Report Number: WUCSE-2006-51

2006-01-01

CiAN: A Language and Middleware for Collaboration in Ad hoc Networks

Rohan Sen, Gruia-Catalin Roman, and Andrew Frank

Designing software that supports collaboration among multiple users in mobile ad hoc networks is challenging due to the dynamic network topology and inherent unpredictability of the environment. However, as we increasingly migrate to using mobile computing platforms, there is a pertinent need for software that can support a wide range of collaborative activities anywhere and at any time without relying on any external infrastructure. In this paper, we adopt the workflow model to represent the structure of an activity that involves multiple tasks being performed in a structured, collaborative fashion by multiple users. Using the workflow model as a... **Read complete abstract on page 2.**

CiAN: A Language and Middleware for Collaboration in Ad hoc Networks

Rohan Sen, Gruia-Catalin Roman, and Andrew Frank

Complete Abstract:

Designing software that supports collaboration among multiple users in mobile ad hoc networks is challenging due to the dynamic network topology and inherent unpredictability of the environment. However, as we increasingly migrate to using mobile computing platforms, there is a pertinent need for software that can support a wide range of collaborative activities anywhere and at any time without relying on any external infrastructure. In this paper, we adopt the workflow model to represent the structure of an activity that involves multiple tasks being performed in a structured, collaborative fashion by multiple users. Using the workflow model as a base, we developed an XML based specification language called CiAN that can be used to build workflows that can be fragmented and distributed across the hosts of participating users so that the collaborative activity is executed in a distributed manner. The tasks specified in the CiAN language are executed by our Java based CiAN middleware, which runs on mobile hosts. Communication of task results between hosts occurs via a novel protocol that uses the workflow structure to make routing decisions on data packets. Complete implementation details and an evaluation of our approach are also presented.

2006-51

CiAN: A Language and Middleware for Collaboration in Ad hoc Networks

Authors: Rohan Sen, Gruia-Catalin Roman, and Andrew Frank

Corresponding Author: rohan.sen@wustl.edu

Abstract: Designing software that supports collaboration among multiple users in mobile ad hoc networks is challenging due to the dynamic network topology and inherent unpredictability of the environment. However, as we increasingly migrate to using mobile computing platforms, there is a pertinent need for software that can support a wide range of collaborative activities anywhere and at any time without relying on any external infrastructure. In this paper, we adopt the workflow model to represent the structure of an activity that involves multiple tasks being performed in a structured, collaborative fashion by multiple users. Using the workflow model as a base, we developed an XML based specification language called CiAN that can be used to build workflows that can be fragmented and distributed across the hosts of participating users so that the collaborative activity is executed in a distributed manner. The tasks specified in the CiAN language are executed by our Java based CiAN middleware, which runs on mobile hosts. Communication of task results between hosts occurs via a novel protocol that uses the workflow structure to make routing decisions on data packets. Complete implementation details and an evaluation of our approach are also presented.

Type of Report: Other

CiAN: A Language and Middleware for Collaboration in Ad hoc Networks

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

Abstract

Designing software that supports collaboration among multiple users in mobile ad hoc networks is challenging due to the dynamic network topology and inherent unpredictability of the environment. However, as we increasingly migrate to using mobile computing platforms, there is a pertinent need for software that can support a wide range of collaborative activities anywhere and at any time without relying on any external infrastructure. In this paper, we adopt the workflow model to represent the structure of an activity that involves multiple tasks being performed in a structured, collaborative fashion by multiple users. Using the workflow model as a base, we developed an XML-based specification language called CiAN that can be used to build workflows that can be fragmented and distributed across the hosts of participating users so that the collaborative activity is executed in a distributed manner. The tasks specified in the CiAN language are executed by our Java-based CiAN middleware, which runs on mobile hosts. Communication of task results between hosts occurs via a novel protocol that uses the workflow structure to make routing decisions on data packets. Complete implementation details and an evaluation of our approach are also presented.

1 Introduction

In recent years, we have witnessed a growing emphasis on systems that are designed to facilitate collaboration. A large number of these collaborative systems use workflows to define the structure of the collaborations. In simple terms, a workflow defines a group of tasks, the ordering among those tasks, and the movement of data from one task to another. An example of a workflow is loan processing, which is a sequence consisting of the tasks "start", "enter applicant details", "obtain loan approval", "transfer money to borrower", and "finish", where all of the tasks need to be completed by people whose expertise matches the requirement of the task. Workflow languages such as BPEL [3], WfXML [14], and XLANG [15] are used to specify the workflow. Workflow Management Systems (WfMSs) such as FLOWer [1], AgentWork [8], Caramba [4], and I-Flow [5] execute the specification, invoking software services or notifying human users to complete tasks. Most WfMSs are designed to execute on servers that are connected to each other via reliable wired networks.

However, the increasing ubiquity of mobile devices and wireless networking has resulted in a shift away from traditional wired networks to dynamic, wireless networks. As such, it is essential that the systems supporting collaboration are also able to function in the more dynamic setting of a wireless network. There have been several efforts by the research community to achieve this in a nomadic wireless setting, e.g., MoCA [11]; but there has not been much work in designing systems for mobile ad hoc networks (MANETs) which lack fixed infrastructure.

There is a strong motivation for developing WfMSs targeted to MANETs. Such systems could be used for collaboration practically anywhere without the need for any external infrastructure. Consider, for example, a large toxic spill that occurred in some remote area. Cleaning up the spill requires coordination among many specialized teams equipped with mobile devices and autonomous robots in a structured fashion so that they do not interfere with each other and complete tasks according to their priority. Such an activity can be modeled as a workflow. However, it is unlikely that the teams cleaning up the spill will have access to a fixed network to run the workflow on a traditional WfMS. All communication and workflow management must occur over a MANET that is formed dynamically between the mobile devices of team members and the robots, thus mandating a WfMS that can work in a MANET setting.

Unfortunately, migrating a WfMS from a wired or nomadic setting to MANETs is not trivial. In wired systems, the WfMS and all the services it exploits to complete tasks can coordinate with each other at any time due to the pres-

ence of the fixed network. In a MANET, we do not have this luxury, so the ability to communicate and coordinate with different participants becomes much more difficult. Also, the WfMS cannot execute on a single host because the loss of that host compromises the collaboration among all other hosts in the MANET. Solving these problems requires a redesign of WfMSs at the most fundamental level.

To our knowledge, designing WfMSs targeted to MANETs without any reliance on external resources has never been attempted before and building such a system from the ground up represents a significant software engineering challenge. Consider the issue of specifying a workflow. The specification must be designed in a way that is intuitive and human-readable, while at the same time being machine-parseable for automated execution of the workflow. In addition, it should be easily fragmentable for distribution across hosts in the MANET. Given a fragmented specification, the system must also be able to decide *how* and *to whom* to distribute each fragment based on individual properties of hosts. Finally, a run-time system must be available to execute each of the fragments in a distributed and disconnected fashion, and disburse results as appropriate, all in the context of a resource poor device in a dynamic MANET. In this paper, we focus on designing the specification and runtime system. We plan to address the problem of distribution or allocation in the future.

Collaboration in Ad hoc Networks (CiAN) is a language and middleware supporting collaboration in MANETs. Section 2 covers the current state of the art. In Section 3, we describe our conceptual model for a collaborative system in a mobile setting. We then present our language in Section 4 and justify our choices for language features. In Section 5, we describe the software components in our system, the system architecture, and its implementation in Java. We discuss the performance of our system in Section 6 before concluding in Section 7.

2 Background and Related Work

Describing a Workflow. Workflows are described using a workflow language that specifies a syntax and offers constructs to specify the workflow structure and function. The capabilities offered by the language are crucial, because they determine the expressive power and versatility of the workflow specification. For example, in the simplest sense, a workflow can be modeled as a directed graph with the nodes representing tasks, and the edges imposing an order among the tasks. This type of specification gives us the requisite structural information about the workflow, but very little functional information related to the actual tasks. An improvement to this technique is the use of Petri-nets [10], a structural specification that can incorporate more functional information by careful use of places and tokens. In current generation languages such as BPEL [3], XLANG [15], WfXML [14], and YAWL [17], the specification is in the form of an XML document. Each language provides its own unique syntax and constructs; e.g., in BPEL, *sequence* is used to execute tasks in order, *flow* for parallel tasks, *switch* and *pick* for conditional execution, etc. In all these cases, the language provides features that help the developer specify requirements for each task and the flow of control through the tasks, which is an improvement over a plain graph specification.

However, these structural constructs have a downside. The workflow specification is not as easily partitionable as a graph-based specification. Currently, a majority of the WfMSs supporting the languages mentioned above execute on powerful servers in a wired network where partitioning is not necessary. However, in a MANET setting, where there is no powerful central server, partitioning the workflow and executing it in a distributed fashion is vital. Attempts have been made to facilitate the partitioning of workflow specification by adding special actions like DoStart, ReceiveStart, DoEnd, and ReceiveEnd to handle the coordination among the different pieces, described in [6]. While this work describes a method for partitioning the specification, it does not support fully decentralized execution, and as such falls short of our needs. Another approach is described in [2] where the authors parse a BPEL specification, discard all the structural constructs and use the *link* construct to build a more graph-like specification. This approach is specific to BPEL and is still fairly rigid e.g., it does not allow optional redundant edges. In addition to the partitioning problem, the languages have two other significant shortcomings: 1) with the exception of YAWL, they do not support (or support only in a very non-intuitive fashion), all the basic structural and synchronzation constructs for workflows [16] as described by van der Aalst in [18], which limits the expressive power of the workflow, and 2) there is no facility to specify mobility and network topology information, which may be exploited in a MANET environment to improve the chances for success.

Executing a Workflow. While having a workflow language that has sufficient features to specify a workflow that will execute in a MANET setting is crucial, it is equally crucial to have a system that is capable of delivering all the features of that language in a mobile setting. The MANET environment is distinct from the stable, wired network environment in that the network is made up of devices that are significantly less powerful and the network topology is dynamic and evolves rapidly over time. As such, a simple port of a system designed for a wired environment to a MANET is likely to be unsuccessful. A WfMS designed for the MANET setting must have three features that are not available in systems today: 1) the architecture of the system must support decentralized and distributed execution of the

workflow, 2) coordination constructs should be available so that the different pieces of the workflow can communicate with each other over the MANET, and 3) the infrastructure should exploit non-functional information about the network to mitigate failures.

Current designs for WfMSs do not completely meet these requirements. For example, in BPEL, transfer of data between tasks is done by centralized shared variables an approach that is not practical in a MANET. A workaround to this was proposed in [2] which uses message passing to distribute data in a wired setting but cannot handle the dynamism of MANETs. MoCA [11] is better suited to mobility due to its use of proxies. Mobile users maintain proxies on stable wired nodes which communicate with the MoCA directory service (DS), configuration service (CS) and context information service (CIS) over a wired link. These three modules coordinate the collaborations of multiple mobile hosts. However, since most of the MoCA infrastructure resides on a wired network, only nomadic mobility can be realistically supported as mobile hosts are dependent on the DS, CS, and CIS for coordination. In AWA/PDA [13], the authors adopt a mobile agent based approach based on the GRASSHOPPER agent system. They define five types of agents, the Workflow Agent (WA), Process Agent (PrA), Task Agent (TA), Worklist Agent (WlA), and Personal Agent (PA). The PA is a daemon agent on a mobile device and is not logically mobile. The rest of the agents are logically mobile and can opportunistically move from one host to another. For example, a TA, which coordinates a single task can migrate to a PDA, work while disconnected, and then report results when a connection is available to the WA. Since all necessary components of the system are logically mobile, this type of architecture is better suited to mobile environments. However, the dependence on a WA or WlA means that the hosts carrying these agents must always be within communication range. While this may be simulated by moving the process from one host to another this is computationally expensive and it may not be possible to guarantee connectivity with all hosts.

WORKPAD [7] is designed to meet the challenges of collaboration in a peer-to-peer MANET involving multiple human users. WORKPAD works by augmenting the basic workflow specification with directives (mainly involving moving a host so that it can communicate with another) so that the tasks in the workflow can hand off data to subsequent tasks and thereby advance the execution of the workflow. WORKPAD's shortcoming is that it requires at least one member of a MANET to coordinate with a central entity that coordinates the mobile devices, manages disconnection, and augmentis the workflow specification with mobility directives. This dependence means that WORKPAD cannot survive in ad hoc mode for an extended time. Our work is targeted to an environment similar to that of WORKPAD. However, our approach is different. Rather than insert directives to move hosts from one place to another, we allow hosts to move freely and use other constructs to ensure successful completion of workflows as described in subsequent sections.

3 Our Perspective on Mobile Collaboration

The CiAN language and middleware are attempts to overcome some of the restrictions of the systems described in Section 2. In this section, we describe our computational model and the philosophy behind our design of the CiAN language and middleware.

Computational Model. For the purposes of this paper, we assume that a workflow is used to coordinate the actions of multiple human users, each of whom are equipped with a mobile device such as a PDA. Each mobile device is carried by the user and hence is physically mobile. However, we assume that the user stores a schedule on the device from which we can infer where a user will be at certain points in time. The devices can communicate with each other using 802.11b or a similar standard when they are within communication range. However, due to the user's physical mobility, such *windows of communication* may be transient. During these transient windows, hosts trade non-functional knowledge about themselves such as their mobility pattern via a gossiping protocol as described in [12]. This knowledge is stored in a local knowledge base on each host and is made available to all components of the CiAN middleware. Finally, hosts may have sensors attached to them, such as a GPS receiver, which the CiAN middleware can leverage.

A task in the workflow is considered to be ready for execution when all of its inputs are available. Since the first task in the workflow has no inputs, it is considered to be always ready. A task may involve an action in the physical world by a human user or the execution of some code on the user's mobile device. If it is an action to be performed by the human user, he receives a prompt on his screen when the task is ready to be performed. He can enter any information related to the task on his PDA upon completion of the task, e.g., reporting the temperature after taking the reading from a sensor manually. If the task involves execution of a software service, it is handled automatically in CiAN. Any data generated after a task is executed is passed on to other hosts when they are in direct communication (a form of routing, described in Section 5 consisting of multiple disjointed peer-to-peer connections allows data transfer between hosts that never meet directly).

Language Model. We based the CiAN language on the concept of an annotated graph. Rather than using explicit structures that specify whether tasks execute in sequence or in parallel, we simply list the tasks in no particular order. Then, treating these tasks as graph nodes, we build two

adjacency lists for each task, the input adjacency list and the output adjacency list. If a Task A appears in Task B's output adjacency list, then there is an edge from Task A to Task B (we automatically add Task A to Task B's input adjacency list if it is not in the specification). The adoption of the graph model allows us to easily fragment a workflow into its constituent tasks and assign individual tasks for execution on available hosts. While the adoption of the graph model is the most significant design feature, there are several others that relate to mobility support which we discuss in the next section.

Runtime System. The ability to fragment a workflow specification in CiAN is not useful unless there is a runtime system designed to execute the pieces in a distributed manner. Given that most current approaches use a centralized architecture, we chose to develop a new system rather than extend any current system. Our runtime system is designed to run in a completely decentralized fashion with no need for a central coordinating entity. Each task is preceded by an input filter which acts as a controller and marshalls the inputs to a task from one or more hosts. The task is succeeded by an output filter which is a similar controller for distributing the output of the task to interested hosts. Both the input and output filters are available as part of the CiAN middleware. They are dynamically parameterizable at the time that a task is assigned to a host so that they are customized to handle the task they are associated with.

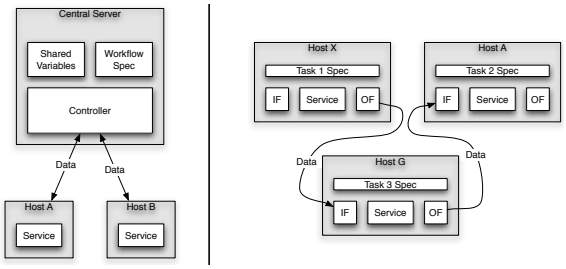

Figure 1. Centralized vs. filter-based architecture

In addition to marshalling the inputs, the filters can perform a small but powerful set of decision and synchronization functions. By exploiting these functions, we can emulate all the workflow patterns suggested by van der Aalst [16] in a simple and intuitive way. The filters can also access the knowledge base for non-functional information about the network and the physical environment to make context-aware choices during the execution of the workflow. Finally, the filters are designed to selectively wait or ignore inputs and outputs. This allows us to build redundancies into the workflow in a simple manner, crucial to protecting against failures that are common in a MANET. Figure 1 illustrates the differences between the two models.

Filters in detail. The input and output filters are the centerpiece of our execution model as they are the structures that allow us to distribute the workflow across multiple hosts and handle synchronization and communication issues. Figure 2 shows a conceptual view of an input filter. An output filter is similar, except that the edges are outgoing.

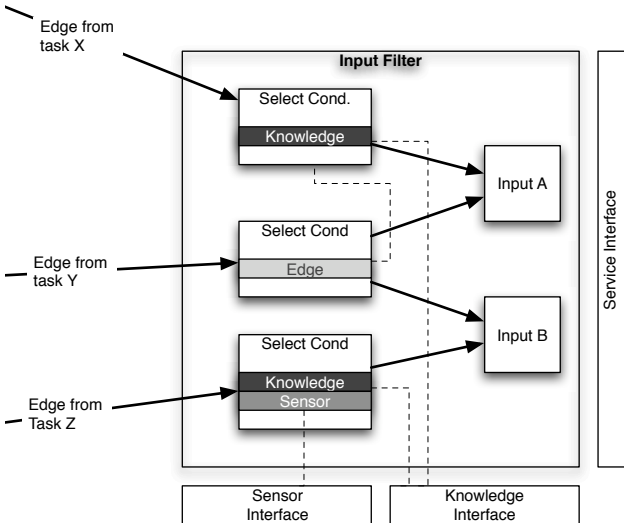

Figure 2. Detail of an input filter

As shown in the figure, each incoming link (an edge in the conceptual workflow graph) may have multiple selection conditions. Selection conditions are groups of conditions that must be met for the value transmitted along that edge to be accepted as an input to the accompanying task. If all the conditions that make up one of the edge's selection conditions are met, and the edge has some value transmitted along it, then the edge is considered valid input. The filters also define acceptance sets. These are subsets of all incoming edges and define which edge combinations represent a valid input to the task. This is useful for building redundancies, because an acceptance set can be built from each redundant set of input edges. When all the edges in any of the accept sets are valid, the task can begin execution. The output filters are similar in behavior, except that the conditions represent whether the results are transmitted along that edge or not, and the accept sets define subsets of outgoing edges that are considered acceptable.

4 A Language for Mobile Collaboration

This section describes the CiAN language for collaboration in MANETs. Figure 3 shows the code for one of possibly many tasks in the workflow.

```
<collaboration>
        <knowledge-base>
                <knowledge-var>host-capabilities</knowledge-var>
                <knowledge-var>motion-profile</knowledge-var>
        </knowledge-base>

        <sensors>
                <sensor-var>location</sensor-var>
                <sensor-var>time</sensor-var>
        </sensors>

        <task>
                <task-name>Task1</task-name>
                    <inputs>
                        <edge>
                                <name>InEdge1</name>
                                <var>InVar1</var>
                                <partner>Task1</partner>
                                <select-cond>
                                   <cond>
                                      <param>sensor:time</param>
                                      <comparator> = </comparator>
                                      <value> 12:30 </value>
                                   </cond>
                                </select-cond>
                        </edge>
                        <accept-set>
                                <set>
                                    <name>InEdge1</name>
                                </set>
                        </accept-set>
                    </inputs>
                <activity>
                        <input-vars>
                                <var>InVar1</var>
                        </input-vars>
                        <service>DummyService1</service>
                        <output-vars>
                                <var>OutVar1</var>
                        </output-vars>
                </activity>

                <outputs>
                        <edge>
                                <name>OutEdge1</name>
                                <var>OutVar1</var>
                                <partner>Task2</partner>
                        </edge>
                        <accept-set>
                                <set>
                                     <name>OutEdge1</name>
                                </set>
                        </accept-set>
                </outputs>
        </task>
</collaboration>
```

Figure 3. CiAN code example

A workflow specification in CiAN is delimited by the `<collaboration>` tags. Within these tags, the specification is split into two elements – the header and the body. The header declares the non-functional information about the hosts and the network that this workflow relies on while the body contains the actual task definitions. The header itself is split into two sections delimited by the `<knowledge-base>` and `<sensor>` tags. The knowledge base section specifies the names of the parameters found in the knowledge base of hosts that this workflow relies on. Each of these parameter names are delimited by `<knowledge-var>` tags. The sensor section specifies the names of the various sensor parameters that the workflow relies upon and these are delimited by `<sensor-var>` tags (the sensors are assumed to be attached to the local device). The absence of these parameter values at runtime does not halt the execution of the workflow but could compromise its flexibility.

The body of the workflow comprises one or more task definitions, delimited by the `<task>` tag. The tasks can be specified in any order regardless of their position in the workflow. A task definition consists of a task name, delimited by the `<task-name>` tag and unique in the scope of a workflow and three additional sections: input, activity, and output, delimited by the `<inputs>`, `<activity>`, and `<outputs>` tag respectively. The input section defines one or more edges, delimited by the `<edge>` tag. Each edge represents a connection to another task in the workflow and is analogous to the edge in the graph representation of the workflow. The input section also defines one or more accept sets, delimited by the `<accept-set>` tag. Each `<set>` element inside is a list of names of edges and represents subsets of the input edges that are considered valid input. For example, if a task had 6 incoming edges labelled 1 to 6, but those 6 edges were really a pair of redundant inputs consisting of 3 edges each, then values from the first of the redundant pairs (edges 1 to 3) are equally valid as the values from the second of the redundant pairs (edges 4 to 6) and it is not necessary to wait for all the inputs. The accept set captures this information. The outputs section is similar to the inputs section in structure except that the edges are outgoing rather than incoming.

The activity section specifies information about the actual service that needs to execute to complete a particular task. This section specifies the names of the input variables to the service, delimited by the `<input-vars>` tag and the names of the variables to which the service writes its output, delimited by the `<output-vars>` tag. It should be noted here that since the input filter, service, and output filter for a given task reside on the same host there is no issue of consistency or overhead in using these variables. Finally, the actual location or URI of the service is delimited by the `<service>` tags.

Workflows specified in CiAN are flexible and context-sensitive due to the way an edge is structured. Each edge has a name, delimited by the `<name>` tag, which is unique in the scope of a task. The `<partner>` tag is used to define the name of the task at the other end of the edge. For incoming edges this is the source task, while for outgoing edges it is the sink task. The `<var>` tag specifies the name of the local variable to which the value transmitted along the edge is written to (in the case of input edges) or read from (in the case of output edges). These variables are the same as those that appear as input variables and output vari-

ables in the activity section. Values transmitted along an incoming edge are written to an input variable from where the service reads it. The service's output is written to an output variable from where it is read and transmitted over an outgoing edge.

In addition to this basic information, each edge can specify zero or more selection conditions (denoted by the `<select-cond>` tag) which consist of multiple sub-conditions. If any one of the selection condition blocks have all their sub-conditions (denoted by `<cond>`) evaluate to true, then the value of the edge is considered acceptable input to the task. Each condition is a three-tuple consisting of a parameter name denoted by `<param>`, a comparator, and a value. The parameter can be of four different types: 1) `knowledge:hostname:paramname`, which refers to the non-functional parameter called `paramname` of a host called `hostname` which may be found in the local knowledge base, 2) `sensor:sensorname`, which refers to the value of a sensor called `sensorname`, 3) `edge:edgename`, which refers to a value transmitted along another edge whose name is `edgename`, and 4) `var:varname` which refers to the value of a local variable called `varname`. The comparator may be the operators {`<, >, <=, >=, ==, !=`}.

The graph like structure of our language helps us support all the basic workflow patterns. If parallelism is required, multiple output edges can fan out from a single task. If sequential processing is desired, then only one output and input may be used. Merges and splits can be built in similar ways. If an XOR merge is desired, each edge can be tagged with a selection condition with require all other input edge values be equal to null. This way, the first edge that yields a value is selected as input. Also, context sensitive selection is implemented in the same way, e.g., an edge is not selected unless the temperature sensor has a value greater than 32.

Implementation. One of the main features of the CiAN specification is that it is easily fragmentable. To fragment the collaboration, we developed a simple Java parser that reads the header information for the CiAN specification. It then parses each task and writes it along with the header information to a separate file. Since tasks are self contained units and not embedded within structural constructs, this process is straightforward. We developed an XML document type definition (DTD) for the CiAN language in order to build a validating CiAN parser using the Java XML Processing API (JAXP). The resultant parser is SAX compatible and can convert each task definition output by the fragmenting parser into executable java objects. It creates input and output filters and parametrizes them using the information in the `inputs` and `outputs` section of each task. It also uses Java reflection to create an instance of the specified service. At this point, the task is ready for execution. Support for this is provided by the CiAN middleware, which is described in the next section.

5 System Design

The CiAN middleware is responsible for executing workflows specified using the CiAN language. The execution of any workflow can be divided into two distinct phases: planning and execution. In this paper, we focus primarily on the execution aspect but we include a brief description of the planning phase for the sake of completeness.

In CiAN, we assume that a group of people equipped with mobile devices such as PDAs assemble at some location to plan the activities of the day. The planning process is executed by a pre-determined group leader and its main goal is to examine the workflow and map each task to a host in the network, thereby delegating the responsibility of executing that task to that host. Such an assumption is not unreasonable, as most group activities usually involve some kind of leader. Figure 4 shows the system architecture for the planning phase, which is executed on the team leader's mobile device. The external application injects the workflow specification (which is in the CiAN language) into the planning system by way of the `Planner API`. This API layer distributes the specification to three units: 1) the `Workflow Splitter` which decomposes the workflow into its constituent tasks as described in Section 4, 2) the `Ordering Generator` which numbers each task to impose a total order among the tasks (for routing purposes described later), and 3) the `Allocator` which determines the mapping of tasks to hosts. The working of the splitter was described in Section 4, while the `Ordering Generator` implements a trivial graph traversal algorithm. The `Allocator` uses information from the workflow specification (i.e., the service requirements of a task), the capabilities, i.e., services, offered by hosts *and* their mobility patterns found in their respective knowledge bases to make allocation decisions. Allocation is essentially a scheduling problem, which is in itself a vast area of study. Due to space constraints, we do not investigate different allocation strategies in this paper. We have implemented a basic strategy and have left more complex options for future study. The `Task Assembler` marshalls the individual task specifications, the task number, and allocation information and sends it out to the host that is allocated to perform that particular task.

Once the planning phase has concluded, execution can begin almost immediately. When the `Task Assembler` transmits the task information to a host, it is received by the `Communication Module`, which handles all incoming and outgoing communication to a host. When the task information arrives, the `Communication Module` passes it to the `Workflow Router`, which in turn passes it to the `Dispatcher`. The `Dispatcher` parses the task

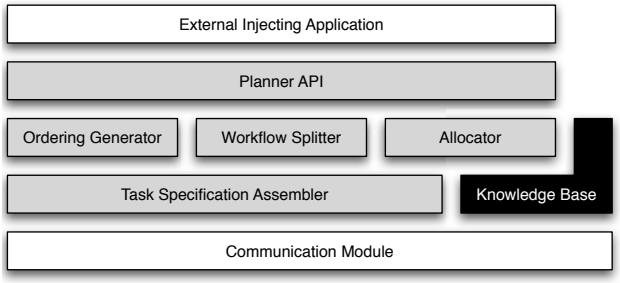

Figure 4. CiAN planning architecture

specification and creates a `Task Manager` for the task. The `Task Manager` contains the input and output filters which are parametrized according to the information in the specification received. The `Task Manager` also has access to the knowledge base to evaluate selection conditions for input or output edges. Finally, it creates *subscriptions* for each of its inputs, which is a request for data generated by its preceding tasks (we will cover subscriptions later in this section). At this point a task is waiting on its inputs before it can start executing.

The first task in any workflow by definition does not have any inputs, and hence can start executing immediately. The `Task Manager` invokes the service and waits for the service to write its output to its output variables. It then executes the output filter to determine if a valid output was generated. If so, it creates a data message that it transmits (via its `Dispatcher` and `Workflow Router`) to the host(s) that are responsible for performing the task(s) immediately following the first task. These tasks wait on their inputs and execute once all the inputs are available. Execution continues until the last task in the workflow is executed.

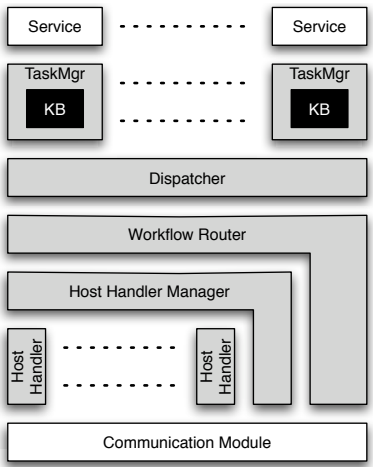

Figure 5. CiAN runtime system architecture

Thus far, we have glossed over an important aspect of our work, which is the fact that this system has to execute in a MANET. The `Communication Module` on each host transmits a beacon periodically. When the `Communication Module` on another host receives such a beacon, it passes it to its `Host Handler Manager` which creates a `Host Handler` for that host. The `Host Handler` tries to establish a direct connection between the hosts. Thus, as long as the hosts are in communication range, the `Host Handler` acts like the proxy of the remote host on the local host and handles communication between them.

Since direct communication is the most reliable form of communication in a MANET, all information in CiAN is transmitted when two hosts are directly connected. Thus, when the `Host Handler` establishes a connection, it synchronizes the knowledge base of the two hosts using the time of acquisition of any knowledge as a tie breaker. It also sends to and receives data or subscription messages from the other host as appropriate. All data and subscription messages received are passed to the `Workflow Router`. If a data message is intended for the local host, it is passed to the `Dispatcher`, which in turn passes it to the appropriate `Task Manager`, which places the value on the correct edge in the input filter.

The remaining piece of our system is the policy for transmitting the data or subscription messages from one host to another. The simplest policy is to simply address the messages by its destination host and use a MANET routing protocol to deliver the message. However, this has two drawbacks: 1) MANET routes do not last often and are expensive to maintain, and 2) it strongly associates a task to the host, which is not desirable if a task were to ever get re-allocated to a different host (a feature not currently supported in CiAN but part of our future plans). Our approach is a store and forward approach based on a routing policy we have developed. When each host receives a task spec, it assigns a number to itself that is the same as the number of the task. If multiple tasks are assigned, then it chooses the lowest numbered task. Subscriptions (generated by tasks to solicit inputs) have the number of the subscribing task, and the number of the task whose input is desired. Similarly, when a task finishes execution, the data is labelled with the generating task number and the number of the task(s) that should receive the data. The messages are routed using one of the following three schemes:

- Scheme 1: Data is routed to any host that has a number between the generating task number and the target task number or has no number in a strictly increasing fashion. Subscriptions are routed similarly but in a strictly decreasing function. Routing to a host with no number is considered neither a decrease nor an increase.

- Scheme 2: Data can be routed to any host that has a number between the starting task number and the target task number in a strictly increasing order. Subscriptions are routed to hosts between the target task number and the ending task number. Routing to hosts without a number is also permitted.

- Scheme 3: This scheme is identical to Scheme 1 with one exception. Data and subscriptions can be routed outside the permissible range but this triggers a counter. If the data or subscription moves to a host in range (as defined by Scheme 1) before the counter expires, the counter is reset, otherwise the data or subscription is destroyed.

Scheme 1 generates the lowest number of messages in the network but is restrictive in the sense that the number of hosts that a message can be routed to is much smaller than the total number of hosts collaborating. Scheme 2 increases the permissible range but generates additional messages. Scheme 3 maintains the low range of Scheme 1 but allows limited transgressions which represents the most favorable tradeoff between number of messages and number of hosts to which the message can be routed. Data and subscriptions are transmitted from host to host using the `Host Handlers` during periods of direct connectivity. The `Host Handlers` pass these messages to the `Workflow Router`, which determines if any of the subscriptions it is aware of matches any data that it is aware of. If a match is generated, then the data is sent to the subscribing host using the AODV routing protocol [9].

Implementation. A prototype of the CiAN middleware has been implemented in Java. We used the Java XML Processing API (JAXP) to build the various parsers that parse the workflow specification and convert it to Java objects, assign numbers to tasks, and allocate them to hosts. The remainder of the system was built using J2SE 5.0. The `CommunicationModule` is used for basic communication in the network. It can be customized to work with any external communication middleware. The `CommunicationModule` uses a beaconing mechanism that multicasts the IP address and port for each host in the network. When a host receives a beacon from another host, it notifiers the `HostHandlerMgr` which creates a HostHandler for the host. It then attempts to create a socket connection to that host using the IP address and port obtained from the beacon. If a connection is established, the hosts synchronize their `KnowledgeBases` and exchange data and subscription. If both hosts have a parameter in their `KnowledgeBase`, the time of acquisition of the knowledge is used to decide which value prevails. Fresher knowledge is preferred to older knowledge.

The `KnowledgeBase` is modeled as a `Hashtable` indexed by host IP address. This points to another

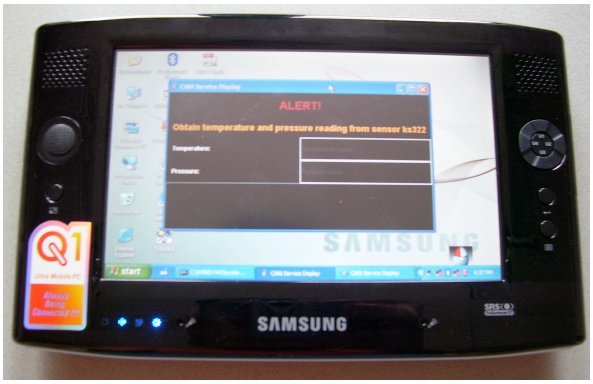

Figure 6. CiAN running on ultramobile PC

`Hashtable` which is indexed by parameter name, e.g., location, and points to the value of that parameter. Hence we can get the value of any parameter by host name and parameter name. This ties in with the `<knowledge-var>` tags on the workflow specification which are in the form hostname:paramname. The various schemes for routing can be implemented in the `WorkflowRouter`. We provide a `DefaultWorkflowRouter` that implements Scheme 1 above. Other schemes can be implemented by extending and overriding the appropriate methods in the `WorkflowRouter`.

The `TaskManagers` for each task run in separate threads and have synchronized access to the `Dispatcher`. The latter initializes the service using Java reflection. The name of the service is assumed to be the class that needs to be initialized. Inputs are passed in as a hashtable that maps variable names to objects that represent their values. The service returns a similar hashtable as output. Any service that is invoked must extend the `CiANService` class so that the appropriate hook methods are available. Figure 6 shows work in progress of the CiAN middleware executing a service on an ultramobile PC. It should be noted that our current implementation is an initial prototype. We are working on developing the next version of our system in which we expect to use existing Web services and related technologies in our middleware, e.g., SOAP for inter-host communication. We also plan to add support for invoking Web services from within CiAN, which will make CiAN an extremely versatile collaboration tool.

6 Evaluation

In the previous section, we showed that the CiAN middleware can execute on ultramobile PCs. In this section, we discuss the performance of the three schemes that we use for routing data between hosts. We simulated the three protocols in the NS2 network simulator. The simulations

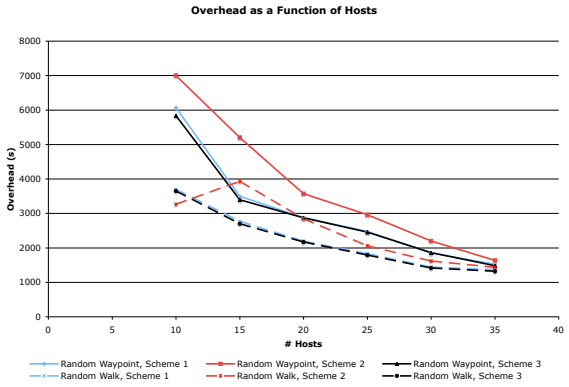

Figure 7. Experiment 3 results

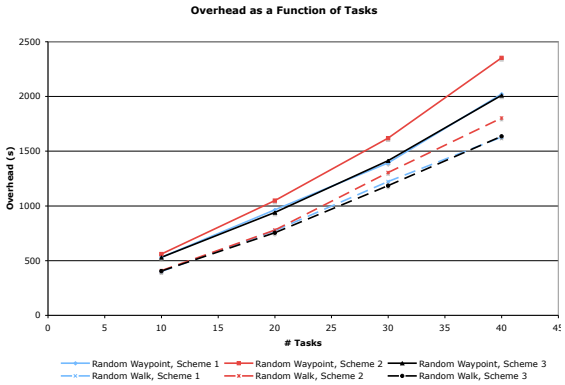

Figure 8. Experiment 3 results

were performed in a 200m x 200m space which represents the size of a large outdoor work area. The transmission range of the mobile devices was set to 25m using the 2-ray ground propagation model and the 802.11b MAC layer was used. Though the range of 802.11b can be higher than 25m, we chose to use a conservative figure since our experience with actual devices showed this to be the most reasonable range in the physical world. The experiments were performed with the hosts moving as per both the random walk and the random waypoint mobility model. In both cases, hosts moved with a uniform speed of 1.7 m/s which is close to human walking speed. In the random walk model, the hosts moved for a random amount of time between 1 minute and 5 minutes. In random waypoint, the host moved until it reached a waypoint. When the hosts paused, they did so randomly in the 1 minute to 5 minute range when ostensibly they were performing some task.

We randomly generated workflows for our tests. We generated a matrix in the range 5x5 to 60x60. The rows and colums of the matrix represented tasks. Using only the cells above the diagonal (to make the resultant graph directed), we marked cells randomly to create an edge between those task pairs. The number of edges chosen was half the total possible number of edges. We then manually added a single source and sink. The tasks were then reordered to ensure they were in increasing order of position in the workflow.

The first experiment performed measured the overhead for executing the workflow while varying the number of hosts. By overhead, we mean the time that was spent transmitting data from one host to another or time that was spent waiting for inputs. As expected, the overhead dropped considerably with increased number of hosts. This is because more hosts equates to more routing options which results in faster data delivery. The results are shown in Figure 7. Each data point is an average of 30 runs with each run representing a different workflow. The overhead time for executing a workflow, while fairly large for small numbers of hosts was still significantly less than the time spent actually performing the tasks in the workflow and as such, considered acceptable. Also as expected, Scheme 3, which is the most flexible performed better than the others. The kink in the trend for Scheme 2 in random walk was due to tasks being assigned to hosts in a small region, which allowed prompt communication and thereby lower overhead of execution.

In the second experiment, shown in Figure 8, we measured the effect of an increase in the number of task in the workflow on overhead. While overhead steadily increased for increasing number of tasks, the per task overhead remained fairly static when the number of tasks was below 30. However, above this number, the overhead per task rose noticeably. This is attributed to the fact that we ran this experiment by keeping the number of hosts fixed at 30. When the number of tasks exceeded 30, some hosts had multiple tasks assigned to them and since we allowed only one task to execute at a time, we found that tasks that were ready in terms of having all their input were waiting for another task on the same host to finish, thereby increasing the overhead.

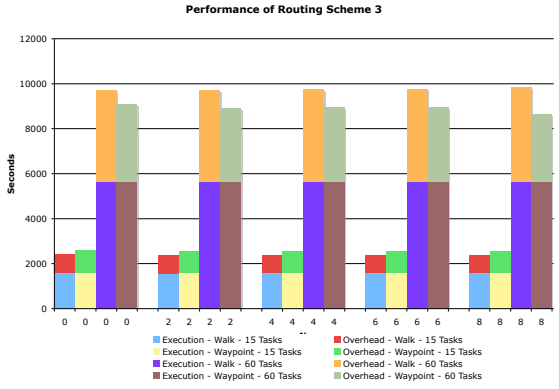

Figure 9. Experiment 3 results

In the final experiment, we studied Scheme 3 in more detail. We ran trials to measure the overhead for a low number of tasks (15) and a high number of tasks (60) in both mobility models when the counter for exceeding the permissible range was varied. Each data point was an average of 20 runs using different workflows. We found that changing the counter value reduced the overhead but not by a significant amount. Given that higher counter values result in higher network traffic, keeping the counter low is desirable and we can do this without much performance penalty.

Our results indicate that our routing protocol based on the task numbers has reasonable performance and a level of overhead that is not a hindrance to the progression of a collaboration. Most significantly, in our trials, higher than 95% of workflows completed successfully despite numerous disconnections and interruptions. The small number that did fail were due to aberrant mobility patterns of one or two hosts that isolated themselves from the rest of the network and did not communicate with their peers, thereby preventing the progress of the workflow. While these results are encouraging, we will focus on optimizing the protocol for reduced network traffic in future work.

7 Conclusion

When WfMSs are ported to a MANET setting, most of the assumptions of stability made by current WfMSs are no longer valid making these systems ill-equipped to function in a MANET. In this paper, we addressed the problem of developing a WfMS for a MANET from the ground up. We started by developing the CiAN language, which allows specification of workflows in a less rigid fashion and incorporates mobility information into the workflow specification. To support the CiAN language, we built a middleware that executes workflows written in CiAN. This middleware is designed to execute workflows in a completely decentralized fashion, relying on non-functional knowledge to make decisions. We also developed a protocol for moving the result of a task from one host to another by exploiting transient communication opportunities among hosts in the MANET. In the future, we plan to study algorithms to determine how tasks get allocated to hosts, which involves work in matching algorithms, as well as strategies such as an auction-based or a marketplace-based strategy, improving the routing protocol for sending data from one host to another, standardizing our middleware using Web services standards, etc. The effort presented in this paper represents an initial foundation upon which we can build more sophisticated features and protocols.

References

[1] P. Athena. Flower user manual, 2001.

[2] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *Proc. of the 13th Intl. World Wide Web Conference*, pages 134–143, 2004.

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

[4] S. Dustdar. Caramba - a process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams. *Distributed and Parallel Databases*, 15:45–66, 2004.

[5] Fujitsu. i-flow developers guide, 1999.

[6] A. Maurino and S. Modafferi. Partitioning rules for orchestrating mobile information systems. *Personal and Ubiquitous Computing*, 9(5):291–300, September 2005.

[7] 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.

[8] R. Muller, U. Greiner, and E. Rahm. Agentwork: A workflow system supporting rule-based workflow adaptation. *Data and Knowledge Engineering*, 2004.

[9] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[10] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[11] V. Sacramento, M. Endler, H. K. Rubinsztejn, L. dos S. Lima, K. Goncalves, and G. A. Bueno. An architecture supporting the development of collaborative applications for mobile users. In *Proceedings of WETICE '04*, pages 109–114, 2004.

[12] 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.

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

[14] K. D. Swenson, S. Pradhan, and M. D. Gilger. Wfxml 2.0: Xml based protocol for run-time integration of process engines. http://www.wfmc.org/standards/docs/WfXML20-200410c.pdf, October 2004.

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

[16] W. M. P. van der Aalst. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.

[17] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[18] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Pattern based analysis of bpel4ws. Technical Report FIT-TR-2002-04, Queensland Univ. of Technology, 2002.