Report Number: WUCSE-2007-41

2007

Distributed Allocation of Workflow Tasks in MANETs

Rohan Sen, Gruia-Catalin Roman, and Christopher Gill

When multiple participants work on a workflow that represents a large, collaborative activity, it is important to have a well defined process to determine the portions of the workflow that each participant is responsible for executing. In this paper, we describe a process and related algorithms required to assign tasks in a workflow, to hosts that are willing to carry out the execution of these tasks, and thereby contributing to the completion of the activity. This problem is a stylized form of the multi-processor scheduling algorithm which has been shown to be NP-Hard. Further complicating the issue is that... **Read complete abstract on page 2.**

Distributed Allocation of Workflow Tasks in MANETs

Rohan Sen, Gruia-Catalin Roman, and Christopher Gill

Complete Abstract:

When multiple participants work on a workflow that represents a large, collaborative activity, it is important to have a well defined process to determine the portions of the workflow that each participant is responsible for executing. In this paper, we describe a process and related algorithms required to assign tasks in a workflow, to hosts that are willing to carry out the execution of these tasks, and thereby contributing to the completion of the activity. This problem is a stylized form of the multi-processor scheduling algorithm which has been shown to be NP-Hard. Further complicating the issue is that we are targeting our approach to mobile ad hoc networks, where hosts are physically mobile, communication links are frequently interrupted, and spatiotemporal considerations become increasingly important. We describe a distributed approach to task allocation in mobile ad hoc networks that employs heuristics to assign tasks in a workflow to mobile hosts based on their capabilities and their mobility patterns. We have implemented our algorithms in the context of CiAN, a workflow management system (WfMS) supporting collaborations in a mobile environment. In addition, we also present performance data of our algorithm and compare it to naive and brute force approaches.

2007-41

Distributed Allocation of Workflow Tasks in MANETs

Authors: Rohan Sen, Gruia-Catalin Roman, and Christopher Gill

Corresponding Author: rohan.sen@wustl.edu

Abstract: When multiple participants work on a workflow that represents a large, collaborative activity, it is important to have a well defined process to determine the portions of the workflow that each participant is responsible for executing. In this paper, we describe a process and related algorithms required to assign tasks in a workflow, to hosts that are willing to carry out the execution of these tasks, and thereby contributing to the completion of the activity. This problem is a stylized form of the multi-processor scheduling algorithm which has been shown to be NP-Hard. Further complicating the issue is that we are targeting our approach to mobile ad hoc networks, where hosts are physically mobile, communication links are frequently interrupted, and spatiotemporal considerations become increasingly important. We describe a distributed approach to task allocation in mobile ad hoc networks that employs heuristics to assign tasks in a workflow to mobile hosts based on their capabilities and their mobility patterns. We have implemented our algorithms in the context of CiAN, a workflow management system (WfMS) supporting collaborations in a mobile environment. In addition, we also present performance data of our algorithm and compare it to naive and brute force approaches.

Type of Report: Other

Distributed Allocation of Workflow Tasks in MANETs

Rohan Sen, Gruia-Catalin Roman, and Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130, U.S.A.
Email: {rohan.sen, roman, cdgill}@wustl.edu

***Abstract*—When multiple participants work on a workflow that represents a large, collaborative activity, it is important to have a well defined process to determine the portions of the workflow that each participant is responsible for executing. In this paper, we describe a process and related algorithms required to assign *tasks* in a *workflow*, to hosts that are willing to carry out the execution of these tasks, and thereby contributing to the completion of the activity. This problem is a stylized form of the multi-processor scheduling algorithm which has been shown to be NP-Hard. Further complicating the issue is that we are targeting our approach to mobile ad hoc networks, where hosts are physically mobile, communication links are frequently interrupted, and spatiotemporal considerations become increasingly important. We describe a distributed approach to task allocation in mobile ad hoc networks that employs heuristics to assign tasks in a workflow to mobile hosts based on their capabilities and their mobility patterns. We have implemented our algorithms in the context of CiAN, a workflow management system (WfMS) supporting collaborations in a mobile environment. In addition, we also present performance data of our algorithm and compare it to naive and brute force approaches.**

I. Introduction

Workflows are useful for modeling large, well-structured activities that can be broken down into smaller tasks and require the participation of multiple hosts (where a host is a mobile device that is carried by a human user). Typically, there are three stages in the life of a workflow: (1) Specification, which is the formulation of the workflow in a machine parseable language, (2) Allocation, which is the assigning of tasks in the workflow to hosts that have the skills to perform those tasks, and (3) Execution, which is the performing of those tasks by the hosts that have been assigned to them and the collation of results. Popular languages for specifying workflows today are BPEL [1] and XLANG [19], among others. The process of allocation and execution is handled by Workflow Management Systems (WfMSs) such as ActiveBPEL [5], BizTalk [2], Groove [3], and Oracle Workflow Engine [11], to name a few. Most WfMSs today are designed to run on enterprise LANs and the workflows they execute are relatively *static*, in the sense that they have rigid semantics that do not change at runtime. For example, business processes, which are arguably the most popular application of workflows today, can have their output influenced only by changes in input and not any other contextual information. In addition, such processes usually run in support of web-based applications and enterprise level systems. However, there is no fundamental restriction in the workflow model that prevents it from being used in more dynamic contexts .

In CiAN [18], we developed a specification that allowed a programmer to formulate workflows that were *dynamic*, i.e., their semantics could change based on context. We also developed a runtime system to execute these dynamic workflows in the setting of a mobile ad hoc network (MANET). However, the algorithm for allocating tasks to hosts that is used by CiAN has two key shortcomings: (1) the algorithm is centralized, which imposed the requirement that all hosts be co-located initially for successful allocation of tasks in the workflow and (2) the algorithm is designed to be run before the execution of the workflow begins, thereby requiring *a priori* allocation of all tasks.

In this paper, we describe a distributed allocation algorithm that addresses these two shortcomings. We take a monolithic workflow and fragment it into smaller "sub-workflows" using a set of pre-defined rules. Each fragment is then assigned to a *local coordinator*, a special host that is responsible for allocating the subset of tasks assigned to it. The allocation process is designed to work in a "just-in-time" manner, with tasks being allocated just before they need to be performed. In addition, we have put in place policies that attract hosts that are not working on a task to the local coordinators which can then assign additional tasks to those hosts from the set of tasks that have yet to be allocated.

This approach is especially beneficial in the dynamic setting of a MANET because: (1) the decentralized approach mitigates the problem of a single point of failure, (2) the just-in-time nature of the algorithm removes the requirement that all tasks be assigned *a priori*, and (3) the policy for attracting available hosts to the coordinators eliminates the need for co-location of all hosts at a single time for allocation of tasks. The remainder of the paper is organized as follows: Section II describes previous work in this area while Section III describes our computational model and the characteristics of MANETs. Section IV desribes our allocation strategy while Section V is a discussion of our assumptions and approach. Section **??** covers implementation details and performance results before we offer conclusions in Section VI.

II. Related Work

A workflow management system (WfMS) has two functions: (1) deciding who does each task in the workflow and (2) transferring data and results between participants in the workflow. The work presented in this paper deals with the first function, i.e., the process of task allocation. Current deployments of WfMSs, such as ActiveBPEL [5] and Oracle Workflow Manager [11], almost always run in a corporate LAN environment or across the Internet where reliable wired connections permit the use of centralized workflow management architectures and instantaneous access to workflow participants. Even WORKPAD [15], which is designed for mobile networks uses a centralized WfMS. Given this environment, the process of allocation is typically handled as follows: the WfMS selects a task to execute. It then looks up a central directory for a suitable software service to perform the task (in some cases even this is not necessary as the services may be hardcoded, e.g., BPEL's `partnerLinks`). Once a service has been found, the WfMS sends the input data to the service and waits for a response. When the response arrives, it selects a service for the next task and repeats the procedure.

In MANETs however, there is neither the opportunity to have a centralized management architecture, nor are the participants always accessible due to wireless links that might break frequently due to host mobility. Hence the allocation process is much more opportunistic in nature, allocating tasks when *a priori*, whenever hosts are within communication range, which is a different approach from those in use today.

In a sense, our task allocation problem is similar to the Job Shop Scheduling problem. In Job Shop Scheduling, a set of jobs is scheduled on a set of machines such that no machine executes more than one job at a time and the total duration for executing the jobs is minimized. In our work, the tasks and hosts are analogous to the jobs and machines respectively. The difference lies in the fact that our primary objective function is to maximize allocation. Minimizing the time required to complete the jobs is only a secondary objective. Also, in addition to jobs being admitted during the scheduling process (the entire workflow is not available for scheduling up front), we also have to accommodate the possibility of additional machines being admitted during the scheduling process. Finally, our approach must also take into consideration the constraints imposed by the physical mobility of hosts and the fact that all jobs cannot be scheduled on all machines.

Job Shop Scheduling has receieved a fair amount of attention from the research community. In [7], the authors describe a heuristic-based method for solving the basic Job Shop Scheduling problem while [6] describes a genetic approach to solving the same problem. More pertinent to our work is [14], which considers the job shop problem with availability constraints. In other words, the set of available machines on which to schedule jobs changes over time. This is analogous to the reachability of hosts changing over time. Another piece of work that is closely related to our effort is reactive Job Shop Scheduling [12] where the schedule is not computed *a priori* but over a period of time. More recently, researchers have considered the use of neural nets to solve the Job Shop problem [20].

Another area from which we draw inspiration is the robot task scheduling. In [9], the authors propose a taxonomy of multi-robot task assignment problems. Our work is closest to the extended time assignment variants of the problem proposed therein. Solutions to this problem involve using a market-based economic model [21], an auction-based approach [8] that uses concept of task utility and fitness of a robot to perform a task to make allocations. Incorporation of spatiotemporal considerations including the formation of organizations and a reward scheme is described in [4] while a scheme for fault tolerant coalition formation is described in [16]. Similar approaches have also been used to allocate resources in wireless sensor networks [13].

In our work, we drew inspiration from each of these bodies of work and developed a scheme that was suited best to our unique environment and requirements. We also ensured that the resulting approach is not overly complex as it is intended for use in MANETs where devices are typically resource constrained. The next section describes our computational model in more detail followed by the description of our algorithm.

III. Computational Model

Before we describe our algorithms, we present a motivating example, and briefly describe our computational model.

A. Motivating Example

Consider a scenario where a bridge is being constructed in a remote area. Workers on the project carry mobile devices (hosts) on their person, on which they receive instructions and task assignments from a sophisticated workflow management system that is responsible for the overall project. Since the project site is in a remote area, the WfMS runs in a distributed fashion over a MANET. Currently, there is only one host that performs task allocations which leads to problems because the host is not always accessible to everyone. Also, because all tasks are allocated *a priori*, the allocation process is not accommodating of emergent events. Realizing the problems with the current approach, the IT support team for the project develops a WfMS that handles task assignments in a distributed fashion. Rather than have one host handle all the allocations, the process is spread out over multiple hosts, with the result that each "subproject" has a coordinator that is responsible for allocating and managing the execution of that subproject. In addition, the coordinators allocate the tasks at the latest possible time so that emergent events are accounted for during the allocation process. With the upgrade to the distributed allocation process, tasks allocated more effectively and the just-in-time behavior ensures that the allocation process is responsive to the environment of the project.

B. Computational Model

For the work presented in this paper, we assume that there exists a *dynamic workflow*, encoded using the CiAN specification, which needs to be allocated and subsequently executed in a well-defined physical space. Each task in the workflow has an earliest start time, a deadline, a duration, the location at which it is to be performed, and a set of qualifications that an host must have in order to perform it. We also assume that there is a closed set of *hosts*, each of which have a set of qualifications. Since we define hosts to be a combination of a human user and a mobile device, the capabilities of a host is the combination of the software capabilities of the user's PDA and human skills of the user, e.g., a metal worker has welding skills. In addition, hosts have a maximum velocity at which they can move, and a schedule. Entries in a host's schedule indicate its commitments. A host's schedule includes commitments related to workflow tasks or external commitments (presumably of a personal nature). Each entry in the schedule consists of a start time, end time, and the location of the host at each of these times. While a host's schedule may not contain workflow-related commitments initially, it may contain personal commitments which must be taken into account when assigning tasks.

A small subset of hosts undertake the function of *coordinators*. Each coordinator is responsible for allocating a part of the overall workflow. Coordinators differ from "worker" hosts in the following ways: (1) they broadcast their schedule so that all worker hosts know where a coordinator is at all times, (2) they advertise a "home" location, which is the location at which they remain unless their schedule requires them to be elsewhere, (3) they do not execute any tasks, instead running the allocation algorithm as their permanent "task". The hosts that serve as coordinators are chosen *a priori*.

Worker hosts in our system can move freely as long as they do not violate certain mobility constraints. During time periods where there is an entry in their schedule, they are required to be at the location dictated by that schedule entry. An entry in the schedule is, in other words, a *commitment* by the host to be at a particular place at a particular time. When they have no entry in the schedule, they are required to gravitate towards the nearest coordinator in search for more work. In other words, we compel the worker host to occupy itself to the maximum possible degree (this is analogous to a human worker completing a task and then returning to a supervisor to be assigned another task.) If a task is allocated to a worker host, it blocks off the time required to perform the task on its schedule, so that it is not assigned another conflicting task.

An host is considered to be qualified to perform a task if it has all the qualifications and is available during the time that the task must be performed. In addition, it must have sufficient time to travel to and from the location of the task without exceeding its maximum velocity and without encroaching on any of its other previously scheduled appointments.

C. MANETs and Related Considerations

A novel feature of the algorithms described in this work is the fact that they are designed to operate in the environment of MANETs, a special class of networks that do not rely on any external or fixed infrastructure. The network infrastructure is borne by the hosts that comprise the network. Due to the physical mobility of hosts, the network topology is dynamic, with communication links being available only intermittently. The transient nature of the links makes multi-hop routes expensive to maintain and prone to failure. Hence, the most reliable way to communicate is by having hosts directly connect to each other when within communication range of each other. This opportunistic style of communication fosters a decoupled style of computing.

Decoupled computing means that hosts may not be reachable at all times. Hence, it is not possible to interact with the host *on-demand*, as is the case with current WfMSs that run in wired networks and have a reliable link to every host. Rather, we are forced to adopt an approach where we develop a short term plan for the host. This plan is then transferred to the host when it is reachable, and is carried out by the host subsequently (including times when it is not reachable). The results are transferred during a subsequent communication opportunity. This makes only a very coarse grained control possible over the host's activities and behavior.

IV. Allocating Tasks to Mobile Hosts

Having presented our computational model and the challenges associated with a MANET environment, we now present our approach for distributed allocation. Our approach can be divided into three main phases: 1) preprocessing steps, which occur prior to the actual allocation process, 2) the core allocation process that is agnostic to whether it is run in a centralized or distributed fashion, and 3) additional resources that distribute the core process and manage the mobility of participating hosts. We describe each phase in turn.

A. Preprocessing Steps

Computing the utility of tasks. Each task in the workflow is assigned a time-dependent utility value that represents how critical it is that the task be allocated. For a task T at time t, its utility $U_T(t)$ depends on (P_T, E_T, t) where P_T indicates how many tasks are in parallel with task T, E_T is the earliest starting time for task T, and t is the current time. The remaining time until a task must be started $(E_T - t)$ is needed to determine the utility so that tasks that start earlier are allocated with higher priority. Similarly, tasks that have no others executing in parallel must be allocated with higher priority since they represent a "bottleneck" in the workflow that must be allocated and completed for the remainder of the workflow to progress. The quantity P_T is a fraction in the range $0 < P_T \leq 1$ and is the inverse of the number of tasks that execute in parallel with task T. To determine the number of parallel tasks, we use a breadth-first search algorithm shown in Figure 1 to mark each task with its *depth in the workflow*, a number that represents the number of preceding tasks that must

For a workflow W with initial task s

```
DETERMINEPARALLELTASKS(s)
    pending ← ⊥
    children ← ⊥
    depth ← 0
    done ← FALSE
    ENQUEUE(pending, s)
    while ¬(done) do
        while ¬EMPTY(pending)do
            ptr ← DEQUEUE(pending)
            if depth[ptr] = ⊥ then
                depth[ptr] ← depth
            else
                depth[ptr] ← MAX(depth, depth[ptr])
            for each c ∈ children[ptr]
                if c ∋ children then
                    ENQUEUE(children, c)
        if LENGTH(children) = 0 then
            done ← TRUE
        pending ← children
        children ← ⊥
        depth ← depth + 1
```

Fig. 1. Algorithm to determine the value of P_T for tasks

be completed before it is ready to execute. Tasks having the same depth in the workflow can execute in parallel. From this, we can determine how many tasks are at a particular depth in the workflow by using a simple graph traversal algorithm. Thus if we know the depth in the workflow of a task, we can easily look up the number of other tasks with the same number and compute P_T The value of E_T is part of the task specification and the value of t can be obtained from the system clock.

Fragmenting the workflow. Since our allocation process is distributed, we must fragment the monolithic workflow we receive as input and allocate each piece in parallel. The number of fragments needed is determined by the number of available coordinators, with each coordinator being assigned one piece of the workflow. We describe two methods for fragmenting the workflow. The user can choose which method is used by way of a parameter to the allocation algorithm.

k-Minimum Cut. We assume that k coordinators are available to allocate tasks. From the algorithm to compute P_T, we know the depth of each task in the workflow and can sort tasks into buckets by their depth. The k-minimum cut approach considers the combined size of adjacent buckets in turn. Cuts are made between the k bucket pairs that have the lowest combined value. The exception to this rule is if any of these cuts result in a fragment having lower than n tasks, where n is a parameter set by the user and has a value less than N/k where N is the total number of tasks in the workflow. In such a case, the next higher cutting point is chosen.

Geographic Cut. Once again, we assume that k coordinators are available. The area in which the workflow is to be executed is divided into k zones. Using a graph traversal algorithm, we examine the task location and put the task specification in a bucket that corresponds to the zone containing the task's location. Thus, the tasks in any fragment are geographically related, i.e., they are in a subset of the total area.

The benefit of the k-Minimum Cut is that it keeps blocks of contiguous tasks under the responsiblity of one coordinator, which is useful when recovering from localized errors (not covered in this paper). However, the geographic cut allows geographically related tasks to be handled by one coordinator. Since each coordinator is responsible for a sub-area of the total area in which the workflow is being executed, the geographic cut allows correlation between the location of the coordinator and the tasks they are allocating.

B. Core Allocation Process

The core allocation process is executed continuously by the coordinator until all tasks have been allocated. Prior to the start of the process, the coordinator sorts the tasks in descending order of utility. Once this is done, it begins the allocation process by formulating and advertising *solicitations* for each task. Interested hosts submit *bids* to perform a task. The coordinator examines the bids and makes a *provisional allocation* of the task to the host that submits the most competitive bid. It then periodically revisits this decision until it is temporally constrained to commit to a particular allocation decision. These steps are described in detail below.

Sorting tasks according to utility. We sort the tasks in ascending order of the difference between the current time and the task's earliest start time. We break ties between tasks having the same difference between the current time and their earliest start time by choosing the task with the higher value of P_T. This sorted list of tasks is placed in a data structure that allows removals only from the head of the list but allows insertions at arbitrary locations provided that they do not violate the utility-based ordering of the tasks.

Formulating and distributing solicitations. Once the tasks are sorted by utility, the coordinator formulates a solicitation for each task which is a 6-tuple of the form $< String : taskName, List : capabilities, Location : taskLocation, Time : duration, Time : start, Time : deadLine >$. Each of the six pieces of information in the solicitation can be obtained from the task specification. More details on this may be found in [18]. The solicitations are placed in a *dynamic task directory*, which is context-aware in the sense that it is notified whenever a worker host comes within range of the coordinator. When such an event occurs, the dynamic task directory creates a direct connection to the worker host that triggered the event, and transmits all the available task solicitations to that host.

Analyzing solicitations and submitting bids. When a worker host receives the list of solicitations from the coordinator, it analyzes them to determine whether it is suited to perform any of the tasks advertised. If so, it submits bids for the tasks it can perform as shown in Figure 2.

For each solicitation, the algorithm checks whether the capabilities required by the task is a subset of the host's capabilities. If so, it checks that host's schedule to ensure that the host does not have any previously scheduled commitments at the time that the task described in the solicitation needs to be performed. This is done using the AVAILABLE function on the host's schedule which returns a boolean value. If this check

Given an host with capabilities C, schedule SC, and capable of a maximum velocity $maxV$ that is analyzing a solicitation list S:

```
ANALYZESOLICITATIONS(C, SC, maxV, S)
    B ← ⊥
    for each s ∈ S
        if capabilities[s] ⊆ C then
            if AVAILABLE(SC, start[s], deadline[s]) then
                precT ← GETPRECEDINGTASK(SC, start[s])
                succT ← GETSUCCEEDINGTASK(SC, deadline[s])
                precV ← (|location[precT] − location[s]|)/
                          (start[s] − deadline[precT])
                succV ← (|location[s] − location[succT]|)/
                          (start[succT] − deadline[s])
                if precV ≤ maxV and succV ≤ maxV then
                    capFrac ← |capabilities[s]|/|C|
                    bid ← {capFrac, precV, succV, maxV,
                              GETDEADLINE(SC, succT)}
                    INSERT(B, bid)
    TRANSMIT(B)
```

Fig. 2. The process by which a host submits a bid for a task

Coordinators order tasks according to:
- how soon they need to be executed
- whether there are other tasks that can be executed in parallel

Hosts submit a bid for a task if:
- they have the skills/qualifications to perform the task
- they can be present at the task location without going back on any previous commitments
- they can travel to and from the task location without going back on any previous commitments

Coordinators provisionally allocate a task to a host if:
- the host is using the highest fraction of its capabilities (among the hosts that have submitted bids)
- the average of its velocity to travel to and from the task is the highest among those hosts that have submitted bids and are using the same fraction of their capabilities

Coordinators bindingly allocate a task to a host if:
- all the conditions for provisional allocation are met
- the time remaining until the start time of the task is below a pre-specified threshold

Fig. 3. Summary of the allocation process

Given a set of sorted buckets with bids B and a sorted list of tasks T, a minimum threshold $minT$, and re-evaluation parameter of n

```
ALLOCATE(T, B, minT, n)
    while T ≠ ⊥ do
        t ← REMOVEFIRST(T)
        bid ← REMOVEFIRSTBLACK(B, t)
        if bid ≠ ⊥ then
            if alloc[t] = ⊥ then
                alloc[t] ← bid
                COLORASGRAY(B, host[bid])
            else
                COLORASGRAY(B, host[MAX(alloc[t], bid)])
                COLORASBLACK(B, host[MIN(alloc[t], bid)])
                alloc[t] ← MAX(alloc[t], bid)
        if start[t]− GETSYSTEMTIME() ≤ minT and alloc[t] ≠ ⊥ then
            NOTIFYHOST(alloc[t])
        else
            notify ← MIN(deadline[alloc[t]], start[t])
            nextCheck[t] ← (1/r)(notify− GETSYSTEMTIME())
            INSERT(T, t)
```

Fig. 4. The allocation algorithm

is successful, then the host is qualified *and* available to do the task. Finally, we factor in travel time. For this, we get the tasks that would immediately precede and succeed the task under consideration, were it to be assigned to this host. This is done using the GETPRECEDINGTASK and GETSUCCEEDINGTASK functions respectively. We then compute the velocity at which the host would need to travel from the location of the preceding task to the location of the task under consideration, and then on to the location of the succeeding task. If both these velocities are lower than the maximum velocity capability of the host, then it is eligible to submit a bid for that task.

Before the submission, it calculates the fraction of its capabilities that it will use in performing the tasks. It then creates a bid, which is a 5-tuple of the form $< double : capabilityFraction, double : precedingVelocity, double : succeedingVelocity, double : maxVelocity, Time : deadline >$. The deadline is computed by the GETDEADLINE function of the schedule, which determines the latest time at which the host must leave the current location so as to have sufficient time to travel to the designated location of a given task before its earliest start time. This bid information is then added to a set. Once all the solicitations are considered, the TRANSMIT function sends all the bids to the coordinator.

Provisional allocations and reallocations. When a coordinator receives a bid, it is placed in the "bucket" that corresponds to the task that it was submitted for. The bids in each bucket are sorted in descending order of the capability fraction of the bid. The capability fraction indicates whether a host is specialized for the task or not. A "jack of all trades" would use fewer of its capabilities for a task than an host that is specialized for the task in question. Sorting the tasks in this manner biases the algorithm to choose more specialized hosts before choosing hosts with broader capabilities, the rationale being that it is desirable to have hosts with broader capabilities available for tasks which may not have specialized hosts. To break ties between bids, we use the average of ratios of the preceding velocity and suceeding velocity to the maximum velocity which gives an indication of how good a fit the task is in the schedule of the host. Higher ratios indicate a more constrained time slot and therefore a better fit in the schedule. When bids are initially inserted into the buckets, they are marked as "black" indicating that the host that submitted the bid is not provisionally allocated to a conflicting task. In the future, as hosts are provisionally allocated for a task, the other bids belonging to the host that are in conflict with that particular provisional allocation are marked as gray.

In parallel with receiving bids, the coordinator runs the allocation algorithm as shown in Figure 4. The algorithm takes the first task from the list of tasks. Due to the ranking of tasks by utility, this task, if allocated, will have the highest effect on the progression of the workflow. It chooses the first bid for that task that is marked black. Since the buckets are sorted, the first bid that is marked as black is the best qualified host that has no other conflicts. This host is then provisionally allocated to perform the task and all other bids

Given a coordinator C, solicitation s, host schedule SC and maximum velocity capability $maxV$

```
ComputeTravelTime(C, s, SC, maxV)
    dist ← |location[C] − location[s]|
    time ← dist/maxV
    if Available(SC, deadline[s], deadline[s] + time) then
        succT ← GetSucceedingTask(SC, deadline[s] + time)
        succV ← (|location[C] − location[succT]|)/
                 (start[succT] − (deadline[s] + time))
        if succV ≤ maxV then
            return true
    return false
```

Fig. 5. Factoring in host mobility for task allocations

submitted by the host that conflict with this allocation are marked "gray". If the task under consideration already has a provisional allocation, the algorithm chooses the better bid using the same criteria that we use to rank bids. If there is a change in the provisional allocation, the conflicting bids are updated accordingly with the new bid's conflicts being marked as "gray" and the old bid's conflicts reinstated as "black". At this point, the coordinator checks whether the current time is within some $minT$ (a parameter to the algorithm) of the earliest starting time of the task. If this is the case, then it makes the allocation final by notifying the host of its newly allocated task. If the current time does not fall within the $minT$ of the earliest start time, the coordinator computes D, the minimum of the deadline advertised by the host in its bid and the earliest start time of the task. It then computes the *reevaluation period* as $(1/r)(D-t)$ where r is a parameter that controls how often a provisionally allocated task is re-evaluated and t is the current time. This task is then re-inserted into the task queue, except this time, its utility is computed as if the difference between earliest start time and the current time is the computed *reevaluation period*. However, it is considered to be of lower utility than another task that may have an actual difference between its earliest start time and the current time that is of the same value. Note that in the process of reinsertion, we may have the value of the reevaluation period be negative, indicating an unallocated task which has passed its earliest start time. These tasks are reinserted at the head of the list of the tasks as top priority potentially starving other tasks. However, this is acceptable because per the workflow model, without completion of the task, subsequent downstream tasks cannot make progress.

The reevaluation of the allocation decision occurs when the a task that is not allocated the first time they are considered is reinserted into the task queue using the scheme described. These tasks eventually come up to the beginning of the task queue where they are considered again. This process continues until a deadline from a host or the constraint of the earliest start time of the task compels a final allocation decision.

C. Accomodating Physical Mobility

Thus far, we have described the algorithm that allocates tasks to interested and qualified hosts. However, given that we operate in a MANET, our algorithm has to be considerate of the physical mobility of participating hosts. When allocating tasks to hosts, we evaluated the suitableness of an host based not only on its qualifications but also on whether it met the spatiotemporal requirements of the task. However, there is one other aspect that we must consider when allocating tasks to hosts–the ability to transfer results after it has finished the task to the host(s) that have been assigned subsequent tasks in the workflow. The preferred method is to transfer the results directly to the intended recipient via a publish-subscribe based protocol described in [18]. However, this may not always be possible due to the lack of a disconnected route [10] (a spatiotemporal series of store and forward hops) between the two hosts in question. In such cases, the source host can attempt to transmit the results to the coordinator using the same publish-subscribe based protocol. If this too is not possible, the source host must physically return to the coordinator and transfer results. The coordinator, once it receives results, stores them until the recipient of those results is within range and then transmits the results to that host.

Since we cannot know of the existence of disconnected routes between hosts *a priori* without knowing their motion profile [17], in our allocation planning, we always assume that the worst case scenario will occur, i.e., the host will need to return to the coordinator. To factor this additional travel during the allocation process, each host adds an additional check in the process for submitting bids, shown in Figure 5. This check is performed just before a bid is added to the list of potential bids that is eventually submitted to the coordinator (see Figure 2). If the check is positive, the bid is added to the list, otherwise the bid is considered invalid and is not added. The additional task simply ensures that the host has sufficient free time to travel back to the coordinator and deposit results without causing a conflict with any other commitments. In this way, we ensure that even while hosts are physically mobile, there is a reliable way for them to exchange data between each other.

D. Distributing the Allocation Process

The final step is to move from what is essentially a centralized allocation process that accounts for mobility to a truly distributed allocation process. This is made possible by using multiple coordinators to allocate a workflow. The transition from one coordinator to multiple coordinators requires three key changes: 1) splitting the workflow into discrete pieces, 2) modifying the behavior of the coordinators, and 3) modifying the behavior of the hosts. These are described in detail below.

Dividing the workflow among coordinators. By definition, the workflow for any activity is a monolithic entity. We assume that initially, a single coordinator has the specification of this monolithic workflow. We refer to this coordinator as the *initiating coordinator*. The initiating coordinator is responsible for fragmenting the workflow using either the k-min-cut or the geographic cut approach. If the k-min-cut approach is used, fragments are assigned randomly to the coordinators. If the geographic cut is used, fragments are assigned such

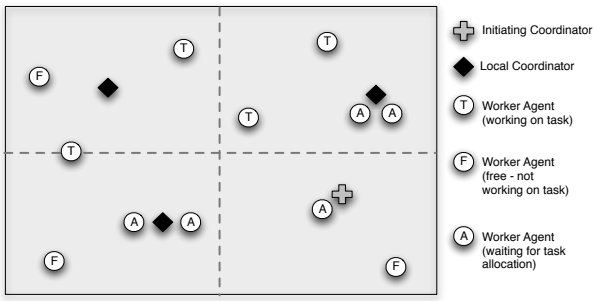

Fig. 6. Hosts during workflow execution

that the coordinators are responsible for allocating tasks in the geographic area in which they operate. Each coordinator, in addition to receiving a fragment of the workflow also receives a table of tasks in the workflow that are not in the fragment allotted to it along with the name of the coordinator responsible for assigning each of those tasks. Once a coordinator receives its fragment of the workflow, it can immediately begin executing the core allocation process as described earlier.

Changes in coordinator behavior. From the coordinator's point of view, the transition from a single coordinator to multiple coordinator requires only two relatively minor changes. The first relates to the bid submission process. If a coordinator has a bid from another host that is better, it immediately rejects the host's bid, which allows it to leave the locality (described in the next paragraph). Second, when calculating the travel time of hosts to return results to the coordinator, the coordinator now has to be aware as to whether the subsequent task that the results are destined for will be allocated by itself or another coordinator. If the task is allocated by the same coordinator, there is no change from the case of the single coordinator. If the task is allocated by a different coordinator, the destination of the travel is changed to be the coordinator that is allocating the subsequent task.

Changes in host behavior. When multiple coordinators are in use, hosts must subtly change their behavior. When hosts have time slots in their schedule that are free, they gravitate towards the *nearest* coordinator. Upon coming within range of a coordinator, they check the solicitations as before. However, if there are no tasks for which they are suited, they immediately leave and go to the next closest coordinator that it has not yet visited. If they do submit bids, the coordinator checks whether the bid is better than the current best choice. If not, the coordinator notifies the host immediately of the failed bid. This ensures that a host is not waiting at a coordinator while other coordinators have tasks that it could perform.

When the system is operating in a fully distributed manner, the distribution of hosts and coordinators may look like Figure 6. Coordinators are essentially stationary and responsible for a fragment of the workflow. Worker hosts gravitate towards a coordinator and remain there if there is a task that needs to be allocated that it can perform competently. If the task is allocated to the host, it leaves, performs the task and then returns to the coordinator. If the task is not allocated, the host may move to another coordinator in search of tasks. The process of searching for tasks and then performing them goes on until all the tasks are completed. At this point worker hosts have no more work to do and they gather around the coordinators. Coordinators then transmit a termination signal that indicates that its portion of the workflow has been completed and shuts down. Eventually all coordinators shut down, indicating to the hosts that the workflow is complete.

V. Analysis and Discussion

Having presented our approach for distributed allocation of workflow tasks in a MANET, we now analyze the computational complexity of our approach and then offer a few comments on our assumptions and design choices.

A. Analysis of computational complexity

In the interest of simplicity, we analyze the computational complexity of our algorithms in three separate stages. The first stage is the preprocessing of the workflow. During this stage, we first compute the number of tasks that execute in parallel with a given task. For this, we start at the root of the workflow graph, and move through it visiting each node exactly once (a check ensures that if two nodes have a common child, the child is not checked twice. Hence this step has a worst case complexity of $O(n)$ where n is the number of tasks. Next, we organize tasks into buckets that correspond to their depth in the workflow. This can be done with a standard traversal algorithm that has complexity $O(n+e)$ where e is the number of edges in the graph. Finally, the minimum cuts are determined by iterating over the buckets organized by depth which in the worst case will be $O(n)$. Thus the overall complexity for the preprocessing is $O(n+e)$.

The next stage is the core algorithm execution. Here, the first step is to sort tasks by utility which in the worst case can take $O(n^2)$ time. The formulation of solicitations is constant time per task and hence has complexity $O(n)$. When hosts submit bids, they are inserted into the correct bucket (the buckets themselves are sorted). In the worst case, searching for the bucket takes $O(n)$ time and inserting it takes $O(h)$ time where h is the number of hosts yielding a total complexity of $O(hn)$. For the actual allocation, each task is reevaluated periodically as per the formula given in Section IV. In the worst case therefore, each task gets reevaluated $log_r(T)$ times where $1/r$ is the fraction of time between the current time and the start time of the task and T is the total time of execution of the workflow. In addition, each time a task is allocated, the algorithm must gray out other bids which takes $O(h)$ time. Hence, the complexity of the allocation algorithm is $O((log_r(T))nh)$ and the total complexity for this stage is $O((log_r(T))n^2h)$.

On the worker hosts, the analysis of each solicitation and the computation of travel can be achieved in constant time with the use of proper data structures and multiple indexes and hence the analysis of solicitations takes $O(n)$ time. It should be noted that the worst case complexities presented are extremely unlikely to occur as the scenarios captured by

Step	Comp-lexity	Total
PREPROCESSING		
Determining Parallel Tasks - The algorithm visits each node in the workflow exactly once.	O(n)	
Fragmenting the graph - Graph traversal to make buckets - Choosing minimum cuts	O(n + e) O(n)	**O(n + e)**
CORE ALGORITHM		
Sorting Tasks by Utility - Sorting of tasks using BubbleSort (once by T - t and then by parallelism	2 * $O(n^2)$	
Formulating solicitations - One solicitation for every task	O(n)	
Collecting bids and keeping them sorted - The list of bids is always maintained in sorted form. Only correct insertion is required	O(hn)	
Allocation and re-allocation - Each task is evaluated atleast once and re-evaluated based on the value of 'n' that is an input to the algorithm	$O((\log_r(T)$ hn)	**$O((\log_r(T)$ $hn^2)$**
PROCESSING BIDS		
Analyzing solicitations + submitting bids - With proper choice of data structure for host schedules	O(n)	
Computing travel time - With proper choice of data structure for host schedules	O(n)	**O(n)**

Fig. 7. Complexity of the allocation process

those cases would make essentially nonsensical workflows. In the sorting of tasks by utility, sequential tasks are already in utility order. The only tasks that may need re-arranging are the ones that occur in parallel, which in most cases is lower than the total number of tasks n. In the bid collection, the analysis is done based on the assumption that every host will submit a bid for every task, an unlikely thing to happen given hosts may not have the capabilities to do all tasks. On the allocation and re-evaluation, the $log_r(T)$ component is an overestimation because it assumes that each task will start at the end of the time interval during which the workflow is going to be executed. In reality, the value of T would be replaced by the start times of tasks which would be half the value of T on average. Also, the h component assumes that all hosts submit a bid for all tasks which is unlikely to occur in practice.

Finally, it should be noted that most of the stages with higher time complexity occur before the core allocation process. The core process itself has a high time complexity but it executes over the duration of the workflow and hence will not suffer from bottlenecks in processor capability.

B. Analysis of our approach and design decisions

The task allocation problem is a stylized version of the multi-processor scheduling algorithm which is NP-Hard. Accommodating ad hoc mobility increases the complexity even further. In order, to keep the problem tractable, we had to make some assumptions and decisions which we discuss here.

Algorithm complexity vs. optimality. For our core allocation algorithm, we adopted a greedy approach with a periodic greedy re-evaluation, which is computationally much simpler but does not always yield an optimal result. The reason for choosing the greedy approach was its "choose and forget" nature, i.e., once we decide to allocate a task, we cannot roll it back. More complex approaches involving rolling back decisions are extremely difficult to achieve in a MANET since hosts may not be reachable and cannot be kept up to date with their allocation and commitment status without the use of distributed transactions which ultimately degrades system performance due to locking of resources and data for extended periods of time.

Contextual information vs. performance. It is generally acknowledged that the motion of hosts participating in a MANET may be completely arbitrary, and as such, it is difficult to compute when hosts will be within communication range of each other unless they advertise their motion pattern. In [17], the authors assume that hosts provide *motion profiles* that describe their location over time. However, in most situations, it is unlikely that we would have access to a complete motion profile *a priori*. Hence, we chose a more practical abstraction for motion information - a personal schedule. However, the downside of using a schedule is that it does not capture opportunistic meetings between hosts while they are in between tasks. Hence, our algorithm cannot exploit these meetings. We chose to reduce the dependence on accurate motion information (which is hard to come by in practice) over an optimization to our approach.

Reasonable behavior of host. In our work, we assume that the host can mark off time on its schedule when it is busy, and must return to the coordinator when it has nothing on its schedule. This is very similar to a workers returning to their supervisor for additional work when they have completed a task, with the rationale being that a worker does not get paid if they do not work a certain number of hours. In addition, the idea that a supervisor may have a degree of control over where a worker goes during work hours is not beyond the realm of possibility. A second assumption we made is that the schedule of the coordinators is known *a priori* and that the coordinators are relatively static as compared to the worker hosts. Once again, this is analogous to common command and control structures where the supervisor is always reachable and usually does not participate extensively in the actual work, instead managing the activity from a static location. We chose this model because we have targeted our system to support collaboration among humans as well as software services. Thus the mobility of the hosts is controlled by the human user to which the mobile device belongs and the mobility of the user is dictated by standard command and control protocols. Thus, in developing our algorithms, we can rely on certain basic behavior patterns which makes the problem much more tractable and allows solutions targeted to reasonable mobility patterns as opposed to completely arbitrary ones.

VI. Conclusion

Due to the volatility of MANETs and the lack of centralized infrastructure, it is desirable to have key processes within a WfMS operate in a distributed fashion. We have described a process by which a monolithic workflow is divided into smaller pieces and then allocated in a distributed fashion by multiple coordinators in a MANET setting. Our approach combines a bidding scheme with measures of utility and fitness to make allocation decision. Our algorithm also supports revision of previous decisions as long as a commitment has not been made for a host to perform a particular task.

References

[1] WSBPEL Committee. Web services business process execution language v2.0. http://www.oasis-open.org/\\committees/download.php/18714/wsbpel-specification-draft-May17.htm, 2006.

[2] Microsoft Corp. The biztalk server. http://www.microsoft.com/biztalk/.

[3] Microsoft Corp. Groove virtual office. http://www.groove.net/home/index.cfm.

[4] T. S. Dahl, M. J. Mataríc, and G. S. Sukhatme. Adaptive spatio-temporal organization in groups of robots. In *Proc. of the Intl. Conf. on Intelligent Robots and Systems*, pages 1044–1049, 2002.

[5] Active Endpoints. ActiveBPEL engine. http://www.active-endpoints.com/active-bpel-engine-overview.htm.

[6] H.-L. Fang, P. Ross, and D. Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In *Proc. of 5th Intl. Conf. on Genetic Algorithms*, pages 375–382, 1993.

[7] A. Garrido, M.A. Salido, F. Barber, and M.A. López. Heuristic methods for solving job-shop scheduling problems. Technical report, Universidad Politécnica Universidad Politécnica de Valencia, 2000.

[8] B. P. Gerkey and M. J. Mataríc. Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, 18(5), October 2002.

[9] B. P. Gerkey and M. J. Mataríc. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, pages 939–954, September 2004.

[10] R. Handorean, C. Gill, and G.-C. Roman. Accommodating transient connectivity accommodating transient connectivity in ad hoc and mobile settings. In *Proc. of Pervasive 2004*, number 3001 in LNCS, pages 305–322, 2004.

[11] Oracle Inc. Oracle workflow. http://www.oracle.com/technology/products/integration/workflow/workflow_fov.html.

[12] N. Liu, M. A. Abdelrahman, and S. Ramaswamy. A multi-agent model for reactive job shop scheduling. In *Proc. of the 36th Southeastern Symposium on System Theory*, pages 241–245, 2004.

[13] G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation*, pages 23–33, 2005.

[14] P. H. Mauguiere, J.-C. Billaut, and J.-L. Bouquard. New single machine and job-shop scheduling problems with availability constraints. *Journal of Scheduling*, 8:211–231, 2005.

[15] M. Mecella, T. Catarci, M. Angelaccio, B. Buttarazzi, A. Krek, and S. Dustdar. Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In *Proc. of the IEEE Intl. Symposium on Collaborative Technologies and Systems*, May 2006.

[16] L. E. Parker. Alliance: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

[17] R. Sen, R. Handorean, G.-C. Roman, and G. Hackmann. Knowledge driven interactions with services across ad hoc networks. In *Proc. of the 2nd Intl. Conference on Service Oriented Computing*, pages 222–231, 2004.

[18] R. Sen, G.-C. Roman, and A. Frank. Cian: A language and middleware for collaboration in ad hoc networks. Technical Report WU-CSE-2006-51, Washington University in St. Louis, 2006.

[19] S. Thatte. Xlang: Web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[20] S. Yang. Job-shop scheduling with an adaptive neural network and local search hybrid approach. In *Proc. of Intl. Joint Conf. on Neural Networks*, pages 2720–2727, 2006.

[21] R. Zlot, A. Stentz, M. Dias, and S. Thayer. Multi-robot exploration controlled by a market economy. In *Proceedings of the IEEE International Conference on Robotics Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.