Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices

Cong Shi*, Vasileios Lakafosis†, Mostafa H. Ammar*, Ellen W. Zegura*
*School of Computer Science
Georgia Institute of Technology
{cshi7, ammar, ewz}@cc.gatech.edu
†School of Electrical and Computer Engineering
Georgia Institute of Technology
vasileios@gatech.edu

ABSTRACT

Mobile devices are increasingly being relied on for services that go beyond simple connectivity and require more complex processing. Fortunately, a mobile device encounters, possibly intermittently, many entities capable of lending it computational resources. At one extreme is the traditional cloud-computing context where a mobile device is connected to remote cloud resources maintained by a service provider with which it has an established relationship. In this paper we consider the other extreme, where a mobile device's contacts are only with other mobile devices, where both the computation initiator and the remote computational resources are mobile, and where intermittent connectivity among these entities is the norm. We present the design and implementation of a system, Serendipity, that enables a mobile computation initiator to use remote computational resources available in other mobile systems in its environment to speedup computing and conserve energy. We propose a simple but powerful job structure that is suitable for such a system. Serendipity relies on the collaboration among mobile devices for task allocation and task progress monitoring functions. We develop algorithms that are designed to disseminate tasks among mobile devices by accounting for the specific properties of the available connectivity. We also undertake an extensive evaluation of our system, including experience with a prototype, that demonstrates Serendipity's performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems —*Distributed applications*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Remote Computing, Task Allocation, Mobile Devices, Opportunistic Networks, Energy Management

1 Introduction

Recent years have seen a significant rise in the sophistication of mobile computing applications. Mobile devices are increasingly being relied on for a number of services that go beyond simple connectivity and require more complex processing. These include pattern recognition to aid in identifying snippets of audio or recognizing images whether locally captured or remotely acquired, reality augmentation to enhance our daily lives, collaborative applications that enhance distributed decision making and planning and coordination, potentially in real-time. Additionally, there is potential for mobile devices to enable more potent "citizen science" applications that can help in a range of applications from understanding how ecosystems are responding to climate change[1] to gathering of real-time traffic information.[2]

Mobile applications have become an indispensable part of everyday life. This has been made possible by two trends. First, truly portable mobile devices, such as smartphones and tablets, are increasingly capable devices with processing and storage capabilities that make significant step improvements with every generation. While power in mobile devices will continue to be constrained relative to tethered devices, advances in battery and power management technology will enable mobile devices to manage longer-lived computations with less burden on available power [22]. A second trend that is directly relevant to our work is the availability of improved connectivity options for mobile devices. These have enabled applications that transcend an individual device's capabilities by making use of remote processing and storage.

Fortunately, a mobile device often encounters, possibly intermittently, many entities capable of lending it computational resources. This environment provides a spectrum of computational contexts for remote computation in a mobile environment. An ultimately successful system will need to have the flexibility to use a mix of the options on that spectrum. At one extreme of the spectrum is the use of standard cloud computing resources to off-load the "heavy lifting" that may be required in some mobile applications to specially designated servers or server clusters. A related technique for remote processing of mobile applications proposes the use of *cloudlets* which provide software instantiated in real-time on nearby computing resources using virtual machine technology [30]. Likewise, MAUI [12] and CloneCloud [11] automatically apportion processing between a local device and a remote cloud resource. In this paper we consider the other spectrum extreme, where a mobile device's contacts are only with other mobile devices, where both the computation initiator and the remote com-

[1]See http://blogs.kqed.org/climatewatch/2011/01/29/citizen-science-the-iphone-app/

[2]See http://www.crisscrossed.net/2009/08/31/citizen-scientist-how-mobile-phones-can-contribute-to-the-public-good/

putational resources are mobile, and where intermittent connectivity among these entities is the norm.

We investigate the basic scenario where an *initiator* mobile device needs to run a computational task that exceeds the mobile device's ability and where portions of the task are amenable to remote execution. We leverage the fact that a mobile device within its intrinsic motion pattern makes frequent contact with other mobile devices that are capable of providing computing resources. Contact with these devices can be intermittent, limited in duration when it occurs, and sometimes unpredictable. The goal of the mobile device is to use the available, potentially intermittently connected, computation resources in a manner that improves its computational experience, e.g., minimizing local power consumption and/or decreasing computation completion time. The challenge facing the initiator device is how to apportion the computational task into subtasks and how to allocate such tasks for remote processing by the devices it encounters.

1.1 Related Work

Our work can be viewed as enabling a truly general vision of *cyber foraging* [4, 5] which envisions mobile applications "living off the land" by exploiting nearby computational resources. At the time of the original conception of the cyber foraging idea almost a decade ago [35] it was hard to imagine the compute power of today's mobile devices (smartphones and tablets) and the vision was, therefore, necessarily limited to constant connectivity to infrastructure-based services. Today, however, it is possible to extend the flexibility of this vision to include "foraging" of the available resources in other mobile devices as we propose to do in this work.

Our work also leverages recent advances in the understanding of data transfer over intermittently-connected wireless networks (also known as disruption-tolerant networks or opportunistic networks). These networks have been studied extensively in a variety of settings, from military [26] to disasters [15] to the developing world [27]. These settings share the characteristic that fixed infrastructure is unavailable, highly unreliable, or expensive. Further, the communication links are subject to disruptions that mean network partitions are common.

Our work is also related to the efforts at developing useful applications over intermittently-connected mobile and wireless networks. Examples of this work include the work by Hanna et al. which develops mobile distributed information retrieval systems [16], and the work by Fall et al. on an architecture for disaster communications response [15] with a specific focus on situational awareness. In this latter work the authors propose an architecture that contains infrastructure-supported servers, mobile producer/consumer nodes and mobile field servers. Related, the Hastily Formed Networks (HFN) project [14] describes potential applications in disaster settings that match well with our vision requiring computation, including situational awareness, information sharing, planning and decision making.

Our work is also closely related to systems that use non-dedicated machines with cycles that are donated and may disappear at any time. In this vein, our work takes some inspiration from the Condor system architecture [31]. Our work also resembles in part those distributed computing environments that have well-connected networks but unreliable participation in the computation, such as those seen in voluntary computing efforts where users can contribute compute cycles, but may also simply turn off their machines or networks at will in the middle of a computation. Examples of these systems include BOINC [2]; other examples are SETI@home [3], and folding@home[6], all leveraging willingness on the part of individuals to dedicate resources to a large computation problem. More recently, the Hyrax project envisions a somewhat similar capability to opportunistically use the resources of networked cellphones [25].

1.2 Paper Outline

The remainder of this paper is organized as follows: we start with the discussion of the problem context and the design challenges in Section 2; we describe the design of a job model and the Serendipity system in Section 3; the task allocation algorithms are presented in Sections 4 and 5; we describe how to enable energy-aware computing in Section 6; we undertake an extensive evaluation of our system on Emulab in Section 7; the implementation and evaluation of Serendipity on mobile devices are presented in Section 8; We conclude this paper and discuss our future work in Section 9.

2 Problem Context and Design Challenges

Network Model: We focus on a network environment that is composed of a set of mobile nodes with computation and communication capabilities. The network connectivity is intermittent, leading to a frequently-partitioned network. Every node can execute computing tasks, the number of which is constrained by its resources, such as processor capability, memory, storage size, and available energy. The period of time during which two nodes are within communication range of each other is called a *contact*. During a contact nodes can transfer data to each other. Both the duration and the transfer bandwidth of a contact are limited. There are some variants of the general network setting. For some mobile devices, a low-capacity control channel (e.g., over satellite link) is available for metadata sharing. In addition, in some special networks, such as networks with scheduled robotic vehicles or UAVs, the node mobility patterns are predictable and, thus, their future contacts are also predictable. All these variants are taken into consideration in our design.

Remote computing usually involves the execution of computationally complex jobs through the cooperation among a set of devices connected by a network. A major class of such jobs, supported by mainstream distributed computing platforms such as Condor [31], can be represented as a Directed Acyclic Graph (DAG). The vertices are programs and the directed links represent data flows between two programs. A traditional distributed computing platform maps the vertices to the devices and the links to the network so that all independent programs are executed in parallel and they transfer the output to their children. As a variant of such computing platforms, MAUI [12] and CloneCloud [11] have a simple network composed of a mobile device and the cloud.

Design Challenges: The intermittent connectivity among mobile devices poses three key challenges for remote computing. First, because the underlying connectivity is often unknown and variable, it is difficult to map computations onto nodes with an assurance that the required code and data can be delivered and the results are received in a timely fashion. This suggests a conservative approach to distributing computation so as to provide protection against future network disruptions. Second, given that the network bandwidth is intermittent, the network is more likely to be a bottleneck for the completion of the distributed computation. This suggests scheduling sequential computations on the same node so that the data available to start the next computation need not traverse the network. Third, when there is no control channel, the network cannot be relied upon to provide reachability to all nodes as needed for coordination and control. This suggests maintaining local control and developing mechanisms for loose coordination. Besides the intermittent connectivity, the limited available energy imposes

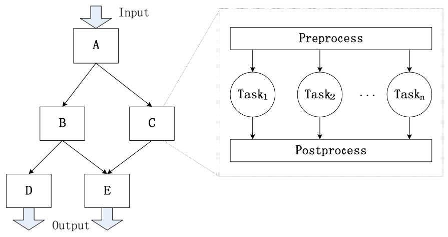


Figure 1: A job model for DTNs is a Directed Acyclic Graph (DAG), the vertices of which are PNP-blocks. Every PNP-block consists of a pre-process, a post-process and n parallel tasks.

another extra constraint on the remote computing among mobile devices.

3 Serendipity System Design

3.1 A Job Model for Serendipity

Our basic job component is called a *PNP-block*. As shown in Figure 1, a PNP-block is composed of a *pre-process* program, n parallel *task* programs and a *post-process* program. The pre-process program processes the input data (e.g., splitting the input into multiple segments) and passes them to the tasks. The workload of every task should be similar to each other to simplify the task allocation. The post-process program processes the output of all tasks; this includes collecting all the output and writing them into a single file.

The PNP-block design simplifies the data flow among tasks and, thus, reduces the impact of uncertainty on the job execution. All pre-process and post-process programs are executed on one initiator device, while parallel tasks are executed independently on other devices. The communication graph becomes a simple star graph. The data transfer delay can be minimized as the initiator device can simply choose nearby devices to execute tasks. In contrast, it is much more difficult for a complicated communication graph, such as the complete bipartite graph used in MapReduce [13], to achieve low delay among intermittently connected mobile devices because the optimization problem associated with mapping the general graph onto them is complex.

The single PNP-block job comprises an important class of distributed computing jobs often called embarrassingly parallel and useful in many applications, among which are SETI@home [3] and BOINC [2]. All jobs are graphically represented by a DAG of PNP-blocks, providing as much computational expressiveness as a regular DAG. For instance, the MapReduce model [13] can be implemented with two sequentially connected PNP-blocks, corresponding to the map phase and the reduce phase, respectively.

3.2 Serendipity System

Figure 2 shows the high-level architecture of Serendipity. A Serendipity node has a *job engine* process, a *master* process and several *worker* processes. The number of worker processes can be configured, for example, as the number of cores or processors of the node. Each node constructs its device profile and, then, shares and maintains the profiles of encountered nodes. A node's device profile includes its execution speed which is estimated by running synthetic benchmarks and its energy consumption model using techniques like PowerBooter [34]. These device profiles when combined with the jobs' execution profiles are used to estimate the jobs' execution time and energy consumption on every node, essential for task allocation. Serendipity also needs access to the contact database, if available, for better task allocation.

To submit a job, a user needs to provide a script specifying the job DAG, the programs and their execution profiles (e.g., CPU cycles) for all PNP-blocks and the input data to the job engine. Con-

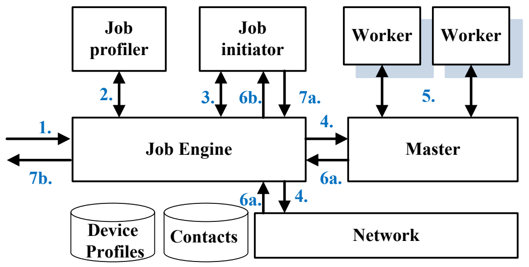


Figure 2: High-level Architecture of Serendipity. After receiving a job (1), the job engine constructs the job profile (2) and starts a job initiator, who will initiate a number of PNP-blocks and allocate their tasks (3). The job engine disseminates the tasks to either local or remote masters (4). After a worker finishes a task (5), the master sends back the results to the job initiator (6a, 6b), who may trigger new job PNP-blocks (3). After all results are collected, the job initiator returns the final results (7a, 7b) and stops.

structing accurate execution profiles of programs is a challenging problem and out of the scope of this paper. We simply follow the offline method used by both MAUI [12] and CloneCloud [11], i.e., running the programs multiple times with different input data.

The script is submitted to the *job profiler* for basic checking and constructing a complete job profile (i.e., tasks' execution time and energy consumption on every node) using its execution profiles and the device profiles. The generated job profile will be used to decide how to allocate its tasks among mobile devices.

If everything is correct, the job engine will launch a new *job initiator* responsible for the new job. It stores the job information in the local storage until the job completes. All PNP-blocks whose parents have completed will be launched by running their pre-process programs on a local worker and assigning a TTL (i.e., time-to-live), a priority and a worker to every task. The TTL specifies the time before which its results should be returned. If a task misses its TTL, it should be discarded, while a copy will be executed locally on the initiator's mobile device. The priority determines the relative importance of a job's different tasks. Section 5 will discuss how to assign the priorities.

Based on the consideration of task allocation and security, the assigned worker can be a single node, a set of candidate nodes, or a wildcard. In fact, only in the specific scenario that the future contacts are predictable while nodes have a control channel to timely coordinate the remote computing, the job initiator will use the global information to allocate tasks and assign a specific node for each task, which will be discussed in Section 4.1. Otherwise, the job initiator only specifies the set of candidate nodes it trusts and lets the job engine allocate the tasks. Finally, these tasks are sent to the job engine for dissemination.

The job engine is primarily responsible for disseminating tasks and scheduling the task execution for the local master. When two mobile nodes encounter, they will first exchange the metadata including their device profiles, their residual energy and a summary of their carried tasks. Using this information, the job engine will estimate whether it is better to disseminate a task to the encountered node than to execute it locally. Such a decision is based on the goal of reducing the job completion time (to be discussed in Section 4) or conserving the device energy (to be discussed in Section 6).

To schedule the task execution, the job engine first determines the job priority. Currently we use the first-in-first-serve policy. But it can be easily replaced by any arbitrary policy. For example, the job from a node that helps other nodes execute a lot of tasks is assigned a high priority. For the tasks of the same job, they are scheduled according to their task priorities.

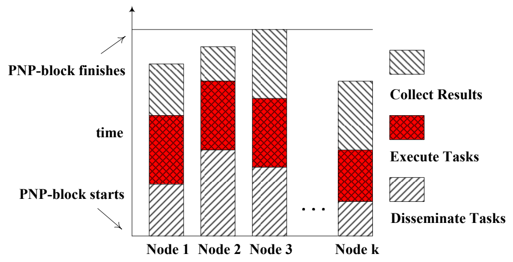


Figure 3: The PNP-block completion time is composed of a) the time to disseminate tasks, b) the time to execute tasks and c) the time to collect results, in addition to the time needed to execute pre-process and post-process programs.

The *master* is responsible for monitoring the task execution on workers. After receiving a task from the job engine, it starts a worker for it. When the task finishes, the output will be sent back to the job initiator using the underlying routing protocols like MaxProp [7]. If the task throws an exception during the execution, the master will report it to the job initiator who will terminate the job and report to the user.

In this paper, we assume that all nodes are collaborative and trustworthy. However, there are also scenarios that some nodes are selfish (i.e., refusing to help other nodes) or even malicious (i.e., distorting the results). To motivate the selfish nodes, we can use some token-based incentive mechanism [24], making use of notional credit to pay off nodes for executing tasks. To protect the remote computing from malicious nodes, we can use reputation-based trust [8] in which nodes construct and share nodes' reputation information.

4 Task Allocation for PNP-blocks

One important goal of remote computing is to improve the performance of computationally complex jobs, especially when mobile nodes have enough energy. In this section, we will design efficient task allocation algorithms to minimize the job completion time. Specifically, since PNP-blocks are the basic blocks to allocate tasks, we will focus on the task allocation for PNP-blocks in various network settings. The problem of task scheduling for multi-processor systems [10, 18] is somewhat related to our task allocation problem. That work, however, does not deal with intermittent connectivity and cannot, therefore, be applied directly to our problem.

Figure 3 illustrates the timing and components of a PNP-block execution. Along the x-axis are the k remote nodes that will execute the parallel tasks of the block. Along the y-axis is a depiction of the time taken at each node to receive disseminated tasks from the initiator, execute those tasks, and provide the result collection back to the initiator. As illustrated, the time for each remote node to receive its disseminated tasks may vary, depending on the availability and quality of the network between the initiator and the remote node. When n tasks of a PNP-block are allocated to k nodes, each node will execute its assigned tasks sequentially, again taking a variable amount of time. After execution of all assigned tasks in the block, the node will send results back to the initiator, with time again being dependent on the network between the initiator and the remote node. Our goal for the task allocation is to reduce the completion time of the last task which equals to the PNP-block completion time.

We consider the design of task allocation algorithms in the context of three models with different contact knowledge and control channel availability assumptions.

4.1 Predictable Contacts with Control Channel

We first consider an ideal network setting where the future contacts can be accurately predicted, and a control channel is available for coordination. The performance in this type of scenarios represents the best possible performance of task allocation that is achievable among intermittently connected mobile devices. It is useful to identify the fundamental benefits and limits of Serendipity.

With future contact information a Dijkstra's routing algorithm for DTNs [20] can be used to compute the required data transfer time between any pair of nodes given its starting time. With the control channel the job initiator can obtain the time and number of tasks to be executed on the target node with which to estimate the time to execute a task on that node. Therefore, given the starting time and the target node, the task completion time can be estimated.

Using this information, we propose a greedy task allocation algorithm, *WaterFilling*, that iteratively chooses the destination node for every task with the minimum task completion time (see Algorithm 1).

Algorithm 1 Water Filling

```
 1: procedure WATERFILLING($T$, $N$)        ▷ $T$ is task set; $N$ is node set.
 2:     current ← currentTime();
 3:     rsv ← getTaskReservationInfo();
 4:     inputSize ← getTaskInputSize($T$);
 5:     outputsize ← estimateOutputSize($T$);
 6:     queue ← initPriorityQueue();
 7:     for all $n \in N$ do
 8:         arrivalT ← dijkstra(this, $n$, current, inputSize);
 9:         exeT ← estimateTaskExecutionTime($n$,$t$);        ▷ $t \in T$
10:         tfinishT ← taskFinishTime(rsv[$n$], arrivalT, exeT);
11:         completeT ← dijkstra( $n$, this, tfinishT, outputSize);
12:         queue.put({$n$, arrivalT, exeT, completeT});
13:     end for
14:     for all $t \in T$ do
15:         {$n$, arrivalT, exeT, receiveT} ← queue.poll();
16:         updateReservation(rsv[$n$], $t$, inputSize, arrivalT, exeT);
17:         send(n, t);
18:         arrivalT ← dijkstra(this, $n$, current, inputSize);
19:         tfinishT ← taskFinishTime(rsv($n$), arrivalT, exeT);
20:         completeT ← dijkstra( $n$, this, tfinishT, resultSize);
21:         queue.put({$n$, arrivalT, exeT, receiveT});
22:     end for
23:     reserveTaskTime(rsv);
24: end procedure
```

For every task, the algorithm first estimates its task dissemination time to every node. With the information of the tasks to be executed on the destination node and the estimated time to execute this task, it is able to estimate the time when this task will finish. Given that time point, the time when the output is sent back can also be computed. Among all the possible options, we choose the node that achieves the minimum task completion time to allocate the task. The allocation of the next task will take the current task into account and repeat the same process. Finally, the job initiator will reserve the task execution time on all related nodes, which will be shared with other job initiators for future task allocation.

4.2 Predictable Contacts without Control Channel

When mobile nodes have no control channels, it is impossible to reserve task execution time in advance. WaterFilling will cause contention for task execution among different jobs on popular nodes, prolonging the task execution time. To solve this problem, we propose an algorithm framework, Computing on Dissemination (CoD), to allocate tasks in an opportunistic way. The algorithm is shown in Algorithm 2.

Algorithm 2 Computing on Dissemination

```
 1: procedure ENCOUNTER($n$)                          ▷ $n$ is the encountered node.
 2:     summary ← getSummary();
 3:     send($n$, summary);
 4: end procedure
 5: procedure GETSUMMARY
 6:     compute ← getNodeComputingSummary();
 7:     net ← getNetworkSummary();
 8:     tasks ← getPendingTaskSummary();
 9:     return {compute,net,tasks};
10: end procedure
11: procedure RECEIVESUMMARY($n$, msg)      ▷ msg is the summary message of node $n$.
12:     updateNodes(msg.compute);
13:     updateNetwork(msg.net);
14:     toExchange ← exchangeTask($n$, this.tasks, msg.tasks);
15:     isSent ← $false$;
16:     while $n$.isConnected() && !toExchange.isEmpty() do
17:         send($n$, toExchange.poll());
18:         isSent ← $true$;
19:     end while
20:     if $n$.isConnected() && isSent == $true$ then
21:         summary ← getSummary();
22:         send($n$, summary);
23:     end if
24: end procedure
25: procedure RECEIVETASK(msg)              ▷ msg contains exchanged tasks.
26:     addTasks(msg.tasks);
27: end procedure
```

The basic idea of CoD is that during the task dissemination process, every intermediate node can execute these tasks. Instead of explicitly assigning a destination node to every task, CoD opportunistically disseminates the tasks among those encountered nodes until all tasks finish. Every time two nodes encounter each other, they first exchange metadata about their status. Based on this information, they decide the set of tasks to exchange. When they move out of the communication range, they will keep the remaining tasks to execute locally or exchange with other encountered nodes in the future.

The key function of this algorithm is the *exchangeTask* function of line 14 that decides which tasks to exchange. In this subsection we assume that future contact is still predictable. Therefore, the task completion time can be estimated when the task arrives at a node as discussed in last subsection. The intuition of CoD with predictable contacts (pCoD) is to locally minimize the task completion time of every task if possible. When a node receives the summary message from the encountered node, it first estimates the execution time of its carried tasks on the other node using the job profiles and the device profiles. For each task it carries, it estimates the task completion time (i.e., the time that its result is received by the initiator) of executing locally and that of executing on the other node by using the contact information. If the local task completion time is larger than the remote one, it sends the task to the encountered node. Every node conservatively makes the decision without considering the tasks the other node will send back.

4.3 Unpredictable Contacts

Finally we consider the worst case that future contacts cannot be accurately predicted. Our task allocation algorithm, CoD with unpredictable contacts (upCoD), is still based on CoD with the constraint that future contact information is unavailable. As shown in Figure 3, minimizing the time when the last task is sent back to the job initiator will reduce the PNP-block completion time. When the data transfer time is unpredictable, we envision that reducing the execution time of the last task will also help reduce PNP-block completion time. This is because the locality property of CoD indicates the existence of a short time-space path between the worker node and the job initiator node. Therefore, when two nodes encounter each other, upCoD tries to reduce the execution time of every task.

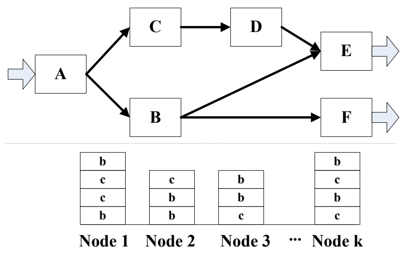


Figure 4: A job example where both PNP-block B and C are disseminated to Serendipity nodes after A completes. Their task positions in the nodes' task lists are shown blow the DAG.

In reality, historical contact information is useful to roughly estimate the future contacts [7] and, thus, should be helpful to task exchange in CoD. Its performance is probably between upCoD and pCoD. We will investigate such possibility as part of our future work.

5 PNP-block Scheduling

Our PNP-block design simplifies the task allocation so that every PNP-block is treated independently. However, it is still possible to further reduce the job completion time by assigning priorities to PNP-blocks since tasks from the same job are executed according to their priority assignment.

Let's consider a simple job DAG shown in Figure 4. PNP-blocks B and C are simultaneously allocated after A completes. Their tasks arrive at the destination nodes unordered. Given a network and a task allocation algorithm, the total time required for both B and C to finish remains almost the same. However, either B or C can have a shorter PNP-block finish time if any of them is given a higher priority over the other. This will be beneficial because their children PNP-block can start earlier.

OBSERVATION 1. *It is better to assign different priorities to the PNP-blocks of a job.*

In the example shown in Figure 4, PNP-block E can only start when both B and D finish. Thus, B and D are equivalently important to E. Meanwhile, there is a time gap between the execution time of C and that of D caused by the result collection of C and task dissemination of D. During that gap, the execution of other tasks (e.g., B) will not affect the PNP-block finish time of D. Therefore, if C is assigned a higher priority than B, the total time for both B and D to finish will be shorter.

OBSERVATION 2. *All parents of a PNP-block are equivalently important to it, while parents have higher priorities than their children.*

The next question arises when B and D are in the task list of the same node, which should have higher priority. We notice that both B and D are equivalent to E, while E and F are equivalent to the job. However, if B finishes earlier, F can start earlier. This is because F only relies on B.

OBSERVATION 3. *When two PNP-blocks have the same priority, the one with more children only depending on it should be assigned a higher priority.*

If there are still PNP-blocks with the same priority, we randomly assign some different priorities to them that keep their relative priorities with other PNP-blocks. Algorithm 3 shows our priority assigning algorithm. The sort method of line 13 is based on Observation 3.

Algorithm 3 PNP-block Priority Assigning

```
1: procedure ASSIGNPRIORITY($J$)                          ▷ $J$ is the job DAG
2:     while !$J$.allPNPblocksHavePriority() do
3:         for all $s \in J$ do                            ▷ s is a PNP-block
4:             if !$s$.haveChild() then
5:                 $s$.priority ← 0;
6:             else if $s$.allChildrenHavePriority() then
7:                 $s$.priority ← $s$.maxChildrenPriority()+1;
8:             end if
9:         end for
10:    end while
11:    for $p = 0 \rightarrow J$.getMaxPriority() do
12:        PNPblocks ← $J$.getPNPblocksWithPriority($p$);
13:        sort(PNPblocks);
14:        for $i = 0 \rightarrow$ PNPblocks.size()-1 do
15:            s ← PNPblocks.get(i);
16:            s.priority ← s.priority + $\frac{i}{PNPblocks.size()}$;
17:        end for
18:    end for
19: end procedure
```

6 Energy-Aware Computing

In the above two sections we focused on how to accelerate the job execution without any consideration of the energy consumption. Because of the limited energy available to some kinds of mobile devices (e.g., smartphones), there are also scenarios when energy conservation is at least as important as execution performance, especially when the applications can tolerate delays. In this section, we describe how to support energy-aware computing with Serendipity.

When a mobile device tries to off-load a task to another mobile device to save energy, the latter may have very limited energy, too. Meanwhile, if all nodes postpone the task execution forever, it definitely saves energy, but meaninglessly. Therefore, a reasonable objective of energy-aware computing among mobile devices makes all nodes last as long as possible while timely finishing the jobs, i.e., maximizing the lifetime of the first depleted node under the constraint that jobs complete before their deadline (i.e., TTL). Unfortunately, without information about the future jobs, it is impossible to solve this optimization problem.

An approximation to this ideal optimization is to greedily minimize a utility function when the job initiator allocates the tasks. Two factors should be considered in the utility functions, the energy consumption of all nodes involved in the remote computing of the task and the residual energy available to these nodes. A good utility function should consume less energy while avoiding nodes with small residual energy. We use a simple utility function that has been considered in energy-aware routing [9]:

$$u(T) = \sum_{i \in N_T} \frac{e_{Ti}}{R_i} \tag{1}$$

where N_T is the set of nodes involved in the remote computing of task T, e_{Ti} is the energy consumption of node i for task T, and R_i is the residual energy of node i.

As discussed in Section 4 the task allocation algorithms, WaterFilling, pCoD and upCoD, try to optimize the job completion time. By replacing the time with the utility function $u(T)$, we can easily adapt these task allocation algorithms to be energy-aware. Specifically, the energy-aware WaterFilling algorithm iteratively chooses the destination node of every task with minimum $u(T)$ while satisfying the TTL constraint. When two nodes encounter, pCoD and upCoD will exchange a task if executing it on current node has higher utility than executing on the other node while satisfying the TTL constraint. If the future contacts are unpredictable, upCoD replaces TTL with the time that task is executed.

7 Evaluation

7.1 Experimental Setup

To evaluate Serendipity in various network settings, we have built a testbed on Emulab [33] to easily configure the experiment settings including the number of nodes, the node properties, etc. In our testbed, a Serendipity node running on an Emulab node has an emulation module to emulate the intermittent connectivity among nodes. Before an experiment starts, all nodes load the contact traces into their emulation modules. During the experiments, the emulation module will control the communication between its node and all other nodes according to the contact traces.

In the following experiments, we use two real-world contact traces, a 9-node trace collected in the Haggle project [19] and the RollerNet trace [32]. In the RollerNet trace, we select a subset of 11 friends (identified in the metadata of the trace) among the 62 nodes so that the number of nodes is comparable to the Haggle trace. The Haggle trace represents the user contacts in a laboratory during a typical day, while RollerNet represents the contacts among a group of friends during the outdoor activity. These two traces demonstrate quite different contact properties. RollerNet has shorter contact intervals, while Haggle has longer contact durations.

We also use three mobility models to synthesize contact traces, namely the Levy Walk Model [28], the Random WayPoint Model (RWP) [29], and the Time-Variant Community Mobility Model (TVCM) [21]. We change various parameters to analyze their impact on Serendipity.

We implement a speech-to-text application based on Sphinx library [23] that translates audio to text. It will be used to evaluate the Emulab-based Serendipity. It is implemented as a single PNP-block job where the pre-process program divides a large audio file into multiple 2 Mb pieces, each of which is the task input.

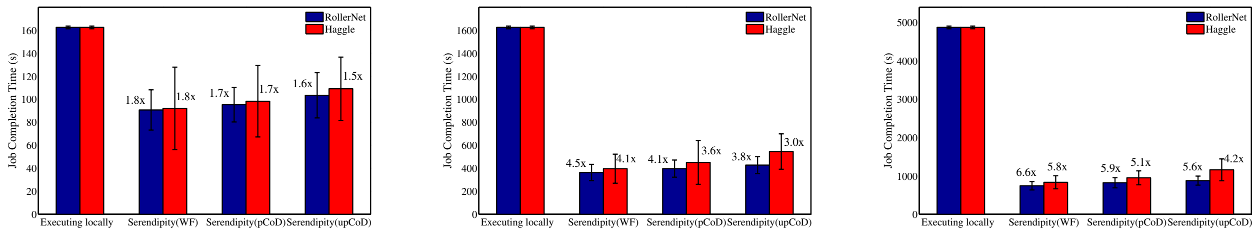
To demonstrate how Serendipity can help the mobile computation initiator to speedup computing and conserve energy, we primarily compare the performance of executing applications on Serendipity with that of executing them locally on the initiator's mobile device. Previous remote-computing platforms (e.g., MAUI [12], CloneCloud [11], etc) don't work with intermittent connectivity and, thus, cannot be directly compared with Serendipity.

In all the following experiments every machine has a 600 MHz Pentium III processor and 256 MB memory, which is less powerful than mainstream PCs but closer to that of smart mobile devices. Every experiment is repeated 10 times with different seeds. The results reported correspond to the average values.

7.2 Serendipity's Performance Benefits

We initiate the experiments with the speech-to-text application using three workloads in three task allocation algorithms on both RollerNet and Haggle traces. The sizes of the audio files are 20 Mb, 200 Mb, and 600 Mb. As mentioned before, it is implemented as a single PNP-block job whose pre-process program divides the audio file into multiple 2 Mb pieces corresponding to 10, 100, and 300 tasks, respectively. The post-process program collects and combines the results. The baseline wireless bandwidth is set to 24 Mbps. We also assume that all nodes have enough energy and want to reduce the job completion time.

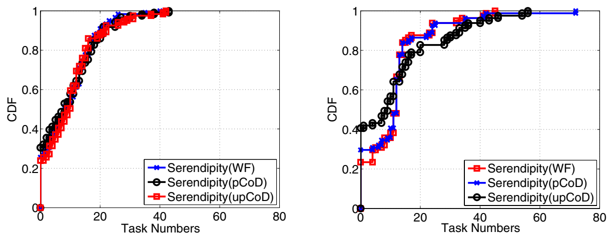
Figure 5 demonstrates how Serendipity improves the performance compared with executing locally. We make the following observations. First, with the increase of the workload, Serendipity achieves greater benefits in improving application performance. When the audio file is 600 Mb, Serendipity can achieve as large as 6.6 and 5.8 time speedup. Considering the number of nodes (11 for RollerNet and 9 for Haggle), the system utilization is more than 60%. More-



(a) 10 tasks (b) 100 tasks (c) 300 tasks

Figure 5: A comparison of Serendipity's performance benefits. The average job completion times with their 95% confidence intervals are plotted. We use two data traces, Haggle and RollerNet, to emulate the node contacts and three input sizes for each.

over, the ratio of the confidence intervals to the average values also decreases with the workload, indicating all nodes can obtain similar performance benefits. Second, in all the experiments WaterFilling consistently performs better than pCoD which is better than upCoD. In the Haggle trace of Figure 5(c), WaterFilling achieves 5.8 time speedup while upCoD only achieves 4.2 time speedup. The results indicate that with more information Serendipity can perform better. Third, although Serendipity achieves similar average job completion times on both Haggle and RollerNet, their confidence intervals on Haggle are larger than those on RollerNet. This is because the Haggle trace has long contact interval and duration, resulting in the diversity of node density over the time.



(a) RollerNet (b) Haggle

Figure 6: The load distribution of Serendipity nodes when there are 100 tasks total, each of which takes 2 Mb input data.

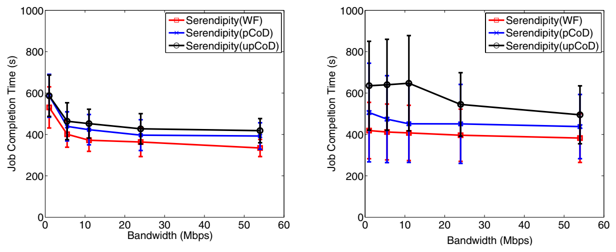
To further analyze the performance diversity, we plot the workload distribution on the Serendipity nodes of Figure 5(b) in Figure 6. In the RollerNet trace, all three task allocation algorithms have similar load distribution, i.e., about 25% nodes are allocated 0 tasks while about 10% of the nodes are allocated more than 20 tasks. In the Haggle trace, WaterFilling and pCoD have similar load distribution, while upCoD's distribution is quite different from them. The long contact intervals of the Haggle trace makes the blind task dissemination of upCoD less efficient. In such an environment, the contact knowledge will be very useful to improve the Serendipity performance.

7.3 Impact of Network Environment

Next, we analyze the impact of the network environment on the performance of the three task allocation algorithms by changing the network settings from the base case.

Wireless Bandwidth: We first consider the effect of wireless bandwidth on the performance of Serendipity. The wireless bandwidth is set to be 1 Mbps, 5.5 Mbps, 11 Mbps, 24 Mbps, and 54 Mbps, which are typical values for wireless links. The audio file is 200 Mb, split into 100 tasks. We plot the job completion times of Serendipity with three task allocation algorithms in Figure 7.

We observe the following phenomena. First, in RollerNet, all three task allocation algorithms accomplish similar performance. Because these nodes have frequent contacts with each other, using the locality heuristic (upCoD) is good enough to make use of the



(a) RollerNet (b) Haggle

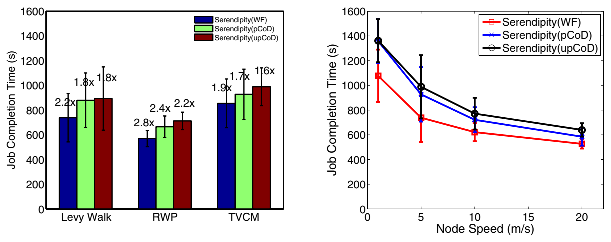
Figure 7: The impact of wireless bandwidth on the performance of Serendipity. The average job completion times are plotted when the bandwidth is 1, 5.5, 11, 24, and 54 Mb/s, respectively.

nearby computation resource for remote computing. Second, when the bandwidth reduces from 11 Mbps to 1 Mbps, the job completion time experiences a large increase. This is because RollerNet has many short contacts which cannot be used to disseminate tasks when the bandwidth is too small. Third, in the Haggle trace, the job completion time of upCoD increases from 545.0 seconds to 647.6 seconds when the bandwidth reduces from 24 Mbps to 11 Mbps. Meanwhile WaterFilling achieves consistently good performance in all the experiments. This is because in the laboratory environment users are relatively stable and have longer contact durations. Thus, the primary factor affecting the Serendipity performance is the contact interval. On the other hand, since the contact distribution is more biased, only using locality is hard to find the global optimal task allocation.

Node Mobility: The above experiments demonstrate that contact traces impact the performance of Serendipity. To further analyze such impact, we use mobility models to generate the contact traces for 10 nodes. Specifically, we use Levy Walk Model [28], Random WayPoint Model (RWP) [29], and Time-Variant Community Mobility Model (TVCM) [21]. These models represent a wide range of mobility patterns. RWP is the simplest model and assumes unrestricted node movement. Levy Walk describes the human walk pattern verified by collected mobility traces. TVCM depicts human behavior in the presence of communities. The basic settings assume a 1 Km by 1 Km square activity area in which each node has a 100 m diameter circular communication range.

In this set of experiments we focus on the two most important aspects of node mobility, i.e., the mobility model and the node speed. The wireless bandwidth is set to 11 Mbps.

The results of this comparison are shown in Figure 8. Figure 8(a) shows that Serendipity has larger job completion time with all the mobility models than it had on Haggle and RollerNet traces. This is because their node densities are much sparser than Haggle and RollerNet traces. Thus it's harder for the job initiator to use other nodes' computation resources. We also observe that Serendipity achieves the best performance when the RWP model is used. This



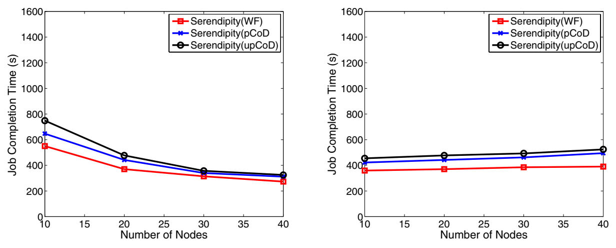
(a) Mobility Models (b) Node Speed

Figure 8: The impact of node mobility on Serendipity. We generate the contact traces for 10 nodes in a 1 km×1 km area. In (a) we set the node speed to be 5 m/s, while in (b) we use Levy Walk as the mobility model.

is because RWP is the most diffusive [28] and, thus, results in more contact opportunities among nodes.

Node speed affects the contact frequencies and durations, which are critical to Serendipity. We vary the node speed from 1 m/s, i.e., human walking speed, to 20 m/s, i.e., vehicle speed. As shown in Figure 8(b), when the speed increases from 1 m/s to 10 m/s, the job completion times drastically decline, e.g., from 1077.1 seconds to 621.6 seconds for WaterFilling. This is because the increase of node speed significantly increases the contact opportunities and accelerates the task dissemination. When the speed further increases to 20 m/s, the job completion time is slightly reduced to 526.4 seconds for WaterFilling.

Number of Nodes: We finally examine how the quantity of available computation resources impacts Serendipity. To separate the effect of node density and resource quantity, we conduct two sets of experiments. In the first set, the active area is fixed, while in the second one, the active area changes proportionally with the number of nodes using the initial setting of 20 nodes in 1 km×1 km square area. Figure 9 shows the results where nodes follows RWP mobility model with wireless bandwidth at 2 Mbps.



(a) Fixed Active Area (b) Fixed Node Density

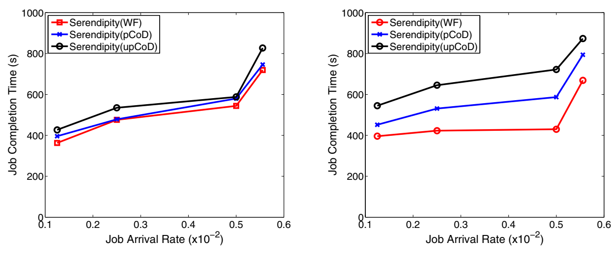
Figure 9: The impact of node numbers on the performance of Serendipity. We analyze the impact of both node number and node density by fixing the activity area and setting it proportional to the node numbers, respectively.

As shown in Figure 9(a), with the increase in the number of nodes in a fixed area, the job completion times of the three task allocation algorithms are reduced by more than 50%, from 550.0, 647.0, and 748.7 seconds to 273.0, 311.7, and 325.0 seconds for WaterFilling, pCoD and upCoD, respectively. Meanwhile, in Figure 9(b), the job completion times are almost constant despite the increase in node quantity.

7.4 The Impact of the Job Properties

Next we evaluate how the job properties affect the performance of Serendipity.

Multiple jobs: A more practical scenario involves nodes submitting multiple jobs simultaneously into Serendipity. These jobs will affect the performance of each other when their execution duration overlaps. In this set of experiments, nodes will randomly submit 100-task jobs into Serendipity. The arrival time of these jobs follows a Poisson distribution. We change the arrival rate, λ from 0.0013 (its system utilization is less than 20%) to 0.0056 (its system utilization is larger than 90%) jobs per second. Figure 10 shows the results on the RollerNet and Haggle traces.



(a) RollerNet (b) Haggle

Figure 10: Serendipity's performance with multiple jobs executed simultaneously. The job arrival time follows a Poisson distribution with varying arrival rates.

As expected, the job completion time increases with the job arrival rate. In both sets of experiments, the job completion time gradually increases with the job arrival rate until 0.005 jobs per second and, then, drastically increases when the job arrival rate increase to 0.0056 jobs per second. According to queueing theory, with the system utilization approaching 1, the queueing delay is approaching infinity. However, even when the system utilization is larger than 90% (i.e., $\lambda = 0.0056$), the job completion times of Serendipity with various task allocation algorithms are still less than 54% of executing locally, showing the advantage of distributed computation.

DAG jobs: The above experiments show that Serendipity performs well for single PNP-block jobs. Since DAG jobs are executed iteratively for all dependent PNP-blocks while parallel for all independent PNP-blocks. The above experiment results also apply to DAG jobs. In this set of experiments we will evaluate how PNP-block scheduling algorithm further improves the performance of Serendipity.

We use the job structure shown in Figure 4, where the processing of one image impacts the processing of another. We use the PNP-blocks of speech-to-text application as the basic building blocks. PNP-block A has 0 tasks; B has 200 tasks; C has 50 tasks; D has 100 tasks; E has 100 tasks; F has 0 tasks. The performance difference between our algorithm and assigning equal priority to the PNP-blocks is shown in Figure 11.

Our priority assignment algorithm achieves the job completion time of 1155.8, 1315.8 and 1383.2 seconds for WaterFilling, pCoD, and upCoD, consistently outperforming that of 1369.2, 1573.4, and 1654.4 seconds when all PNP-blocks have the same priority. These experiments demonstrate the usefulness of priority assigning. Further evaluation of our algorithm on diverse type of jobs will be part of our future work.

7.5 Energy Conservation

In this set of experiments, we demonstrate how Serendipity makes the entire system last longer by taking the energy consumption into consideration. We consider an energy critical scenario where node i has E_i% energy left, where E_i is randomly selected from [0, 20]. The energy consumption of task execution and communication is randomly selected from the measured values on mobile devices. The detailed measurement will be presented in the next section. In this set of experiments, nodes will randomly submit 100-task jobs into Serendipity. The arrival time of these jobs follows a Poisson

Table 1: A comparison of Serendipity's energy consumption. We report the number of jobs completed before at least one node depletes its battery and their average job completion time. Jobs arrive in a Poisson process with $\lambda = 0.005$ jobs per second.

	Haggle				RollerNet			
	Energy Aware		Time Optimizing		Energy Aware		Time Optimizing	
	# Completed jobs	Time (s)	# Completed jobs	Time (s)	# Completed jobs	Time(s)	# Completed jobs	Time(s)
Serendipity(WF)	17.0	2664.7	4.5	409.2	21.8	2823.9	2.5	496.2
Serendipity(pCoD)	10.0	2162.4	3.0	435.3	16.8	2173.6	4.8	539.0
Serendipity(upCoD)	9.3	2080.6	3.0	564.0	16.8	2082.6	3.5	562.7
Executing locally	N/A	N/A	1.3	1614.0	N/A	N/A	1.3	1614.0

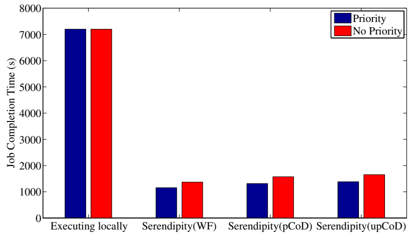


Figure 11: The importance of assigning priorities to PNP-blocks.

distribution with $\lambda = 0.005$ jobs per second. We compare energy-aware Serendipity against "time-optimizing" Serendipity and executing jobs locally. The TTL of energy-aware Serendipity is set to twice the time of executing the job locally.

Table 1 shows the number of jobs completed before at least one node depletes its energy and the average job completion time of those completed jobs. We make the following observations. First, energy-aware Serendipity completes many more jobs than executing locally and using time-optimizing Serendipity. This is because energy-aware Serendipity balances the energy consumption of all the mobile devices through adaptively allocating more tasks to devices with more residual energy. In contrast, the time-optimizing Serendipity will quickly deplete the energy of some mobile devices by allocating many tasks to them. Second, through global optimization, energy-aware Serendipity with the WaterFilling allocation algorithm completes more jobs than those with pCoD and upCoD. Third, the job completion time of energy-aware Serendipity is much larger than that of time-optimizing Serendipity. There exists a tradeoff between energy consumption and performance. Finally, compared with executing locally, time-optimizing Serendipity both completes more jobs and has smaller job completion time. This is because statistically the few devices with limited residual energy will last longer by off-loading the computation to other devices.

8 Implementation

We implemented a prototype of Serendipity on the Android OS [1]. It comprises three parts: the Serendipity worker corresponding to the worker in Fig. 2, the Serendipity controller including all other components in Fig. 2 and a user library providing the key APIs for application development.

We currently use *WifiManager*'s hidden API, *setWifiApEnabled*, to achieve the ad hoc communication between two devices, i.e., one device acts as an AP while the other device connects to it as a client.

We use the Java reflection techniques to dynamically execute the tasks. Every task has to implement the function *execute* defined in the APIs. When the Serendipity worker executes a task, it executes this function.

The separation between the Serendipity worker and the Serendipity controller is based on access control. Android's security architecture defines many kinds of permission to various resources including network, GPS, sensors, etc. The Serendipity worker is implemented as a separate application with limited access permission to these resources, acting as a sandbox for the task execution. When the Serendipity controller receives a task to execute, it will start a Senredipity worker and get the results from it.

8.1 System Evaluation

To evaluate our system, we implemented two computationally complex applications, a face detection application, and a speech-to-text application. The face detection application takes a set of pictures and uses computer vision algorithms to identify all the faces in these pictures [17]. It is implemented as a single PNP-block job where the face detection in each picture is a task. The speech-to-text application takes an audio file and translates the speech into text using the Sphinx library [23]. It is also a single PNP-block job where the pre-process program divides a large audio file into multiple pieces, each of which is input to a separate task.

We tested Serendipity on a Samsung Galaxy Tab with a 1 GHz Cortex A8 processor and a Motorola ATRIX smartphone with a dual-core Tegra 2 processor, each at 1 GHz. Both of them run the Android 2.3 OS. The face detection and speech-to-text applications are used for evaluation.

Table 2: The execution time of two applications on two devices.

	Input size (Mb)	Galaxy Tab (s)	ATRIX (s)
FaceDetection	2.2	17.9	7.2
Speech-to-text	3.0	40.3	18.8

We first executed the two applications locally on the two devices. As Motorola ATRIX smartphone has a dual-core processor, we split the input files into two parts of equal size and simultaneously executed the two tasks to fully utilize its processor. Table 2 shows their execution times. We also measured the TCP throughput between these two devices by sending 800 Mb data. We obtain 10.8 Mbps throughput on average when they are within 10 meters. In fact, they still achieve 5.9 Mbps throughput even when they are more than 30 meters away.

To assess the performance of Serendipity, we construct a simple network in which the two devices are consistently connected during the experiments. As expected, Serendipity speeds up more than 3 times than executing the applications on the Samsung Galaxy Tab.

To generate the energy consumption profiles of the two applications on these mobile devices, we repeatedly execute those applications starting with full battery until the batteries are depleted and count the number of iterations. Similarly, WiFi's energy profiles are obtained by continuously transferring data between them. Table 3 demonstrates the results.

Table 3: The energy consumption of mobile devices. The ratios of consumed energy to the total device energy capacity are reported.

	Input size (Mb)	Galaxy Tab	ATRIX
FaceDetection	2.2	4.14×10^{-4}	3.44×10^{-4}
Speech-to-text	3.0	9.32×10^{-4}	9.01×10^{-4}
WiFi	800	8.02×10^{-4}	2.04×10^{-3}

The energy required to transfer a task only accounts for 0.5 % (i.e., $\max(\frac{8.02\times2.2}{4.14\times800}, \frac{8.02\times3.0}{9.32\times800})$) and 1.6% (i.e., $\max(\frac{20.4\times2.2}{3.44\times800},$

$\frac{20.4\times3.0}{9.01\times800}$)) of the energy required to execute the task on these devices, respectively. It indicates that Serendipity won't consume much extra energy. Instead, by delegating tasks to devices with a lot of energy, it can significantly save the job initiator's energy.

We use an extreme example to show the gains of energy-aware Serendipity. Suppose the ATRIX phone has a lot of pictures for face detection. Assume it only has 5% energy left, and the Galaxy tablet has 50% energy left. Energy-aware Serendipity can detect about 1320 pictures before the ATRIX phone depletes its battery, while time-optimizing Serendipity can only detect about 203 pictures.

9 Conclusion and Future Work

In this paper we have developed and evaluated the Serendipity system that enables a mobile device to remotely access computational resources on other mobile devices it may encounter. The main challenge we addressed is how to model computational tasks and how to perform task allocation under varying assumptions about the connectivity environment. Through an emulation of the Serendipity system we have explored how such a system has the potential to improve computation speed as well as save energy for the initiating mobile device. We have also reported on a preliminary prototype of our system on Android platforms.

In our future work we will complete our experimental evaluation of the prototype systems to include more devices and incorporate intermittent connectivity. We will also consider incentive and reputation systems that are derived from previous work in MANET and peer-to-peer systems and tailored to the Serendipity environment.

As mentioned previously we envision Serendipity as developed here to enable an extreme of a spectrum of remote computation possibilities that are available to mobile devices. Our future work will consider extending our investigation to enable hybrid remote computation where the use of cloud or cloudlet resources is augmented with the use of resources on other mobile devices.

Acknowledgments

We would like to thank our shepherd, Kannan Srinivasan, and the anonymous reviewers for their insightful feedback. This work was supported in part by the US National Science Foundation through grant CNS 0831714.

10 References

[1] Android open source project. http://source.android.com.

[2] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *IEEE/ACM GRID*, 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, November 2002.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002.

[5] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *ACM MobiSys*, 2007.

[6] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *IEEE IPDPS*, 2009.

[7] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *IEEE INFOCOM*, 2006.

[8] L. Buttyán and J.-P. Hubaux. Enforcing service availability in mobile ad-hoc wans. In *ACM MobiHoc*, 2000.

[9] J.-H. Chang and L. Tassiulas. Energy conserving routing in wireless ad-hoc networks. In *IEEE INFOCOM*, 2000.

[10] C. Chekuri, A. Goel, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize flow time with ε resource augmentation. In *ACM STOC*, 2004.

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

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

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[14] P. J. Denning. Hastily formed networks. *Commun. ACM*, 2006.

[15] K. Fall, G. Iannaccone, J. Kannan, F. Silveira, and N. Taft. A disruption-tolerant architecture for secure and efficient disaster response communications. In *ISCRAM*, May 2010.

[16] K. M. Hanna, B. N. Levine, and R. Manmatha. Mobile Distributed Information Retrieval For Highly Partitioned Networks. In *IEEE ICNP*, pages 38–47, Nov 2003.

[17] E. HjelmÃěs and B. K. Lowb. Face detection: A survey. *Elsevier Computer Vision and Image Understanding*, September 2001.

[18] E. S. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. In *IEEE IPDPS*, 1994.

[19] P. Hui, J. Scott, J. Crowcroft, and C. Diot. Haggle: a networking architecture designed around mobile users. In *WONS*, 2006.

[20] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *ACM SIGCOMM*, 2004.

[21] W. jen Hsu, a. K. P. Thrasyvoulos Spyropoulos, and A. Helmy. Modeling time-variant user mobility in wireless mobile networks. In *IEEE INFOCOM*, 2007.

[22] J. G. Koomey, S. Berard, M. Sanchez, and H. Won. Assessing Trends in the Electrical Efficiency of Computation over Time. Technical report, `http://www.intel.com/assets/pdf/general/computertrendsreleasecomplete-v31.pdf`, 2009.

[23] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transaction on Acoustics, Speech and Signal Processing*, 1990.

[24] R. Lu, X. Lin, H. Zhu, X. Shen, and B. Preiss. Pi: A practical incentive protocol for delay tolerant networks. *IEEE Transactions on Wireless Communications*, April 2010.

[25] E. Marinelli. Hyrax: Cloud computing on mobile devices using mapreduce. Master's thesis, Computer Science Dept., CMU, September 2009.

[26] P. Marshall. DARPA progress towards affordable, dense, and content focused tactical edge networks. In *IEEE MILCOM*, 2008.

[27] A. S. Pentland, R. Fletcher, and A. Hasson. DakNet: Rethinking connectivity in developing nations. *Computer*, January 2004.

[28] I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the levy-walk nature of human mobility. In *IEEE INFOCOM*, 2008.

[29] A. K. Saha and D. B. Johnson. Modeling mobility for vehicular ad-hoc networks. In *ACM VANET*, 2004.

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

[31] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurr. Comput. : Pract. Exper.*, February 2005.

[32] P. U. Tournoux, J. Leguay, F. Benbadis, V. Conan, M. D. de Amorim, and J. Whitbeck. The accordion phenomenon: Analysis, characterization, and impact on dtn routing. In *Proc. IEEE INFOCOM*, 2009.

[33] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *USENIX OSDI*, 2002.

[34] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *IEEE/ACM/IFIP CODES/ISSS*, 2010.

[35] J. Zhou, E. Gilman, M. Ylianttila, and J. Riekki. Pervasive service computing: Visions and challenges. In *IEEE CIT*, 2010.