Modeling Distributed Applications for Mobile Ad Hoc Networks using Attributed Task Graphs[1]

P. Basu, Ke Wang, and T.D.C. Little

Department of Electrical and Computer Engineering
Boston University, Boston, Massachusetts 02215, USA
(617) 353-9877
{*basu,ke,tdcl*}*@bu.edu*

MCL Technical Report No. 09-15-2003

Abstract– Mobile ad hoc networks (MANETs) have received significant attention from the research community recently owing to the growth in popularity of portable computing and wireless networking. While researchers have primarily focused on developing lower layer mechanisms such as channel access and routing for making MANETs operational, higher layer issues such as application modeling have largely remained ignored. In this chapter, we present a novel distributed application framework based on *attributed task graphs* that enables a large class of resource discovery based applications on mobile, failure-prone environments such as MANETs. A distributed application is represented as a task comprised of smaller sub-tasks that need to be performed on different classes of computing devices with specialized roles. Execution of a particular task on a MANET requires several logical patterns of data-flow between nodes representing such device classes. As a result, dependencies are induced between the different classes of devices that need to cooperate to execute the application. Such dependencies yield a task graph representation of the distributed application.

We consider the problem of executing distributed tasks on a MANET by means of dynamic selection of specific devices that are needed to execute the sub-tasks. We present simple and efficient algorithms for dynamic discovery and selection of suitable devices in a MANET from among a number of them providing the same functionality. This is carried out with respect to the proposed task graph representation of the application, and we call this process task embedding or anycasting. Since MANETs are prone to disconnections, we advocate periodic monitoring of the selected devices by one another. In the event of an application disruption owing to node mobility or failures, our algorithms adapt to the

[1]In *The Handbook of Mobile Computing*, Eds. Imad Mahgoub and Mohammad Ilyas, CRC Press, 2004. This work is supported by the NSF under grant No. ANI-0073843. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

situation and dynamically rediscover the affected parts of the task graph, if possible. We propose metrics for evaluating the performance of these algorithms and report simulation results for a variety of application scenarios differing in complexity, traffic, and device mobility patterns. We demonstrate by simulation that our protocol can instantiate and re-instantiate TG nodes effectively in mobile scenarios; also the delivered effective throughput is near perfect at low to medium degrees of mobility and moderately high for high mobility scenarios.

Keywords: mobile ad hoc networks, distributed application execution, anycasting, device/service discovery, task graphs

1 Introduction

The shrinking size of tetherless computing devices and increasing diversity of their capabilities has dramatically increased the value of pervasive computing. Exploiting the full potential of a large network of such devices while not frustrating the end-user with interminable configuration tasks poses several interesting challenges for a developer of distributed applications. Wireless networking technologies such as IEEE 802.11b (or WiFi) [12], Bluetooth [9], and Zigbee [18] have begun to enable several distributed applications on truly tetherless computing environments for end-users. These technologies are capable of enabling connectivity between possibly mobile users through infrastructureless networking, also known as mobile ad hoc networking. Formally, a mobile ad hoc network (MANET) is a rapidly deployable, autonomous system of mobile devices which are connected by wireless links to form an arbitrary graph at any instant of time.

With the ubiquity of portable devices and wireless network connectivity, MANETs are likely to gain popularity in the near future, especially in settings where a networking infrastructure is impossible, cumbersome, or expensive to establish. We can conceive scenarios in which the environment surrounding us consists of a large number of specialized as well as multipurpose devices, many of them portable, and linked through wireless connections, albeit with fluctuating link availability. When a large number of computing devices become equipped with wireless connectivity, and they form an ad hoc network, they can offer services to other devices for performing several tasks. Ideally, such pervasive networks can enable a broad range of distributed applications that need exchange of information between multiple devices. In such scenarios, since the service providing devices may themselves be mobile, a user cannot rely on one particular device for a certain service since its reachability or availability is not guaranteed. Instead, a user must be prepared to access the required service from any of several devices in the MANET providing similar services. Besides, the user may not have a preference for a specific device as long as her task is accomplished in a seamless manner.

In order to realize the above features we advocate the decoupling of the logical structure of a distributed application or task (consisting of sub-tasks) from the actual physical devices than execute the application. We propose the use of the Task Graph (TG) abstraction for representing the structure of the user applications in terms of smaller sub-tasks. It is a graph composed of *nodes* and *edges*, where the nodes represent the *classes*[2] of devices/services needed for processing data related to the task while the edges represent necessary associations

[2]Printer, Photocopier, Digital Picture Frame etc. are examples of classes

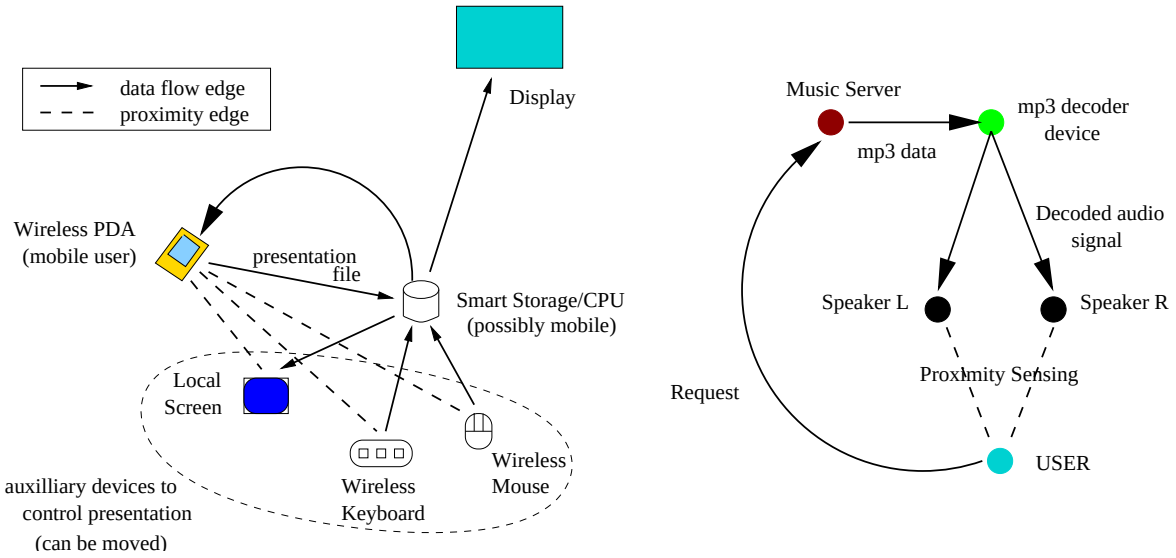


Figure 1: Smart Office and Home Applications: (a) Smart Presentation Task, and (b) Stereo Music Service

between different nodes for performing the task.

Thus when a task is to be executed, specific devices are selected (in other words, *instantiated*) at runtime, and are made to communicate with one another according to the specifications of the TG. More specifically, for each class of device in TG, one suitable instance needs to be chosen to take part in task execution. We call this process *Dynamic Task-based Anycasting* or *Embedding* [6].

When a participating device becomes unavailable, a new substitute device with similar capabilities is selected to continue the task. Therefore, a basic proposition in our model is that as long as there is one *accessible* device in the entire network capable of performing a particular sub-task as requested by the user-level application, the latter can proceed. Obviously, the application should be elastic enough to adapt to the changing conditions of the mobile multi-hop network.

The TG abstraction of a distributed task is advantageous in many ways. It is inherently distributed, as most pervasive applications and services of the future are likely to be, since more and more specialized devices will need to communicate with one another to offer more and more powerful services. It also offers hierarchical composability, as collections of devices can be logically grouped together to constitute a single node in a TG [5].

The rest of the chapter is organized as follows: Section 2 introduces the basic modeling framework with the necessary terminology; Section 3 presents task graph instantiation

algorithms for mapping applications onto MANETs; Section 4 presents simulations results under various degrees of mobility; Section 5 presents related work on the topic; Section 6 concludes the chapter.

2 Modeling Distributed Tasks with Task Graphs

In the past few decades, a variety of distributed applications have been enabled by many advances in computer networking. A distributed networked application or task is composed of several components or sub-tasks. These components often execute on different hardware devices and communicate among each other in order to yield a desired result. Traditional parallel and distributed computing platforms are comprised of high performance nodes internetworked with static high capacity links. However, as mentioned in Section 1, the computation and communication substrate offered by a MANET is potentially mobile and hence, prone to link failures. Therefore, it is necessary to develop a model for a distributed application which decouples the bindings between its logical components and the actual hardware devices that they are executed on until application runtime. Additionally, the model should utilize the component-level structure of an application in order to dynamically discover and select appropriate devices in the network with desired capabilities for hosting and executing the aforementioned application components.

2.1 A Modeling Framework for Task Execution

In this section we propose the modeling framework which advocates the decoupling of the needs and structure of a distributed task from the physical network. We begin with an introduction of the necessary terminology.

2.1.1 Preliminaries

A **device** in our context is a physical entity that performs at least one particular function such as interaction with its physical surroundings, computation, and communication with other devices. It may be equipped with an embedded processing element, sensors and actuators for interacting with the physical environment, a wireless communication port, and/or a user interface.

If a device primarily performs one specific function, it is called a "specialized device,"

otherwise, it is referred to as a "multipurpose device." Examples of the former type include digital cameras, speakers, printers, keyboards, display devices etc., while examples of the latter include Personal Digital Assistants (PDA) and portable notebook computers.

The capabilities of each device can be summarized in their *attributes*. Attributes can be static (i.e., time-invariant) or dynamic (i.e., time-variant). For example, a network digital camera can have a static attribute "resolution" which can take values like 320x240, 640x480 etc. Examples of dynamic attributes include location (absolute or relative, depending on the availability of GPS), available computational power, and current load. In this dissertation, we only consider devices with their principal attribute, (i.e., their primary function). Multi-attribute extensions are possible and are considered elsewhere [1].

A **service** is a functionality provided by a device or a collection of cooperating devices. A service provided by a single device is referred to as a *simple* service whereas one provided cooperatively by a collection of devices is referred to as a *composite* service.

Multiple devices can exist in the MANET for providing the same service. For example, there can be multiple wireless cameras in the network which a user can choose from for taking a picture. We refer to this situation as "multiple instances of wireless camera services."

Service composition is the process of construction of an instance of a composite distributed service from other simple or composite service instances available in the current networked physical space. In this chapter we concentrate on the composition of composite services from simple services only. However, service composition can be carried out in a hierarchical manner – complex services can be constructed from composite services using hierarchical task graphs [5].

A **node** is an abstract representation of a device or a collection of devices characterized by a minimal set of attributes that can offer a particular service.

A node is *simple* when it represents a single physical device. It is *complex* when it represents multiple simple nodes. We refer to the principal attribute of a node or a device as its *class* or *category* or *type*. Examples of classes include printer, speaker, joystick etc.

An **edge** is a necessary association between two "nodes" with attributes that must be satisfied for the completion of a task. Examples of edge attributes include causal ordering, relative importance in the overall task, required data rate between nodes, allowable bit error rate, and physical proximity.

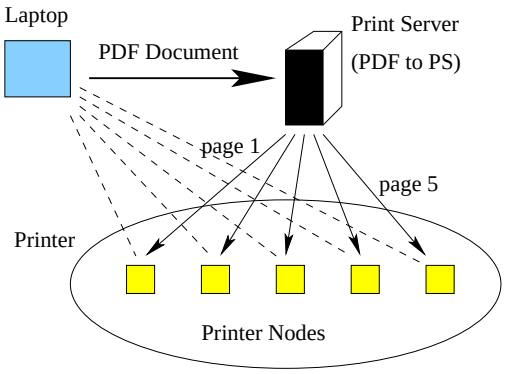


Figure 2: A Smart Printing Service

2.1.2 Tasks and Task Graphs

A **task** can be described as work executed by a node with a certain expected outcome. The work done by a component of a complex node is considered a *sub-task* of the larger task. An *atomic* task is an indivisible unit of work that is executed by a simple node. Atomicity is related to the core capability of a device, described through its attributes, and is partially constrained by subjective design choices.

A **task graph** is a graph $TG = (V_T, E_T)$ where V_T is the set of *nodes* that need to participate in the task T, and E_T is the set of *edges* denoting data-flow between participating nodes.

Instantiation or **Embedding** of a task graph TG on a MANET represented by a graph G is the process of mapping all nodes of TG to nodes in G such that their attributes match. The process also maps edges in TG to paths (single-hop or multi-hop) in G.

We explain the abstractions developed so far by means of a simple example. Consider a scenario in which there is a PostScript (PS) printer connected to a computer (print server) running conversion software that can convert Portable Document Format (PDF) files to printable PS format. The printer node and the computer node each represent *devices* that offer particular *services*. The printer is considered a *specialized device* offering the service of converting PS files into printed pages, while the computer is a *multipurpose device* which has among its many offered services the one service of converting PDF files into PS format. This example is illustrated in Figure 2 where the task of printing a PDF document to a single or multiple printers has been logically represented as a task graph.

The printer is a physical device representation of a *simple node* with certain *attributes*

(such as print resolution, color capabilities) and it offers the service of converting PS files into printed pages. Analogously, the print server computer plus its conversion software can be viewed as a representation of a *PDF → PS converter node*. By taking these two nodes together we can form a *complex node* that offers a "PDF printing service." Let a *task* be the printing of one PDF document. In this specific case, based on subjective criteria, we define an *atomic task* to be the printing of one page of the document.[3] The entire document can be then printed on a set of available printers as shown in Figure 2. The mechanisms of how appropriate physical devices are discovered and selected to perform a sub-task are discussed later in this section.

Note that in the above scenario, we formed a new composite service, *PDF printing*, by composing simpler existing service instances. Although this example is simplistic, we believe that research that enables such capability in today's MANETs for *arbitrary* device types and quantities is essential for exploiting the network's full potential.

2.1.3 A Taxonomy of Tasks

We broadly classify tasks into the following distinct categories:

Preassigned Tasks In this category of tasks, specific devices need to participate – nodes in the task graph already have devices mapped to them and hence discovery is not required. These nodes are referred to as *bound* nodes. Therefore, the problem of embedding a task is equivalent to finding suitable (not necessarily the shortest) routes between pairs of devices that are directly connected by an edge in the task graph. If the optimization variable is "load" on intermediate forwarding devices instead of delay, algorithms for load balancing should be executed instead of a shortest path algorithm.

Non-preassigned Tasks This category of tasks entails a number of homogeneous or heterogeneous computing devices in the network providing specific services. Unlike the *preassigned* case, nodes in the task graph are logical entities and do not signify devices with specified physical addresses. In fact, any device that can satisfy the requirements specified in a TG node's attribute set is a candidate for participating in the task. We, therefore, refer to such tasks as "anycastable." Communication between selected devices need to satisfy the edge attributes as well. Since all nodes in a task graph corresponding to such a task are free

[3]We assume that the printer API does not work at the granularity of printing a dot.

to be chosen, we refer to them as *free* nodes, as opposed to those in a preassigned task which are referred to as *bound* nodes. Optimization of certain performance measures is desirable during the process of instantiation of task graphs. This is described in more detail in Section 2.3.

Partially *preassigned* tasks have a subset of TG nodes that are bound. These *bound* devices have to be selected in the physical network whereas the remaining *free* nodes can be chosen smartly. As in anycastable tasks, the choice of *free* nodes is governed by certain optimization criteria.

Most existing networked distributed applications fall into the *preassigned* category as there is no freedom in the choice of devices and the user decides beforehand which devices participate in the application. We believe that with the advent of pervasive computing, a whole class of *anycastable* tasks will emerge by exploiting the philosophy of loose coupling between services and the devices offering them.

In the context of the smart presentation application, a pocket PDA containing the presentation slides and a particular overhead display can be *bound* devices but the keyboard, the mouse and the smart storage are *free* devices, instances of which can be smartly chosen from the available network.

2.1.4 A Data-flow Tuple Representation Model for Distributed Tasks

In this section, we propose a simple data-flow tuple based model for the high level representation of the logical relationships between different components of a distributed application. The entire application is modeled by a set of tuples each corresponding to a particular data-flow in the application. In other words, each tuple corresponds to a logical unit of data processing that is needed between the distributed components of an application. Every application component is characterized by a *tuple node* with the same semantics as that of a *node* described in Section 2.1. Each unit of data-flow is originated at a certain tuple node and is consumed at one or more terminal tuple nodes (called *sinks*) after being processed and relayed by a set of intermediate tuple nodes. Consider the *smart presentation* application described in Section 1. The following data-flows can characterize a sample presentation:

1. Presenter's PDA (U) sends presentation data (e.g., a Powerpoint slide) to Smart Storage (SS) which hosts appropriate presentation software.

2. Keystrokes are originated at a wireless keyboard (K) by the presenter.

Table 1: Data-Flow Tuples for the Smart Presentation Task

ID	Node	Data-flow Tuples
1	U	$[-; SS]_{ppt}\ [SS; -]_{notes}$
2	SS	$[U; LS, D]_{ppt}\ [K; LS, D]_{keys}\ [M; LS, D]_{clicks}\ [U; \{ppt \rightarrow notes\}; U]_{notes}$
3	K	$[-; SS]_{keys}$
4	M	$[-; SS]_{clicks}$
5	LS	$[SS; -]_{ppt,keys,clicks}$
6	D	$[SS; -]_{ppt,keys,clicks}$

3. Mouse commands are originated at a wireless mouse (M) by the presenter.

4. SS receives presentation data, keystrokes and mouse clicks, processes the data and displays them on a projected display (D) and a local screen (LS). SS also extracts and sends the ASCII part of the presentation and some corresponding notes to the user on her PDA screen (U).

To represent such application data-flow between nodes we employ a generalized tuple architecture. If a node of type X receives data from nodes of types A, B and C, and sends the processed data to nodes of types D and E for a certain application flow (e.g., mouse commands or keystrokes or something more application specific), we can represent this data-flow schematically using the following tuple:

$$X : [A, B, C; \{processing\}; D, E]_{tag}$$

Each data-flow can be uniquely identified at any node by its *tag* attribute. We denote by $\{processing\}$ the transformation of the incoming data units from source nodes before they are transmitted to the destination nodes.

Generating Task Graphs from Tuples: The user node is expected to specify the data-flows in the distributed application as a set of tuples using a standardized language. A Task Graph (TG) representation can be easily generated from a tuple representation – each TG node is derived directly from the corresponding tuple node since it bears one-to-one correspondence with the latter. A TG edge is created between TG nodes X_i and X_j if a data-flow exists between the tuple nodes corresponding to X_i and X_j respectively.

The application data-flows for the smart presentation application can be depicted as tuples as shown in Table 1 and they translate to the task graph shown in Figure 1(a).

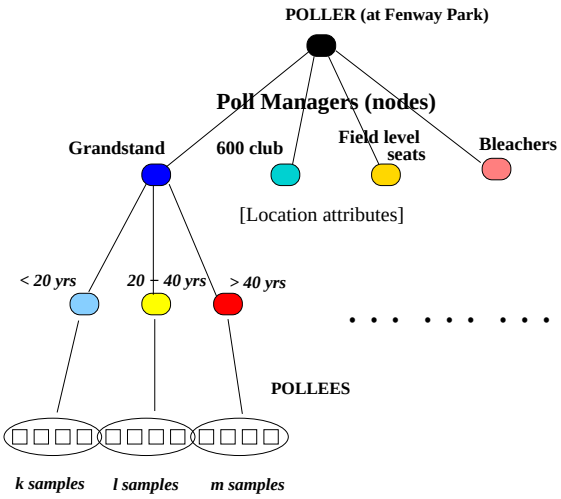


Figure 3: A Task Graph for Location-based Wireless Polling

Advantages of the Tuple Representation: Having a data-flow tuple representation for a task serves two purposes: (1) It is a natural and structured specification of the data-flows in a task from which a task graph can be derived easily, and (2) after the logical resources specified in the task graph are mapped to physical devices in the MANET, tuples govern the flow of actual application data at each participating device.

Examples of data-flow tuples presented in this section contain only the essential information for data exchange, namely the data source and the data destination, and whether the incoming data needs any processing before it is relayed to another device. In general, the edges in a TG can have attributes such as upper bounds on channel error rates, bandwidth, etc. which reflect the quality-of-service (QoS) needs of a distributed application. These, and requirements such as *proximity* (since devices like keyboard, mouse etc. should be located as near the user as possible) can also be integrated in the TG via the tuple architecture. A direct way of incorporating such requirements and task constraints is by specification of *edge attributes* in the tuple. For example, consider a scenario where a node of type X needs to communicate with another node of type D such that the separation between them is no more than 3 MANET hops and that the average delay over that path does not exceed 10 milliseconds. These two requirements are specified as attributes of the edge $e = (X, D)$ in the corresponding task graph: $e.separation \leq 3$ and $e.delay \leq 0.01s$. Implementation details of most of these edge attributes are beyond the scope of this research, and are not considered further.

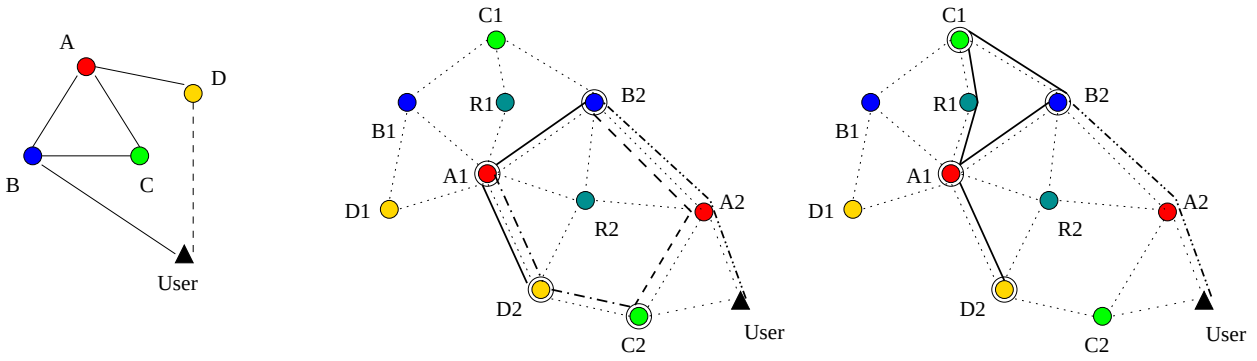
Now we give another example of an application, location based wireless polling, that can be enabled by the proposed attributed task graph framework. Imagine a full capacity

Fenway Park (approximately 34,000) hosting a Red Sox game. As Nomar Garciaparra hits a home run, the stadium authorities decide to poll the people in the stadium with a question: "Was Ted Williams a better hitter than Nomar?" Polling can be achieved over the wireless ad hoc network in the stadium formed by the PDAs owned by the fans.

Suppose that one wants to conduct a poll in a scientific or controlled fashion. Instead of broadcasting the query to *all* PDAs in the stadium and processing all responses, one wants only a fraction of people in the audience to reply as long as people from most profiles are represented proportionally in the poll results. The advantages of doing this are twofold: (i) Less wireless bandwidth will be consumed in the polling process, and (ii) The poll results are likely to represent samples from different sections of the population in a fair and controlled fashion. The extent of fairness and control in the polling process can be defined by the *poller* quantitatively by means of a task graph.

A sample task graph depicting a structured poll is shown in Figure 3. The *POLLER* wants a specified proportion of votes (specified by parameters $k, l, m, \ldots$) from spectators in particular age groups sitting in specific sections of Fenway Park as shown in the figure. The simplest way to perform the poll would be as mentioned before: flood the query throughout the MANET and collect the responses. In addition, only $k, l, m, \ldots$ responses need to be processed by the *POLLER*. Since this wastes wireless bandwidth, expanding ring search (also known as TTL scoping) can be used until the requisite amount of responses have been gathered. However, even this suffers from one problem that virtually all *pollees* will respond to the single *POLLER* node which will be swamped with incoming traffic. In fact the nodes within a few wireless hops of the *POLLER* will be busy forwarding/routing the incoming packets towards it.

A task graph based solution can mitigate the above problems by delegating the task of polling to an intermediate layer of nodes which have enough computing resources and are less power constrained in their operation. We call these nodes *Poll Managers*. They conduct the polls based on the set of profiles that they are responsible for and act as *aggregators* of poll results which are processed and then returned back to the POLLER. If the Poll Managers are spatially spread out uniformly across the network (they are selected based on their location attributes), it will result in less channel contention and hence reduce hot spots in the network. Another advantage of using intermediate poll managers is that they can localize the detection of mobility of a device in the middle of a poll transaction.



(a) Task Graph (b) Embedding #1 (c) Embedding #2

Figure 4: Example of Task Graph Embedding

2.2 Embedding Task Graphs onto Networks

The first step in executing a distributed application on a set of specialized devices is to *discover* appropriate devices in the network and to *select* from the ones who responded, the devices that are suitable for the execution of the more complex application. Mathematically speaking, embedding a task graph $TG = (V_T, E_T)$ onto a MANET graph $G = (V_G, E_G)$ involves finding a pair of mappings (φ, ψ) such that $\varphi : V_T \rightarrow V_G$ and $\psi : E_T \rightarrow P_G$, where the type or *class* of $v \in V_T$ is the same as that of $\varphi(v)$ and P_G is the set of all source-destination paths in G. Figure 4(a) depicts a hypothetical task graph. Figures 4(b-c) show a sample network topology with two possible embeddings of TG on it.

The complete process of device discovery, selection of a device from multiple instances of devices in the same category, and the assignment of a physical *device* to a logical *node* in the task graph is referred to as *instantiation*. We also refer to the collective process of instantiating all TG nodes as *task embedding* or *task-based anycasting* [6].

2.3 Metrics for Performance Evaluation

The embedding function (φ, ψ) maps nodes and edges in $TG = (V_T, E_T)$ to *devices* and *paths* in G. **Average (Maximum) Dilation** of an embedding is the average (maximum) length of such paths taken over all edges in TG. Mathematically, if $\|a, b\|_G$ denotes the length of a shortest path between nodes a and b in G, average and maximum dilation are respectively

given by:

$$D_{avg} = \frac{1}{|E_T|} \sum_{e \in E_T} \|\psi(e)\|_G = \frac{1}{|E_T|} \sum_{(x,y) \in E_T} \|\varphi(x), \varphi(y)\|_G \quad (2.1)$$

$$D_{max} = \max_{e \in E_T} \|\psi(e)\|_G = \max_{(x,y) \in E_T} \|\varphi(x), \varphi(y)\|_G \quad (2.2)$$

Average dilation is a significant metric since it impacts the throughput between instantiated devices. An embedding with large dilation signifies long paths between directly communicating devices, which is undesirable in MANETs since TCP throughput drops significantly with increase in hop distance [15]. In contrast, an embedding with low dilation results in better task throughput. We consider the weighted version of the metric in Section 3.1 where we formally describe the optimal embedding problem.

Instantiation time is a metric which measures the time taken to embed or instantiate all nodes in TG onto G. **Re-instantiation time** measures the time taken to find a replacement device after an embedding is disrupted owing to node, link, or route failures.

Average Effective Throughput, ($AvgEffT$), is the average number of application data units (ADUs) actually received at instantiated data sinks divided by the number of ADUs that were supposed to be received at the intended targets in an ideal situation.[4] Therefore, $0 \leq AvgEffT \leq 1$. It is a useful metric for measuring the resilience of the protocols to failures.

Source-to-sink delay is the latency suffered by an ADU as it funnels itself through various intermediate relay nodes in the instantiated task graph. This metric is useful for measuring application performance during transmission of task data.

The above metrics are useful in the performance evaluation of our embedding algorithms (see Section 4). Additional metrics that have not been investigated in this research have been listed in [4].

3 Algorithms and Protocols for Task Graph Instantiation

In this section we describe how task graphs can be mapped onto MANETs with respect to certain optimization criteria. First, we formulate an optimization problem and show how

[4] If a relaying node in the path from source to sink becomes uninstantiated, effective throughput will be affected because some data-flows will be discarded and will not reach the data sinks.

it can be efficiently solved *exactly* for tree TGs. We then give a greedy heuristic than is amenable to a simple distributed implementation.

3.1 Optimization Problem Formulation

We formulated the constrained task graph embedding problem as the following optimization problem:

> If C be a set of principal attributes (or classes) of specialized devices; $G = (V_G, E_G)$ represents the MANET topology, with the class of each device in V_G belonging to C; $TG = (V_T, E_T)$ is a task graph such that the class of each node in V_T belongs to some $S \subseteq C$; and function $w : E_T \rightarrow \mathbb{R}^+$ defines edge weights which could signify application data-flow requirements, find mappings $\varphi : V_T \rightarrow V_G$ and $\psi : E_T \rightarrow P_G$, where the *class* of $v \in V_T$ is same as that of $\varphi(v)$ and P_G is the set of all *paths* in the network G, such that the weighted average dilation given by:
>
> $$D_{avg}^{(wt)} = \frac{1}{\sum\limits_{e \in E_T} w(e)} \sum_{e=(x,y) \in E_T} w(e) \, \|\varphi(x), \varphi(y)\|_G \tag{3.3}$$
>
> is minimized, where $\|a, b\|_G$ denotes the shortest path between devices a and b in G.

The computational complexity of the general version of the problem where a task graph can have multiple nodes belonging to the same class, and then that of a more specialized version of the problem where all nodes in a task graph belong to distinct classes has been investigated in [4]. The above problem has been shown to be NP-complete in both these situations. However, the problem becomes tractable when the task graph is a tree with nodes belonging to distinct classes; we give an exact polynomial time algorithm for this scenario in Section 3.2. The solution approach in Section 3.2 assumes that the user node possesses the knowledge of the entire network topology as well as that about the capabilities of the devices in the network. In section 3.4, we propose distributed algorithms for embedding, which albeit suboptimal, operate locally and are efficient.

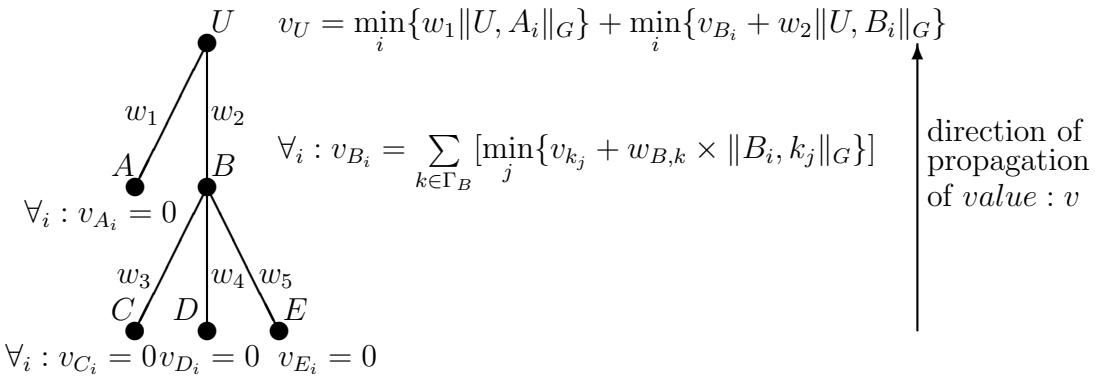


Figure 5: Outline of the Exact Optimal Polynomial-Time Algorithm

3.2 An Optimal Polynomial-time Embedding Algorithm for Tree Task graphs with Distinct Labels

Although the `CC-EMBED` problem is NP-complete with respect to the average dilation metric for the general graphs, there is an interesting special case of a tree which lends itself to an optimal polynomial time solution.

We present below TreeEmbed, an optimal algorithm (with respect to D_{avg}) for embedding a *tree* task graph TG onto a host network G. The running time is polynomial in $|G|$ as well as $|TG|$. The algorithm minimizes searching in the solution space by exploiting the *tree* structure of TG, and is based on the principle of optimality.[5] The algorithm requires that the node executing the algorithm have complete knowledge of the snapshot of the network topology at the given instant of time.

For each node X in TG, algorithm 1 seeks to discover the *best* embedding for each child node Z at every instance (x) of X in G. After the best child candidates are known for all instances, the optimal cost embedding φ^* is selected starting at root node U.

The algorithm proceeds by the propagation of a certain value function $v(.)$ from the leaf nodes of TG towards the root node U. The crux of the idea is that the principle of optimality holds because of the tree structure of TG: if a device instance x of node X is selected by its parent and is optimal, then the choice of instance z (of X's child Z) is optimal too. This greatly reduces the search space for an exact optimal embedding. Moreover, embedding of children nodes can proceed independently of each other because they possess distinct attributes. After carrying out this step for all children of X for each instance x, assign the

[5]The Principle of Optimality holds for problems whose structure is such that their optimal solutions contain the same for the smaller sub-problems [8].

sum of the calculated minimum values to $v(x)$. Figure 5 illustrates the procedure for a task graph of 6 nodes. $\Gamma_B = child(B)$ is the set of children of B in TG. k_j is an instance in G of child k of B in TG.

Algorithm 1 TreeEmbed(TG, G, w, c_1, c_2)

1: **Given:** Tree Task Graph, $TG = (V_T, E_T)$; $w : E_T \to \mathbb{R}^+$; $c_1 : V_T \to C$),
Host Network Graph $G = (V, E)$; $c_2 : V \to C$);
/* C: attribute universe; c_1, c_2: attribute fns.; c_1 is injective; */
2: $\forall X \in V_T : X$ is a leaf in TG, $L[X] \leftarrow 0$; /* assign levels to each leaf node */
3: $\forall X \in V_T : X$ is not a leaf in TG, $L[X] \leftarrow 1 + \max_{Z \in child(X)} L[Z]$; /* and the rest */
4: **for all** $(X : L[X] == 0)$ **do**
5: $\forall x : (c_2(x) == c_1(X))$, $v(x) \leftarrow 0$; /* assign value to matching instances */
6: **end for**
7: **for** $(\ell \leftarrow 1; \ell \leq L_{max}; \ell \leftarrow \ell + 1)$ **do**
8: **for all** $(X \in V_T : L[X] == \ell)$ **do**
9: **for all** $(x \in V : (c_2(x) == c_1(X))$ **do**
10: **for all** $(Z : Z \in child(X))$ **do**
11: $z^* \leftarrow \arg \min_{z \in V \wedge c_2(z) == c_1(Z)} \{v(z) + w_{(X,Z)} \| x, z \|_G\}$;
12: $\varphi_x(Z) \leftarrow z^*$; /* best instance of child node Z for x */
13: $v(x) \leftarrow v(x) + \{v(z^*) + w_{(X,Z)} \| x, z^* \|_G\}$; /* update value of x */
14: **end for**
15: **end for**
16: **end for**
17: **end for**
18: **for** $(\ell \leftarrow L_{max};\ \ell \geq 0;\ \ell \leftarrow \ell - 1)$ **do**
19: $S \leftarrow \{X \,|X \in V_T \ \wedge \ L[X] == \ell\}$;
20: **while** $(X \in S \wedge child(X) \neq \phi)$ **do**
21: $x \leftarrow \varphi^*(X)$; /* note that $\varphi^*(U) = U$ */
22: **for all** $(Z : Z \in child(X))$ **do**
23: $\varphi^*(Z) \leftarrow \varphi_x(Z)$; $\psi^*(X, Z) \leftarrow \|\varphi(X), \varphi(Z)\|_G$; /* optimal embedding */
24: **end for**
25: **end while**
26: **end for**

The running time of the TreeEmbed algorithm can be calculated as follows: assigning "levels" to TG nodes takes $O(|V_T|)$ time. In the worst case, the maximum level of a TG, $L_{max} = |V_T| = O(|V_T|)$; although in more balanced trees, $L_{max} = O(\log |V_T|)$. Suppose there are $|C|$ classes of devices in G with $\frac{|V|}{|C|}$ instances of each class, on average. For every parent instance $x \in V$, each child instance $z \in V$ is considered by the embedding algorithm: the shortest path between x and z is computed (in $O(|V|^2)$ time); the minimization step in line 11 of Alg.-1 is performed (in $O(\frac{|V|}{|C|})$ time). Since this process is performed for all edges in TG

the time complexity of Alg.-1 (lines 7–17) is $O(|E_T| \times \frac{|V|}{|C|}(\frac{|V|}{|C|} \times |V|^2 + \frac{|V|}{|C|})) = O(|E_T|\frac{|V|^4}{|C|^2}) = O(|V_T|\frac{|V|^4}{|C|^2})$. Note that the "for loops" in Alg.-1 (lines 7–8) are subsumed in this calculation and since $|V|$ is the dominant term, the time complexity is given by the above expression itself.

If Warshall-Floyd's all-pairs shortest path algorithm is used (running time is $O(|V|^3)$ and extraction of shortest path cost is $O(1)$ assuming random access storage), then the running time of TreeEmbed is $O(|E_T|\frac{|V|^2}{|C|^2} + |V|^3) = O(|V|^3)$.

3.3 A Greedy Algorithm for Task Graph Embedding

If TG is a general graph (and not a tree), then the task embedding problem is much harder since the principle of optimality may not hold in that case. This is because the optimal embedding of every pair of nodes and the edge connecting them in TG cannot be done independently of other edges and nodes in TG, as can be done if TG were a tree. In the case of a tree TG, as we propagate the *values* from the leaves to the root, the optimal embeddings of each subtree are retained and used later while embedding a node closer to the root. This is not possible for any general task graph with greater connectivity than a tree.

Algorithm 1 suffers from large time complexity even though it is polynomial, the main reason for this being that all devices in the network G are considered as candidates for embedding and the dynamic programming algorithm chooses the best subset among them systematically. Moreover, the algorithm may often fail to run in polynomial time if a few nodes of the same class occurs more than once in TG. Due to these reasons, we developed a simple greedy algorithm GreedyEmbed which is suboptimal (even for trees) but has lower time complexity and works for the case where all node types in TG are not distinct. We briefly describe it below.

The greedy algorithm begins the search for candidate devices from the user node U itself and conducts it in a breadth-first manner. At every step of the Breadth First Search (BFS) process, an unvisited TG node is instantiated greedily by the nearest candidate device in G which matches the requested attributes. Ties are broken arbitrarily and there is no lookahead. Since only nearby devices in G (from the current location) are considered as candidates, searching for the nearest suitable instance of a TG node may not require a complete traversal of G. Hence the algorithm trades off optimality for time efficiency.

GreedyEmbed also possesses a few clear advantages over TreeEmbed in its functionality

and implementation. Unlike the latter, GreedyEmbed can handle the case in which multiple nodes in TG possess the same attributes. Moreover, distributed implementations of GreedyEmbed are facilitated easily due to the nature of breadth first search. We describe a distributed approach based on these principles in the next section.

3.4 A Distributed Algorithm for Task Graph Instantiation

In this section, we present a distributed approach for solving the task graph embedding problem in a MANET with an objective of minimizing D_{avg}. We assume here that each heterogeneous device can provide a single type of service, and that all nodes in the network are *simple*. The additional nuances of the homogeneous case have been elaborated upon in [4]. We assume the presence of a MANET routing protocol (DSR) and a reliable transport protocol (TCP) for control and application data packet transmission.

All devices in the network execute copies of the same algorithm except the user node, U, which executes a different algorithm since it acts as a *state synchronizer* or coordinator in the initial phases of the embedding process. In our opinion, the user devices are best suited for acting as coordinators since they usually originate the application data flows, and even under mobility, always remain near the user.

The embedding process begins at U with a distributed search which proceeds through the MANET G hand-in-hand with a *breadth-first search* (BFS) through TG. Figure 10 depicts a task graph with its BFS and non-BFS edges. We call the spanning tree on TG induced by BFS and rooted at U, a BFS tree ($BFST_{TG}$) of TG. We propose a *greedy* solution much like the GreedyEmbed Algorithm described in section 3.3 to keep the dilation of the embedding low: the algorithm begins from U by progressively mapping the nodes of $BFST_{TG}$ to nearest devices and the edges to shortest paths in G. Instantiation of any pair of nodes $x, y \in V_T$ cannot affect each other if x is not a parent of y in $BFST_{TG}$, or vice-versa. Hence, the search can proceed in a distributed manner along the branches of $BFST_{TG}$.

A sample path of the instantiation protocol helps illustrate the salient steps of the algorithm. These have been shown in Figure 6 as a message exchange diagram, and are also summarized below. Details of the protocol including finite state machine descriptions can be found in [6, 4].

1. U broadcasts search queries for each neighbor category in TG[6] (A and B).

[6]The broadcast is controlled by sending the query packet to all one-hop neighbors which examine its

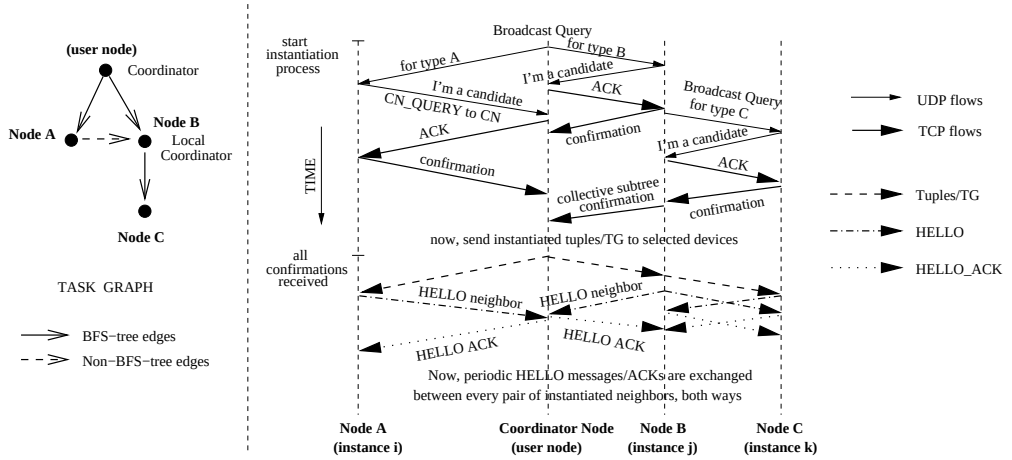


Figure 6: Task Graph Instantiation Protocol

2. Available instances of each queried node reply to U. Candidate devices that reply first (A_i, B_j) become the chosen *instances* at U.

3. U sends an ACK to these selected devices which send back confirmations.

4. If there are any uninstantiated nodes rooted at any instance in TG (such as C below B_j), then it broadcasts search query packets for all those node categories and the instantiation proceeds further[7].

5. When confirmations from all nodes reach U, the data transmission can begin[8].

The task graph itself is sent as control data during the instantiation process. After the selection of a device, control packets and application data are transmitted using TCP since packet losses due to route errors are very common in MANETs.

3.4.1 Handling Device Mobility

When mobility causes network partitions or disconnections, the instantiated devices may no longer be able to communicate if the partition breaks all paths between them. In such situations, new instances need to be selected. The necessary first step in this direction

contents and decide whether to rebroadcast it. A time-to-live (TTL) field in the packet also prevents it from causing a storm.

[7] B_j here acts as the local coordinator responsible for instantiation of nodes rooted below it.

[8] In an ideal situation, all data originating at the source should reach the instances of the sink nodes in TG (A_i and C_k in the example in Figure 6) after having been massaged and relayed by the intermediate devices (B_j).

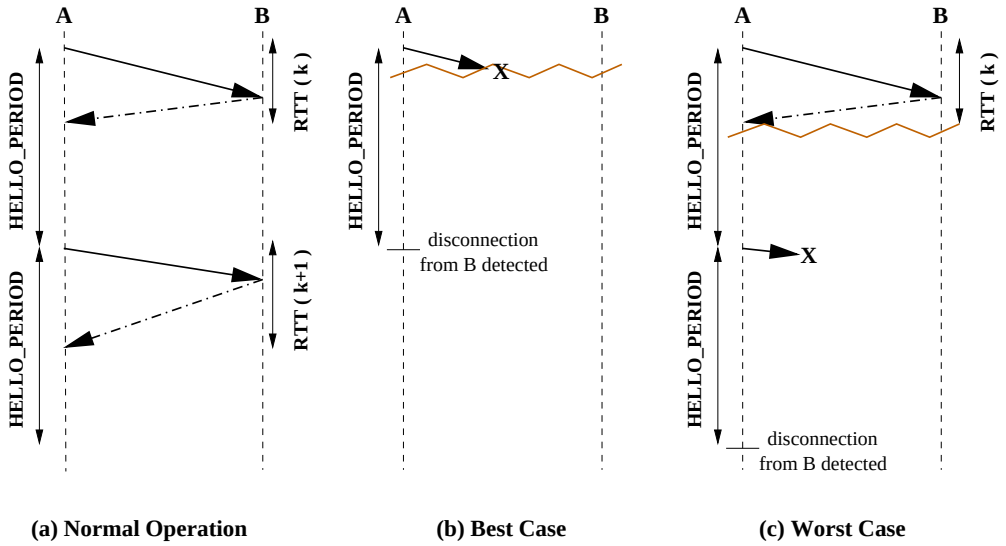
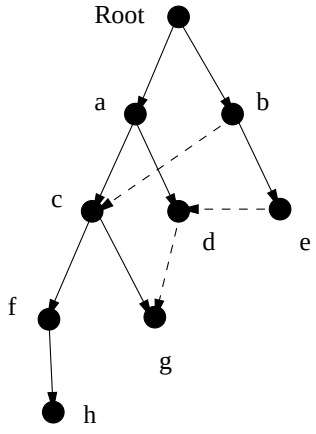


Figure 7: Detection of Disconnections: A and B are Parent–Child Instances



Node	Parents	Children	Grandchildren	Siblings
Root	–	a,b	c,d,e	–
a	Root	c,d	f,g	b
b	Root	{c},e	d,f,g	a
c	a,{b}	f,g	h	d
d	a,{e}	{g}	–	c
f	c	h	–	–
g	c,{d}	–	–	–
h	f	–	–	–

Figure 8: Logical Neighbor Table Information at Instantiated Nodes: letters enclosed in {..} represent devices that are non-BFS parents and non-BFS children.

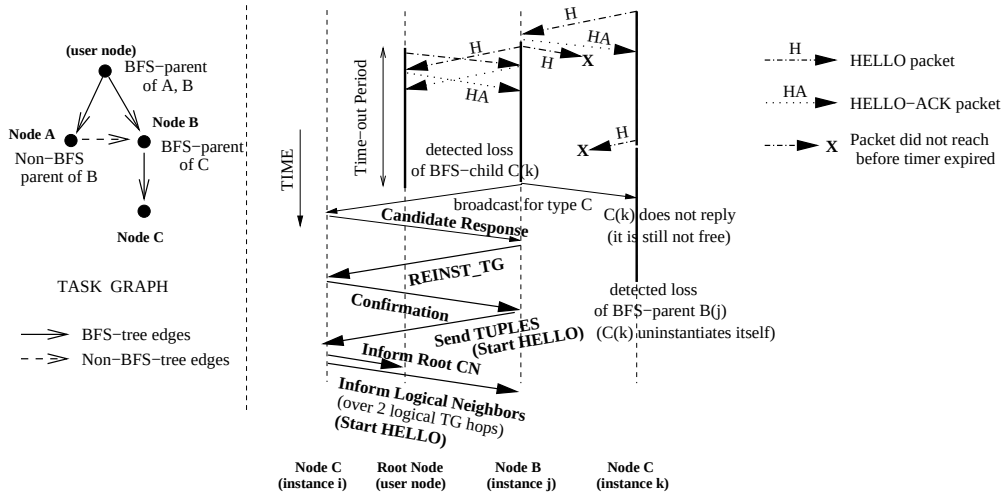


Figure 9: Handling Task Disruptions by Re-instantiation of Devices

is the *detection* of disconnections. We propose a lightweight, soft-state exchange protocol for detecting disconnections in an instantiated TG. The protocol requires each instantiated device to send *periodic* `HELLO` messages (with period T) to its logical neighbor instances in TG, which reply with a `HELLO-ACK`. This has been demonstrated in Figure 9.

Specifically, each instantiated device keeps track of its BFS parent and BFS children and some additional information. Figure 8 shows the information that each instantiated device stores for detecting disconnections and performing recovery – we refer to this as the 2-hop logical neighborhood information [4]. The instance of node C (denoted by 'c') keeps track of the instance of node A (its BFS parent) as well as of the instances of nodes F and G (BFS children). If a BFS parent device stops hearing from one of its BFS children[9], it uninstantiates its child and starts looking for a replacement of the same type. The child meanwhile would stop hearing `HELLO-ACK`s from the parent (assuming bidirectional links), and will uninstantiate itself. This has been illustrated in Figure 9. On average, if the `HELLO` timer at every instantiated device is set to time period T, disconnections will be detected after time $\frac{3}{2}T$.

Although 1-hop logical neighborhood information is sufficient for detecting disconnections in the instantiated TG, 2-hop logical neighborhood information as illustrated in Figure 8 is necessary and sufficient to recover from *single* persistent disconnections without any extra unnecessary re-instantiations of downstream nodes in TG [4]. The process of re-instantiation involves (1) discovery of a suitable replacement; (2) transfer of instantiated

[9]The parent concludes this if it does not get a `HELLO-ACK` from that child before the expiry of its `HELLO` timer

logical neighborhood information from parents and children; and (3) resumption of application data transfer. We do not discuss these schemes in detail in this chapter and direct the reader to references [6, 4].

Impact of Disconnections on Application Layer The application layer of every participating device keeps up-to-date (in-out) tuple information for parent and children devices. If disconnection of some participating devices disrupts a running task, then it is the responsibility of the BFS-parent node to transfer the application state to the newly instantiated replacement device, and then resume the application data-flow. Meanwhile data packets reaching old node instances are dropped by those devices. The average effective throughput ($AvgEffT$) metric tries to capture the effectiveness of our disruption handling algorithm by measuring the fraction of the data that actually reached the "current" data sinks from the source. An application layer buffer management scheme at the BFS-parent node instance can increase the reliability of task completion. We plan to investigate these issues in future.

Mobility of devices may also result in lengthening or shortening of routes between device instances, and ideally, if there is no disconnection/partition, the application should proceed without disruption. But such ideal conditions may not hold in reality where route failures can trigger route discovery which along with TCP re-transmissions after timeouts may sometimes take several seconds to complete. Hence, this can result in `HELLO-ACK`s not coming back in T seconds which results in the conclusion that a disconnection has happened, even when the nodes are reachable from one another.

Researchers have proposed solutions to the above problem based on explicit notification of route errors to TCP [11]. However, in this work, we do not attempt to alter TCP or DSR (including their default timer settings); we simply develop our protocol on top of these protocols. Hence, if a device does not receive a `HELLO-ACK` from its neighbor in T seconds, we deem the neighbor to be disconnected. A reasonable value of T is one which is not low enough to cause significant control overhead[10], and not high enough such that disconnections are not detected fast enough. For our simulations, we chose $T = 7\,seconds$ ($> 6s$, the default TCP re-transmission timer).

[10]Although exchanging `HELLO` messages with higher frequency could result in the DSR caches having fresher routes

4 Performance Evaluation

We simulated the greedy instantiation/re-instantiation algorithms proposed in Section 3.4 using the network simulator *ns-2*[2]. 100 mobile devices with specialized roles (represented by their principal attributes) were simulated in a 1500×600 areas. Node motion was governed by the *random waypoint* mobility model; the velocity was randomly chosen from $[0, v_{max}]$ where $v_{max} = \{1, 5, 10, 15, 20\}$ m/s. We assume that the devices are constantly moving between waypoints. The simulated transmission range for each node was 250m. We show simulation results for task-graphs in Figure 10 – we refer to them as Tree TG, Non-Tree TG-1 and Non-Tree TG-2 respectively. The principal attribute of each device belonged to one of 12 different classes. The attributes were uniformly distributed across the MANET.

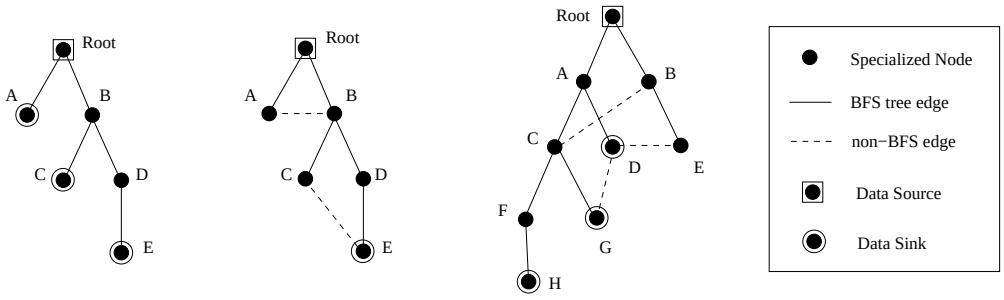


Figure 10: Task Graphs used in Simulation Studies

The total simulation time was 400s – the instantiation process began at 200s, and at 600s, the *user/root* node started sending data to the data sinks. The data flow consisted of a CBR source, with a burst of S bytes of data every T seconds. We report results for $(S, T) = (2500, 1)$. Devices which are not part of the instantiated TG do not forward packets, and such packets are not buffered; in other words, if a device which was part of an instantiated TG becomes disconnected while there is a packet in transit, the packet is lost.

Dilation First we analyze the constant mobility scenarios for different simulation parameters. We first evaluate the quality of embedding using the average dilation metric. For every mobility scenario, dilation is measured initially after completion of instantiation and subsequently after every re-instantiation event. These values are then averaged over the simulation time period to yield one number. We observe from Figure 11 that average dilation for the embedding scheme does not vary greatly with speed; in fact d_{avg} lies between 1.25 and 2 for all three task graphs at all different values of MaxSpeed. This means that the average number of physical hops between two instantiated nodes in TG is low and remains approximately

constant under mobility. This is because of the approximately uniform spatial distribution of device categories and the reasonable abundance of devices of each category in the network (5 to 13 of each type).

However, we do observe that d_{avg} increases when the maximum speed is increased above 1 m/s. The principal reason for this is the following: at 1 m/s speeds, re-instantiations are rare and the d_{avg} does not deviate too much from its value after initial instantiation. However, at greater speeds, re-instantiation events occur more frequently because of logical neighbor instances either having moved far away from each other or having been disconnected by a network partition. Either of these events disrupts the usual smooth exchange of `HELLO` message resulting in re-instantiations. Owing to the uniform distribution of device categories in space, the re-instantiation process will find another device with similar attributes within its vicinity. Although that keeps the contribution of the new path length towards d_{avg} low, the hop distances between existing instances along other TG edges are likely to have increased over time (although not high enough to cause re-instantiations along those edges). This causes d_{avg} to increase at higher speeds on the whole.

Another observation from Figure 11 is that at lower speeds, d_{avg} is lower for TG1 (a tree) than TG2. This is obvious because, our heuristic algorithm attempts to minimize the hop count only along the BFS-tree edges of a task graph both during instantiation as well as re-instantiation; since TG2 has extra edges, the minimization does not occur along those edges, thus yielding a higher dilation, in general. The above reasoning does not hold at high rates of mobility as all instantiated paths break more often and device category distribution is spatio-temporally more uniform in the neighborhood of a device. Hence, non-BFS-tree edges are likely to be mapped onto paths with similar lengths as BFS-tree edges quite often, and that causes d_{avg} to be similar for both TG1 and TG2.

Embedding Time Table 2 compares the times taken for embedding each task graph on the network. We depict the minimum, maximum, and median times for each TG for three different maximum speeds. We show the median instantiation time instead of the average instantiation time since the time samples are skewed. Generally, the times for TG2 exceed those for TG1 and Tree since the former is a larger task graph and it needs exchange of packets between a larger number of devices during instantiation. Some samples are much greater than the rest owing to the role of TCP (over DSR routing protocol) in the instantiation process. After the candidate's response reaches a coordinator node, it sends ACKs encapsulated in TCP packets since they can be lost if sent using an unreliable

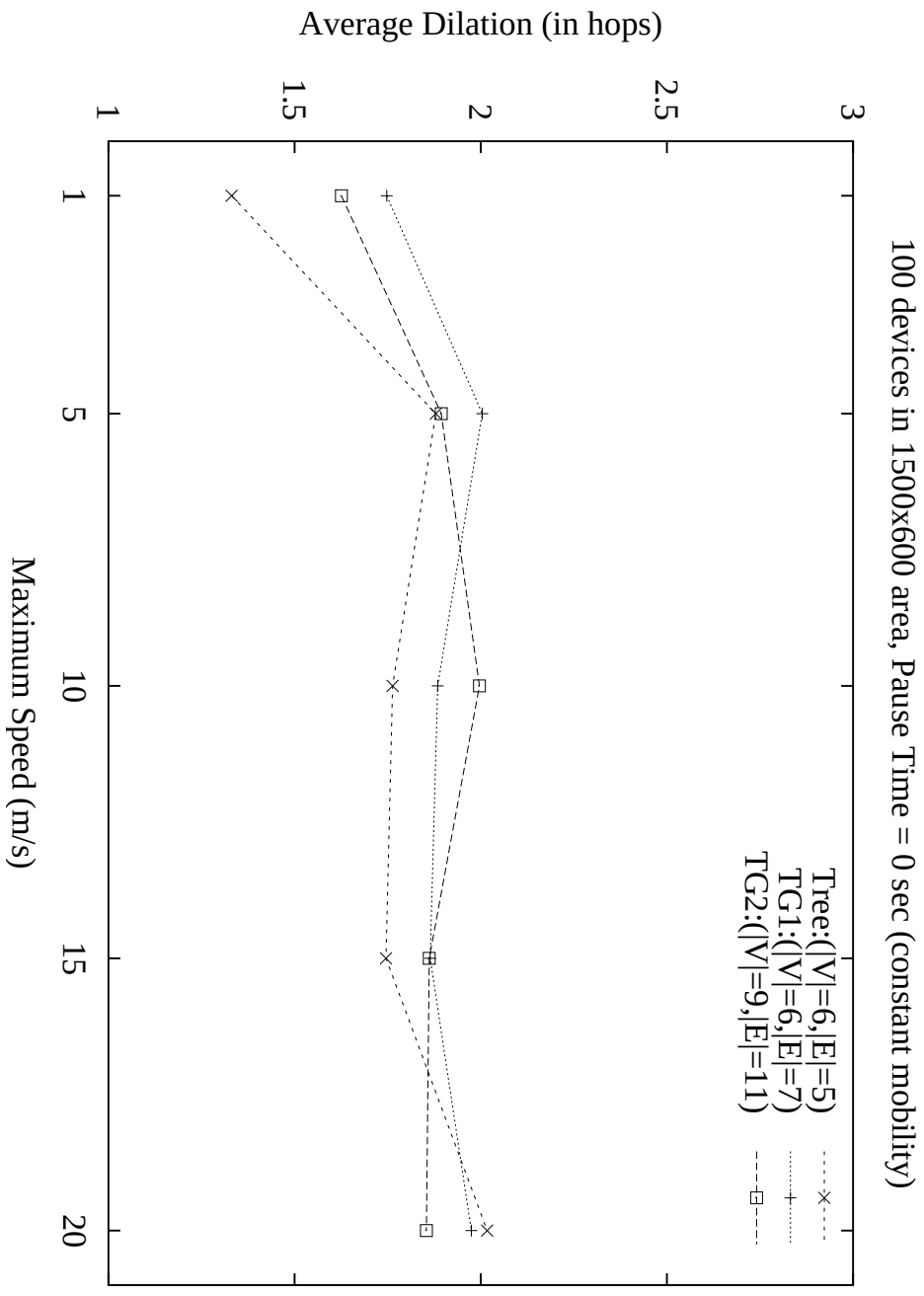


Figure 11: Average Dilation vs. Maximum Speed

TG/Scenario	Minimum (s)	Maximum (s)	Median (s)
Tree (1m/s)	0.795719	6.561610	1.435320
TG1 (1m/s)	0.810867	6.819640	1.399530
TG2 (1m/s)	2.170060	7.957830	6.674960
Tree (5m/s)	0.670853	6.111210	1.728970
TG1 (5m/s)	0.536686	7.708620	6.278840
TG2 (5m/s)	1.742180	9.537000	7.827160
Tree (10m/s)	0.643709	1.438240	1.216280
TG1 (10m/s)	0.842213	6.694860	1.530080
TG2 (10m/s)	3.337190	9.168950	7.275040
Tree (15m/s)	0.749414	4.039460	1.062140
TG1 (15m/s)	0.446600	6.511620	0.909011
TG2 (15m/s)	1.520370	4.090240	3.241610
Tree (20m/s)	0.651414	2.062220	1.088190
TG1 (20m/s)	0.717359	4.022630	1.484370
TG2 (20m/s)	1.361380	7.674460	5.262870

Table 2: Task Embedding Time

transport protocol. TCP is also used in all subsequent communication (except broadcast and candidate response packets).

Now, if for some reason a route error occurs while a TCP transmission has not completed, TCP attempts redelivery only after waiting for a period of time even if a new route is rediscovered immediately by DSR. This period can be as large as 6 seconds (default retransmission timer of TCP) if no prior communication has happened between the two communicating devices. If a route error occurs shortly after two devices have communicated using TCP but before another TCP transmission is completed, the retransmission timer is set based on the round trip time estimate between those two devices and hence it can be lower than 6 seconds. Hence we see instantiation time samples greater than 6 seconds on several occasions. If mechanisms such as explicit feedback [11] are added to TCP, then these times can be reduced significantly. Also, no monotonic pattern is observed as a result of the increasing mobility of devices. This can be attributed to the uniform spatial distribution of device classes in all random mobility patterns as well as the large variability in TCP timers during the multiple steps of the instantiation process.

Effective Throughput After the completion of the instantiation process, we begin data transmission from the user node (source) to the various sinks shown in Figure 10 according to particular tuple specifications. In Tree TG, instances of A, C and E receive one flow

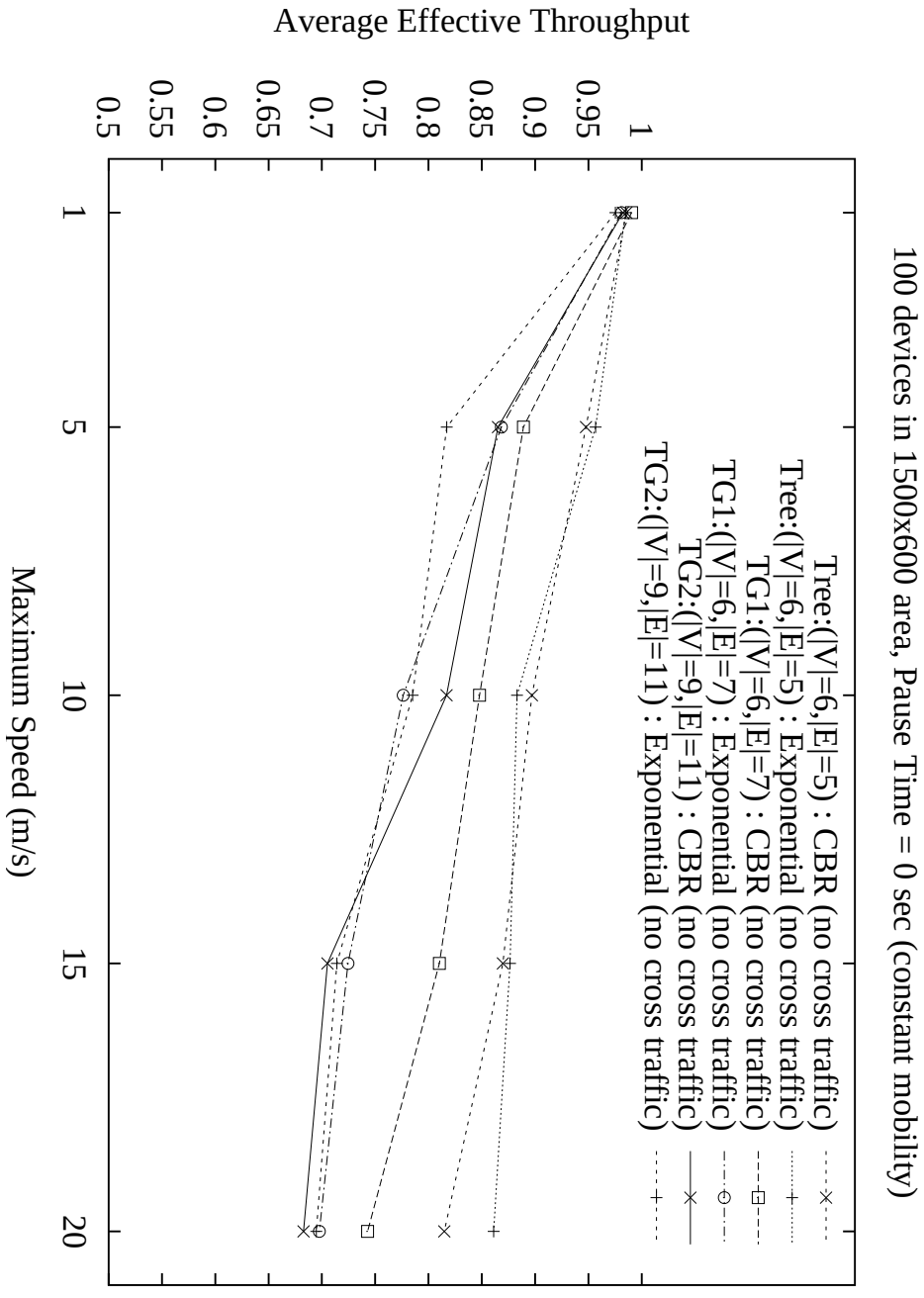


Figure 12: Average Effective Throughput vs. Variation of Maximum Speed

each. In TG1, the instance of E receives 4 flows through instances of various relay nodes. In TG2, instances of D and H receive one flow each and the instance of G receives 4 flows. We plot normalized $AvgEffT$ for all three TGs in Figure 12. We generate task data traffic using two different patterns: periodic constant bit rate (CBR) bursts and bursts with exponentially distributed sizes after exponentially distributed inter-arrival times (resulting in Poisson distributed bursts). The mean burst sizes and inter-arrival times are kept constant for both cases. A maximum aggregate throughput of $300kbps$ can be reached for the TG2 scenario assuming simultaneous transmission at all instantiated devices in accordance with the underlying tuple architecture.

In Figure 12 we can see that at low mobility, $AvgEffT$ is almost perfect (close to 1.0). We can also observe that in general, $AvgEffT$ drops with increase in the maximum speed of devices for most situations. This is to be expected since higher speeds generally result in more re-instantiations and that results in more ADUs not reaching their intended destinations. However, $AvgEffT$ rarely drops below 70% in the simulated scenarios even under heavy mobility. This demonstrates that our protocols adapt fairly well to mobility and are able to recover from disruptions in task data flow. We can make some more observations from the two figures: (1) Exponential traffic pattern occasionally results in a lower throughput than the CBR traffic pattern in scenarios involving non-tree task graphs, and (2) TG1 usually yields lower throughput than Tree TG.

Exponentially distributed data generation times can occasionally result in large periods without much network activity, and this causes the on-demand routing protocols to lose routes to destinations. More route errors cause more frequent TCP back-offs and sometimes result in re-instantiation even if the devices are graph-theoretically reachable from one another. Loss of throughput is greater in the case of non-tree TGs than Tree TG because recovery from the loss of a non-BFS child usually takes more time than a BFS child. On the contrary, in the CBR case, periodic generation of packets keeps routes fresh and hence the TG suffers less re-instantiation.

Number and Time of Re-instantiation Figure 13(a) shows the average number of re-instantiations underwent during the entire simulation time (400s). The rate of change in network topology increases with mobility causing more network partitions or route errors. These events in turn prevent `HELLO` packets from arriving in time, and thus triggering more re-instantiations. Since packets caught in transit during the re-instantiation process are dropped (as mentioned earlier, we do not consider application layer buffering in this work),

Figure 13: (a) Number of Re-instantiations, (b) Average Re-instantiation Time

$AvgEffT$ is directly affected by re-instantiations.

Although Tree TG is a sub-graph of TG1, for the CBR data case, TG1 suffers less re-instantiations because data flow along the non-BFS edges of TG1 results in the presence of more valid alternate routes (or parts of them). Hence, when a route error happens along a BFS edge (the primary cause of re-instantiations) of TG1, often these alternate routes come to the rescue before the `HELLO` timer expires, thus reducing the rate of re-instantiations. TG-2 generally suffers more re-instantiations since it is a larger graph with more depth.

In spite of Tree TG having more re-instantiations than TG1, it experiences better $AvgEffT$ than TG1. This is because the data tuples of TG1 (as well as TG-2) involve flows along non-BFS edges in the graph. Also, the set of re-instantiation events is only a subset of the set of all disruptions. When a non-BFS parent loses a child instance momentarily due to partitions or `HELLO` timeouts, a re-instantiation will not be triggered since that is the responsibility of the BFS parent of the child instance; Hence, the throughput is affected until a new instance is found by a BFS parent and the non-BFS parent is informed of this event by a 1-logical-hop broadcast, or a route to the old instance is restored. Also, Tree TG has sinks at all depths unlike TG1 – hence the latter's effective throughput suffers more from a re-instantiation of an intermediate relay node. Exponential traffic generally affects re-instantiations more than CBR traffic especially for the non-tree graphs as explained before. The result of that is slightly lower throughput in the respective cases.

Figure 13(b) shows the variation of times taken to re-instantiate a TG node, i.e. the times taken to discover a new replacement for a disconnected device which can participate in the task. This time is measured from the time when the rediscovery broadcast is sent out until the time instant when a confirmation is received from the new candidate (this involves 2 round-trip handshaking steps including the broadcast). Our re-instantiation protocol is able to find a new device nearby within 1 second. In fact, in most cases, these times are only a few hundred milliseconds. Local network effects are dominant factors in the determination of this metric at higher speeds, hence there is not much correlation between the values in such cases.

Cumulative ADU Delay Distributions We now examine the nature of the delay distributions that occur as a result of sending task data using CBR and Exponential traffic patterns. Figure 14 shows the empirical cumulative probability distributions (cdf) of ADU delay samples. A logarithmic scale is used for the delay samples in order to differentiate between delays at lower and higher ends more effectively. In Figure 14(a), delays for the static case are

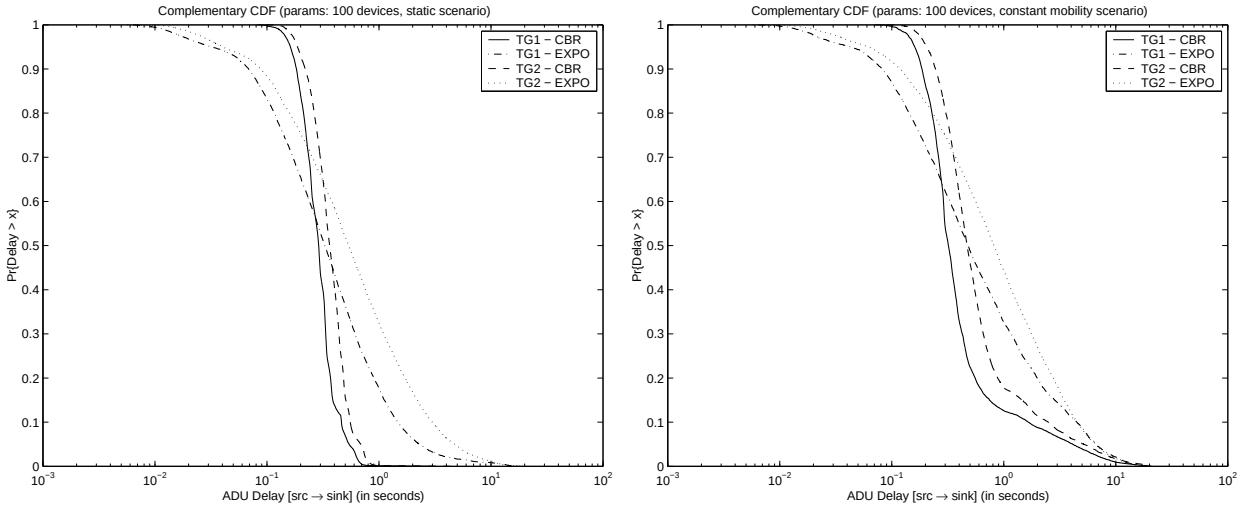


Figure 14: Cumulative Probability Distribution for ADU Delay: (a) Static, (b) Constantly Mobile

plotted. We observe that CBR delay values span a much smaller range than their exponential counterparts. The shape of the task graph does not seem to affect that of the CDF curves. That is primarily because the distribution of sinks in both TG1 and TG2 have a common aspect which is a dominant factor in the determination of ADU delays – two sinks each in TG1 are 3 and 4 logical hops away from the source, respectively, while in TG2, four sinks are 3 logical hops away and two sinks are 4 logical hops away from the data source.

CDF curves of delays in the constant mobility scenario have been plotted in Figure 14(b). We can easily see that although the shapes of the curves are similar at lower values of delay, they become much flatter and somewhat heavy tailed at larger values for both TGs and traffic patterns. These samples correspond to ADUs which had to experience delays due to route errors and expiry of TCP timers. In this work, we do not attempt to investigate the exact statistical nature of the distribution, and leave that as a topic of future research.

5 Related Work

Service discovery in networks has been a popular topic of research in the industry as exemplified by SLP [13] and Sun's Jini [17]. In both these schemes, a service providing computer registers itself with its attributes at a centralized directory server which the clients can lookup on demand. MOCA is a variation of Jini without any centralized registry [7]. It is specifically designed for mobile computing devices – every device has a service registry component which only the applications running on the local and surrounding devices can benefit from. Our

approach is different from these as it operates at a logical layer above service discovery and it can co-exist with any of these schemes. Also, it does not depend upon any centralized directory service.

INS proposes to capture *user-intent* for discovering appropriate devices suitable to them. The user intent is abstracted into collections of attribute-value pairs that describe the needs of the user. The specific devices that will perform the desired service will be selected by special entities called Intentional Name Resolvers. INS has a feature called Intentional Anycast and late binding which is somewhat similar to what we call *instantiation* of TG nodes. However, INS does not to attempt to systematically utilize the logical structure of a distributed task for resilient application execution.

Hodes et al. [14] have investigated means of composing services for heterogeneous mobile clients. Their work primarily focuses on controlling office equipment from mobile devices and design of client-device interfaces. They too have not addressed the issues involved in composing complex services from simple devices with specific interaction patterns between them. In general, none of the aforementioned approaches consider scenarios in which multiple specialized devices need to offer their services in a cooperative manner for the provision of a more complex service, a case which we believe will be increasingly common in a ubiquitously networked world.

IBM's PIMA [3] has a vision somewhat similar to ours. Although they argue briefly for the design of applications in terms of sub-tasks instead of specific devices, they have not mentioned any approach for realizing this vision so far. Our task-graph concept on the other hand is a systematic and concrete approach which can help realize this vision.

The concept of a task graph was originally proposed in the parallel computing and scheduling literature for representing tasks that can be split *temporally* into sub-tasks and then allocated to different homogeneous processors connected by a fixed high-performance interconnect for reducing the total completion time [10, 16]. Our notion of a task graph is different from this classical one. We are not necessarily concerned with tasks that are distributable among multiple homogeneous processors for speed-up. Rather, most tasks that we are concerned with in this work involve several specialized heterogeneous devices that communicate with each other and are possibly mobile, and there is no notion of minimizing the total completion time. However, if we are interested in solving a large scale distributed computing task on a network of homogeneous mobile devices, then our notion of a task graph will be similar to the classical one. Therefore, our task graph formulation is more general than the one used in the parallel computing context.

6 Conclusion

In this chapter, we presented a framework for embedding and executing a distributed application on a network of specialized, potentially mobile devices. We developed a *task graph* abstraction for applications by taking into account the dependencies induced by the data flows existing between the components of an application. We described the task embedding problem and presented an optimal polynomial time algorithm with respect to an average hop-count measure called dilation, for the special case where the task graph is a tree. We also described how it can be heuristically extended for general graphs. Owing to the unreasonable requirements and time complexity of the aforementioned algorithm, we presented a more practical distributed heuristic algorithm (and protocol) for embedding a given task graph onto a MANET. We also presented a scalable, local disconnection detection and repair mechanism for recovering from task disruptions caused by node mobility and failures.

We showed by simulations that our protocols are able to instantiate and re-instantiate task graphs satisfactorily in constantly mobile scenarios although the use of a better reliable transport protocol than TCP can yield better performance. As a part of future work, we plan to investigate mechanisms of developing user level applications on top of the TG layer described in this chapter. These applications will be completely oblivious of the TG node–physical address mappings during their execution and this can be a major benefit in failure prone mobile networked environments.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. "The Design and Implementation of an Intentional Naming System". In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[2] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. "Improving Simulation for Network Research". Technical Report 99-702, University of Southern California, Los Angeles, CA, March 1999. URL: `http://www.isi.edu/nsnam/ns`.

[3] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. "Challenges: An Application Model for Pervasive Computing". In *Proceedings of the 6th ACM MobiCom Conference*, Boston, MA, August 2000.

[4] P. Basu. *A Task Based Approach for Modeling Distributed Applications on Mobile ad hoc Networks*. PhD thesis, Boston University, Boston, Massachusetts, May 2003. Available online at: `http://hulk.bu.edu/projects/adhoc/Basu-PhDThesis2003.pdf`.

[5] P. Basu, W. Ke, and T. D. C. Little. "Scalable Service Composition in Mobile Ad hoc Networks using Hierarchical Task Graphs". In *Proceedings of the 1st Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, Sardegna, Italy, September 2002. IFIP.

[6] P. Basu, W. Ke, and T. D. C. Little. "Dynamic Task Based Anycasting in Mobile Ad Hoc Networks". *ACM/Kluwer Journal for Mobile Networks and Applications (MONET)*, 8(5), October 2003. In Press.

[7] J. Beck, A. Gefflaut, and N. Islam. "MOCA: A Service Framework for Mobile Computing Devices". In *Proceedings of the International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, Seattle, WA, August 1999.

[8] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, 1995.

[9] Bluetooth Consortium. URL. `http://www.bluetooth.com`.

[10] S. Bokhari. "On the Mapping Problem". *IEEE Transactions on Computers*, 30(3), 1981.

[11] K. Chandran, S. Raghunathan, S. Venkatesan, and R. Prakash. "A Feedback-Based Scheme for Improving TCP Performance in Ad Hoc Wireless Networks". *IEEE Personal Communications Magazine*, February 2001.

[12] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai. "IEEE 802.11 Wireless Local Area Networks". *IEEE Communications Magazine*, 35(9):116–126, September 1997.

[13] E. Guttman. "Service Location Protocol: Automatic Discovery of IP Network Services". *IEEE Internet Computing*, July 1999.

[14] T. Hodes, R. Katz, E. Servan-Screiber, and L. Rowe. "Composable Ad-Hoc Mobile Services for Universal Interaction". In *Proceedings of the 3rd ACM MobiCom Conference*, Budapest, Hungary, September 1997.

[15] G. Holland and N. Vaidya. "Analysis of TCP Performance over Mobile Ad Hoc Networks". In *Proceedings of the 5th ACM MobiCom Conference*, pages 219–230, Seattle, WA, August 1999.

[16] R. Monien and H. Sudborough. "Embedding one Interconnection Network in Another". *Computing Suppl.*, 7:257–282, 1990.

[17] S. Oaks and H. Wong. *JINI in a Nutshell*. O'Reilly, first edition, March 2000.

[18] ZigBee Alliance. URL. `http://www.zigbee.org`.