1.2 Device Characteristics. To evaluate the performance of Femtocloud under realistic helper characteristics scenarios, we identify the available compute capacity in a variety of mobile devices.

Table 3: Experimental helper's characteristics.

Devices	Computation Capacity
Galaxy S5	3.3 MFLOPS
Nexus 7 [2012]	7.1 MFLOPS
Nexus 7 [2013]	8.5 MFLOPS
Nexus 10 [2013]	10.7 MFLOPS

We use matrix multiplication operations to emulate the compute capacity of a set of mobile and handheld devices. To achieve this, we measure the time needed to finish a load of 200 MFLOPs in a background thread using Galaxy S5, Nexus 7 and Nexus 10 devices and use the measured time to calculate the device compute capacity in MFLOPS. For each device, we repeat this process and take the average of 20 runs to measure the average compute capacity of the device. Table 3 summarizes the capacities we establish for the mobile devices we test.

4.1.3 Metrics and parameters. We are interested in the following performance metrics:

- **Job Completion Time:** This is the average amount of time needed to completely execute a job.
- **Job Admission Ratio:** This is the ratio of the number of admitted jobs to the total number of incoming job requests.

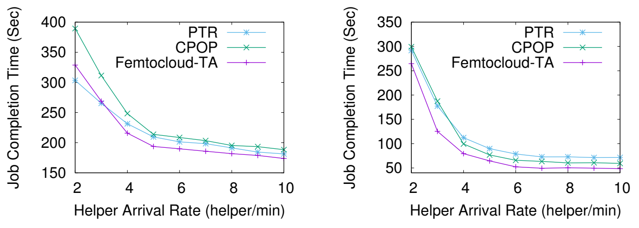
To assess the impact of each of our workload management mechanisms, we measure the performance of the system running our task assignment algorithm while turning other mechanisms (e.g., admission control, task checkpointing, early task pick up) ON and OFF.

4.1.4 Baselines. We compare our task assignment algorithm with the following baseline techniques:

(1) **Critical path in a processor (CPOP)[26, 35]**: The CPOP scheduler aims to assign the critical path to the node that will execute it faster. To assign all the remaining tasks, it works in two phases. In the first phase, it assigns priorities to the tasks based on the amount of computations leading to them and the amount of computations after them. In the second phase, it assigns the highest priority task first to the node that will finish it faster.
(2) **Per-task risk-controlled assignment (PTR)**: The PTR scheduler takes the scheduling decision for every task independently. Once a task is ready to execute (all its dependencies are fulfilled), it applies the same risk-controlled assignment mechanism as Femtocloud to select its executing helper. If more than one task is ready to execute, it assigns the one with the largest compute resource requirements first. This derived from the risk-adjusted return economic principle and is currently used by systems like COSMOS[31]

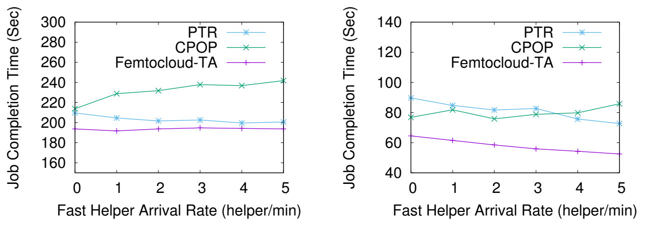
4.2 Results

In this set of experiments, we organize the helpers and the job originators in four groups each of which represents an enterprise network. The bandwidth and latency between two nodes in our experiment are modeled using a Normal Distribution where the



(a) Pipeline Jobs (b) General Parallel Jobs

Figure 3: Impact of changing the helper's arrival rate on the task assignment performance.



(a) Pipeline Jobs (b) General Parallel Jobs

Figure 4: Impact of helper's presence time heterogeneity on the task assignment performance.

standard divination is set to 20% the mean. The average bandwidth and latency between two nodes in the same group is 30 Mbps and 25 msec, respectively. However, the average bandwidth and latency between two nodes in different groups are 10 Mbps and 100 msec, respectively. We use a Poisson arrival process to model the arrival of new helpers. The helper arrival rate is set to 5 helpers/min and arriving helpers are randomly assigned to one of the groups. In addition, the helper presence is modeled as Normal(5 min, 1 min).

In this section, we only present the results from using the **General Parallel Path Model** and the **Pipeline Job Model**. We select these two models since (1) they cover a wide range of applications and real-jobs, and (2) they are considered the two extremes of our job model spectrum and they reveal all the interesting insights[3]. We use a Poisson arrival process to model the arrival of new jobs, where the job arrival rate is set to 5 Jobs/min. In addition, each job has 30 tasks that form its dependency graph. For the jobs that follow the **General Parallel Path Model**, we set the number of parallel paths to be equal to 5.

To assess the efficiency of each of our mechanisms independently, we start by disabling all our mechanisms and compare the performance of our task assignment mechanism to the base-line assignment mechanisms in Section 4.2.1. We then analyze the performance of our admission control mechanisms under different configurations in Section 4.2.2. Section 4.2.3 shows how using our task checkpointing mechanism helps in high churn situations. Section 4.2.4 analyzes the sensitivity of our workload management mechanisms to estimation errors. Each experiment represents the average of 10 runs.

4.2.1 Risk Controlled Assignment Performance. In this section, we study the impact of using our Femtocloud task assignment mechanism and compare its performance with the two baseline task assignment mechanisms (CPOP and PTR). To ensure fairness, we allocate all the helpers that join our system to every job manager while running each of the task assignment mechanisms. We focus in this section on the impact of (1) changing the helper arrival rate and (2) presence time heterogeneity. To avoid repetition, we will study the impact of the changing the average helper presence time in Section 5 using our implemented prototype.

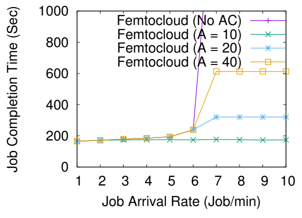
Impact of changing helper arrival rate: Figure 3 shows the impact of the helpers arrival rate on the job completion time for the **Pipeline Job Model** and the **General Parallel Path Model**. The figure shows that when the helper arrival rate is low, resource sharing and competition between jobs increases and thus the average job completion time increases as well. Figure 3(a) shows that, in case of the pipeline job model, our Femtocloud task assignment mechanism (Femtocloud-TA) outperforms CPOP due to its ability to take the helper's presence time model into account while assigning tasks to them. It also outperforms PTR due to Femtocloud-TA's ability to assign multiple jobs back to back. However, when the helper arrival rate is low, making the number of helpers present at any point in time low compared to the number of jobs, assigning a full path of tasks increases the probability of losing the whole path and re-executing it which leads to wasting significant compute resources.

Impact of helper presence time heterogeneity: To understand the impact of helper presence time heterogeneity, we add a new set of helpers to our helper set. These new helpers (Fast Helpers) have the capacity of 10.7 MFLOPS and their presence times are modeled as Normal(30 Sec, 10 Sec). We use a Poisson process to model the arrival of these helpers. Figure 4 shows the impact of changing the arrival rate of the fast helpers.

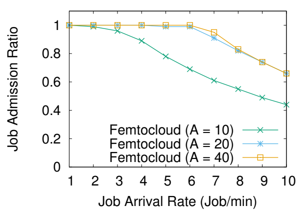
Figure 4(a) shows that, in case of the pipeline job model, Femtocloud-TA does not utilize the availability of the fast helpers due to the high risk of losing the job's computations due to the fast helper's churn. CPOP, however, did not take these helper's churn probability into account and assigned tasks to them. Such decision led to wastage of computation resources leading to an increase in the average job completion time. PTR, however, was able to utilize the additional capacity introduced by these helpers due to (1) its fine-grained per-task assignment mechanism, and (2) its ability to take the risk probability into account while assigning tasks to their executing helpers. Figure 4(b), however, demonstrates that the increased complexity of a job can result in its division into multiple stages allowing Femtocloud-TA to utilize the fast helpers to decrease average job completion time.

4.2.2 Admission Control Performance. We next study the impact of using our Femtocloud admission control mechanism. We compare the performance of using Femtocloud task assignment mechanism with and without using our admission control mechanism under different loads. In the following experiments, we disable (1) the checkpointing mechanism, and (2) the early task pick up mechanism.

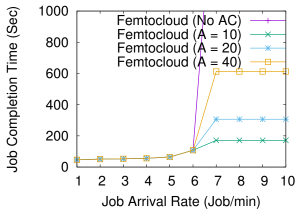
[3]We shed the light on the results from using the four job models in Section 5.



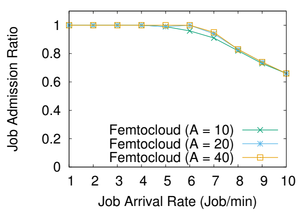
(a) Job Completion Time (Pipeline)



(b) Job Admission Ratio (Pipeline)

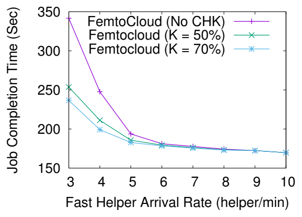


(c) Job Completion Time (G-Parallel)

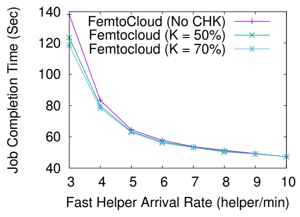


(d) Job Admission Ratio (G-Parallel)

Figure 5: Impact of job arrival rate on the performance of Femtocloud.

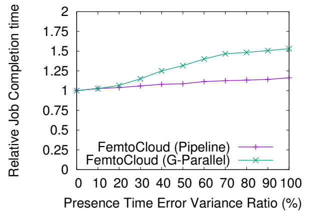


(a) Pipeline Jobs

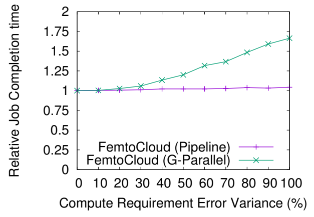


(b) General Parallel Jobs

Figure 6: Impact of changing the helper's presence time on the checkpointing performance.



(a) Presence Time Errors



(b) Task Requirement Errors

Figure 7: Sensitivity to estimation errors (presence time model parameters and task compute requirements.

Figure 5 shows the impact of changing the job arrival rate on the performance of Femtocloud with and without using admission control. While using our admission control mechanism, we experiment with the total relative work remaining maximum threshold ($\mathcal{A}_{\max}$) values of 10, 20, and 40. An increase in the job arrival rate causes the average job completion time to increase. Without admission control, we observe that the job completion time increases drastically once the number of jobs in the system exceeds its capacity. With admission control, job completion time can be limited to an acceptable value. It is important to carefully select the value of $\mathcal{A}_{\max}$. A low value of $\mathcal{A}_{\max}$ decreases the efficiency of the system and leads to the rejection of jobs that the helpers are capable of executing as shown in Figure 5(b) ($\mathcal{A}_{\max} = 10$).

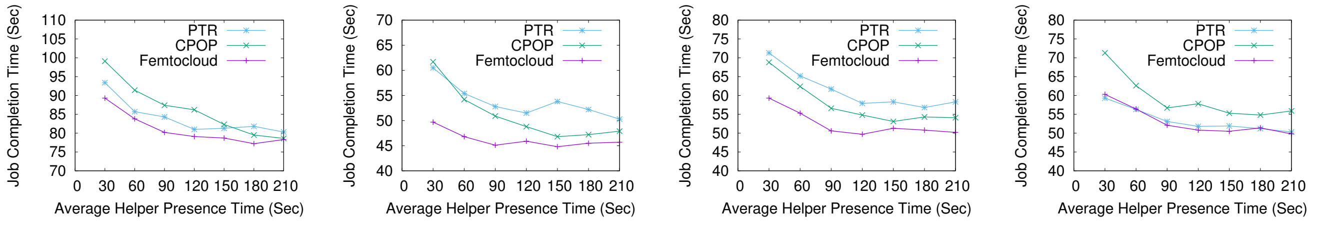
4.2.3 *Task Checkpointing Performance.* We compare the performance of task assignment in Femtocloud with and without using task checkpointing while disabling all the other mechanisms.

Figure 6 shows the impact of changing the presence time of helpers on the performance of Femtocloud with and without using our checkpointing mechanism. We compare various reliability thresholds, ($\mathcal{K}$), when checkpointing is used. The figure shows that when the average helper presence time is low, there is increased need for task checkpointing to avoid losing finished tasks, re-executing them, and increasing the overall job completion time. However, when the average presence time is large the probability of losing task results decreases and thus the checkpointing mechanism is not as important. Note that the checkpointing mechanism is more critical in case of the pipeline job model since the assigned path length increases.

4.2.4 *Sensitivity Analysis.* We analyze the sensitivity of our mechanisms to estimation errors. In this set of experiments, we enable all our mechanisms. We set the value of the reliability threshold ($\mathcal{K}$) for our checkpointing mechanism to be 70%. We analyze Femtocloud's sensitivity to estimation errors of (1) presence time model parameters and (2) task compute requirements. In both cases, our errors follow a Normal distribution with a mean of 0. We set the variance of the error distribution to be a percentage of the correct value and we vary this percentage from 0% to 100%. To show the error sensitivity, we report the ratio between the average job completion time with and without errors

Figure 7 (a) shows, for both the pipeline job model and the general parallel job model, the job completion time increases with the increase of the presence time error variance. We notice that the pipeline job model is relatively less sensitive to errors in estimating the presence time model parameters because the checkpointing mechanism maintains copies of the intermediate task results leading to efficient recovery from the churn when it occurs. The figure also shows that the general parallel job model can sustain up to 20% error variance with approximately 10% increase in the average job completion time.

Figure 7 (b) shows the impact of changing the variance of the compute requirement estimation error on the performance of Femtocloud. The figure demonstrates that the pipeline job model is insensitive to this type of error because the helper's fair queue management mechanism prevents a job task estimation error from influencing the performance of other jobs. The general parallel model, however, can accept up to 30% variance of the compute requirement estimation error without a significant increase in the job completion time.



(a) Pipeline Jobs (b) Parallel Jobs (c) General Parallel Jobs (d) Pyramid Jobs

Figure 8: Impact of helper's presence time on the performance.

5 PROTOTYPE IMPLEMENTATION AND EVALUATION

5.1 System Implementation

In this section, we present the implementation details of our Femtocloud prototype. We implement the Femtocloud controller logic as a python script that carries the responsibilities described in Section 2.1. We run this script on a local Linux machine. To emulate running on the cloud, we enforce an additional communication latency between the controller and the helpers of 168 ms, which is our measured average latency between a mobile device in Georgia Tech's enterprise network and Amazon Web Services in Europe (Frankfurt and Ireland).

We implement the Femtocloud helper as an Android application that allows users to enter their resource sharing policies. Based on these policies, the helper connects to the controller and shares the user profile accordingly. Upon joining the Femtocloud, the helper service estimates the mobile device capabilities and shares them with the controller. Additionally, while being used by a job manager, it shares with it the estimated fair share of resources that the job may receive from the helper. To ensure fair resource sharing between different jobs, it implements our mechanisms described in Section 3.3. It also carries the responsibility of estimating some contextual information as described in Section 2.2.

Our job manager is implemented as an Android service that can be assigned to the helper with the estimated longest presence time, or the job originator. The job manager's main responsibilities are described in Section 2.3. The state of the job manager is periodically replicated at the controller to enable recovery in case the device running the job manager functions opts out of the system.

Job originators are Android applications designed to generate jobs each of which consists of a set of tasks organized in directed-acyclic dependency graph. We emulate real tasks in the job by synthesizing a matrix multiplication task with the same input size, output size, and compute requirements of the target tasks. When a job manager assigns a task to a helper, the code is directly sent to the helper from the originator and is executed by the helper using the Java Reflection API.

In our prototype, we run the job originators inside Android x86 virtual machines. We configure the job originators to be willing to carry the job management responsibility. Therefore, once the controller accepts a job it starts the job manager service at the originator's VM.

5.2 Results

In this section, we present the results acquired using our prototype implementation. Due to the limitations imposed by the scale of our experiments, we modified our checkpointing mechanism such that a task is considered well-maintained if its results are available in at least two helpers. We also set the number of full jobs allowed by Femtocloud to be relatively high such that all the incoming jobs are admitted by the controller. Therefore, we only present the job completion time results in Figure 8.

Our helper set consists of 6 devices, a Galaxy S5, a Nexus 10[2013], 2 Nexus 7 [2013], and 2 Nexus 7 [2012]. All these helpers are connected to the same enterprise network through WiFi. We use the Normal distribution to model the helper's presence time. In our experiments, we change the mean of the helper presence time distribution from 30 to 210 seconds. Once a helper leaves, it returns after an OFF period that follows a Normal distribution with mean equals 25% of the presence time mean maintaining the helper's duty cycle to be 75% on average.

We use a Poisson arrival process to model the arrival of new jobs. The job arrival rate is set to be equal to 3 jobs per minute. Each of the generated jobs consists of 15 tasks. We set the number of parallel paths in the jobs that follow the **General Parallel Path Model** to be equal to 3. In our results, we show the average of 5 runs.

Figure 8 shows the impact of changing the average presence time of the helper on the job completion time of different types of jobs. It is clear from the plots that with the increase of the helper presence time the job completion time decreases for all the scheduling mechanisms. Also all job models, Femtocloud outperforms the CPOP task assignment mechanism due to its ability to control the risk associated with assigning tasks to helpers that may leave before completing them. The relative advantage of Femtocloud over CPOP decreases with the increase of the helper presence time and decrease of churn probability. The figure also reveals that Femtocloud outperforms the PTR assignment mechanism under all job models except the pyramid model. For both the parallel and the general parallel job model, Femtocloud outperforms the PTR assignment mechanism due to its ability to identify the job's bottleneck (critical path) and prioritize it while assigning tasks to their executing nodes. For the pipeline job model, however, Femtocloud slightly outperforms PTR due to its ability to send multiple tasks back to back to the helper instead of waiting for each task to finish in order to assign the next one. In case of the pyramid job model, the Femtocloud loses its back-to-back assignment edge over PTR since all the paths at each every stage will consist of only one tasks. This will lead to Femtocloud schedule to operate on a per-task basis.

6 DISCUSSION

In this section, we discuss two key questions associated the Femtocloud system: (1) How can the system provide users with incentives to share their mobile compute resources? (2) How can the system overcome the security challenges that will be introduced by malicious users (helpers/job originators). Although properly addressing those two questions are out of scope of our current work, they require some discussion.

6.1 User Incentives

We have witnessed increasing numbers in those willing to share their compute resources. For instance, BOINC projects demonstrate the willingness of millions of users to share the compute resources of their personal computers and mobile devices in support of scientific applications [1]. To achieve similar success, the Femtocloud system has to provide users with proper incentives to share their compute resources not only for scientific applications but also for a wide-range of edge computing applications.

To identify effective incentive mechanisms and assess users willingness to share their mobile compute resources while employing each of these mechanisms, we conducted a pilot user study. We surveyed approximately 50 students taking networking courses at Georgia Tech, at the undergraduate and graduate levels. Our survey consisted of two main sections. In the first section, we asked the students about the generic factors that might influence their decision when it comes to sharing their mobile compute resources (e.g., battery life, device type). These questions were intended to ensure the students' awareness to these factors prior to responding to the sharing-decision related questions. The latter section of the survey asked about their willingness to share in four specific scenarios: (1) in support of a for-profit company, (2) in support of gaming, (3) in support of science, and (4) in support of finding a lost child. In addition to the simple yes/no responses, students were asked to write additional comments, if needed.

Our pilot study reveals that individuals rationalize sharing their computational resources differently depending on who is utilizing the resources and why. All the students who were willing to share their resources with a for-profit business noted that they must receive some compensation (e.g., money). For the other three scenarios, the cause and the trustworthiness of the computation borrower were the driving factors behind their decision. The science scenario and the lost child scenario appealed to 76% and 83% of the demographic we surveyed, respectively. The gaming scenario, however, appealed to only 29% of the users surveyed.

The outcome of our pilot study suggests that people are willing to share their mobile compute resource if (1) they are getting compensated by the computation borrower, or (2) the cause of the computation is significant (e.g., common good, emergencies). Accordingly, a mix of these incentive mechanisms can be developed and integrated with our Femtocloud system to insure its adoption and future success.

6.2 Security and Privacy

In the Femtocloud system, ensuring the security and privacy of our helpers is critical for them to share their resources. Generally, helpers need to guarantee protection against malicious originators who may try to infringe on their privacy or compromise their security. Fortunately, using sandboxing allows the Femtocloud helper client service to control data access privileges and thus ensures helper data privacy [2, 15].

A Job originator must be protected against malicious helpers that may try to access the job's private data or provide incorrect results without executing the job. First, to protect against job-data leakage while running on untrusted resources, task execution over encrypted data was proposed [11, 14, 24, 36]. In these mechanisms, the originator encrypts the job's private data prior to submitting the job to the Femtocloud system. Upon the completion of the job, the Femtocloud system will return the results that only the job originator will be able to decrypt and understand. Second, to enable originators to verify the correctness of the results and and the work done by the helpers, cryptographically verifiable approaches can be used [13, 20]. These mechanisms enable the job originator to ensure the correctness of the results and verify that the tasks have been fully executed by the helpers. In the absence of these approaches, task replication over independent helpers can be used to detect inconsistencies and protect against malicious entities.

7 RELATED WORK

Cloud-Based Computational Offloading: Computational offloading has emerged as a result of the lack of computational resources in the early generations of smart-phones, the limited battery these devices operates on, and emerging trends of developing compute intensive applications [9]. Early developed computational offloading systems argued for offloading the heavy computations from the resource-constrained mobile devices to cloud, which has virtually unlimited compute capacity [6, 7, 16, 21, 31]. For instance, MAUI [7] and CloneCloud [6] primarily focused enabling code offloading without placing too much burden on the application developers. COSMOS [31], however, focused on building a cost-efficient computational offloading service on top of commercially-available, on-demand, elastic clouds. These systems demonstrated combined energy-savings and execution-speedups for applications with appropriate properties. In contrast, these systems also showed that the communication bandwidth and the latency between the mobile device and the cloud are the main performance bottlenecks for the cloud-based computational offloading systems.

Edge Computing: To minimize the network delays between the mobile devices and the computing servers, edge computing proposes bringing compute resources closer to their users. One early realization of edge computing was Cloudlets, which introduced a middle tier between mobile devices and traditional clouds [17, 28, 39]. Cisco's fog computing also shares the same vision of introducing a layer between data centers and end devices [4, 33, 34]. The end goal of both Cloudlets and fog computing is covering the edge with dedicated computing servers. Therefore, they share two key limitations: (1) They need capacity provisioning to ensure meeting users requirements at any point in time; and (2) They face a significant deployment barrier due to the potentially staggering cost of deploying their servers.

To reduce this deployment cost and overhead of edge computing services, recent efforts, including our own work on Femtoclouds [18] and Serendipity [32], proposed leveraging the resources of underutilized mobile devices at the edge [5, 10, 12, 18, 19, 23, 25, 25, 27, 32]. Femtoclouds, uniquely, propose taking advantage of clusters of devices that tend to be co-located in places such as public transit, classrooms, theaters, or coffee shops. These clusters can perform computations for jobs generated from within our outside of the cluster. We observed that these devices could operate as a cloudlet if managed properly from a nearby controller, especially if deployed in scenarios that suggest membership stability. Our work in this paper fills in an important gap in our earlier work by designing and evaluating a new architecture along with realistic workload management functions. We also consider a more general job model as well as realistic models for helper presence time.

Data Center Workload Management: We draw on earlier research in workload management in traditional cloud computing paradigms [8, 22, 35, 37, 38]. There are two fundamental differences in our environment that require innovation beyond existing research in workload management: (1) the fragmented and heterogeneous nature of the compute resources provided by mobile devices, and (2) the potentially significant churn in the availability of this compute resource.

8 CONCLUDING REMARKS

In this paper, we presented an enhanced architecture for the Femtocloud system in which we rely on mobile device clusters at the edge to provide the compute resources while moving the cluster control and management functionalities to the cloud. Within this new architecture, we developed a set of adaptive workload management mechanisms and algorithms that make the Femtocloud system efficient for real computation workloads, and reliable and scalable in the presence of device heterogeneity and churn. We implemented a small scale prototype of our system and use it to demonstrate the feasibility and efficiency of the system. We used simulations to further understand the gains imposed by each of our workload management techniques in larger scale systems.

In addition to workload management, research in Femtocloud systems needs to address several important issues before they are fully deployable. These include: (1) Implementing an effective user incentive mechanism for mobile devices to opt in as helpers; (2) Implementing efficient security protocols that detect malicious helpers and prevent them from participating in the system; and (3) how to enable large scale deployment where widely scattered helpers can collaborate to form more tightly knit compute resources.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful feedback. This work was supported in part by the US National Science Foundation through grant NETS 1409589 and the Qatar National Research Fund, a member of Qatar Foundation through grant 8-1645-1-289.

REFERENCES

[1] BOINC: Open-source software for volunteer computing. https://boinc.berkeley.edu/. (????). Online; accessed 23-April-2017.

[2] Paul A Ashley, Anthony M Butler, Ghada M ELKeissi, and Leny Veliyathuparambil. 2017. Dynamic security sandboxing based on intruder intent. (2017). US Patent 9,535,731.

[3] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. 2002. The case for cyber foraging. In *ACM EW*.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog computing and its role in the internet of things. In *SIGCOMM MCC*.

[5] M. Chen, Y. Hao, Y. Li, C. Lai, and D. Wu. 2015. On the computation offloading at ad hoc cloudlet: architecture and service modes. *IEEE Communications Magazine* (2015).

[6] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. 2011. CloneCloud: elastic execution between mobile device and cloud. In *ACM EuroSys*.

[7] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. 2010. MAUI: making smartphones last longer with code offload. In *ACM MobiSys*.

[8] C. Devi and R. Uthariaraj. 2016. Load Balancing in Cloud Computing Environment Using Improved Weighted Round Robin Algorithm for Nonpreemptive Dependent Tasks. *The Scientific World Journal* (2016).

[9] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. 2013. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing* (2013).

[10] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan. 2013. The case for mobile edge-clouds. In *IEEE UIC/ATC*.

[11] Zhangjie Fu, Kui Ren, Jiangang Shu, Xingming Sun, and Fengxiao Huang. 2016. Enabling personalized search over encrypted outsourced data with efficiency improvement. *IEEE TPDS* (2016).

[12] Hend Gedawy, Sannan Tariq, Abderrahmen Mtibaa, and Khaled Harras. 2016. Cumulus: A distributed and flexible computing testbed for edge cloud computational offloading. In *Cloudification of the Internet of Things (CIoT)*. IEEE.

[13] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*. Springer.

[14] Craig Gentry and others. 2009. Fully homomorphic encryption using ideal lattices.. In *STOC*.

[15] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2015. Rethinking security of Web-based system applications. In *ACM WWW*.

[16] M. S Gordon, D. Jamshidi, S. A Mahlke, M. Mao, and X. Chen. 2012. COMET: Code Offload by Migrating Execution Transparently.. In *OSDI*.

[17] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. 2013. Just-in-time provisioning for cyber foraging. In *ACM MobiSys*.

[18] K. Habak, M. Ammar, K. A Harras, and E. Zegura. 2015. Femtoclouds: Leveraging mobile devices to provide cloud service at the edge. In *IEEE CLOUD*.

[19] Karim Habak, Cong Shi, Ellen W Zegura, Khaled A Harras, and Mostafa Ammar. 2017. Elastic Mobile Device Clouds: Leveraging Mobile Devices to Provide Cloud Computing Services at the Edge. *Fog for 5G and IoT* (2017).

[20] Ahmed E Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F Sayed, Elaine Shi, and Nikos Triandopoulos. 2014. TRUESET: Faster Verifiable Set Computations.. In *USENIX Security*.

[21] K. Kumar and Y. Lu. 2010. Cloud computing for mobile users: Can offloading computation save energy? *IEEE Computer* (2010).

[22] Y. Kwok and I. Ahmad. 1996. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE TPDS* (1996).

[23] Abderrahmen Mtibaa, Khaled A. Harras, and Afnan Fahim. 2013. Towards Computational Offloading in Mobile Device Clouds. In *IEEE CloudCom*.

[24] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can homomorphic encryption be practical?. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM.

[25] T. Nishio, R. Shinkuma, T. Takahashi, and N. B. Mandayam. 2013. Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. In *MobileCloud*. ACM.

[26] Maria Alejandra Rodriguez and Rajkumar Buyya. 2016. A taxonomy and survey on scheduling algorithms for scientific workflows in IaaS cloud computing environments. *Concurrency and Computation: Practice and Experience* (2016).

[27] Ahmed Saeed, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. 2015. Vision: The case for symbiosis in the internet of things. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*. ACM.

[28] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* (2009).

[29] John A Sharp. 1992. *Data flow computing: theory and practice*. Intellect Books.

[30] William F Sharpe, Gordon J Alexander, and Jeffery V Bailey. 1999. *Investments*. Prentice-Hall Upper Saddle River, NJ.

[31] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura. 2014. Cosmos: computation offloading as a service for mobile devices. In *ACM MobiHoc*.

[32] C. Shi, Va. Lakafosis, M. H. Ammar, and E. W. Zegura. 2012. Serendipity: enabling remote computing among intermittently connected mobile devices. In *ACM MobiHoc*.

[33] I. Stojmenovic. 2014. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *IEEE ATNAC*.

[34] I. Stojmenovic and S. Wen. 2014. The fog computing paradigm: Scenarios and security issues. In *IEEE FedCSIS*.

[35] H. Topcuoglu, S. Hariri, and M. Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS* (2002).

[36] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*.

[37] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and others. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM.

[38] F. Wu, Q. Wu, and Y. Tan. 2015. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing* (2015).

[39] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee. 2015. The design and implementation of a wireless video surveillance system. In *MobiCom*.