Dynamic Task Based Anycasting in Mobile Ad Hoc Networks[1]

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

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

MCL Technical Report No. 07-22-2003

Mobile ad hoc networks (MANETs) have received significant attention in the recent past owing to the proliferation in the numbers of tetherless portable devices, and rapid growth in popularity of wireless networking. Most of the MANET research community has remained focused on developing lower layer mechanisms such as channel access and routing for making MANETs operational. However, little focus has been applied on higher layer issues, especially application modeling. In this paper, we present a novel distributed application framework based on *task graphs* that enables a large class of resource discovery based applications on MANETs. A distributed application is represented as a complex 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 involves several logical patterns of *data flow* between classes of such specialized devices. These data flow patterns induce dependencies between the different classes of devices that need to cooperate to execute the application. Such dependencies yield a *task graph* representation of the application.

We focus on the problem of executing distributed tasks on a MANET by means of dynamic selection of specific devices that are needed to complete the tasks. In this paper, 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 *Dynamic Task-based Anycasting*. Our algorithm periodically monitors the logical associations between the selected devices, and in the event of a disruption in the application owing to failures in any component in the network, it adapts to the situation

[1]*ACM/Kluwer Journal for Mobile Networks and Applications (MONET)*, Vol. 8, Issue 5 pp.593 - 612, October 2003. 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.

and dynamically rediscovers 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. From our simulation studies, we observed that our protocol was able to instantiate and re-instantiate TG nodes quickly and yielded high effective throughput at low to medium degrees of mobility and not much below 70% effective throughput for high mobility scenarios.

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

1 Introduction

With the shrinking size of tetherless computing devices and increasing diversity of their capabilities, the value of pervasive computing is rapidly becoming real. As these devices proliferate in number, exploiting the full potential of a network of such devices while not frustrating the end-user with interminable configuration tasks and wire-plugging exercises proves to be a challenging issue indeed.

Established technologies such as IEEE 802.11 [10] and burgeoning technologies such as Bluetooth [6] are poised to bring true tetherless computing applications to the end user in every possible way. 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 increase in popularity of portable devices and wireless connectivity standards, MANETs are likely to gain popularity in the near future, especially in settings where a networking infrastructure is expensive, cumbersome, or impossible to construct. 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. Ideally, such pervasive networks can enable a broad range of distributed applications that need exchange of information between multiple devices.

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. In such a situation, since the service providing devices may themselves be mobile, a user cannot rely on one particular device for a certain service since its reachability/availability is not guaranteed. Instead, a user must be prepared to access the required service from any

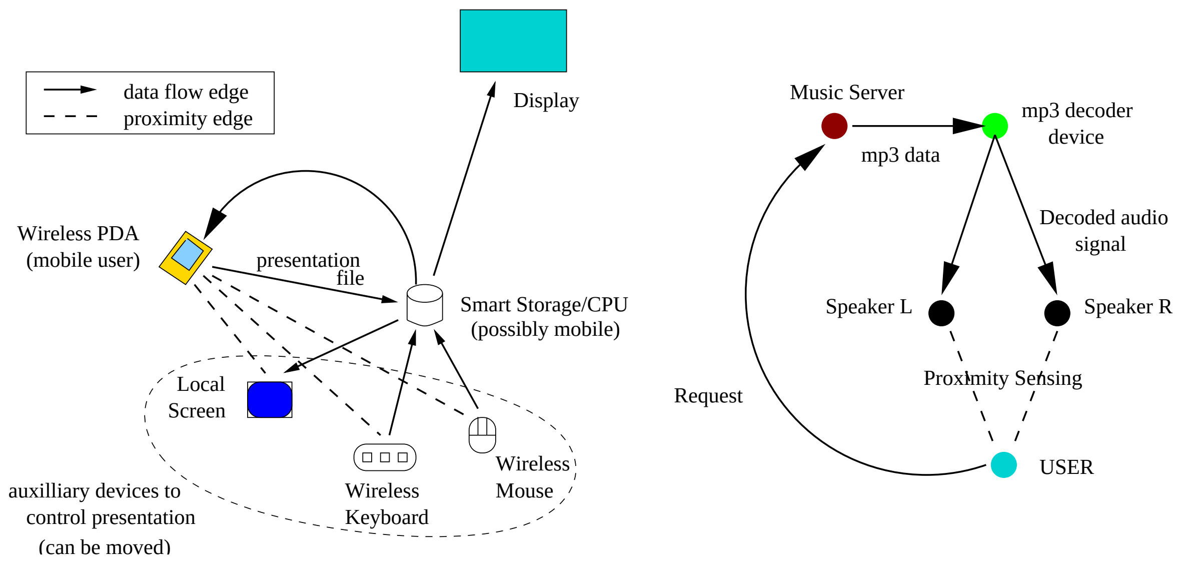


Figure 1: Smart Office and Home Applications: (a) Smart Presentation Task, and (b) Stereo Music Service

of several devices in the MANET providing similar services, if possible.

Significant previous research focused on lower layer MANET issues such as channel access [18, 5] and routing [17, 20, 16]. However, higher layer issues related to application design have received little attention in this context. We bridge this gap with a novel scheme for modeling and executing distributed applications on MANETs that rely on services offered by other devices scattered across the network.

We introduce the abstraction of a *Task-Graph* (TG) for representing higher-level tasks (or applications) that a user may want to perform, 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 between different nodes for performing the task. Fig. 1 shows two examples of task graphs in the pervasive computing domain. Fig. 1(a) shows the TG of a smart presentation application, in which a user, carrying only a PDA with wireless connectivity capacities, automatically finds and instantiates the devices in a room needed to make a full presentation. Fig. 1(b) shows the TG of a music service application, in which the application tracks the physical location of the user and instantiates the appropriate speaker devices that will play the music the user has selected. Both applications are mentioned in further detail in the next section.

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

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

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 *Task-based Anycasting* or *Embedding*. The adjective *dynamic* is added since the choice of device instances may change with time due to mobility.

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.

Although we do not prove it explicitly, we argue that optimal embedding of a general graph onto a general MANET is a hard problem. However, if TG is a tree, we show that it is possible to perform optimal embedding in polynomial time, and present one such algorithm. But since the running time of this algorithm is high and it assumes knowledge of the entire topology, we present a simpler and practical distributed algorithm for dynamic instantiation of TGs on a MANET for executing the application corresponding to the TG. In the event of a disruption in the application owing to failures in any component in the network, our algorithm quickly adapts to the situation and dynamically *re-instantiates* the affected parts of the TG, if possible.

We measure the success of our distributed instantiation algorithm by certain metrics such as average time for instantiation, quality of the instantiation (in this paper, the technical term used for this is *dilation*), frequency of disruption of tasks due to mobility or route failures, and average effective throughput and application data delay that are experienced after task data transmission begins.

The rest of the paper is organized as follows: Sec. 2 briefly describes the related work in the literature. Sec. 3 introduces example scenarios that motivate research in our Task-Graph approach. Sec. 4 introduces the theoretical foundations for the TG concept. Sec. 5 presents a distributed algorithm (and protocol) for TG instantiation in a network. Sec. 6 presents simulation results of the proposed algorithm and protocol. Sec. 7 concludes the paper.

2 Related Work

In recent years, *service discovery* in networks has been a popular topic of research in the industry as exemplified by SLP [13] and Sun's Jini [21]. 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 [4]. 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.

Some researchers have proposed to capture *user-intent* for discovering appropriate devices suitable to them. The Portolano project [11] emphasizes the use of data centric networking which is really a low-level implementation technique that can potentially have TG as a higher-level abstraction. In INS [1], the user intent is abstracted into collections of attribute-value pairs that describe what the user wants. 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, it does not to attempt to systematically represent the relationships between the components of a distributed task, which is our principal focus. 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 has a vision somewhat similar to ours. In their vision paper [2], they argue very briefly for the design of applications in terms of sub-tasks instead of specific devices. However, 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 used in 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 [7, 19]. 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 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. We do not discuss the homogeneous device scenario in this paper. To the best of our knowledge, this is the first attempt towards modeling distributed applications on a MANET using task graphs.

3 Examples of Application Scenarios

In this section we describe a few application scenarios which motivated our research. We give examples from smart offices and homes, disaster relief, and sensor networks.

Smart Pervasive Computing Applications Here we consider an application for future smart offices, conference halls, or ad-hoc meeting rooms. Imagine a large hall with a number of display screens, projectors, wireless keyboards and pointing devices. Also assume that there are a number of *smart storage* devices which can run popular file-viewing software. These devices can store a presentation file in their local storage and can render them on a display screen using appropriate software[3]. Devices are tetherless and therefore can be moved around in the room freely. The presenter carries only a PDA (with wireless connectivity) that contains the presentation file. The file, or portions of it, are transferred from the PDA to a *suitable* smart storage device depending on the type of presentation software and other factors to be outlined later. The presenter will need a pointing device, a keyboard, perhaps, and a local screen to control the presentation, but does not care about which devices actually perform the presentation task.

[3]Laptops of audience members with wireless connectivity can easily take up this role.

Fig. 1(a) depicts the presentation task with a graph where the vertices stand for categories of devices involved in the task and the edges stand for data flow between device categories or other factors like physical proximity. Assuming that there is no infrastructure support in the hall, a multi-hop ad-hoc network is formed by these devices among each other.

In this scenario, a self-organizing protocol is required to discover the most suitable display screen, smart storage device, keyboard, mouse and local screen with respect to the presenter's location and data requirements. The presenter can move around in the hall either naturally or for displaying multiple items at the same time, and a different set of peripherals may need to be discovered for optimal effect. Note that the nearest smart storage device containing the suitable viewer may not be the best one for the nearest keyboard and the nearest display to communicate with. The discovery process must consider the relationships between vertices in the corresponding task graph. Thus the presentation task can be achieved seamlessly without bothering the user with configuring the chosen peripheral devices.

Fig. 1(b) illustrates an example of an application for future smart homes. Imagine a user wearing a proximity sensor embedded in his/her shirt roaming around in the house while listening to music. When he/she requests a particular song from a music server (hosted in the house or somewhere in the Internet), the music server streams the mp3 file to a suitable mp3 decoder in the house which may be a hardware player or a software decoder residing on a PC. The proximity sensor in the users shirt senses the nearest set of speakers, and the selected mp3 decoder device starts streaming the audio to the selected set of speakers. If the user moves to another room, the music *follows* the user automatically since a new set of devices is dynamically selected, triggered by the proximity sensor.

Scalable Coordination in Disaster Relief In disaster relief situations involving earthquakes, fires or floods, there are various teams of workers involved such as law enforcement officers, fire fighters, rubble removers, rescue workers, paramedics, and ambulance personnel, among others. Each of these groups of people have specialized roles in the rescue operations, and they are expected to be in constant touch with each other through their personal handheld devices. However, usually all rescue workers do not need to communicate with all others. For example, the fire fighters receive data from sensors and communicate with paramedics, which in turn need to communicate with the ambulance personnel only. Hence, one can establish certain communication patterns between different types of rescue workers such that only messages that are relevant to them arrive at their PDAs.

Therefore, instead of forming one huge multicast tree, one can form a number of smaller multicast groups (each corresponding to a team of workers) connected together by a small *anycast* tree. The nodes in the anycast tree are similar to the nodes in an embedded task graph, each belonging to a specialized type defined by their roles, e.g. Police, Ambulance etc. An *edge* in an anycast tree joins one representative member of a specialized team to a similar member of another specialized team, if those particular teams need to communicate. e.g., if fire fighters need to communicate with paramedics, one fire fighter's PDA acts as the gateway of communication, and so does one paramedic's PDA. Communicating along the anycast tree could result in more scalable management of resources than communicating via pure multicast which could waste a significant amount of bandwidth. Our instantiation algorithms can efficiently establish an anycast tree along which such communication can occur. If the selected anycast nodes are lost due to network partitions, then replacement nodes can be selected dynamically.

Smart Sensor Networks Several "smart" specialized sensors (e.g. temperature, location, humidity, motion etc.) with some local computing power may collaborate with one another to perform a complex sensing task rather than transmitting all data to a centralized controller who makes the decision. Abstracting the complex interdependencies and relationships among the different sensor classes into a task graph can be advantageous in such situations.

4 A Task Graph Based Modeling Framework

The use of graph theoretic approaches is well known in traditional parallel and distributed computing systems [7, 19]. We believe that it can be a promising and elegant tool for modeling distributed applications not only on static distributed platforms but also in mobile distributed computing environments such as MANETs. In this section, we first lay the foundations of a task graph based modeling framework and then introduce the task embedding (or anycasting) problem. We then present an optimal polynomial-time algorithm for embedding a tree Task Graph onto a MANET, and then argue why the arbitrary graph embedding problem is hard. In Sec. 5 we present a more practical distributed algorithm which can function efficiently in dynamic, mobile environments.

Definitions of Terms 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 is usually 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 PDAs and laptops.

The capabilities of each device can be summarized in their *attributes*. Attributes can be static (which do not change with time) or dynamic (which change with time). 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), power levels, available computational power (or load), and available communication bandwidth. In this paper, we only consider specialized devices with their principal attribute, i.e., their main function[4].

A *service* is a functionality provided by a device or a collection of cooperating devices. Although hierarchical service composition[5] is possible using our modeling scheme, in this work, we only consider services provided by single devices. Multiple devices can exist in the MANET for providing the same service.

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 consider simple nodes only in this paper. 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, weight, required data rate between nodes, allowable bit error rate, and physical proximity.

4.1 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 bigger task. An *atomic* task is an indivisible unit of work which is executed by a simple node. Atomicity is

[4]Multi-attribute extensions are possible to our work on the lines of [1].

[5]creating more complex services from simpler services that are available in the current networked physical space.

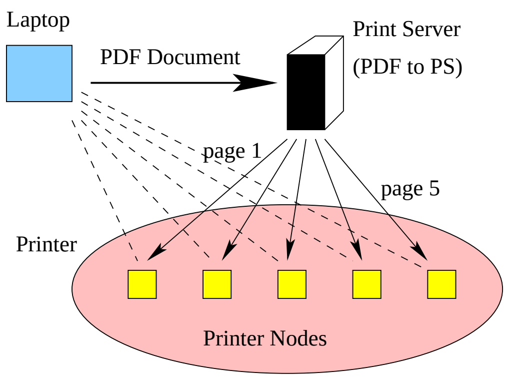


Figure 2: A Smart Printing Service

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, and E_T is the set of *edges* denoting data flow between participating nodes.

An example may bring further clarity to the abstractions developed so far. Consider a scenario in which there is a postscript (PS) printer connected to a small computer (print server) running filtering software that can convert PDF files to printable PS format. The printer and the computer are each *devices* that offer a particular *service*. 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. The example has been illustrated in Fig. 2.

(i)	$U:\ [-;SS]\ [SS;-]$	**(iv)**	$M:\ [-;SS]$
(ii)	$SS:\ [U;LS,D]\ [U;U]\ [K;LS,D]\ [M;LS,D]$	**(v)**	$LS:\ [SS;-]$
(iii)	$K:\ [-;SS]$	**(vi)**	$D:\ [SS;-]$

Table 1: Data Flow Tuples for the Presentation Task

The printer is a physical device representation of a *simple node* with certain *attributes* (such as print resolution, ink color) that offer the service of converting PS files into printed pages. Analogously, the print server computer plus its filtering software can be viewed as a representation of a *PDF* $\rightarrow$ *PS converter node*. By taking these two nodes together we can form a *complex node* that offers a "PDF printing service". A *task* we have in mind is 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[6]. The entire document can be then printed on a set of available printers as shown in Fig. 2. Note that in this scenario we formed a new service (PDF printing) by combining two existing ones. Granted that this example is simple (even trivial), we believe that research that enables such capability in today's MANETs for *arbitrary* device types and quantities is essential for tapping into the networked environment's full potential.

4.2 A Data-flow Tuple Architecture

Every class of device participating in a task possesses a set of *tuples* representing the task requirements in terms of the data flow from other classes of devices into the current class, and from the current class to other classes. Each tuple corresponds to a logical unit of data processing that is needed between the distributed components of an application. Consider the *smart presentation* application described in Sec. 3. The following data flows can characterize a sample presentation: (i) Presenter's PDA (U) sends data to Smart Storage (SS) with appropriate presentation software. SS then displays the data on the Display (D), Local Screen (LS), and some corresponding notes on the PDA screen (U)(ii) Keyboard (K) data (from the presenter typing on the keyboard) and Mouse (M) clicks are sent to SS which interprets them and relays them to D and LS.

To represent such application data flow between nodes we employ a generalized tuple architecture. If a node of class X receives data from nodes of classes A, B and C, and sends the processed data to nodes of classes D and E for a certain application flow (e.g. mouse commands or keystrokes), we can represent this flow using the following tuple: $X : [A, B, C; D, E]$. Such an application data flow can be specified in a file using a standardized language and a Task Graph (TG) can be generated from it easily. The aforementioned application data flows can be written as tuples as shown in Table 1 and they translate to the task graph shown in Fig. 1(a).

Having a tuple architecture for a task serves two purposes: (1) It is a structured specification of the data flows in a task from which a task graph can be derived, and (2) after the embedding of the task graph on the MANET, it governs the flow of actual application data at each participating device.

[6]We did not come up with any objectively quantifiable argument as to why the atomic task should not be the printing of a dot in the paper. Subjectively, we believe atomic operations at that level are too cumbersome to manage at such high-level abstraction.

The example of a data-flow tuple presented above contains only the essential information for data exchange, namely the data source and the data destination. In general, the edges in a TG can have attributes such as channel error rates, bandwidth, etc. which reflect the QoS needs of the 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. One direct way of implementing it is to introduce a *proximity attribute* of the edge and specify it in the tuple. Implementation details of the proximity attribute are beyond the scope of this paper, and we reserve it for future study.

4.3 Embedding Task Graphs onto Networks

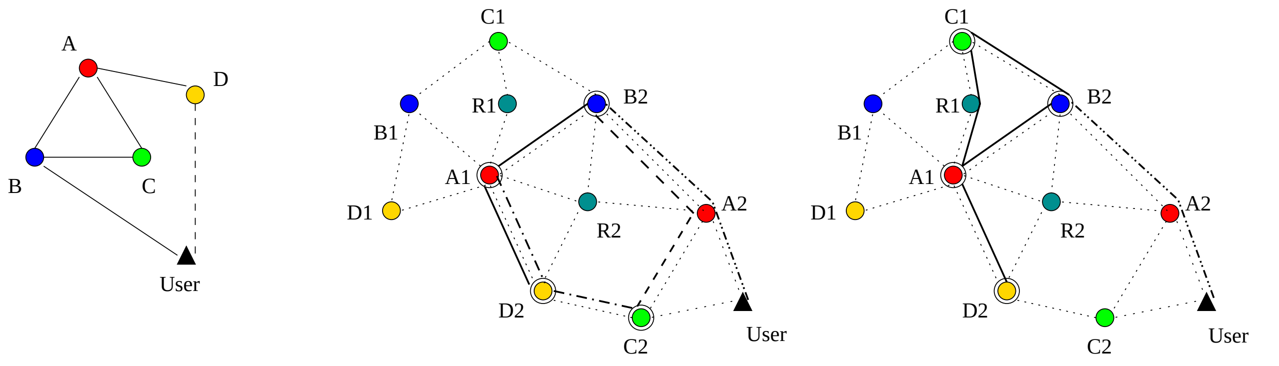
The first step in executing a distributed application on a set of specialized devices is to *discover* appropriate devices in the network and to *select* from those, some suitable devices that are needed to execute the more complex application. Mathematically speaking, embedding a task graph $TG = (V_T, E_T)$ onto a MANET $G = (V_G, E_G)$ 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 *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. Fig. 3(a) depicts a hypothetical task graph. Figures 3(b-c) show a sample network topology with two possible embeddings of TG on it.

The entire 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.* The collective process is called *embedding* or *task-based anycasting.*

4.4 Metrics for Performance Evaluation

In general, the embedding process maps edges in TG to *paths* in G. *Average Dilation* of an embedding is the average 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, dilation is given by:

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



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

Figure 3: Example of Task Graph Embedding

Dilation is an important 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 Sec. 4.5 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. When an embedding is disrupted owing to network failures, *reinstantiation time* measures the time taken to find a replacement device.

A useful metric for measuring the resilience of the protocols to failures is *Average Effective Throughput*, ($AvgEffT$), which 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[7]. Therefore, $0 \leq AvgEffT \leq 1$. Another useful metric for measuring application performance during transmission of task data is the source-to-sink delay suffered by an ADU as it funnels itself through various intermediate relay nodes in the instantiated task graph. In Sec. 6 we describe the performance of our algorithms with respect to the above metrics.

4.5 Optimization Problem Formulation

We formulated the minimum embedding problem as the following optimization problem:

[7]If a relaying node in the path from source to sink gets uninstantiated, effective throughput will be affected since certain data flows will be dumped and will not reach the data sinks.

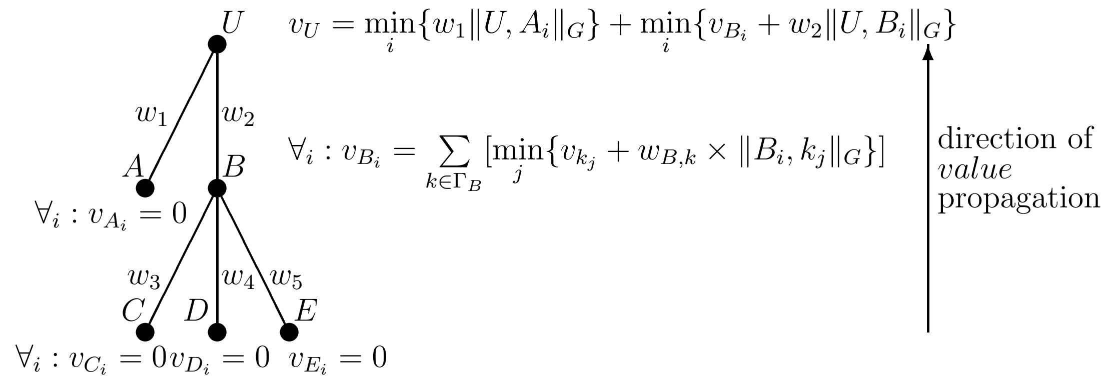

U
$v_U = \min_i\{w_1\|U, A_i\|_G\} + \min_i\{v_{B_i} + w_2\|U, B_i\|_G\}$
w_1
w_2
A
B
$\forall_i : v_{B_i} = \sum_{k\in\Gamma_B}[\min_j\{v_{k_j} + w_{B,k} \times \|B_i, k_j\|_G\}]$
direction of *value* propagation
$\forall_i : v_{A_i} = 0$
w_3
w_4
w_5
C
D
E
$\forall_i : v_{C_i} = 0 v_{D_i} = 0 \; v_{E_i} = 0$

Figure 4: Optimal Solution based on the Principle of Optimality

Let C be a set of principal attributes (or classes) of specialized devices. Let $G = (V_G, E_G)$ be a graph corresponding to a MANET of specialized devices, with the class of each device in V_G belonging to C. Let $TG = (V_T, E_T)$ be a task graph such that the class of each node in V_T belongs to some $S \subseteq C$. Also, a function $w : E_T \rightarrow \Re^+$ defines edge weights which could signify data rate requirements or importance of edges with respect to the task. 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:

$$D_{avg} = \frac{1}{\sum\limits_{e\in E_T} w(e)} \sum_{e=(x,y)\in E_T} w(e)\,\|\varphi(x), \varphi(y)\|_G \qquad (2)$$

is minimized, where $\|a, b\|_G$ denotes the shortest path between devices a and b in G.

4.6 An Optimal Polynomial-time Algorithm for Embedding *Tree* Task Graphs

We present an optimal algorithm (with respect to D_{avg}) for embedding or anycasting a *tree* task graph onto a MANET at a given instant of time. The running time is polynomial in $|G|$ as well as $|TG|$. The algorithm minimizes searching in the solution space by taking advantage of the *tree* structure of TG, and is based on the principle of optimality[8]. The

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

algorithm requires that the node executing the algorithm have a complete knowledge of the topology of the network at the given instant of time.

The algorithm proceeds by the propagation of a certain *value* function from the leaf nodes of TG towards the root node U. It comprises of the following steps:

1. Perform a BFS traversal of the tree TG and assign a level L to each node starting from U which has $L = 0$.

2. Now begin assigning a *value*, v to nodes starting from the highest level, $L_{max} \leq |V_T|$. If the node under consideration, say X, is a leaf node in TG, for all instances X_i in G of class X, assign value zero, i.e., $\forall_i : v_{X_i} = 0$.

3. If X is not a leaf node, consider its children at every instance X_i of X in G. For instance X_i, for every child node Y of X, carry out the following optimization step: sweep through all the instances Y in G, and select the instance Y_j which minimizes the quantity $v_{Y_j} + w_{(X,Y)} \| X_i, Y_j \|_G$.

   The crux of the idea is that the principle of optimality holds because of the tree structure of TG: if X_i is selected by its parent and is optimal, then the choice of Y_j is optimal too. This greatly reduces the search space. After carrying out this step for all children of X for X_i, assign the sum of the calculated minimum values to v_{X_i}. Fig. 4 illustrates the procedure for a task graph of 6 nodes[9].

4. Complete this value assignment for all instances of all nodes at a given level in TG before considering lower level nodes. At U, choose the children such that v_U is minimized (note that there is only one instance of U). Now, we have the optimal instances of all nodes which minimize the dilation metric in Eq. 2, and hence have an optimal embedding.

Now, we calculate the running time of the above algorithm. Step 1 takes $O(|V_T| + |E_T|)$ time. If the nodes with the same level are pushed onto a stack after traversal, then nodes with $L = L_{max}$ will reside at the top and value propagation can occur efficiently. Now, since in the worst case, $L_{max} = |V_T| = O(|V_T|)$ (although in more balanced trees, it is logarithmic in $|V_T|$), Step 2 can loop $O(|V_T|)$ times. In Step 3, every edge in TG is mapped to a shortest path in G. Suppose there are $|C|$ classes of devices in G with $\frac{|V_G|}{|C|}$ instances of each class. For every parent instance, each child instance is considered, the shortest path is computed (in

[9] Γ_B is the set of children of B in TG. k_j is an instance in G of child k of B in TG.

$O(|V_G|^2)$ time) and a minimization step performed (in $O(\frac{|V_G|}{|C|})$ time). Hence this mapping process for all edges will take $O(|E_T| \times \frac{|V_G|}{|C|}(\frac{|V_G|}{|C|} \times |V_G|^2 + \frac{|V_G|}{|C|})) = O(|E_T|\frac{|V_G|^4}{|C|^2})$ time. Note that the Step 2 loop is subsumed in this calculation. Since $|V_G|$ is the dominant term, the time complexity is given by the above expression itself. If we use All Pairs Shortest Path algorithm (running time is $O(|V_G|^3)$ and extraction of shortest path cost is $O(1)$ assuming a random access storage) instead of the Dijkstra's Source-Destination Shortest Path, then running time of the entire process is $O(|E_T|\frac{|V_G|^2}{|C|^2} + |V_G|^3) = O(|V_G|^3)$.

This algorithm has the following drawbacks: (1) it is centralized and does not have a low time complexity, (2) it needs entire topology information at a particular device which we believe is unrealistic for large MANETs that we consider in this paper, and (3) it is not adaptive to mobility of devices. Hence, in Sec. 5 we propose a distributed greedy heuristic approach which albeit suboptimal, is much simpler and less time consuming, and reasonably efficient in operation.

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

Since there are $|V_T|$ distinct classes of devices in TG, and $\frac{|V_G|}{|C|}$ instances of each class in the network G ($|V_T| \leq |C|$), the total number of possible embeddings is $\frac{|V_G|^{|V_T|}}{|C|^{|V_T|}}$. Hence, the search space for an optimal brute force algorithm is exponential in $|V_T|$, which is clearly unacceptable. The combinatorial structure of the problem prompted us to look for a heuristic solution, the basic idea of which is very simple: find a spanning tree ST_{max} of TG with the maximum weight and then embed that tree in G using the algorithm described before. It is easy to see that the edges of TG which are not in ST_{max} will not be considered by the embedding process and can be mapped to very long paths and give highly suboptimal results without any bounded approximation factor. Any better solution will have to involve partitioning TG into smaller components which can be embedded *independent* of each other. Finding such components and then embedding them onto the network both are likely to have high time complexity. Hence, we trade off complexity for optimality and look for greedy heuristic solutions which are simple and distributed in nature.

5 A Distributed Task Embedding Approach

A centralized approach is often preferable when there is some existing static infrastructure with a fixed base station node or a directory server, devices are less mobile and more connected with each other, and the probability of node and link failure is low. In the Sec. 4.6 we described a centralized solution which requires knowledge of the entire network and yields optimal results for tree TGs. However, when we consider a dynamic and mobile distributed environment such as a MANET of low power devices, no device in the network may possess adequate computational power to execute that algorithm. Also the MANET graph could change by the time the optimal dilation is calculated. Moreover, it is hard for one device to track the changes in topology between participating instances using reactive MANET routing protocols such as DSR.

It is also hard to guarantee that a device acting as a centralized controller will always be connected to the rest of the network. When a disconnection or partition occurs in the network, a new controller may have to be elected, and that device will have to compute a new embedding. Although it may be easier to find close-to-optimal solutions (with respect to particular performance metrics) using a centralized approach than a distributed one, the latter is more robust and adaptive to mobility since there is no single point of failure. Hence, we feel that localized distributed algorithms are better suited for such dynamic MANET environments.

Services such as device discovery and selection can be implemented in an intermediate layer (say, TG layer) between the application and network layers. Our approach assumes the presence of routing support in the MANET. However, one can also envision the TG layer to be embedded in the network layer for performance improvements (not considered in this work).

5.1 A Distributed Algorithm for Instantiation of Task Graph Nodes

In this section we present a distributed algorithm for the instantiation of nodes (or anycasting of the corresponding task) in a given task graph. The corresponding distributed protocol has been specified by the finite state machine shown in Fig. 7.

The principal goal of the algorithm is to produce an embedding of a TG onto a MANET with the objective of optimizing the performance metrics proposed in Sec. 4.4. In this work, we assume that each heterogeneous device can provide a single type of service, and that all

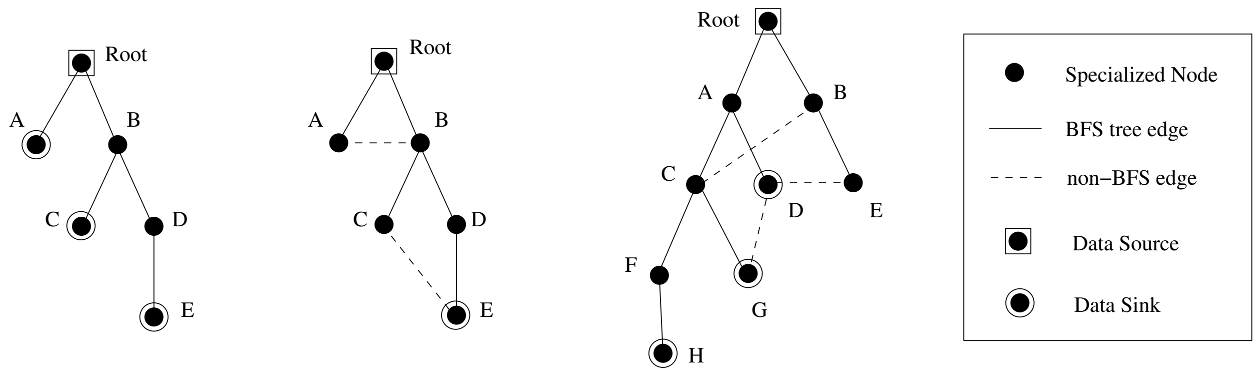


Figure 5: Task Graphs: (a) Tree, (b,c) Non-Tree Graphs

nodes in the network are *simple*. Our concept can be easily extended to incorporate the case where multiple types of services are provided by one particular device. We assume the presence of a MANET routing protocol (e.g. DSR, AODV etc.) and a reliable transport protocol (TCP). A task graph TG is submitted by the user node's application layer to its TG layer which begins the embedding process. The instantiation/embedding process is completely on demand and no proactive exchange of device state is assumed in the MANET.

All devices in the network execute copies of the same algorithm. Every device in the system exists in a state s such that $s \in S_1 \times S_2$, where S_1={COORDINATOR, NON_COORDINATOR} and S_2={UNINSTANTIATED, WAIT_FOR_ACK, INSTANTIATED, SUBTREE_INSTANTIATED}. A device plays the role of a "Coordinator" when it coordinates the process of instantiation of a subset of the nodes in TG. Only five of the eight possible states in $S_1 \times S_2$ are necessary in the protocol. They have been depicted in Fig. 7. All devices except the user device U start in the UNINSTANTIATED state. The latter starts execution in the (INSTANTIATED, COORDINATOR) state which means that it itself is instantiated and is playing the role of a coordinator. It also means that TG has not been instantiated yet, since in that case U would make a transition to the (SUBTREE_INSTANTIATED, COORDINATOR) state.

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. Fig. 5 depicts three task graphs with their BFS and non-BFS edges. We call the spanning tree on TG induced by BFS and rooted at U, a BFS-tree of TG ($BFST_{TG}$)[10]. In our algorithm, a coordinator is responsible for coordinating the instantiation of nodes that are its children in $BFST_{TG}$. In

[10]We assume here that all edges of TG have equal weight. If the weights are unequal, a Maximum Weight Spanning Tree should be computed and its edges should be used for performing the embedding instead of the BFS-tree. Rest of the steps are similar to the ones taken for embedding a BFS-tree with obvious changes in nomenclature.

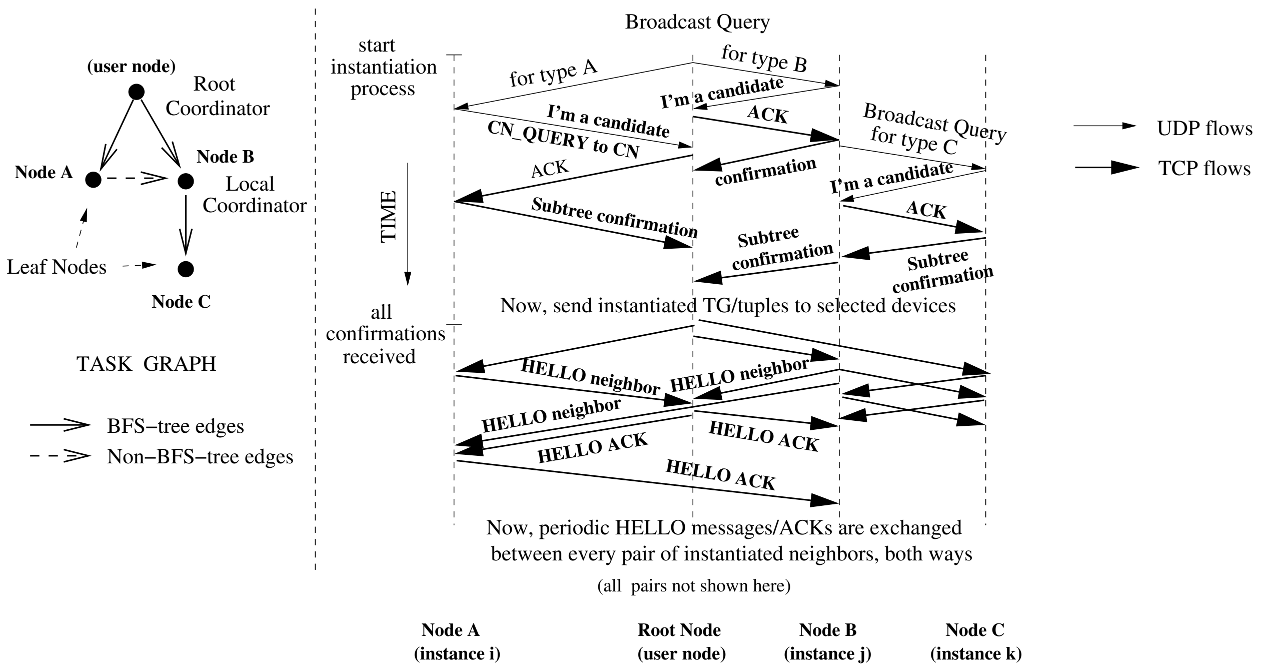


Figure 6: Dynamics of the Embedding Scheme

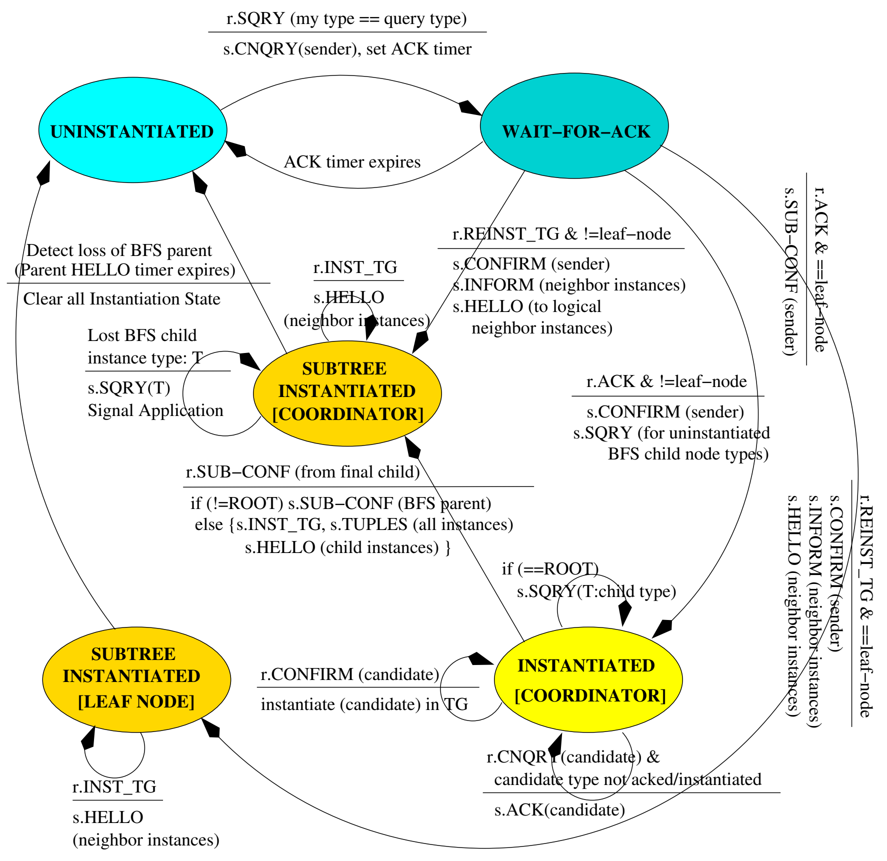
other words, every chosen instance of a TG node in the network has exactly one BFS parent which acts as its coordinator.

We propose a *greedy* solution 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 $BFST_{TG}$. Note that unlike the optimal algorithm proposed in Sec. 4.6, this algorithm greedily instantiates nodes in TG starting from U and searches only the local space around an instantiated device for instantiating the next node.

The salient steps of the algorithm have been illustrated in Fig. 6 by means of a time-based message diagram. First, TG nodes which are neighbors of the user node, U are visited parallely, and their instances are discovered in the network. U issues broadcast search queries for each neighbor (types A and B) in TG[11]. The query types in the search packet are A and B respectively. A packet is rebroadcast by the TG layer of a device, D if it does not match the search query type, the time-to-live (TTL) value in the packet has not reached zero, and if D has not seen the same broadcast (identified by a broadcast sequence number) before.

[11]The broadcast is controlled by executing it at the TG layer rather than the IP layer. Essentially, a device in the network broadcasts a packet to all its one-hop neighbors which examine the state of the packet and decide whether to rebroadcast it or not.

When a free instance B_j of type B gets a search (S_QRY) packet, it sends a candidate query (CN_QRY) packet to its coordinator (BFS parent) U expressing its willingness to participate in the task. B_j then changes its state to WAIT_FOR_ACK. U checks the instantiation state of the TG node B and sends an acknowledgment (ACK) to B_j if B has not been instantiated yet in its TG. The coordinator device is responsible for accepting and rejecting responses from several candidate devices in the network before the discovery is completed. Selection of a candidate device from a set can be performed on the basis of several heuristics. In this paper, we select the first candidate device that replies to a search query packet of type B to become the instance of B at U. This is a simple method of selecting a nearby device in an attempt to minimize average dilation. More sophisticated rules can be used for the selection process; for example, one can attempt to maximize the energy efficiency and remaining battery lifetime of a candidate. For that purpose, the candidates need to encapsulate their current attributes in the CN_QRY packet. Another goal may be to minimize the overlap of routes between any pair of communicating instances of TG nodes in order to reduce bottlenecks. We plan to investigate these alternative goals in future.



r.SQRY (my type == query type)
s.CNQRY(sender), set ACK timer
UNINSTANTIATED
WAIT–FOR–ACK
ACK timer expires
r.ACK & ==leaf–node
s.SUB–CONF (sender)
r.REINST_TG & !=leaf–node
s.CONFIRM (sender)
s.INFORM (neighbor instances)
s.HELLO (to logical neighbor instances)
Detect loss of BFS parent (Parent HELLO timer expires)
Clear all Instantiation State
r.INST_TG
s.HELLO (neighbor instances)
Lost BFS child instance type: T
s.SQRY(T)
Signal Application
SUBTREE INSTANTIATED [COORDINATOR]
r.ACK & !=leaf–node
s.CONFIRM (sender)
s.SQRY (for uninstantiated BFS child node types)
r.SUB–CONF (from final child)
if (!=ROOT) s.SUB–CONF (BFS parent)
else {s.INST_TG, s.TUPLES (all instances)
s.HELLO (child instances) }
r.REINST_TG & ==leaf–node
s.CONFIRM (sender)
s.INFORM (neighbor instances)
s.HELLO (neighbor instances)
if (==ROOT)
s.SQRY(T:child type)
SUBTREE INSTANTIATED [LEAF NODE]
r.CONFIRM (candidate)
instantiate (candidate) in TG
INSTANTIATED [COORDINATOR]
r.CNQRY(candidate) &
candidate type not acked/instantiated
s.ACK(candidate)
r.INST_TG
s.HELLO
(neighbor instances)

Figure 7: Finite State Machine Representation of the Salient Characteristics of the Instantiation Protocol

Along with the acknowledgment, U sends a copy of the task graph listing the device types that are in the sub-tree of TG rooted at B (such as C). U does not send explicit negative acknowledgments to all other instances of B. Instead, a candidate B_k starts a timer as soon as it sends a CN_QRY to U. If the timer expires before it receives an ACK, it puts itself in the UNINSTANTIATED state again. This is more scalable than U sending explicit NACKs to the coordinator, albeit at the cost of tying up B_k in the WAIT_FOR_ACK state for a longer period of time. An acknowledged candidate B_j changes its state to (INSTANTIATED, COORDINATOR) and confirms its role to U by sending a CN_CONFIRM packet. B_j assumes the role of a coordinator since there are uninstantiated nodes rooted at B in the TG, and it has to take the responsibility of instantiating those nodes. B_j then broadcasts S_QRY packets for all uninstantiated child node types and the instantiation thus proceeds further. Thus, B_j acts as a *local coordinator* for all nodes in its subtree.

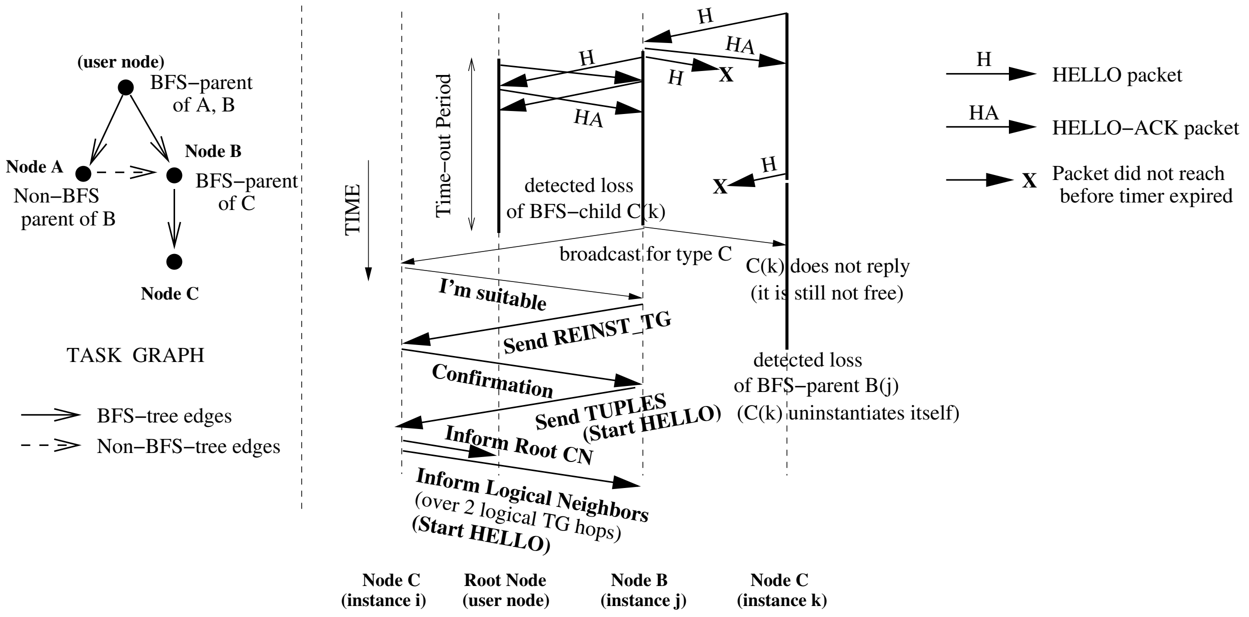


Figure 8: Re-instantiation of Devices

Instantiation of B at U is not complete until the CN_CONFIRM packet arrives from B_j. If a device is an instance of a leaf node in BFS_{TG} it sends a CN_CONFIRM packet with the SUBTREE_CONFIRM flag *set* since it does not have to propagate the search further. This has been illustrated in Fig. 6 where C_k (which is an instance of a leaf node in TG) sends a subtree confirmation to its BFS parent and coordinator B_j. The instantiation of C at B_j is complete at this stage.

After receiving subtree confirmations from all selected child node instances, the parent node sends a subtree confirmation to its BFS parent along with a partially instantiated task graph; in the running example, B_j sends a CN_CONFIRM(subtree) packet to U after it receives a similar packet from C_k. Simultaneously, it also transits to the (SUBTREE_INSTANTIATED, COORDINATOR) state. On the other hand, a leaf node instance upon instantiation transits to the (SUBTREE_INSTANTIATED, LEAF_NODE) state after sending the subtree confirmation upstream.

When U receives subtree confirmations for all its child node types in TG, it concludes the instantiation or embedding process. At this stage, U is aware of the instances of all nodes in TG. It then sends the instantiated task graph and tuples to the chosen devices (node instances) so that each device knows the addresses of instances of its neighboring nodes in TG.

An important point to be noted here is that the root coordinator U needs to send the instantiated TG to certain devices whose corresponding TG nodes have incoming or outgoing non-BFS edges incident upon them. For example, since the search process was carried out along the BFS tree edges of TG, B_j and A_i will not know about each other's existence unless U tells each about the other. Ideally, for any two given nodes connected by a non-BFS edge in TG, a node which is their nearest common ancestor in BFS_{TG} will have knowledge of their instances. However, for simplicity we delegate the responsibility of keeping such information on the root U. For tree task graphs, the above situation cannot arise, and hence U need not disseminate task graph information at all; the nature of the instantiation process ensures that each instantiated device keeps all the information that is essential to it for task data flow, i.e. the addresses of all its parent and children nodes[12].

Salient details of state transitions and control packet exchanges in our protocol have been illustrated in Fig. 7 by means of a finite state machine diagram[13]. In our opinion, the user devices are best suited for acting as root coordinators since they usually originate the application data flows, and even under mobility, always remain near the user. Since the instantiation process is distributed among nodes in BFS_{TG}, the root coordinator is not overwhelmed with control packet transmissions and receptions as would be the case in a centralized approach.

MANET multicast protocols could have a role to play in this algorithm, especially for the dissemination of TG information from the coordinator after the instantiation is completed.

[12]Actually, regardless of the nature of TG each node needs to keep slightly more information than parent and child addresses in order to perform local repair after disconnections – this issue will be addressed in detail in Sec. 5.2.

[13]Packets arriving at a device in a state different from the intended state in sender's view are discarded.

However, assuming the presence of multicast support in all ad hoc networks may not be realistic. Hence our protocol does not depend on it.

After the instantiation information is exchanged, the user application can start data transmission. The flow of data will be governed by the instances of the corresponding tuples, and in the 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 Fig. 6) after being relayed by the intermediate devices (B_j).

The task graph itself is sent as control data during the instantiation process. Once the instantiation process is complete, the selected set of devices communicate with each other using the underlying TCP layer which sits on top of the MANET routing layer (DSR in our case). TCP is used to exchange control and data packets since packet losses due to route errors are very common in MANETs. Our protocol does not assume anything specific about the routing protocol running on the MANET although we favor reactive/on-demand protocols over proactive ones since they tend to have lower overhead. Hence we chose DSR for our simulation experiments.

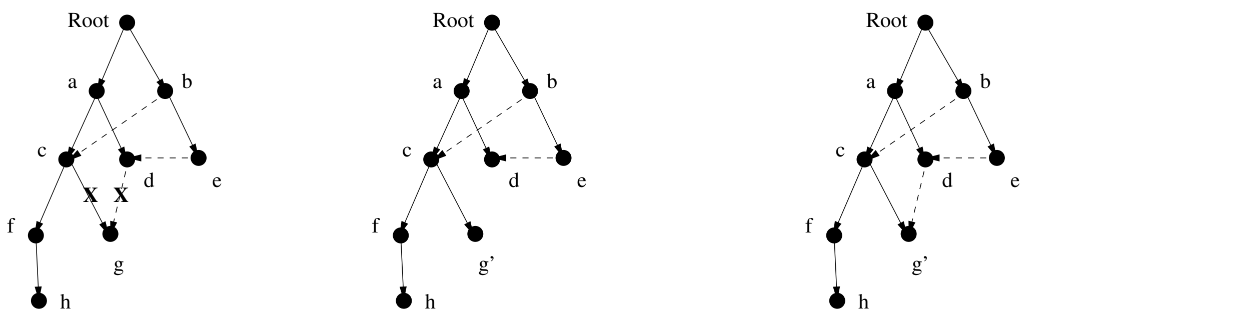
5.2 Handling Mobility of Devices

In this section, we present in detail how our instantiation algorithms react to the mobility of devices. If the devices in the network are highly mobile during the lifetime of a distributed application running on the network, the network topology and previously established connections may change, and this may disrupt the application. Therefore, in the presence of mobility, it is no longer sufficient to permanently select specific devices and appoint them to execute the application – continual monitoring must be performed for detecting disruptions, and replacement devices must be selected for resuming the application.

5.2.1 Detection of Disconnections

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 is the *detection* of disconnections.

Mobility of devices may cause network partitions or disconnections, and two instantiated devices may no longer be able to communicate if all paths between them are broken.



g gets disconnected from c
c detects disconnection
c searches for new g
d does not do anything

c finds g' (g' doesn't know about d)
c sends portion of TG to g'
g' knows about d now
d does not know about g' yet
g' 1–logical–hop broadcasts its neighbor table

d knows about g' now
c knows that g' knows about d
c,d 1–logical–hop broadcast their neighbor tables
a,b,e come to know about g'
(this is useful if c or d are disconnected)

Figure 9: Bookkeeping after Re-instantiation

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 Fig. 8. Note that all instantiated devices are in either of the two states when they start exchanging HELLO messages: (SUBTREE_INSTANTIATED, COORDINATOR) or (SUBTREE_INSTANTIATED, LEAF_NODE).

Specifically, each instantiated device keeps track of its BFS parent and BFS children. For example, in Fig. 5(c), the instance of type C keeps track of the instance of type A (its BFS parent) as well as of the instances of types F and G (BFS children). If a BFS parent device stops hearing from one of its BFS children[14], it uninstantiates its child and starts searching for a replacement of the same type. The child meanwhile would stop hearing HELLO-ACKs from the parent (assuming bidirectional links), and will uninstantiate itself. This has been illustrated in Fig. 8. On average, if the HELLO timer is set to T seconds at every instantiated device, disconnections will be detected after T seconds. State transitions that result from this process have been shown in the FSM diagram in Fig. 7.

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-ACKs not coming back in T seconds which results in the conclusion that a disconnection has happened, even when

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

the nodes are reachable from one another.

Recently, researchers have proposed solutions to the above problem based on explicit notification of route errors to TCP [9]. However, in this study, we do not attempt to alter TCP or DSR (including their default timer settings), and simply build 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[15], 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).

5.2.2 Re-instantiation and Bookkeeping

If a BFS parent device detects a disconnection with its child at the expiry of its HELLO timer, it issues a broadcast search query for the same type of the child. The re-instantiation process then proceeds like the earlier instantiation process with a small simplification that the CN_CONFIRM step is omitted, since the BFS parent here handles one disconnection at a time. Also, the coordinator device is not involved in this process, and the BFS parent device acts like a *local* coordinator. After finding a new device, the BFS parent instantiates it in its own copy of TG, and sends a portion of the re-instantiated TG to the new child device. Fig. 9 helps appreciate the reason behind doing this. $a, b, \ldots, h, g'$ are device instances.

Our protocol addresses the problem of state maintenance after disconnections, and attempts to solve it *locally*, i.e. without involving the coordinator node. Each instantiated device knows the addresses of its parents (both BFS and non-BFS), its children, its children's parents, and its children's children[16]. With a little thought, one can see that information about this portion of TG is enough to handle single node disruptions in the instantiated task graph. However, there is one situation which cannot be adequately handled by this technique: a network partition such that 2 nodes connected by a non-BFS edge get disconnected from their parents. e.g., d and g get disconnected from c and a respectively, in the same HELLO period. a finds d', and c finds g' but a tells d' about g and c tells g' about d (from their 2-logical-hop neighbor tables), and a deadlock situation occurs where d' tried to contact g and d' tries to contact d. To solve this problem, we adopt an approach where a non-BFS parent asks for the coordinator's help if it loses its non-BFS child. Since the coordinator is

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

[16]Essentially, this is 2-logical-hop information except for parent's parents information which is unnecessary.

Simulation Parameter	Value
Number of Devices	50, 100
Simulation Area	$1500m \times 300m$, $1500m \times 600m$
Transmission Radius	$250m$ (nominal range for IEEE802.11b radios)
Mobility Model	Random Waypoint ([8])
MANET Routing Protocol	Dynamic Source Routing (DSR)
#Classes of Devices	12 (assignment of devices classes to devices done with uniform probability)
Size of TG: $(\|V_T\|, \|E_T\|)$	$(6,5), (6,7), (9,11)$ (task graphs shown in Fig. 5)
Simulation Period	$400s$ (difference between end and start times of sending Task data)
MaxSpeed	$1, 5, 10, 15, 20$ m/s
Pause Time	$0, 100, 200, 300, 400, 500, 600$ sec
Data Traffic Pattern (at Source U)	CBR: burst: 2500 bytes; period: 1 sec (no cross traffic) CBR: burst: 2500 bytes; period: 1 sec (with cross traffic) Exponential: mean burst: 2500 bytes; mean period: 1 sec (no cross traffic) Exponential: mean burst: 2500 bytes; mean period: 1 sec (with cross traffic
Cross Traffic Pattern (if applicable)	10 random (source,destination) pairs exchange 512 byte packets after every 1 second interval

Table 2: Simulation Parameters

kept informed by newly instantiated nodes, such a situation can be avoided by taking its help. Note that if the coordinator is unreachable from this node instance, it is because of a network partition alone (since we do not have any timers for such messages), and in that case, a chunk of the instantiated network has been disconnected from the rest (containing the coordinator). In such situations, repair occurs soon at the BFS-parent node instances whose HELLO timers expire.

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

6 Simulation Results

We simulated the algorithms (and protocols) described in Sec. 5 using the popular network simulator *ns-2* [22] for several different scenarios. The simulation parameters have been listed in Table 2. We show simulation results for task-graphs in Fig. 5 – we refer to them as Tree TG, Non-Tree TG-1 and Non-Tree TG-2 respectively. The instantiation process begins at a time when steady state is reached in the network with respect to MaxSpeed and Pause Time parameters, and almost immediately, a data source at the *user/root* node started sending data to the data sinks using the four different traffic load patterns specified in Fig. 2. The results have been averaged over seven different sets of mobility scenarios for each case. Devices which are not part of the current instantiated TG do not relay data packets, and such packets are not buffered[17].

Scenario	Minimum	Maximum	Median	Scenario	Minimum	Maximum	Median
Tree (1m/s)	0.795719	6.561610	1.435320	Tree (1m/s)	0.278449	0.538200	0.433553
TG1 (1m/s)	0.810867	6.819640	1.399530	TG1 (1m/s)	0.213863	0.416054	0.319194
TG2 (1m/s)	2.170060	7.957830	6.674960	TG2 (1m/s)	0.623054	6.832330	2.170670
Tree (5m/s)	0.670853	6.111210	1.728970	Tree (5m/s)	0.313390	0.416645	0.409284
TG1 (5m/s)	0.536686	7.708620	6.278840	TG1 (5m/s)	0.312488	0.486092	0.383662
TG2 (5m/s)	1.742180	9.537000	7.827160	TG2 (5m/s)	0.576274	13.137300	6.616830
Tree (10m/s)	0.643709	1.438240	1.216280	Tree (10m/s)	0.232924	6.315140	0.551869
TG1 (10m/s)	0.842213	6.694860	1.530080	TG1 (10m/s)	0.321746	6.153450	0.387705
TG2 (10m/s)	3.337190	9.168950	7.275040	TG2 (10m/s)	0.529770	7.515390	0.614241
Tree (15m/s)	0.749414	4.039460	1.062140	Tree (15m/s)	0.331452	0.421195	0.353379
TG1 (15m/s)	0.446600	6.511620	0.909011	TG1 (15m/s)	0.313852	0.548396	0.417565
TG2 (15m/s)	1.520370	4.090240	3.241610	TG2 (15m/s)	0.718282	13.072200	7.310430
Tree (20m/s)	0.651414	2.062220	1.088190	Tree (20m/s)	0.308923	6.224250	0.517880
TG1 (20m/s)	0.717359	4.022630	1.484370	TG1 (20m/s)	0.309559	6.320650	0.409159
TG2 (20m/s)	1.361380	7.674460	5.262870	TG2 (20m/s)	0.472178	6.247620	0.637519

Table 3: Embedding Time (in seconds) (a) 100 devices, (b) 50 devices

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 reinstantiation event. These values are then averaged over the simulation time period to yield one number. We observe from Fig. 10 that average dilation for the embedding

[17]In other words, if a device which was part of a TG becomes disconnected while there is a packet in transit, the packet is lost.

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, reinstantiations are rare and the d_{avg} does not deviate too much from its value after initial instantiation. However, at greater speeds, reinstantiation 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 reinstantiations. 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 reinstantiations along those edges). This causes d_{avg} to increase at higher speeds on the whole.

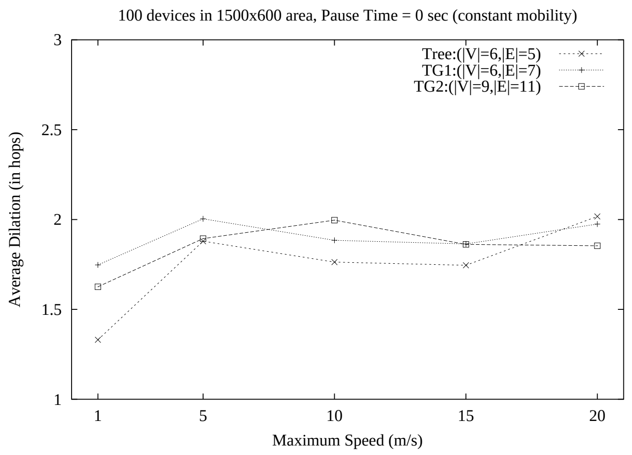


Figure 10: Average Dilation vs. Speed

Another observation from Fig. 10 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 reinstantiation; 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 3 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 CN_QRY packet reaches a coordinator node, it sends ACKs encapsulated in TCP packets since they have a good chance of getting lost if sent using an unreliable transport protocol. All subsequent communication (except broadcast and CN_QRY packets) also uses TCP.

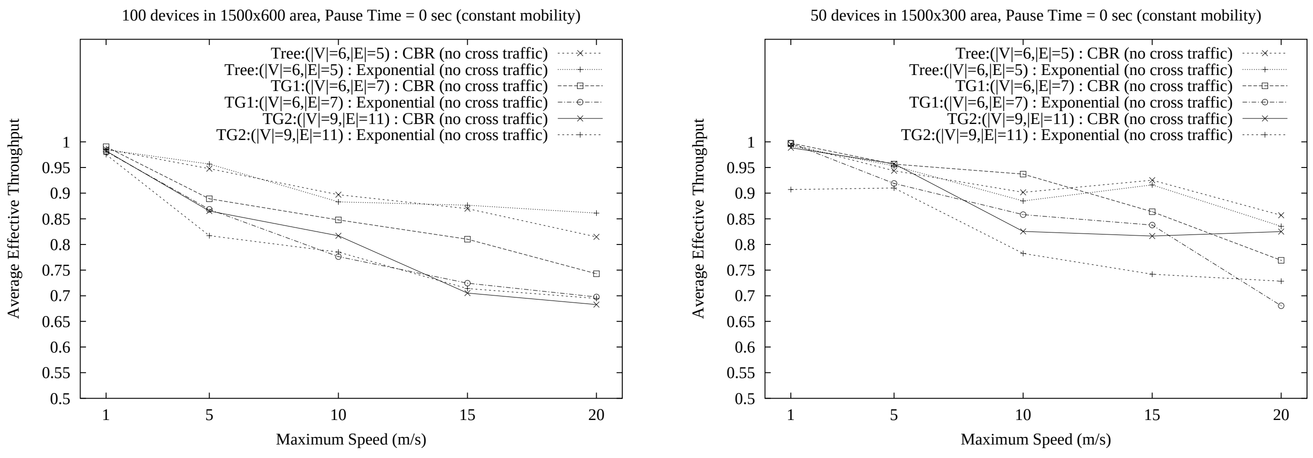


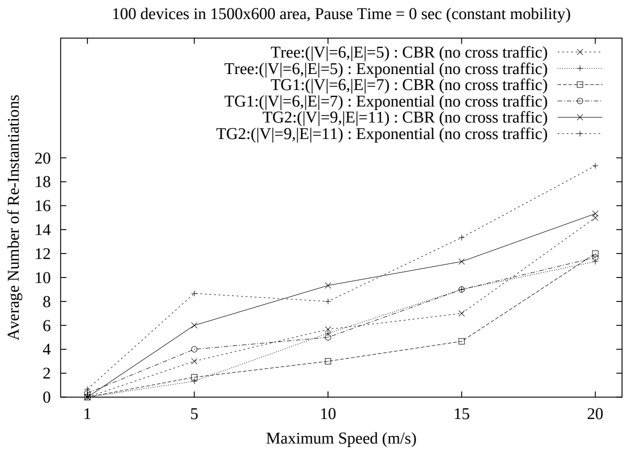
Figure 11: Average Effective Throughput: Variation of Maximum Speed

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 (as explained briefly in 5.2) are added to TCP, then these times can be reduced significantly.

Also, we observe no monotonous pattern as a result of increasing mobility of devices. This can be attributed to the uniform spatial distribution of device categories in all mobility patterns as well as large variability in TCP timers during the multiple steps of the instantiation process. For the 50 device scenario with the same device density, we observe that instantiation times are in the same range as in the 100 device network. The median times are generally lower for the 50 device case since in a smaller network, broadcast traffic consumes lower bandwidth and also a lesser number of devices (approximately half) are likely to respond to S_QRY packets, thus resulting in lower queuing delays for other packets in the network.

Effective Throughput After the instantiation process is completed, we begin data transmission from the user node (source) to the various sinks shown in Fig. 5 according to particular tuple specifications. In Tree TG, instances of A, C and E get one flow each. In TG1, the instance of E gets 4 flows in all through instances of various relay nodes. In TG2, instances of D and H get one flow each and the instance of G gets 4 flows. We normalize and then plot $AvgEffT$ for all three TGs for both the 100 and 50 device MANETs in Fig. 11. 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 Fig. 11 we can see that at low speeds, $AvgEffT$ 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 reinstantiations 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 a couple of 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.



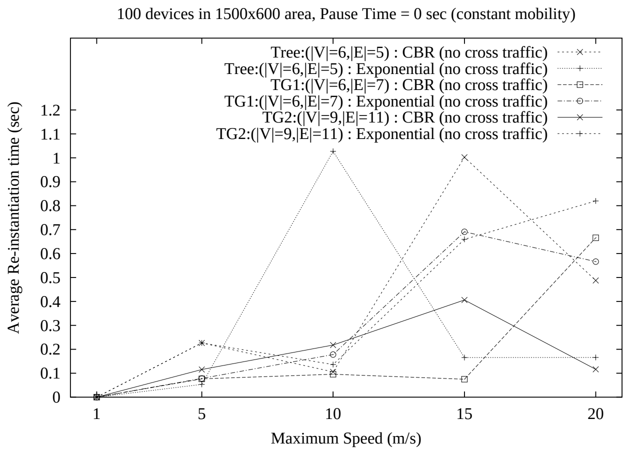


Figure 12: (a) Number of Re-instantiations, (b) Average Re-instantiation Time

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 TCP to back off more often and sometimes results in reinstantiation because of reasons explained earlier. 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.

Number and Time of Reinstantiation In Fig. 12(a), we plot the average number of re-instantiations underwent during the entire simulation time (400s) for the 100 device case. Results are quite similar for the 50 device scenario and have been omitted for brevity. The rate of change in network topology increases with speed 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), $AvgEffT$ 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 (as shown in Fig. 9), 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 reinstantiations 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.

Fig. 12(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 reinstantiation 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.

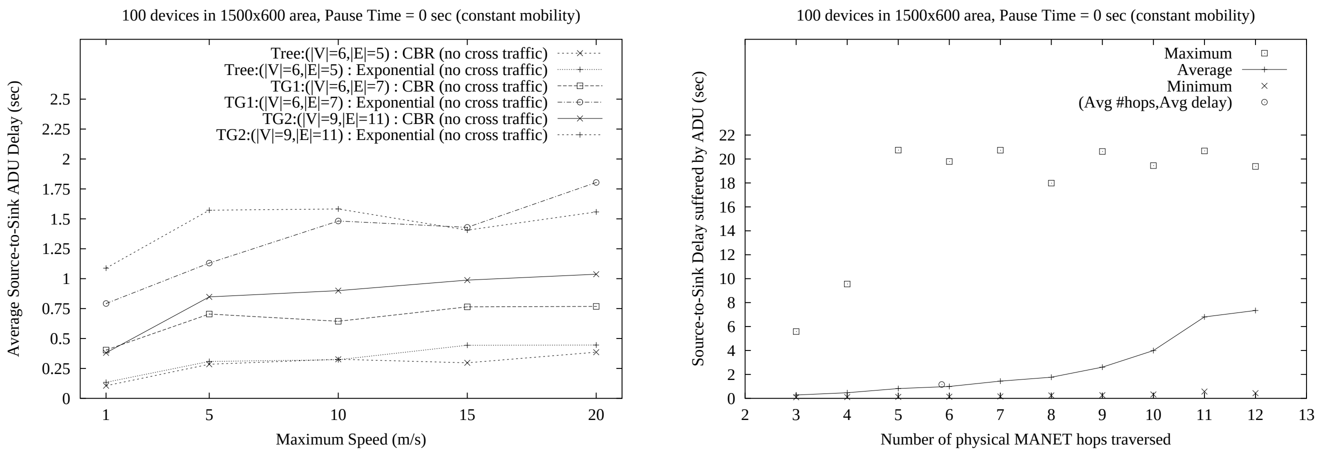


Figure 13: Source-to-Sink ADU Delay: (a) Average, (b) vs Hops (for TG2; MaxSpeed = 10m/s)

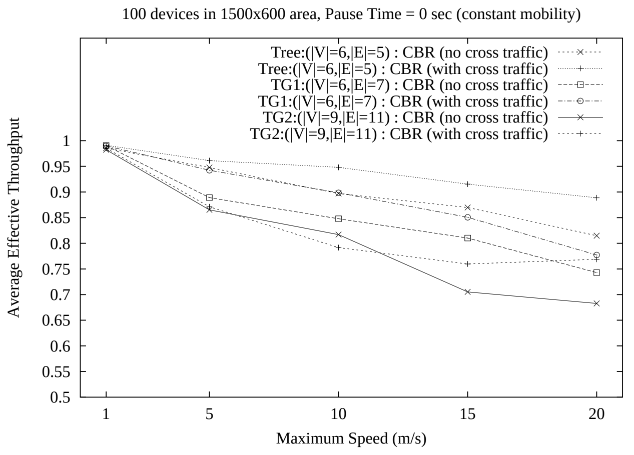
Average Source to Sink Delay We now analyze the effect of varying mobility speed on the source to sink delays suffered by application data units. The average delay values for all TGs have been plotted in Fig. 13(a). We observe very low delays for Tree TG (less than 0.5 sec) at all speeds for both CBR and Exponential traffic patterns although delays increase slightly with MaxSpeed. However, we observe greater delays for the non-tree TGs – TG2 has higher delays because it is a larger graph and has sinks at greater depths in the task graph than the other two graphs. TG1 has greater delay than Tree TG because all its sinks are at the greatest possible depth in the graph unlike the latter.

We also observe that exponential traffic suffers greater delay than CBR traffic in almost all cases. This is due to several reasons: first, although the mean burst size of exponential traffic is the same as that of CBR, larger ADUs suffer greater delays in a mobile network since they are subjected to route changes and hence TCP delays for a longer period of time, and these delays are not compensated adequately by the low delays experienced by small ADUs. Secondly, occasionally the time between ADU transmissions is much larger than mean 1 sec., and that results in a greater probability of route changes between successive ADU transmissions. Also, if two large bursts are generated in quick succession while reinstantiation is happening somewhere in the network or DSR is discovering new routes, the probability of experiencing greater delays increases.

Only average delay values were shown in Fig. 13(a). To give an idea about how delay varies for a particular scenario as a function of MANET hops, we plot a subset of the delay values for a representative scenario (TG2 ; MaxSpeed = 10m/s ; Pause Time = 0sec ; CBR traffic) in Fig. 13(b). We can see that the delay expectedly increases with the number of MANET hops traversed by the ADU. Although the maximum delays are very high (above 20 sec.), the average delays are reasonable. On the same graph we can see that an ADU on average traverses less than 6 MANET hops and experiences a delay of approximately 1 second. Average delay increases linearly up to 9 hops (at a rate of about 250ms per hop) but then increases non-linearly for a greater number of hops. We examine some more interesting properties of the ADU delay distribution later in this section.

Since the spatial device density is the same for both the 50 and 100 device scenarios, most metrics except embedding time exhibit similar behavior for both cases. The effect of device density on some of the above performance metrics has been discussed elsewhere [3].

Effect of Cross Traffic In our analysis so far, we have assumed that there is no additional traffic in the MANET except task data. In this section we examine the effect of other traffic



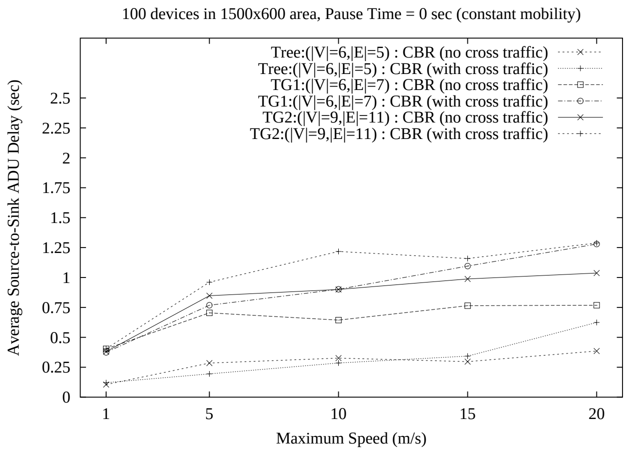


Figure 14: Effect of Cross Traffic: (a) Effective Throughput, (b) Average Source-to-sink Delay

in the network on the performance of our protocols. We choose 10 source-destination device pairs at random from the 100 devices, and each source sends a 512 byte UDP data packet to the destination every second. We report the effect of such cross traffic on our throughput and delay metrics in Fig. 14. We observe from Fig. 14(a) that introduction of cross traffic actually improves the effective throughput in the simulated scenarios. This is not surprising because the use of an on-demand underlying routing protocol like DSR causes routes along all paths which are not currently in use to become stale. This results in lower throughput at higher speeds when route errors and partitions cause reinstantiations. Introduction of cross traffic keeps alternative routes and route caches fresh at various nodes in the network and this results in quick recovery from reinstantiations or even lower number of reinstantiations (graph not shown) due to HELLO message timeouts. We see as much as 15% increase in effective throughput for the TG2 case.

However, one has to pay a price due to the flow of cross traffic. Fig. 14(b) shows the average source-to-sink delays suffered by an ADU for the simulated scenarios. We observe that introduction of cross traffic has resulted in slightly greater delays especially for the non-tree task graphs and for Tree TG at high speeds. This is because as mobility increases, more route errors occur and a significant number of control packets are injected into the network by DSR. This results in task packets getting delayed in interface queues since control data packets related to routing get preference over regular application data packets. The most significant increases in delay are seen for TG1 – from 750ms to above 1.25 sec at 20 m/s speed, primarily because all its data sinks lie far away from the data source.

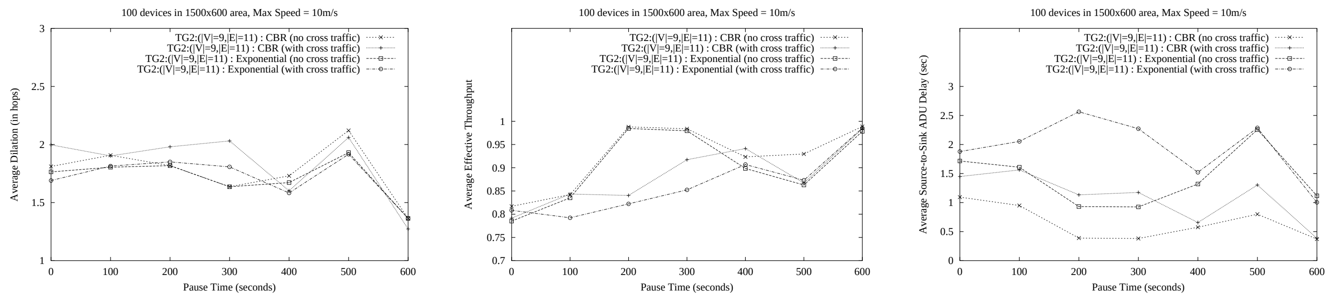


Figure 15: Effect of Varying Pause Time at 10 m/s for TG2: (a) Dilation, (b) $AvgEffT$, (c) ADU Delay

Effect of Variation in Pause Times Now we investigate the effect of variation in pause time while keeping MaxSpeed constant at 10 m/s. The pause time parameter PT was varied from 0 sec i.e. constant mobility (already considered earlier) to 600 sec (completely static) at steps of 100 sec. Note that the scenario generator that was used pauses *all* nodes for PT seconds before they start moving. Hence, for accuracy in simulations, we began the experiments and data collection at time $t > k \times PT$ where $k > 1$ and is sufficiently large such that a steady state that is characteristic of the movement pattern (in terms of fraction of nodes moving) was reached. In order to realize this we had to create mobility scenarios much greater in duration than 600 sec for $PT = 200, 300, 400, 500$. $PT = 0$ and $PT = 600$ are special cases and were handled by scenarios of duration 600s. Task execution protocols were run for 400 seconds from $t = 1100s$ to $t = 1500s$ for the former cases and from $t = 200s$ to $t = 600s$ for $PT = 0, 100, 600$.

The results have been shown in Fig. 15 for TG2 for all possible traffic patterns. We observe that average dilation is minimum ($d_{avg} \approx 1.4$) for the static scenario whereas it is more or less unaffected by PT (lies in the 1.6–2.1 range). This is expected as the spatial uniformity of devices is the dominant factor in determining this metric at all pause times.

From Fig. 15(b) we observe that our protocols manage to yield almost perfect effective throughput in the static scenario. This is because once the devices are instantiated, no reinstantiations happen due to mobility, and hence the sink node instances hardly lose any ADUs. Moreover, $AvgEffT$ does not drop below 0.8 even for the constant mobility scenario. This metric hovers in the 0.8–1.0 range for all the intermediate values of PT. Monotonously increasing patterns that are observed in the constant mobility and varying MaxSpeed case (Fig. 11) are not observed here because the rate of route changes between any pair of instantiated devices may not decrease monotonously with decrease in PT if most devices in that part of the network are static[18]. However, we believe that if a much larger number of scenario and random device pattern samples are taken, we are likely to observe a monotonous behavior.

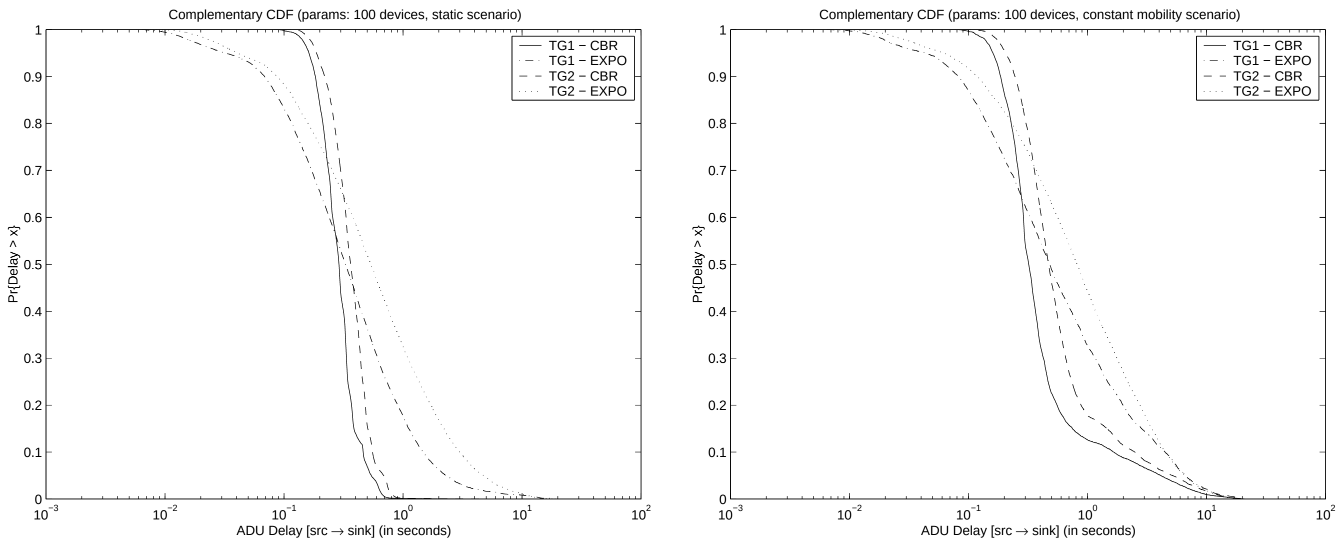


Figure 16: Cumulative Probability Distribution for ADU Delay: (a) Static, (b) Constantly Mobile

We also observe that Exponential traffic results in lower throughput and higher average delays (both with and without extra cross traffic) than CBR traffic. Reasoning for this is similar to the one mentioned in an earlier discussion (on effect of variation of MaxSpeed), and we do not repeat that here. However, we do observe that cross traffic is not able to improve the effective throughput in these scenarios. In fact cross traffic begins to help only at greater MaxSpeeds (15 and 20 m/s) for the TG2 case (see Fig. 14(a)) i.e. when route errors increase significantly.

Cumulative ADU Delay Distributions So far we have investigated only the average delays experienced by ADUs. We now examine the nature of the delay distributions that occur as a result of sending task data using CBR and Exponential traffic patterns. Fig. 16 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 Fig. 16(a), delays for the static case are 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.

[18]This could explain the extremely high throughput for $PT = 200$ and $PT = 300$ – actually very similar average delay values in Fig. 15(c) point to the same conclusion.

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 Fig. 16(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 paper, we do not attempt to investigate the exact statistical nature of the distribution, and leave that as a topic of future research.

7 Conclusions

In this paper we presented a task-based framework for embedding and executing a distributed application on a network of specialized, mobile devices. We developed a *task graph* abstraction for applications by taking into account the dependencies induced by the data flows existing between components of an application. We described the task-based anycasting (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 anycasting a given task graph onto a MANET. We also presented a scalable, local detection and repair mechanism for recovering from task disruptions caused by mobility of devices.

We showed a plethora of simulation results for evaluating the salient properties of the proposed distributed protocol. From our simulation studies, we observed that our protocol was able to instantiate and re-instantiate TG nodes quickly with low dilation, and yielded high effective throughput at low to medium mobility speeds and not much lower than 70% effective throughput for high mobility scenarios. We also noted that in some high mobility scenarios, the performance of our protocols suffered due to the non-aggressive retransmission timeout mechanism of TCP and aggressive route caching done by the underlying routing protocol, DSR. Introducing cross traffic load actually helped improve the effective throughput in such situations.

In this work, we did not simulate heavy loads since standard TCP acts as a major bottleneck over multiple hops under heavy loads [15]. Our focus in this work was to demonstrate the viability of the proposed dynamic task-based anycasting approach in MANETs using existing routing and transport protocols. We believe that much superior task execution throughput can be achieved at more demanding loads if standard TCP is replaced by feedback based schemes such as the one proposed in [9] – our future efforts will be directed towards such endeavors. Although we do not focus on reliable task execution in this paper, we note that it can be achieved with buffering and re-transmissions in the application.

Acknowledgments

This work is supported by the National Science Foundation (NSF) under grant No. ANI-0073843. We thank the anonymous reviewers for their suggestions for improving the paper.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," *Proc. 17th ACM SOSP*, Kiawah Island, SC, December 1999.

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

[3] P. Basu, W. Ke, and T.D.C. Little, "A Novel Approach for Execution of Distributed Tasks on Mobile Ad hoc Networks," *Proc. IEEE WCNC 2002*, Orlando, FL, March 2002, pp. 579-585.

[4] J. Beck, A. Gefflaut, and N. Islam, "MOCA: A Service Framework for Mobile Computing Devices," *Proc. ACM MobiDE*, Seattle, WA, August 1999.

[5] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "MACAW: a media access protocol for wireless LAN's," *Proc. SIGCOMM '94: Conference on Communications Architectures, Protocols and Applications*, London, England, September 1994, pp. 212-225.

[6] Bluetooth SIG, *http://www.bluetooth.com*

[7] S. H. Bokhari, "On the Mapping Problem," *IEEE Trans. on Computers,* Vol. 30, No. 3, 1981, pp. 207-214.

[8] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu and J. Jetcheva, "A Performance Comparison of Multi-Hop Ad Hoc Network Routing Protocols", *Proc. 4th ACM MobiCom*, Dallas, TX, 1998.

[9] 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.

[10] IEEE 802.11 Standard, *http://www.ieee802.org/11/*

[11] M. Esler, J. Hightower, T. Anderson, and G. Borriello, "Next Century Challenges: Data-Centric Networking for Invisible Computing," *Proc. 5th ACM MobiCom*, Seattle, WA, August 1999.

[12] M. Gerla and J. T.-C. Tsai, "Multicluster, mobile multimedia radio network," *Wireless Networks 1*, 1995, pp. 255-265.

[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," *Proc. 3rd ACM MobiCom*, 1997.

[15] G. Holland and N. Vaidya, "Analysis of TCP Performance over Mobile Ad Hoc Networks," *Proc. 5th ACM MobiCom*, Seattle WA, August 1999, pp. 219-230.

[16] P. Jacquet, P. Muhlethaler, A. Qayyum, A. Laouiti, L. Viennot, and T. Clausen, "Optimized Link State Routing Protocol," Internet-Draft, draft-ietf-manet-olsr-04.txt, September 2001. Work in Progress.

[17] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks", in *Mobile Computing*, edited by Tomasz Imielinski and Hank Korth, chapter 5, pages 153-181, Kluwer Academic Publishers, 1996.

[18] P. Karn, "MACA – A new channel access method for packet radio," *Proc. 9th ARRL/CRRL Amateur Radio Computer Networking Conference*, September 1990.

[19] R. Monien and H. Sudborough, "Embedding one Interconnection Network in Another," *Computing Suppl. 7*, 1990, pp. 257-282.

[20] C. E. Perkins, E. M. Royer, and S. R. Das, "Ad Hoc On-Demand Distance Vector (AODV) Routing," Internet-Draft, draft-ietf-manet-aodv-08.txt, March 2001. Work in Progress.

[21] Sun Microsystems, "Jini Technology Core Platform Specification," *http://www.sun.com/jini/specs*

[22] VINT Network Simulator - ns (version 2). *http://www.isi.edu/nsnam/ns/*