Report Number: WUCSE-2006-18

2006-01-01

MobiWork: Mobile Workflow for MANETs

Gregory Hackmann, Rohan Sen, Mart Haitjema, Gruia-Catalin Roman, and Gill

The workflow model is well suited for scenarios where many entities work collaboratively towards a common goal, and is used widely today to model complex business processes. However, the fundamental workflow model is very powerful and can be applied to a wider variety of application domains. This paper represents an initial investigation into the possibility of using workflows to model collaboration in an ad hoc mobile environment. Moving to a mobile setting introduces many challenges as the mobility of the participants in a workflow imposes constraints on allocation of workflow tasks, coordination among participants, and marshaling of results. We... **Read complete abstract on page 2.**

MobiWork: Mobile Workflow for MANETs

Gregory Hackmann, Rohan Sen, Mart Haitjema, Gruia-Catalin Roman, and Gill

Complete Abstract:

The workflow model is well suited for scenarios where many entities work collaboratively towards a common goal, and is used widely today to model complex business processes. However, the fundamental workflow model is very powerful and can be applied to a wider variety of application domains. This paper represents an initial investigation into the possibility of using workflows to model collaboration in an ad hoc mobile environment. Moving to a mobile setting introduces many challenges as the mobility of the participants in a workflow imposes constraints on allocation of workflow tasks, coordination among participants, and marshaling of results. We present an algorithm that heuristically allocates tasks to participants based on their capabilities and mobility and discuss the architecture and implementation of MobiWork, our prototype system that allocates and executes workflows in an ad hoc mobile environment. An evaluation of the performance of our heuristic algorithm is also presented.

2006-18

MobiWork: Mobile Workflow for MANETs

Authors: Gregory Hackmann, Rohan Sen, Mart Haitjema, Gruia-Catalin Roman, Christopher Gill

Corresponding Author: rohan.sen@wustl.edu

Abstract: The workflow model is well suited for scenarios where many entities work collaboratively towards a common goal, and is used widely today to model complex business processes. However, the fundamental workflow model is very powerful and can be applied to a wider variety of application domains. This paper represents an initial investigation into the possibility of using workflows to model collaboration in an ad hoc mobile environment. Moving to a mobile setting introduces many challenges as the mobility of the participants in a workflow imposes constraints on allocation of workflow tasks, coordination among participants, and marshaling of results. We present an algorithm that heuristically allocates tasks to participants based on their capabilities and mobility and discuss the architecture and implementation of MobiWork, our prototype system that allocates and executes workflows in an ad hoc mobile environment. An evaluation of the performance of our heuristic algorithm is also presented.

Type of Report: Other

MobiWork: Mobile Workflow for MANETs

Gregory Hackmann, Rohan Sen, Mart Haitjema, 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.
{ghackmann, rohan.sen, mart.haitjema, roman, cdgill}@wustl.edu

ABSTRACT

The workflow model is well suited for scenarios where many entities work collaboratively towards a common goal, and is used widely today to model complex business processes. However, the fundamental workflow model is very powerful and can be applied to a wider variety of application domains. This paper represents an initial investigation into the possibility of using workflows to model collaboration in an ad hoc mobile environment. Moving to a mobile setting introduces many challenges as the mobility of the participants in a workflow imposes constraints on allocation of workflow tasks, coordination among participants, and marshaling of results. We present an algorithm that heuristically allocates tasks to participants based on their capabilities *and* mobility and discuss the architecture and implementation of MobiWork, our prototype system that allocates and executes workflows in an ad hoc mobile environment. An evaluation of the performance of our heuristic algorithm is also presented.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Algorithms, Design

Keywords

Mobile Ad hoc Networks, Workflow Management, Software Architecture, Algorithms

1. INTRODUCTION

Groupware is a special class of systems that support and facilitate collaboration among people and stand-alone software services. Groupware is becoming increasingly relevant in today's world given the growing need for teams of people to collaborate with each other effectively while working on a common project. Workflow Management Systems (WfMSs) [18] represent a type of groupware that has been widely successful, especially in the domain of business computing as evidenced by popular standardized languages such as BPEL [2] and WfXML [6]. WfMSs are based on the *workflow* model, which is informally defined in Wikipedia as "the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks and how tasks are tracked" [7]. In other words, WfMSs coordinate and oversee the performance of tasks by multiple active agents (people and/or software services) that result in the realization of a common goal.

Traditionally, WfMSs have been used to model business processes such as expense authorization, loan approval, and insurance claim processing, to name a few. While these processes require the input of multiple agents, they are considered fairly static, in that the group of agents collaborating on such a process remains the same for long periods of time and the execution of the workflow is seldom affected by external conditions. Additionally, the software that supports the execution of such workflows is designed to work on high-end servers connected by reliable wired connections.

This paper represents a first effort to apply the workflow model to a more dynamic setting, specifically that of mobile devices that interact with each other using a Mobile Ad hoc Network (MANET). The benefit of engineering a workflow based software system for mobile devices is that it opens the door to a vast new application domain where collaboration among individuals in the physical world is supported and aided by collaborative software that runs on PDAs, cellular phones, and other such devices, and is not constrained by the physical limitation of a wired network. For example, consider a construction site where several tasks must be completed by a group of workers in a particular order to build a variety of structures. In such an environment, there is no scope for setting up a centralized server and a set of connected workstations for the workers to access. Rather, it is much more practical for workers to carry small, ruggedized PDAs that execute and manage the workflow in a distributed manner. When it is time for a worker to do a particular task, his or her PDA would play an audible alert and display the details of the task on the screen. Once the work has been completed, the worker can use the PDA to acknowledge completion of the task, attaching a report if required. The underlying system would then file the data and notify all relevant parties of the completion of the task, which may trigger initiation of the next task in

the workflow and also inform the supervisor of the status of that task. This approach allows the workers to focus on the actual tasks at hand, leaving the system to absorb the collaboration overhead.

Developing a WfMS targeted to a MANET setting is challenging because the fundamental workflow model relies heavily on reliable and always available connections to coordinate the distributed agents that take part in a workflow's execution. In MANETs however, the network topology changes rapidly resulting in frequent disconnections between participants in the workflow and reducing the scope for coordination among them. In addition, tasks are likely to be constrained by location and time, e.g., "complete wiring on the 4th floor between 3PM and 4PM" cannot be done at any other location at any other time. Given such constraints and the dynamic execution environment, we were compelled to re-examine the components of a WfMS in the context of a mobile environment. Our investigations led to the development of a basic WfMS architecture that retains the salient features of current WfMSs and adds additional components to support mobility and disconnected operation.

The contributions of this paper can be summarized as follows: (1) an allocation algorithm that assigns tasks in the workflow to mobile hosts, using a dynamic constraint list associated with each agent to choose agents based on their capabilities *as well as* their ability to satisfy the spatiotemporal constraints of the task at hand; (2) a software architecture for WfMSs in mobile settings based on the concept of knowledge exploitation, using freely-traded attributes of agents to make allocation and execution decisions, and (3) a Java implementation and simulation results of the performance of our allocation algorithm.

The remainder of this paper is organized as follows. Section 2 covers the basic aspects of MANETs and WfMSs as well as related work. In Section 3 we introduce our workflow model for mobile settings, and describe the algorithms that we use to allocate tasks to entities that will perform them. Section 4 describes the architecture and selected implementation details of our system. We evaluate our approach in Section 5, and offer concluding remarks in Section 6.

2. BACKGROUND

Since this paper is concerned with providing workflow in MANETs, in this section we present a brief overview of workflow and MANET technologies.

2.1 Workflow Management

Workflows are a powerful model for describing collaboration between individuals which has traditionally been very popular in business contexts. Owing to this popularity, a workflow has commonly been defined as the automation and management of a business process where a business process is the sequence of tasks which must be done to achieve a certain business goal [13]. The structure of a workflow can be conceptualized as a directed graph where the nodes represent the individual tasks and the edges impose the relevant ordering between the tasks. To provide a basic example, the automation of a loan request represents a workflow since an ordered sequence of tasks, such as credit checks and paperwork, must be performed by various individuals to process the request. Although such business processes are common applications for workflow, the fundamental workflow model itself is general and can be applied whenever a group of individuals must perform a set of tasks in some order to achieve a common goal.

While a workflow defines how individuals collaborate by describing the ordered series of tasks that must be performed, the WfMS is the software system that actually supports the execution of the workflow. A WfMS accepts as input a computerized representation of the workflow, called the workflow specification, and manages the workflow, which includes delegating tasks to be performed. Tasks defined in the workflow specification are simply descriptions of the work that needs to be performed along with the constraints on its execution. At the heart of the WfMS is the workflow management engine which is responsible for allocating agents (such as a human user or automated service) to perform tasks, and for providing the appropriate interfaces to call the required services or user interactions to support the execution of these tasks.

Workflow Management Systems have evolved from isolated legacy systems designed to automate processes for individual businesses, to systems that support workflow in a broader scope. In particular, the Web services community, fueled by the e-commerce revolution, have used the concept of a workflow as a tool for composing existing Web service infrastructure into orchestrated or choreographed distributed applications. Several standardized workflow specification languages such as WS-CDL [11], Wf-XML [6], and BPEL [2] have surfaced to allow tasks to be defined in terms of Web services descriptions. Commercial [3, 5] and open source [1] workflow management engines then match Web services to tasks to execute the workflow. While these WfMSs allow tasks to be executed across geographic and organizational domains, Web services depend on reliable permanent connections and are not designed with mobility in mind.

Recently some systems have been developed to address workflow in mobility explicitly. A series of systems such as Exotica/FMDC [8], DOORS [16] and ToxicFarm [10], adapt workflow to mobility by supporting workflows in the face of network disconnections. Clients in these systems hoard the needed data from a centralized server before they disconnect from the network. Clients may then continue to perform their task(s) while disconnected and the server merges any changes upon reconnection. These systems, therefore, rely on some fixed network infrastructure and assume disconnections are temporary. They also do not exploit the potential for collaboration among clients which are not connected to a central server but which may communicate directly with each other. Another approach to workflows in mobile settings has been through the use of mobile agent technology. The Agent-based Workflow Architecture (AWA) [17] is a workflow system that consists of mobile Task Agents which can migrate to mobile devices to execute workflow tasks. The task execution may occur while the device is disconnected provided the Task Agent eventually has the opportunity to migrate back to a Workflow Agent which oversees the execution of the workflow. This agent-based approach is more flexible and hence more appropriate for dynamic settings but its single point of failure (the Workflow Agent) makes it undesirable for MANETs.

2.2 MANETs

A MANET is a wireless network which is formed opportunistically by wireless, physically mobile devices called hosts. These hosts act as routers and communicate with

each other via wireless radios, typically with a very limited range. The network is decentralized and does not rely on any fixed infrastructure. The hosts, therefore, must discover the network topology for themselves and must form routes to other hosts, perhaps in a multihop fashion. Since hosts move freely, the network topology may change rapidly and unpredictably. Disconnections occur frequently and hosts may join and disconnect at any time as they move in and out of communication range of other hosts in the network.

The goal of our work is to merge the concepts of workflows and MANETs, and this represents a mostly unexplored area of research. One emerging system is WORKPAD [15], which supports workflows in MANETs but it is designed specifically for emergency/disaster scenarios. WORKPAD operates by centrally coordinating the activities of small teams which perform various disaster recovery activities. These teams operate in separate MANETS but the system assumes that hosts in each MANET operate in tight proximity with a central coordinating host that predicts disconnections and reallocates tasks to other agents or replans the workflow when disconnections occur. Additionally, WORKPAD assumes that the coordinator in each MANET maintains a reliable satellite connection to a central system of P2P servers.

Having presented some of the basic concepts of workflows, MANETs, and work related to our own, we now present the technical aspects of MobiWork. In particular, the next section describes an algorithm to allocate workflow tasks, given the dynamic aspects of MANETs.

3. ALLOCATING TASKS IN THE PRESENCE OF MOBILTY

For the purposes of this paper, we assume a scenario where a group of individuals (*group members*) carrying mobile devices comes together at the beginning of the day to work together to complete some activity. The various tasks that must be completed as part of the activity are specified in a pre-defined workflow, which is loaded onto the PDA of the *group leader*. Initially, all people involved in the workflow (and therefore the mobile *hosts* they carry on their person) are co-located. Each mobile host can run multiple software processes, which we refer to as *agents*. However, for simplicity of presentation, we assume that each host runs only one agent, thereby eliminating the distinction between hosts and agents, which we use interchangeably in the remainder of this paper. Thus the planning stage, where individual tasks in the workflow are assigned to agents on group members' hosts, can occur in a centralized fashion with the group leader in charge of running the allocation algorithm. Once the allocations are complete, the group disbands to do their assigned tasks. We also assume that the mobility of the hosts is *flexible*. By this we mean that a host can be directed to a certain location at a certain time to perform a particular task as per the requirements of the workflow: for example, a construction foreman could send a worker to the 5th floor to test the electrical fittings. However, agents, which are the software processes running on the hosts have their mobility pattern constrained to be identical to that of the mobile host on which they are executing. Finally, should an error occur after the group disbands, then any replanning required to re-assign tasks occurs in a decentralized manner using only the knowledge available in the knowledge base of the *replanning host*. In the remainder of this section, we describe our basic mobile workflow model, and an algorithm that allocates tasks to agents taking into account the fact that the workflow executes in a mobile setting.

3.1 Basic Mobile Workflow Model

We use a simple mobile workflow model to illustrate our algorithms and our system architecture and its implementation. We chose a simple model so that we could reason about the implications of mobility on WfMSs without being encumbered by implementation specific complications. In this section, we briefly describe the model that we use and then describe our algorithm to allocate tasks to hosts.

In our model, we conceptualize the workflow specification for any activity as an annotated, directed graph that we call a *plan*. The nodes in the graph represent the *tasks* that need to be completed, while the edges impose an ordering among these tasks. Each task has several attributes: (1) a *task identifier* which is unique in the scope of the plan, (2) a *qualifications list* which describes the qualifications an agent must have to perform a particular task (for simplicity, we assume that an agent must meet all the qualifications listed to be considered as a candidate to perform a task), (3) a *location* at which the task must be performed, (4) a *start time* which represents the earliest time at which the task can be started, (5) an *end time* which represents the deadline by which the task must be completed, (6) a list of inputs specifying the type of the input and the task from which the input will arrive, and (7) a list of outputs specifying the type and recipient task. Each node in the plan has two annotations, *allocation* and *completion status*. The allocation annotation indicates the agent to which a task has been allocated while the completion status indicates whether a task is waiting to be started, in progress, completed, or in an error state.

The mobile agents that participate in completing the tasks in the workflow also conform to certain requirements. Each mobile agent has (1) a unique identifier within the scope of the group that is working on a particular plan, (2) a list of qualifications that it possesses, and (3) the maximum speed at which it can move. These qualifications determine whether an agent is capable of completing a particular task.

3.2 Allocation Strategies

The key to a successful execution of a workflow is to find group members that can do the various tasks associated with the workflow. The process of assigning tasks to group members is referred to as task allocation. Task allocation takes on added significance in a MANET setting since proper allocation results in fewer errors and therefore fewer instances where expensive re-planning is required. We now describe our allocation algorithm that takes into account the qualifications of group members and mobility issues to create well-formed allocations for workflows.

A well-formed allocation is a mapping of actions in the graph to agents that can carry them out. Each action in the graph may specify a set of requirements that agents must meet in order to perform the action, and each agent advertises its qualifications. We say that an agent may carry out an action if its qualifications are a superset of the action's requirements. For example, consider a construction job that requires a supervisor to inspect the work site before the job is officially completed. The workflow for this job will contain an "inspect" action that requires the corresponding agent to have the appropriate job qualifications.

Action 1	
Agent A	2, 3
Agent B	3, 4
Agent D	3, 4, 5

Action 2	
Agent A	1, 4
Agent B	3, 4
Agent C	4, 5

Figure 1: Example constraint tables

Well-formed allocations are also subject to a series of *constraints*. Constraints disallow certain undesirable allocations, and can be divided into two categories. First, *agent constraints* are constraints on agents' behavior that may change during the course of the allocation algorithm. Examples of agent constraints include:

- If Agent A is allocated to Action 1, then it must also be allocated to Action 2.
- If Agent B is allocated to Action 2, then it cannot also be allocated to Action 3.
- Agent C must be allocated to Action 4.

Second, *spatiotemporal constraints* are often a product of mobility, and do not change as the graph is allocated. Examples of spatiotemporal constraints include:

- One agent cannot be allocated to two actions whose start and end times overlap.
- One agent cannot be allocated to two actions with location constraints that are infeasible. That is, if two actions are separated by time t and distance d, then only an agent with a maximum speed of at least $\frac{d}{t}$ can be allocated to both.
- Two different agents cannot be allocated to two sequential actions if the second agent cannot reach the first agent in time to receive and use its results.

For the purposes of illustration, we will only consider spatiotemporal constraints in this section. Agent constraints can be incorporated without changing the algorithm, by representing them as functions of the current state of the allocation.

We represent constraints as 3-tuples $< a_1, A, a_2 >$, which indicate that the agent A cannot be allocated to action a_2 if it is also allocated to action a_1. This uniform representation simplifies the allocation algorithm, as discussed below. For the sake of simplicity, we assume that all constraints are symmetric, i.e., that the constraint $< a_1, A, a_2 >$ implies the constraint $< a_2, A, a_1 >$. This restriction could be lifted with a few additions to our algorithm: when we collect these constraints into tables as described below, we must add annotations to indicate the "direction" of asymmetric constraints and take these directions into account when deciding if an allocation will violate a constraint. These changes are conceptually straightforward but add complexity to our basic algorithm, and hence we do not discuss them here.

Once these constraints are established, we collect them in a series of tables. For each action in the graph, we create a table like the ones shown in Figure 1. We fill the first column with the agents that are capable of performing the action, constraints non-withstanding, and place an empty list in the second column. The first column can be generated using an existing agent matchmaking scheme, e.g., if agents' capabilities and actions requirements are expressed using a uniform ontology such as OWL-S [14], then matching actions to capable agents is straightforward [12]. Then, for each constraint $< a_1, A, a_2 >$, we check the table for action a_1 to see if it has a row for agent A. If it does, then we add a_2 to the corresponding list in the second column. If not, we ignore the constraint, since it is impossible to violate it.

Once the tables are populated, two important kinds of data can be derived easily from them. First, for each action, we have a list of agents to which it can be allocated. Then, for each agent to which we can allocate a given action, we have a list of future allocations that are made impossible by that decision. For example, according to the first table in Figure 1, we can allocate Action 1 to Agents A, B, or D. If we choose to allocate Action 1 to Agent A, then we can never choose later to allocate Actions 2 or 3 to the same agent.

We will now discuss how these tables are used to perform the actual allocation. First, we will consider a simple algorithm that simply iterates through all possible allocations until a suitable one is found. Then, we will discuss some modifications and heuristic refinements to this naïve algorithm which can greatly improve its average case performance.

```
void createConstraintTables(actions, agents)
    for each A in actions
        for each G in agents
            if G.capabilities ⊆ A.requirements
                conflicts := computeConflicts(G, A)
                addRow(A.table, [G, conflicts])

boolean allocateAction(actions, allocation, n)
    A := actions[n]
    for each row [G, conflicts] in A.table
        for each C in conflicts
            if [C, G] ∈ allocation
                next row

        if n = |actions| or allocateAction(actions, allocation, n + 1)
            allocation := allocation ∪ [A, G]
            return true

    return false

map allocate(actions, agents)
    allocation := ∅
    createConstraintTables(actions, agents)

    allocateAction(actions, allocation, 1)
    return allocation
```

Figure 2: Psuedo-code for naïve allocation algorithm

3.2.1 Naïve Algorithm

The naïve algorithm, which is shown as pseudo-code in Figure 2, begins by populating the constraint tables as described above. The tables are placed in a list, sorted by the actions' IDs. The algorithm then iterates through the list as follows.

For each action, it selects the first agent in the corresponding table, and marks its row in the table. This indicates that it is attempting to allocate this action to the first agent. Next, it collects the agent's conflict list from the second column. For each conflicting action, it consults their tables to find which row has been marked. (It ignores actions whose IDs are greater than the current action's, since they have not yet been allocated.) If none of these actions have been allocated to the agent that it has just selected, then it moves on to the next action.

If at least one of these actions has already been allocated, then it has just violated a constraint. It un-marks the row it just marked, selects the next agent in the table, and marks that agent's row instead. Then, it repeats the

conflict-checking procedure described above with the newly-selected agent. The algorithm continues iterating through the agents until an acceptable one is found.

If the entire list of agents is exhausted without finding an acceptable one, then it has made an error earlier in our algorithm. It un-marks all of the rows in the current table, and returns to the previous table in the list. Again, it un-marks the previous table's current row and proceeds to the next agent.

This algorithm enumerates all possible action/agent pairs until a well-formed allocation is found. At each step the algorithm verifies that our decision will not violate any constraints. Therefore, any allocation that our algorithm provides will be valid; and if any valid allocation of all actions exists, our algorithm will eventually find it. However, since the algorithm blindly iterates through all actions and agents in no particular order, its performance is often poor. If there are m agents and n actions in a workflow, then the algorithm may consider $O(n^m)$ allocations before finding a well-formed one. This is unacceptable for all but the smallest plans.

We now discuss some enhancements to this algorithm which aim to combat this poor performance. In the worst case, this enhanced algorithm may still enumerate all possible allocations until an acceptable one is found; so its asymptotic performance is also $O(n^m)$. However, the enhancements direct the algorithm towards the most fruitful decision paths first, greatly improving its average case performance.

```
boolean enhancedAllocateAction(actions, A, allocation)
    for each row (G, conflicts) in A, ordered by |conflicts|
        if |conflicts| = 0
            allocation := allocation ∪ (A, G)
            return true

        myToken := new AllocationToken(A, G)
        push(stack, myToken)
        allocation := allocation ∪ (A, G)

        for each C in conflicts
            if C ∉ allocation
                push(stack, new DisableRowToken(C, G))
                disableRow(C.table, G)

        for each C in conflicts, ordered by |conflicts|
            if C ∉ allocation
                if not enhancedAllocateAction(actions, C, allocation)
                    do
                        token := pop(stack)
                        undo(token)
                    until token = myToken

                    push(stack, new DisableRowToken(A, G))
                    next row

        return true
    return false

map enhancedAllocate(actions, agents)
    allocation := ∅
    createConstraintTables(actions, agents)

    for each A in actions, ordered by |A.table|
        if A ∉ allocation
            enhancedAllocateAction(actions, allocation, A)
    return allocation
```

Figure 4: Psuedo-code for enhanced allocation algorithm

3.2.2 Enhanced Algorithm

The enhanced algorithm, shown in Figure 4, begins by populating the constraint tables in the same way as the naïve algorithm. The algorithm then places the actions in a list, sorted in ascending order by the number of agents that can carry each action out. Then, it sort the rows in each table by the number of conflicts in the second column.

The algorithm begins by picking the first unallocated action in the list. Again, it selects the first agent in the corresponding table. If this agent has no conflicts, then it marks this row in the table, and continues to the next unallocated action.

If the first agent has at least one conflicting action, then we must begin a "sub-algorithm" to handle this action. The sub-algorithm creates a stack which represents its decisions, as shown in Figure 3. It marks the first agent's row in the table, and pushes a token onto the stack reflecting this marking. Next, it collects the list of conflicting actions from the table. For each conflicting action, its grays out its row in the corresponding action's table, and pushes onto the stack a marker that represents this change. When it later visits these actions, it will disregard all the rows that have been grayed out, since they reflect decisions that would violate a constraint. The sub-algorithm then recursively attempts to allocate all the actions whose tables it has just modified. Once all conflicting actions have been recursively allocated, it returns to the original list of actions and continues sequentially allocating them as before.

While the sub-algorithm executes, it may encounter an action with no capable agents left. This means that it has made an incorrect decision earlier, and that it must roll its state back to that place. It does this by popping elements off the stack, undoing the changes that they represent, until it reaches a change to a table that marked one of at least two remaining rows. This indicates a place where it made a decision that may have been incorrect. It un-marks the agent chosen at this point, and grays out its row so that it doesn't try that agent again. (Again, it pushes a token onto the stack to reflect the row that has just been grayed out.) Finally, the sub-algorithm attempts to re-allocate the action to the next un-grayed agent in the table.

This algorithm improves on the naïve algorithm's performance in two significant ways. First, rather than allocating actions in an arbitrary order, it first allocates the actions that are hardest to satisfy; also, these actions are first allocated to the agents that will cause the fewest conflicts later. This will reduce the amount of backtracking that the algorithm must do, since it will first consider the paths that are least likely to cause irresolvable conflicts. Second, the improved algorithm recurses through agents' conflict lists, effectively dividing the workflow into sub-workflows. Because of the order that the algorithm recurses through actions, it is guaranteed to first consider the entire "conflict closure" of an action, i.e., all the actions that recursively conflict with it. Since by definition actions in one closure cannot conflict with actions in another closure, they are allocated completely independently of each other. So, once a closure has been fully allocated, the algorithm will never revisit any of the actions in it; this greatly reduces the cost of backtracking.

Note that neither of these algorithms considers the actual data flow when computing a well-formed allocation. This decision has two implications. First, the constraint that two agents must "meet up" before exchanging data becomes more complex to describe when one agent must receive results from multiple predecessors. This constraint can be

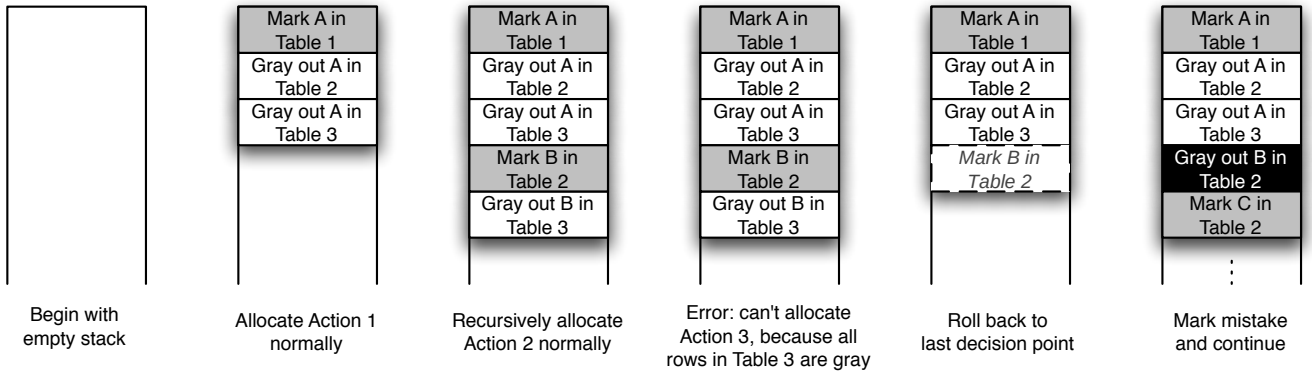

Figure 3: An example allocation stack, and how it is used to track and roll back changes

simplified by requiring that all nodes that join to a common node on the graph must take place in the same physical location. This behavior can be enforced by adding "move to a common location" actions to all the paths immediately before the join point.

Second, our allocation is conservative: we assume that all actions in the workflow will be executed, even though the workflow may split into multiple, mutually-exclusive paths. Thus, valid allocations may exist which do not execute all actions, and which our algorithm will not find. This shortcoming can be worked around by enumerating all possible traces through the workflow and attempting to allocate each trace individually until one feasible allocation is found. As we show in Section 5, the cost of running the enhanced algorithm is low enough to make this approach feasible. Nevertheless, in future work we may consider ways to better incorporate data flow information into our algorithm's decisions.

4. THE MOBIWORK SYSTEM

Having described our basic mobile workflow model and allocation algorithm, we now describe the architecture of MobiWork, which is the system that implements the model and algorithm. This section is organized as follows: we first present details of the system architecture, highlighting the separate planning and execution infrastructure and how they work together. This is followed by descriptions of sample runs of both the planning and execution activities and select details of our Java implementation.

4.1 System Architecture

Recall that in Section 3 we made the assumption that the group leader acts as a central planner and assigns tasks to group members who are then responsible for the performance of those tasks. This assumption ordinarily motivates two discrete architectural designs, one targeted to the planning role to be used by the group leader, and another targeted towards execution of tasks, to be used by group members. However, given that we are in a MANET setting, we cannot rely on the central planner beyond the initial planning stage. Thus, in the case of errors during the execution of tasks, all re-planning is conducted in a distributed manner. Thus, each host must possess the infrastructure to (re)allocate tasks as well as to execute them. Figure 5 shows the architecture of the MobiWork system that runs on the PDAs of all group members as well as the group leader. The system is differentiated into four kinds of components based on their roles: (1) planning components, (2) execution components, (3) common components, and (4) external components. We present details of the external components first, before describing the common, planning, and execution components.

External Components. The external components shown in the figure represent components that are not part of the MobiWork implementation but form an integral part of the overall system. The *Planning Application* is a special application that is used only by the group leader during the initial planning stage. The planning application is used to inject the plan for the activity to be completed, into the system. The injection of the workflow begins the bootstrapping process for MobiWork. The *Monitoring Application* can be instantiated optionally by the group leader to monitor the progress of the tasks in the plan. The monitoring application gets all completion information from the Workflow Manager. It should be noted however, that the completion information may not be up-to-date all the time, as information about group members that are disconnected from the remainder of the group may not update until they rejoin the group later. *User Applications* are instantiated by a person using MobiWork in order to assist him or her in completing an assigned task. The outputs of the user applications, e.g., a PDF file, a JPEG image, etc. must be uploaded manually into the system by the person using the UI component. *Communication Middleware* is any third party software that supports host discovery and communication between hosts in a network. While our system is designed to work with any such middleware, a communication middleware that is capable of gracefully handling and recovering from frequent disconnections is especially desirable in the MANET setting.

Common Components. The common components of MobiWork represent centralized resources that are used by both the planning and execution components of the system and the components which form the bridge between the planning and execution roles of the system. The *Workflow Manager* is the central component of the MobiWork system managing both planning and execution activities. During the planning stage, the Workflow Manager accepts the plan from the planning application and passes it on to the planner for allocation. It reports the allocation information to the monitoring applications (if they are instantiated). During execution, the Workflow Manager gets regular progress updates from the Executor (an execution component described later) which it can also pass on to the monitoring applications. However, the most important role of the Workflow

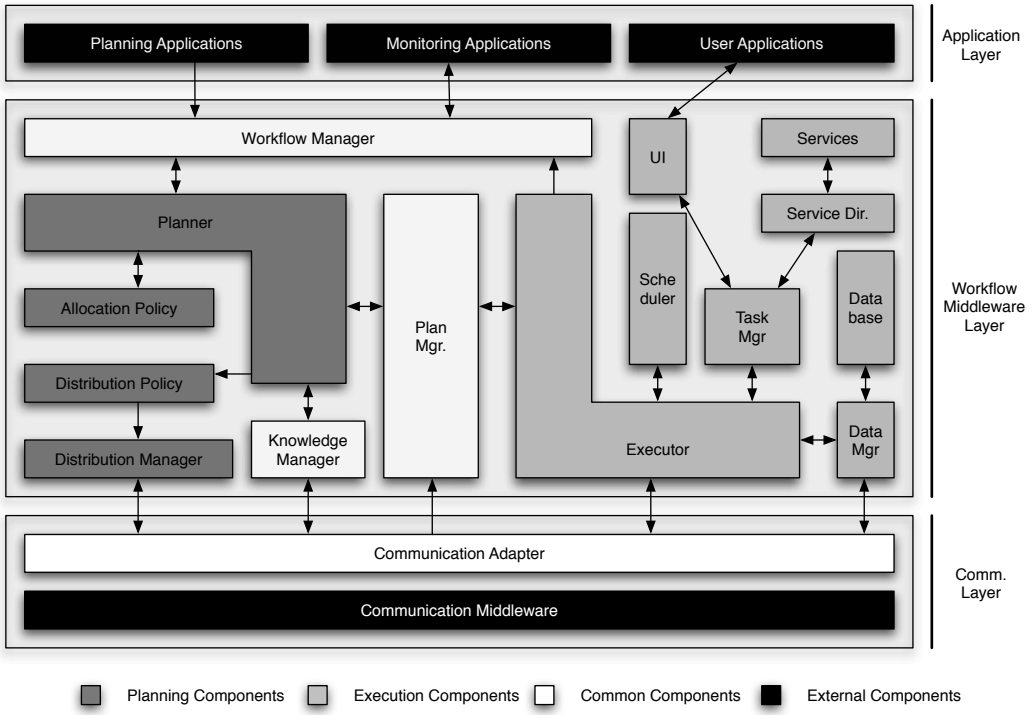

Figure 5: The architecture of MobiWork

Manager is during *re-planning*. If the Executor detects an error in execution, it reports the error to the Workflow Manager which determines the extent of the error and tasks the planner to re-allocate that section of the plan and disburse the new allocations.

The *Plan Manager* stores and maintains the plan. The plan manager may contain the entire plan (in the case of the group leader) or a portion of the plan (in the case of a group member) which reflects the tasks that have been allocated to the particular group member(s). The plan manager is a common component, as the Planner (a planning component described later) needs information about the plan for allocating tasks while the Executor needs the plan information to determine the hosts from which the inputs for a task are going to come and the hosts to which the results should be disbursed. The Plan Manager can be updated by the Planner when a new plan is injected into the system (on the group leader) or by the group leader via the Communication Middleware when task allocations are made. The *Knowledge Manager* gathers information about hosts in a MANET using an out of band gossiping protocol and stores the gathered information within a knowledge base that it owns. The knowledge base contains additional kinds of information such as group members' capabilities and motion patterns, among others. It also contains an instantaneous acquaintance list of hosts within communication range. The contents of the knowledge base is used by the allocation algorithm to make allocation decisions as described in Section 3.2 as well as to predict potential errors during execution (information that a host is not behaving as intended can be used to re-allocate the plan preemptively so that the errant host is not used). The *Communication Adapter* simply adapts the communication interface of the MobiWork middleware to the interface of the communication middleware being used.

Planning Components. The planning components of the MobiWork architecture are responsible for allocating the tasks in the plan among available group members and distributing the allocation information to the relevant group members. The *Planner* is responsible for coordinating these activities. The Workflow Manager initially provides the planner with the plan, which it stores within the Plan Manager. Subsequently, upon receiving the appropriate stimulus from the Workflow Manager, the Planner retrieves the stored plan from the Plan Manager. At the same time, it retrieves the knowledge about various group members' capabilities from the knowledge base and passes the plan and the knowledge to the *Allocation Policy* component. The Allocation Policy component contains the algorithm which allocates tasks to hosts based on the knowledge provided by the planner, an example of which was presented in Section 3.2. The Allocation Policy component returns the plan with each task annotated with the host that has been assigned to perform it. The *Distribution Policy* component is responsible for breaking the plan up into pieces that can be sent to group members. By default, the Distribution Policy component distributes to any host the tasks which have been allocated to it along with the identifiers of the hosts that provide the inputs to that task or take outputs from that task. Alternate policies may distribute entire subsections of the plan to facilitate more efficient distributed re-planning. The *Distribution Manager* sends the pieces of the plan to the recipients using the Communication Middleware.

Execution Components. The execution components of MobiWork support the actual execution of tasks in the plan. The *Executor* is the main coordinating entity during the execution of a plan. It is responsible for setting up events with the scheduler, executing tasks on time, and propagat-

ing the results to the recipients as indicated in the plan. The *Scheduler* stores events corresponding to the start time of each task assigned to the local host. The scheduler reports to the Executor every time an event fires and provides the identifier of the task to be started. The *Data Manager* is the repository of input data to the task and output data generated by the task. It also may provide other services such as versioning. The *Task Manager* is responsible for executing any task provided to it. Tasks may be completed by executing a software service on the host automatically or by soliciting input from the person using the host. The *Service Directory* contains listings of all available software services. The *UI* component forms the link from the MobiWork system to the person using the host should his or her input be required in the course of completing a task.

4.2 Anatomy of Standard Runs

Having described the different architectural components, we now describe how they interact with each other to deliver the system functionality. For clarity, we separate the bootstrapping, planning, and execution activities into sequential phases.

Bootstrapping. During this phase, all system components are initialized. The Knowledge Manager is populated with information about the local host such as its capabilities. Once the information about the local host is entered into the Knowledge Manager, it initiates a gossiping protocol via which it exchanges information about other group members. Since all group members are assumed to be initially co-located, the Knowledge Manager eventually has information about all group members. At this time, the acquaintance list within the Knowledge Manager is also initialized and updated. This concludes the bootstrapping phase.

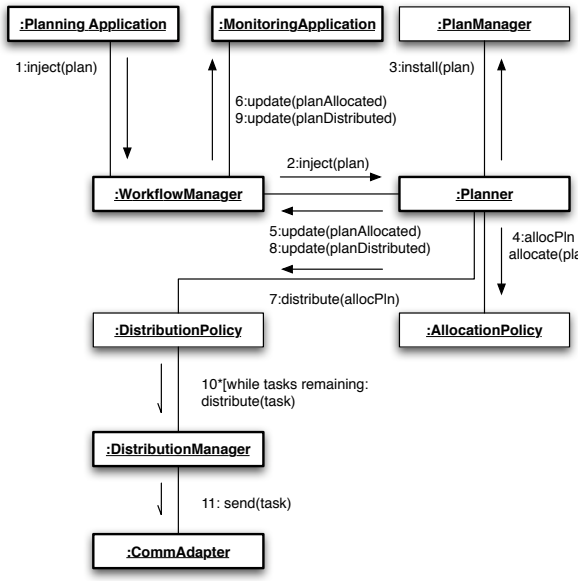

Figure 6: Sequence of actions during planning

Planning. The planning phase starts when a Planning Application injects a plan into the system via the Workflow Manager. The Workflow Manager passes the plan to the Planner which stores it in the Plan Manager. After the plan has been injected, the Planning Application directs the Workflow Manager to allocate the injected plan to available group members. This request is channelled to the Planner which retrieves the plan from the Plan Manager, and passes it to the Allocation Policy, along with a handle to the Knowledge Manager. The Allocation Policy uses the knowledge base within the Knowledge Manager to determine the allocation of tasks in the plan to group members. It should be noted that it is during this phase that MobiWork accommodates the mobility of the participating hosts. The knowledge base contains information regarding how fast hosts can move. Thus, by looking at the times and locations at which tasks are to be performed (specified in the plan) in combination with the maximum speed a host can move, it can compute the effects of allocating a particular task to a host on its ability to be at a different location at a different time to perform another task. After the allocation is completed, the Allocation Policy It returns the plan, with each action annotated with its allocation, to the Planner. The Planner then informs the Workflow Manager of a successful allocation (or generates an error if the allocation failed). The Workflow Manager in turn notifies any Monitoring Application(s) that may have been initialized. The Planner then hands the allocated plan to the Distribution Policy. The Distribution Policy divides the plan into smaller sub-plans that are distributed to the group members. The sub-plan distributed to a group member contains at a minimum the tasks allocated to him or her and may contain additional tasks in the same region of the plan. The Distribution Policy passes the sub-plans to the Distribution Manager which sends out the sub-plans using the Communication Middleware. The collaboration diagram for planning is shown in Figure 6.

Execution. The Plan Manager on every Group Member listens for incoming sub-plans which are distributed by the Distribution Manager. Similarly, the Data Manager listens for incoming data. When a plan comes in, it is stored in the Plan Manager, which notifies the Executor of the presence of a new plan. The Executor then gets the plan from the Plan Manager and examines the allocated tasks. The Executor then sets up events with the Scheduler which coincide with the start time of all allocated tasks. The system is quiescent until the start time for the first task arrives. At that time, the Scheduler notifies the Executor. The Executor fetches the task specification from the Plan Manager. It then fetches the input data for the task from the Data Manager (which was listening for this data). If the data is not available, an error is generated and the Executor notifies the Workflow Manager to initiate re-planning. In the normal case, the Executor passes the task specification and the data to the Task Manager. The Task Manager queries the Service Directory for a suitable service. A match is generated by comparing the required qualifications for the task with the capabilities of the services. The Task Manager then invokes the chosen service passing in the input data. Note that a qualifying service is guaranteed to be present because the service list is advertised as knowledge and the task allocation is based partially on the availability of services that can perform the task. Once the Service has finished executing, it passes its output to the Data Manager and notifies the Executor of completion. The Executor then fetches the output from the Data Manager. The plan stored in the Plan Manager lists the recipients of this task's output. The Executor references this information and uses the Communication Middleware to send the data to the appropriate recipients. The entire

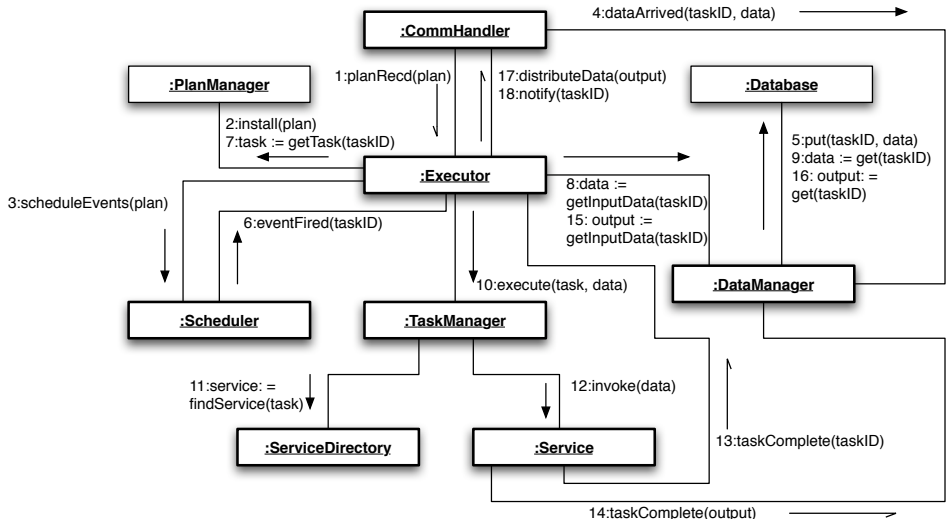

Figure 7: Sequence of actions during execution

process is shown pictorially in Figure 7.

4.3 Implementation Details

A prototype version of the MobiWork system has been implemented in Java using Limone [9] as the communication middleware. Our choice of Java was motivated by its cross-platform compatibility, which is desirable in MANETs where devices may not be of a uniform type. Limone, a lightweight communication middleware for MANETs was developed previously by our research group and uses a tuple-space based communication model to interact with hosts in MANETs. The current implementation of MobiWork is targeted towards laptops and the recently announced Origami PCs [4]. Though we considered an implementation targeted towards PDAs, we decided not to focus on those platforms for the following reasons: (1) the Java Virtual Machines (JVMs) available for PDAs are not as efficient and do not offer the same degree of capabilities as the standard JVM; (2) Java-based graphics and user interface performance on PDAs is unacceptably slow; (3) the emergence of small PCs that are only slightly bigger than PDAs but offer laptop-like capabilities indicates that we are fast approaching the point where the performance gap between PDAs and laptops will shrink significantly; and (4) the screen sizes on PDAs are currently, in our opinion, too small and low resolution. Ultimately we favor the benefits of using Java and having bigger screens to work with, and observe that PDA technology appears likely soon to catch up to the technology level for which we are currently implementing.

The MobiWork system has been implemented to address the asynchronous interactions that are common in a MANET. The `CommAdapter` is an abstract class that acts as the interface between MobiWork and the communication middleware. In our case, we extended the abstract `CommAdapter` to create a concrete `LimoneCommAdapter` class that contains the specific implementation needed to interact with Limone. The `LimoneCommAdapter` acts like a dispatcher, receiving messages from Limone, and dispatching it to any `MessageListeners` registered with it. Any objects registering with the `LimoneCommAdapter` as a listener can specify the type(s) of messages in which they are interested. Available options are `KNOWLEDGE` (information about other hosts), `PLAN` (sub-plans distributed to hosts), `DATA` (data that is transmitted from host to host as part of the data flow of the plan), and `HOST` (information about a host coming into or going out of communication range).

The `KnowledgeManager` registers itself as a listener for `KNOWLEDGE` and `HOST` while the `PlanManager`, and `DataManager` register themselves as listeners for `PLAN`, and `DATA` respectively. When the `PlanManager` receives an event, it alerts the `Executor`, which begins the execution process that was described in Section 4.2. The execution of tasks also occurs in an asynchronous fashion. The `TaskManager` calls the `executeService(...)` method on the `ServiceDirectory` which then finds a suitable service and executes it. Both the `Executor` and `DataManager` register themselves as `TaskCompletionListeners` on the `ServiceDirectory`. Thus when a task is completed, the `DataManager` is notified to retrieve the output data and the `Executor` can then proceed with retrieving this data and sending it to the host responsible for the next task.

On the planning side, `AllocationPolicy` and `DistributionPolicy` are abstract classes for the algorithms that implement the allocation and distribution functionality. With our initial system, we provide a `StandardAllocationPolicy` as we described in Section 3.2 and a `StandardDistributionPolicy` as we described in Section 4.1. Additional implementations of these policies can be added by users of the system and the system can be directed to use the correct policy by way of command line flags. At present, our system is a basic prototype implementation. We plan to address code complexity as well as performance as part of our ongoing research in this area.

5. EVALUATION

We evaluated our mobile workflow approach by implementing a prototype plan allocator in Java. This prototype was deployed on a computer equipped with a 3.2 GHz Pentium 4 CPU, 512 MB of RAM, Linux 2.6.16, and Java 2 Standard Edition 5.0_06. We tested this prototype by generating a series of random plans and measuring the time the allocator spent to find a solution. The generation of these plans is parameterized by several values, as we discuss be-

low. For comparison, our implementation allows selection between the naïve and enhanced versions of the algorithm.

Randomly generating a set of *realistic* plans is difficult, mainly because the "realism" of plans is hard to quantify. Instead, our random plan generator generates a *diverse* range of plans based on several parameters:

- r, the number of requirements that actions may draw from
- a, the number of actions in the plan
- g, the number of agents in the system
- p_r, the probability that an action has a specific requirement
- p_c, the probability that an agent has a specific capability
- p_o, the probability of an agent having a constraint between two actions

By varying these parameters, we can determine the effect that certain properties of plans have on allocation performance. For the sake of simplicity, we do not subdivide spatiotemporal constraints from agent constraints. Rather, we generate a set of constraint tuples directly, without first generating a set of causes for those conflicts.

We performed 50 allocations of fully random plans using a wide range of values for these parameters, and recorded the time taken for each version of the algorithm either to find an allocation or to determine that the plan was impossible to allocate. For comparison, we repeated this procedure with 50 more random plans that were first filtered to ensure that an allocation existed. Since the decision space that the naïve algorithm traverses quickly becomes intractable as the number of actions and agents increases, we enforced an upper-bound of 30 seconds to find an allocation. In the interest of space, we will not present here the results of all combinations of parameters. However, we will note that the parameters that had the greatest effect on algorithm performance were the number of actions in the graph, the ratio of actions to agents, and the probability of conflicts. Figure 8 shows the effect of varying the first two of these parameters with $p_o = 0.1, r = 8, p_r = 0.1$, and $p_c = 0.1$; Figure 9 shows the effect of repeating these experiments with $p_o = 0.3$ and the other parameters unchanged. We also note that the enhanced algorithm required no more than 10 ms to allocate any plan of up to 24 actions, whereas the naïve algorithm frequently required more than 30 seconds to allocate the same plans.

The performance difference between the two versions of the algorithm is striking, especially when they are provided with plans that cannot necessarily be allocated. At first, we were concerned that this gap was caused by a bug in the implementation. However, after tracing through the execution of the two versions, we confirmed that their behavior was correct, and discovered an explanation for the performance gap. The algorithm would often make an incorrect decision while allocating one of the first few actions. The naïve version would then iterate through a large portion of the decision space before it could return to that early mistake and correct it. As we noted in Section 3, the enhanced version of the algorithm effectively divides the plan into sub-plans, and allocates each sub-plan independently. Hence, when it makes a mistake early in one sub-plan, it only has to explore the relatively small decision space of that sub-plan before revisiting the incorrect decision. Furthermore, the naïve algorithm must traverse nearly the entire decision space before it can conclude that plan is not allocatable. If the enhanced algorithm traverses the smaller decision space of one sub-plan and fails to find an allocation, then it immediately concludes that the entire plan cannot be allocated.

6. CONCLUSION

Workflow-based collaborative technologies hold much promise, but thus far systems supporting workflow based collaborations (WfMSs), have been designed for stable wired networks or nomadic mobile networks. In this paper we sought to lay a foundation for WfMSs that can operate in the unpredictable environment of a MANET without depending on centralized resources or reliable links between participants. We extended the algorithm for the allocation of tasks from a simple one based only on task requirements and agent capability matching to a necessarily more complex one that additionally considers the mobility of all participants. We use a unified concept of constraints to determine non-allocatability of a particular agent to a task due to mobility, lack of capabilities, or plan-specific restrictions, with the capacity to add any other types of constraints as necessary. Our investigations showed that a heuristic version of our algorithm which allocates tasks for which there are fewest qualified agents first clearly outperforms the naive algorithm which allocates tasks in a numerological order. Our implementation of MobiWork in Java has served as a proof of concept that workflow-based collaboration can be supported to a satisfactory degree in a MANET environment. However, much work needs to be done to optimize the systems and improve their sophistication for greater reliability and error handling.

7. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation Division of Information and Intelligent Systems research award IIS-0534699. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not represent the views of the research sponsors.

8. REFERENCES

[1] ActiveBPEL engine. `http://www.activebpel.org/`.
[2] OASIS web services business process execution language (WSBPEL) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.
[3] Oracle BPEL process manager. `http://www.oracle.com/technology/products/ias/bpel/index.html`.
[4] The origami project. http://origamiproject.com/default.aspx.
[5] WebSphere process server. `http://www-306.ibm.com/software/integration/wps/`.
[6] Wf-XML 2.0. `http://www.wfmc.org/standards/wfxml_demo.htm`.
[7] Workflow. http://en.wikipedia.org/wiki/Workflow.
[8] G. Alonso, R. Gunthor, M. Kamath, D. Agrawal, A. E. Abbadi, and C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. In *Proc. 3rd International Conference on Cooperative Information Systems (CoopIS)*, pages 99–110, Vienna, May 1995.

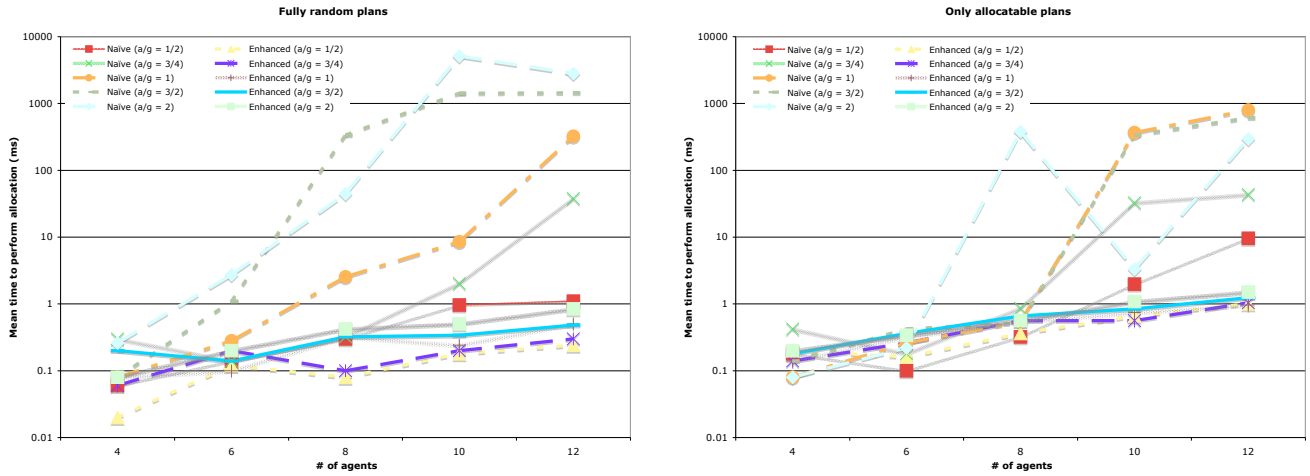

Figure 8: Algorithm performance when $p_o = 0.1, r = 8, p_r = 0.1, p_c = 0.1$. Left: mix of allocatable and unallocatable plans; Right: all plans are allocatable.

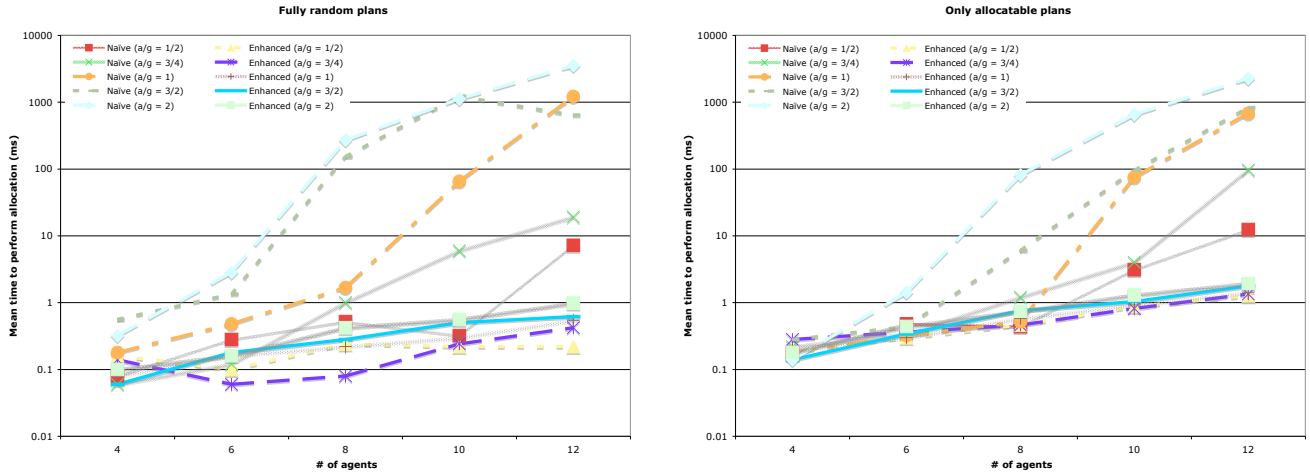

Figure 9: Algorithm performance when $p_o = 0.3, r = 8, p_r = 0.1, p_c = 0.1$.

[9] C.-L. Fok, G.-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 135–151, Bologna, Italy, February 2004. Springer-Verlag.

[10] C. Godart, P. Molli, G. Oster, O. Perrin, H. Skaf-Molli, P. Ray, and F. Rabhi. The toxicfarm integrated cooperation framework for virtual teams. *Distributed and Parallel Databases*, 15(1):67–88, 2004.

[11] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. `http://www.w3.org/TR/ws-cdl-10/`, November 2005.

[12] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339. ACM Press, 2003.

[13] R. T. Marshak. *Groupware: Technology and Applications*, chapter Workflow: Applying Automation to Group Processes, pages 71–97. Prentice-Hall, 1995.

[14] D. Martin and et. al. OWL-S: Semantic markup for web services. `http://www.w3.org/Submission/OWL-S/`, November 2004.

[15] M. Mecella and et. al. Workpad: an adaptive peer to peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In *IEEE International Symposium on Collaborative Technologies and Systems*, 2006.

[16] N. Preguiça, J. L. Martins, H. Domingos, and S. Duarte. Integrating synchronous and asynchronous interactions in groupware applications. *Lecture Notes in Computer Science*, 3706:89–104, 2005.

[17] H. Stormer and K. Knorr. Pda- and agent-based execution of workflow tasks. In *Proceedings of the Informatik 2001*, pages 968–973, 2001.

[18] W. van der Aalst and K. van Hee. *Workflow Management : Models, Methods, and Systems*. MIT Press, March 2004.