Transient Clouds: Assignment and Collaborative Execution of Tasks on Mobile Devices

Terry Penner, Alison Johnson, Brandon Van Slyke, Mina Guirguis, Qijun Gu
Texas State University
San Marcos, TX 78666
{t_p68,abj25,brandon.vanslyke,msg,qijun}@txstate.edu

***Abstract*—Mobile devices are evolving into powerful systems due to recent advances in their communication, storage and computation technologies. They are poised to play a key role in providing a rich collaborative computing platform for various applications. This paper proposes "Transient Clouds" – a collaborative computing platform that allows nearby devices to form an ad-hoc network and provide various capabilities as cloud services. Transient Clouds utilize the collective capabilities of the devices present, along with their social and context awareness that cannot be provided efficiently by the traditional clouds. We present a modified algorithm of the Hungarian method for assigning tasks to devices in order to achieve various goals (e.g., load balancing, collocating executions, etc...). We evaluate the performance of our proposed algorithms through simulation and provide a real implementation on the Android platform using the Wi-Fi Direct framework. We envision Transient Clouds to be utilized in temporal scenarios in which the cloud is created on-the-fly by the devices present in an environment and would disappear as the devices leave the network.**

I. INTRODUCTION

Despite recent technological advances in mobile devices, a single device has limitations – in terms of its computation and storage capabilities. This has promoted a lot of research efforts in the areas of cloud computing and mobile cloud computing in which computation and storage are offloaded to distant clouds [1]–[5]. Offloading computation and providing storage services, however, require Internet connections which may not be available in some situations (e.g., disaster impacted sites). Thus, the challenge remains of how mobile devices can rely on their collaborative efforts to provide computational services.

Mobile devices vary greatly between each other – some have powerful processors, some have long lasting batteries, while others have different types of sensors (e.g., cameras, GPS, etc.). Currently, a device cannot run an application that requires a particular capability that is not supported by the software/hardware of the device. Therefore another challenge is how to create platforms in which a device can exploit the capabilities offered by nearby devices to execute tasks that cannot be executed locally on the requesting device.

To address the above challenges, this paper presents "Transient Clouds" – a collaborative computing platform that allows nearby devices to form an ad-hoc network and provide various capabilities as a cloud service. Through utilizing the collective power of the group, devices become no longer constrained by their local software and hardware capabilities. Moreover, in many applications, the devices present in an environment posses social and context awareness that are harder to exploit with distant clouds due to the delay in accessing traditional cloud services. Transient Clouds resemble peer-to-peer networks and must cope with the dynamic nature of devices joining and leaving the network for distributing the load. Their existence is closely related to their context so when all the devices leave the network, the Transient Cloud disappears.

Transient Clouds are poised to play a key role in the future due to the increase in number of mobile devices coupled with the increase in computation demands by new applications. As devices get more powerful, their battery life will shrink; battery technology is not growing as quickly as computation consumption rates [6]. Through enabling collaborative computing among devices, the load on each device decreases and the lifetime of the network gets extended. Moreover, since devices rely on direct communication between the nodes, they do not require cellular or access points to communicate.

To enable Transient Clouds, assignment algorithms are needed to assign tasks to devices for execution based on their capabilities. Task assignment is a well studied problem with many proposed algorithms [7]–[9]. In such algorithms, each device submits a cost value for each task it can execute (if the device implements this capability). The goal of an assignment algorithm is to find the minimum total cost assignment. The main issue with this approach is that devices do not have the ability to choose costs corresponding to what they *actually* get assigned. To illustrate this point, consider a device that can execute 3 tasks based on its capabilities. The device should have the ability to inflate the costs of tasks 2 or 3 by some factor, if task 1 has been assigned to it. If it declares higher costs for all of the 3 tasks, it may not get any task assigned to it. In Transient Clouds we seek to provide the devices with the ability to balance the assignments among them, so we propose to *dynamically adjust the costs as assignments are being made*.

Contributions: This paper makes the following contributions:

1) We present a modified version of the Hungarian method [7] for task assignment that adjusts the costs dynamically based on previous assignments. This can be used to provide various properties such as load balancing and collocating task executions.
2) We evaluate the effect of task assignments under different arrival and departure distribution models using a discrete-time event simulation.

3) We assess the feasibility of Transient Clouds through our developed prototype implementation on Android using today's new technology (e.g., Wi-Fi Direct).

Paper organization: In Section II we put this work in context with other related work. In Section III we discuss the foundations of Transient Clouds, models, and our task assignment algorithm. Our proposed platform is evaluated in Section IV through simulations and implementation. We conclude the paper in Section V.

II. Related Works

There has been much research work that has focused on mobile cloud computing in which offloading computation to external servers is sought to reduce the load on the devices. COMET [1], MAUI [2], DPartner [3], COFA [4], and Cuckoo [5], among others, offload computation from a mobile device to a remote server in the public cloud to achieve various performance properties. The above works, however, rely on a functioning Internet connection to perform the offloading – something that may be not be possible in many scenarios.

Mobile Clouds, on the other hand, do not rely on public clouds because the computation is performed on the mobile devices. In their paper on mClouds [6], Miluzzo et al. discuss the theory behind the concept as well as what possible incentives might be for getting users to connect their devices to the cloud. Transient Clouds go beyond the concept of Mobile Clouds by providing a working prototype that fills in the details about how the cloud services are assigned to devices. Another work related to Transient Clouds is the OSGi system [10]. The OSGi system features remote installation and execution of code on other devices; the difference is that Transient Clouds do not need to install any code on the devices. Our framework simply sends pre-compiled code to another device and has it execute only the exact method that we desire, without any installation.

Other works in the area includes Apache Hadoop [11] and sensor clouds [12]. Hadoop is a system for distributing computation across multiple machines. Hadoop, however, is designed for a well-known network of servers and server racks, while our framework deals with a heterogenous set of mobile devices with individual devices leaving and joining the network at various intervals. Sensor clouds, on the other hand, aim to virtualize the sensors available for various applications. Transient Clouds support various types of sensors with different capabilities (e.g., sensors, smart-phones, cameras, robots, etc...) and must deal with the dynamic nature of their presence. Moreover the assignment algorithm is capable of load balancing and collocating task executions.

III. Methodology

A. The Model

We envision Transient Clouds according to the following model. Let $\mathcal{D}$ denote the whole set of devices such that $\mathcal{D} = \{d_i | 0 \leq i < N\}$, where N is the total number devices. Let $\mathcal{C}$ denote the set of capabilities offered by mobile devices such that $\mathcal{C} = \{c_i | 0 \leq i < M\}$, where M is the total number of capabilities. Each device, d_i supports a subset of the capabilities, $C(d_i)$, such that $C(d_i) = \{c_j | c_j \ is \ offered \ by \ device \ d_i\}$. Thus, the whole system can be represented by a two-dimensional matrix, P of 1's and 0's with rows corresponding to devices and columns to capabilities. An element p_{ij} in P is $\mathbb{1}\{c_j \in C(d_i), 0 \leq i < N, 0 \leq j < M\}$, where $\mathbb{1}\{\}$ is an indicator function. Let $d(t)$ denote the set of devices present in the system at time t, where $d(t) \subseteq \mathcal{D}$. Figure 1a shows an example of the P matrix, where d_0 can offer all capabilities, d_1 can offer capability c_0, and so on.

Given a requested service that is composed of a subset of capabilities to be supported by the Transient Cloud, we need an assignment algorithm that would assign each requested capability to a device. In general, the assignment can be done in a centralized or decentralized approach. In this paper we focus on a centralized approach since our implementation prototype of Transient Clouds relies on the Wi-Fi Direct framework which assumes the presence of a mobile device that acts as a group leader.

B. The Assignment Algorithm

As mobile devices join and leave the network, the Transient Cloud must cope with the varying set of capabilities offered by the devices present. Thus it is important to come up with an efficient assignment algorithm.

Due to the heterogeneity of mobile devices, some capabilities will be very common among devices (such as "has a processor"), while others will be very rare (such as "can measure temperature"). Because of this, we want to make sure that no devices are assigned too many tasks. Consider a network where there are 10 devices but only 1 of them has the capability to measure temperatures. We should not assign both processor-intensive and temperature-related tasks to that one device. Any of the other 9 devices in the network could be assigned the processor-intensive task, which would help lighten the load on the device that is capable of measuring temperatures. Situations like this are what require an assignment algorithm that assigns no device more tasks than absolutely necessary.

In some situations, however, it may make more sense to collocate tasks together on a single device, since this may reduce the delay in getting the results. Consider again the example above. If the measured temperature needs to be processed through a computation, then it would make sense to collocate that part of the computation with the one that takes the measurement on the same device.

Note that the assignment algorithm must be able to return a device for every capability requested – as long as such a device is available. The assignment need not involve every single device, but it must achieve certain properties such as load balancing or collocating capabilities across devices.

The Hungarian method [7] is one of the methods used in assigning tasks to workers. Consider a 2-dimensional square matrix, Q, in which the rows represent the workers and the columns represent the tasks. An entry q_{ij}, in matrix Q, denotes the cost of assigning task j to worker i. The Hungarian method achieves an optimal assignment in which the aggregate cost is minimized whereby every task must be assigned to only one device. Its running complexity is $\mathcal{O}(n^3)$, where n is the number of tasks/workers. Since with our setup, device d_i may offer $C(d_i)$ capabilities, we may want to assign multiple tasks to the same device. Also, unlike the Hungarian method, a device might not be assigned any task. Furthermore, we need a way to load balance and/or collocate tasks across devices. Thus, we propose an online modification to the Hungarian method to make it flexible for Transient Clouds.

Devices	Capabilities 0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	0	0	0	0	0
2	0	1	0	1	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	1	0
6	1	0	1	1	1	0

(a) Compressed capability table

Devices with Available Capabilities	Capabilities 0	1	2	3	4	5
0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0
0	0	0	**1**	0	0	0
0	0	0	0	**1**	0	0
0	0	0	0	0	**1**	0
0	0	0	0	0	0	**1**
1	1	0	0	0	0	0
2	0	1	0	0	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	1	0
6	1	0	0	0	0	0
6	0	0	1	0	0	0
6	0	0	0	1	0	0
6	0	0	0	0	1	0

(b) Standard selection

Devices with Available Capabilities	Capabilities 0	1	2	3	4	5
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	**1**
1	**1**	0	0	0	0	0
2	0	**1**	0	0	0	0
2	0	0	0	1	0	0
3	0	0	0	**1**	0	0
4	0	0	0	0	**1**	0
5	0	0	0	0	1	0
6	1	0	0	0	0	0
6	0	0	**1**	0	0	0
6	0	0	0	1	0	0
6	0	0	0	0	1	0

(c) Modified selection

Fig. 1. Standard vs. modified Hungarian method selection

In our proposed modification of the Hungarian method, we split each device into multiple rows; one row for each capability as shown in Figure 1b. The columns show $\{c_j | 0 \leq j < M\}$ and the rows show each capability in $\{d_i | 0 \leq i < N\}$ for each device separately. Running the Hungarian method directly on this matrix form would not solve the issues presented above. For example, we may get a selection – like the one presented in Figure 1b – where d_0 is overloaded by having to execute all capabilities. In order to fix this imbalance, we modified the algorithm so that each time we select a capability from a device we would *inflate* (e.g., by doubling) the cost of other capabilities in $C(d_i)$ (referred to as siblings) for the same device. This will reduce their chance of being selected. If we would like to collocate capabilities on the same device, we *deflate* the costs of the siblings. Figure 1c shows the assignment made with our approach when a load-balanced assignment is requested and thus costs are inflated (lines 5 and 11). One can observe that every device has a maximum

```
1: Transpose matrix
2: Subtract minimum from each row
3: Subtract minimum from each column
4: Mark first uncovered 0 element from each row as starred
5: Double and add 1 to the cost of any siblings
6: Cover each column with a starred 0
7: **if** Not all capabilities are covered **then**
8:     Find uncovered 0
9:     **if** 0 found **then**
10:        Mark found 0 as primed
11:        Double and add 1 to the cost of any siblings
12:        Look for starred 0 in same row
13:        **if** Starred 0 found **then**
14:            Cover row, uncover starred 0 column
15:            GOTO Line 8
16:        **else**
17:            Find starred element in same device columns
18:            **while** Element has been found **do**
19:                Find primed element in same row
20:                Find starred element in same device columns
21:            **end while**
22:            Un-star all found starred elements
23:            Star all found primed elements
24:            Erase all other prime marks and reset covers
25:            GOTO Line 4
26:        **end if**
27:    **else**
28:        Find smallest uncovered element
29:        **if** Smallest value is Infinity **then**
30:            Return starred 0 elements
31:        **else**
32:            Add to every element in a covered row
33:            Subtract from every element in uncovered column
34:            GOTO Line 8
35:        **end if**
36:    **end if**
37: **else**
38:    Return starred 0 elements
39: **end if**
```

Fig. 2. The assignment algorithm

of one task assigned to it.

The psuedocode for our proposed algorithm – that load balances the assignments – is shown in Figure 2. The input that the algorithm expects is a matrix like that shown in Figure 1a. It is important to note that a p_{ij} of 0 in our case corresponds to ∞ in the traditional Hungarian method.

To explain some of the terms used in the algorithm outlined in Figure 2, a *covered* row or column simply means that we have already made a selection in it so it is unavailable to select more from, a *starred* element is one that is in our current selection (although it may be replaced later), and a *primed* element is one that is marked as an option for changing our current selection to. Lines 5 and 11 are the ones that achieve load-balancing and can be modified for collocating tasks on devices. Since all devices start off with a cost of 1 for all of their capabilities, when we execute lines 2 and 3 the entire matrix will be reduced to 0's and ∞'s. That is why when we double the cost we also add 1; it is the only way we are guaranteed to have a larger cost than any of the other devices. Since all of the costs start out the same, they all increase to the same values as they are selected. This leads to the property that if devices d_a and d_b ($0 \leq a, b < N$) have the same number of capabilities chosen, then the elements in $C(d_a)$ and $C(d_b)$ will all have the same cost as each other. This property ensures that the workload will be balanced fairly across all devices.

IV. Evaluation

A. Android Prototype

We developed a prototype of Transient Clouds on Android for testing its feasibility. There have been other implementations for partitioning and offloading Android applications, but all of them required users to modify their devices in some way, either by editing core components of the Android system (such as the Dalvik VM [1] or Java language itself [3]) or by requiring users to obtain root access on their device. We wanted to show that it is possible to perform these tasks without modifying the existing system in order to make this technology more accessible.

We built our Transient Clouds on the underlying ad-hoc network. As of Android 4.0 (Ice Cream Sandwich, API 14), the Wi-Fi Direct framework has been built into the operating system [13]. Wi-Fi Direct is a technology that allows devices to create an ad-hoc network and connect to each other using standard Java sockets. In Wi-Fi Direct, only one device (called the Group Owner) acts as a router, and all of the other peer devices that connect to it create a single network. However, this limits the size of our network to be a maximum of one-hop from the Group Owner. Because we are unable to create an ideal multi-hop ad-hoc network, we decided to embrace this limitation and modify our network scheme to fit in with Android's Wi-Fi Direct implementation. We took advantage of the fact that every device must connect to the Group Owner and decided to make it the hub device of the network that is responsible for maintaining the network's state.

In Wi-Fi Direct, none of the devices can see the IP addresses of any other devices, with the exception that all of the devices can see the IP address of the Group Owner. However, it is possible to see the MAC Address of every device that is nearby. To work with this, we wrote our network protocol so that when a device connects to the Group Owner, the Owner saves the connecting device's IP address in a data structure along with its MAC address and the tasks that it is capable of executing. This way, when we want to execute a certain task, we simply have to link from the task number to the MAC address to the IP address, and then open a socket to the device. Therefore, the network is designed as a forwarding service where all of the peer devices can execute tasks and the Group Owner simply acts as an intermediary, forwarding tasks from the requester to the executor.

To enable sending code from a device to another, we used the Reflection library. Reflection allows for Java programs to load, explore, and execute compiled classes at runtime with no prior knowledge of them. All it needs is both the class' and method's names and any parameters to pass to the method as an array of *Object*s. It can then load the class from the **.class** file as a generic *Class<?>* type and create a new instance of it. The desired method can be captured from the generic as a *Method* type and the class instance can be used to invoke the desired method with the given parameters.

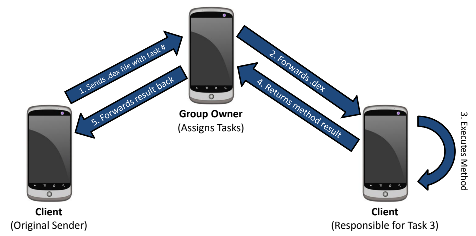


Fig. 3. Illustration of code distribution in Transient Clouds

Android uses its own modified VM called Dalvik. Instead of using **.class** files like the JVM, Dalvik uses **.dex** files to store its compiled classes. Fortunately, when an app is created, a **.dex** file with all of the classes used in the app is automatically generated and easily accessible.

To send the code, we place the auto-generated **.dex** file containing all of the compiled code of a task into the app's resources folder. From there it is sent as a file stream through a socket to the Group Owner, along with the task number, the names of the class and method to execute, and the method parameters. Based on the task number, the Group Owner decides which device to forward the task to, and sends it all of the information that it has received. The device that receives the **.dex** file then uses the Reflection library to extract and execute the desired method. It captures the result of the method in an *Object* type and returns the result to the Group Owner, who then forwards it on to the peer device that requested the task. Figure 3 illustrates this process.

To assess the performance of distributing tasks over a Transient Cloud, we created an Android application prototype as a proof-of-concept. This was run on several devices connected to each other using the Wi-Fi Direct framework. In this setup, teh Group Owner would issue commands for computations to it's connected peers. With each command, the group owner sent parameters for the computation, along with an arbitrary amount of data to simulate sending different sizes of bytecode payloads (e.g., dex files).

The setup composed of two Google Nexus 7's with 1.2 GHz quad-core CPUs and one HTC One with a 1.7 GHz quad-core CPU. One of the Nexus devices acted as the group owner and issued commands to the other two devices. The commands specified one of two tasks to be run on the peer devices, (1) a CPU-intensive task running a loop of a size specified by the command issuer and (2) a memory-intensive task that filled a specified amount of memory with numbers and reversed the range of memory.

In our experiments, we varied (1) the size of the loop ran in task 1, (2) the size of the memory block written in task

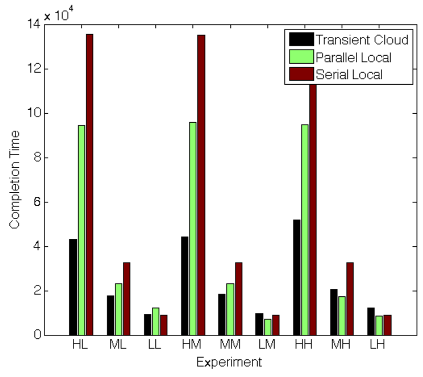


Fig. 4. Comparing completion times between Transient Clouds, parallel and serial executions

2, and (3) the size of the data chunk sent along with the tasks. We measured the time elapsed between the group owner sending the commands to the peers and receiving all of the results of their computations. We then tested the performance of running each of these trials locally on the group owner, with the same parameters for computation size and payload size, running both of the tasks serially and recording the time elapsed. We then repeated each of the trials locally again, this time running the two tasks in parallel with each other.

Figure 4 shows a set of results obtained under different experiments in which we vary the computational aspect between Low (L), Medium (M) and High (H) and the data transfer size between Low (L), Medium (M) and High (H), respectively. For instance, experiment 'HL' means the computation was high and the transfer size was low. The Y-axis represents the completion time. The results are averaged over 5 independent runs. One can observe that when the ratio of computation versus transfer size is high, Transient Clouds are more efficient than local execution and vice versa.

B. Impact of dynamics on Transient Clouds

In order to test our algorithm, we would need a massive number of devices to create an accurate network. Due to constraints on our resources, we developed an idealized version of a Transient Cloud in simulation. We created it primarily for the purpose of seeing how a Transient Cloud is likely to act over its lifetime and if the cloud could be stable enough to be useful. To measure stability, we wanted the simulation to tell us whether or not all of the capabilities in the network are covered by some device. If every capability has some device assigned to it, then we say that the network has one *unique complete coverage*. If we remove the chosen tasks and, after reassigning tasks to devices, we find another complete coverage, then we say that the network has two unique complete coverages. We can repeat this process until we find the total number of unique complete coverages. Providing replicas of the Transient Cloud is important in order to deal with the dynamic nature of devices joining and leaving the network.

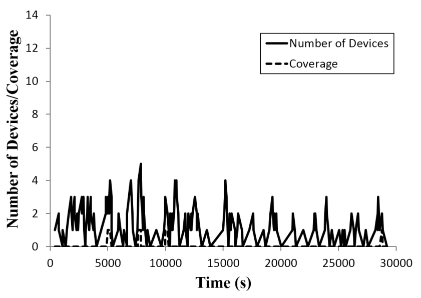
To study the performance of a Transient Cloud, we need to understand the following: (i) how the arrival and departure rates and distributions affect the number of coverages, (ii) on average how many devices must be in the network in order for it to have at least one unique complete coverage, and (iii) the percentage of time that the network has at least one unique complete coverage.

In our simulation experiments, each device is given an arrival time and a departure time as well as its $C(d_a)$ that is generated at random, where each capability has a different probability of occurring. This is to account for the various frequencies of different capabilities being present in a device. The cloud service is composed of a representation of all the unique capabilities.

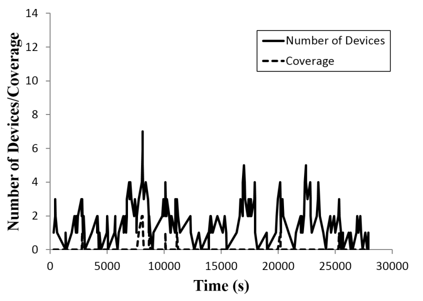
We varied two metrics as we generated our devices' arrival and departure times: the rate at which they join the network, and the distribution model for their duration in the network (e.g., on period). We held the device arrival distribution constant as a Poisson distribution, because we were assuming that users would be joining the cloud at a fairly regular interval. Likewise, we kept 5 minutes as the average amount of time that a device would stay in the network in all our tests so that all of our results can be compared in a consistent manner. The three device departure distributions that we tested were: uniform, exponential, and pareto. For each distribution that we tested, we varied the arrival rate to be: 1 device arriving every 10 minutes, 1 every 7 minutes, 1 every 5 minutes, 1 every 2 minutes, and 1 every minute.

The simplest departure distribution that we tested was a uniformly random distribution. For this, we simply generated a pseudo-random number between 100 and 500 seconds, resulting in an average time of 5 minutes in the network. A sample run is shown in Figure 5a. Next, we tested an exponential distribution because we wanted to know what the network would look like if people exited with the same distribution that they joined with. Recall that Poisson arrivals imply exponential inter-arrival times. A sample run from this test is shown in Figure 5b. Finally, we tested a Pareto distribution with an exponent of 0.83. This is what is known as a *heavy-tailed* distribution, because most devices leave quickly, but there are a few that remain for a very long time. This distribution is often observed in peer-to-peer networks [14], which we followed here. A sample run can be seen in Figure 5c. We repeated every experiment four times and took the average of their results.

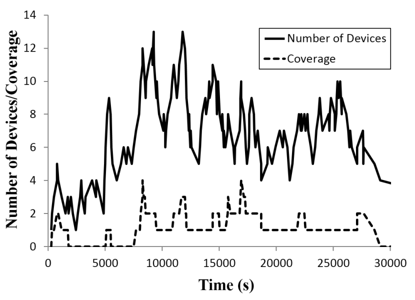
After running all of our experiments, we created the histograms in Figure 6 showing the relative length of time that the network spent having a total of n unique complete coverages. These histograms show that if devices are arriving and departing with the same distribution, they must arrive at a much higher rate than they depart if the Transient Cloud is to be useful. Figure 6c on the other hand, shows us what we believe to be the more likely scenario. This is the heavy-tailed Pareto distribution, which is seen in peer-to-peer networks. We believe that a Transient Cloud will behave very similarly to a peer-to-peer network, with a small number of users founding the network and remaining in it for almost the entire duration, while most users will join for a short time, and then leave again. As you can see in the chart, if devices enter the network with the same frequency as the average departure, the network has at least one coverage just over 70% of the time, making



(a) Uniform random distribution

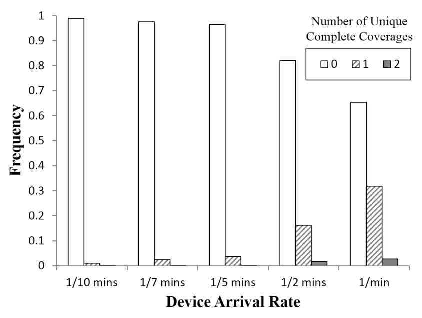


(b) Exponential distribution

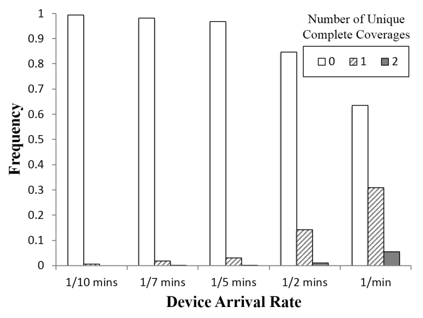


(c) Pareto distribution

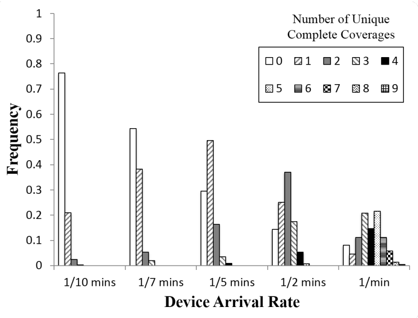
Fig. 5. Examples of network state over time (1 device arriving every 5 minutes).



(a) Uniform random distribution



(b) Exponential distribution



(c) Pareto distribution

Fig. 6. Histograms of network coverage with varying arrival rates.

it quite viable. If arrivals occur just over twice as fast as the departures do, then the network is actually very stable, with about 85% of the time spent having at least one coverage and nearly 40% of the time spent with two unique complete coverages.

V. Conclusion

In this paper we have presented Transient Clouds – a collaborative computing platform that allows devices to create an ad-hoc network and provide their capabilities as a cloud service. Tasks are assigned to devices based on a modified version of the Hungarian method that can achieve certain assignment properties such as load-balancing or collocating tasks. We have evaluated Transient Clouds using simulation and implementation. We have discovered that the current technology allows for a partial implementation of Transient Clouds within the confines of stock Android. This partial implementation is more than enough to show off the potential of the technology, and hopefully will be able to garner more interest and support from developers to have these features built in as a fundamental part of the operating system.

Acknowledgement

This research is funded by NSF REU award #1156712 that is co-funded by the Department of Defence and by NSF CNS award #1149397.

References

[1] M. Gordon, D. Jamshidi, S. Mahlke, Z. Mao, and X. Chen, “COMET: Code Offload by Migrating Execution Transparently,” in *Proceedings of OSDI*, Hollywood, CA, October 2012.
[2] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making Smartphones Last Longer With Code Offload,” in *Proc. of the International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, June 2010.
[3] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, “Refactoring Android Java Code For On-Demand Computation Offloading,” *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, vol. 47, no. 10, pp. 233–248, 2012.
[4] D. Shivarudrappa, M. Chen, and S. Bharadwaj, “COFA : Automatic and Dynamic Code Offload for Android,” Boulder, CO, USA, 2011.
[5] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, “Cuckoo: A Computation Offloading Framework for Smartphones,” in *Mobile Computing, Applications, and Services*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, M. Gris and G. Yang, Eds., 2012, vol. 76, pp. 59–79.
[6] E. Miluzzo, R. Cáceres, and Y. Chen, “Vision: mClouds - Computing On Clouds of Mobile Devices,” in *Proceedings of the third ACM workshop on Mobile cloud computing and services*, UK, June 2012.
[7] H. Kuhn and B. Yaw, “The Hungarian Method for the Assignment Problem,” *Naval Research Logistics Quarterly*, pp. 83–97, 1955.
[8] D. P. Bertsekas, “The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem,” *Annals of operations research*, vol. 14, no. 1, pp. 105–123, 1988.
[9] R. Cohen, L. Katzir, and D. Raz, “An efficient approximation for the generalized assignment problem,” *Information Processing Letters*, vol. 100, no. 4, pp. 162–166, 2006.
[10] OSGi Alliance Staff, “OSGi Alliance,” http://www.osgi.org/Main/HomePage, 2013.
[11] The Apache Software Foundation Staff, “Apache Hadoop!” http://hadoop.apache.org/, 2013.
[12] M. Yuriyama and T. Kushida, “Sensor-Cloud Infrastructure-Physical Sensor Management with Virtualized Sensors on Cloud Computing,” in *Proceedings of NBiS*, Takayama, Gifu, Japan, Septemeber 2010.
[13] Google Android Staff, “Android 4.0 APIs,” http://developer.android.com/about/versions/android-4.0.html, 2013.
[14] J. Li, J. Stribling, R. Morris, and M. Kaashoek, “Bandwidth-Efficient Management of DHT Routing Tables,” in *Proceedings of NSDI*, Boston, MA, May 2005.