Routing in Delay Tolerant Networks Using Storage Domains

Padma Mundur, Sookyoung Lee, and Matthew Seligman

Abstract— **In this paper, we present a routing algorithm for a class of dynamic networks called the Delay Tolerant Networks (DTNs). The proposed algorithm takes into account the quintessential DTN characteristic namely, intermittent link connectivity. We modify the breadth first search (BFS) algorithm to take into account link state changes and find the quickest route between source and destination nodes. We adopt a message drop policy at intermediate nodes to incorporate storage constraint. We also introduce the idea of time-varying storage domains where all nodes connected for a length of time act as a single storage unit by sharing the aggregated storage capacity of the nodes. We evaluate the routing algorithm with and without storage domain in an extensive simulation. We analyze the performance using metrics such as delivery ratio, incomplete transfers with no routes and dropped messages. The DTN topology dynamics are analyzed by varying: number of nodes generating traffic, link probability, link availability through combinations of downtime/uptime values, storage per node, message size, and traffic. The delay performance of the proposed algorithms is conceptually the same as flooding-based algorithms but without the penalty of multiple copies. More significantly, we show that the Quickest Storage Domain (Quickest SD) algorithm distributes the storage demand across many nodes in the network topology, enabling balanced load and higher network utilization. In fact, we show that for the same level of performance, we can actually cut the storage requirement in half using the Quickest SD algorithm.**

Index Terms—**Delay Tolerant Network (DTN), Routing algorithm, Quickest delivery algorithm, and Storage domain algorithm**

I. Introduction

THE topic of this paper is efficient data delivery in dynamic network topologies with intermittent links. Specifically, we will focus on the design and development of routing algorithms for a class of networks that is distinctly different from the traditional TCP/IP-based networks. The class of dynamic networks under consideration is also referred to as delay tolerant or disruption tolerant networks (DTNs). As network technologies have evolved over the years, many non-traditional networks have been developed for instance, wireless, sensor, and mobile ad hoc networks. Reliance on infrastructure-based networking seems to be slowly eroding as we discover potential uses for these self-configuring networks. Applications for these networks range from military combat situations to civilian applications of vehicle-based mobile data centers; disaster relief situations where fixed infrastructure may have been destroyed; a commuter bus as it moves through rural areas provides connectivity by acting as a store and forward switch. Some industries are anticipating advanced vehicle-to-vehicle and vehicle-to-enterprise capabilities to set up vehicle-based mobile datacenters [www.erpdaily.com/news/2005], particularly useful for law enforcement surveillance vehicles.

DTNs are an emerging class of networks that define a new approach and a framework to provide networked services in non-TCP/IP networks, sometimes also referred to as "challenged" networks. Some unique challenges arise as we move away from the underlying assumptions for traditional IP-based networks [1]. To operate TCP protocol, there must be an end-to-end path between the source and the destination and the round trip delays must be small enough that there can be a "conversation" about the data transfer between the source and the destination. Neither of these assumptions is valid in a DTN -- intermittent connectivity makes it difficult to guarantee an end-to-end path for an ongoing data transfer and long round trip delays make it impossible to provide timely acknowledgements and retransmissions. The proposed DTN architecture offers a set of choices to counter these challenges: application-specific data units, known as *bundles* versus stream of packets; hop-by-hop delivery with optional in-network storage versus end-to-end routing. Given these new operational semantics, efficient data delivery becomes an important design issue with the objective of maximizing delivery, minimizing buffer/storage usage, and minimizing overhead due to routing protocols.

In this research, our objective is to design and develop efficient routing algorithms, protocols and other support services that take into consideration the absence of an end-to-end path and long network delays. For this paper, we focus on developing routing algorithms. Assuming a store and forward type of data transfers, our main objective in designing routing algorithms is to minimize the delay and maximize

Padma Mundur is with the Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD 20742 (e-mail: pmundur@umiacs.umd.edu).

Sookyoung Lee is with the Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, MD 21250 (e-mail: slee22@cs.umbc.edu).

Matthew Seligman is with the Laboratory for Telecommunication Sciences (LTS), 8080 Greenmead Road College Park, MD 20742 (e-mail: seligman@ltsnet.net).

reliability through higher delivery ratio subject to storage constraints on intermediate nodes connected by intermittent links. This simple formulation of the routing problem raises some important design considerations: (1) nature of disconnections – predictable or random; (2) links activated by mobile nodes with opportunistic intent for data delivery; (3) policies that govern message rejections and conservation of storage for more important messages – priorities and class of service; (4) congestion control – unlike IP-based network, congestion in DTNs manifests as lack of available storage on intermediate nodes due to high usage on a particular route.

In this paper, we present a routing algorithm that assumes predictable link connectivity and storage constraints on intermediate nodes. We modify the Breadth First Search (BFS) algorithm to take into account link state changes and find the quickest route possible between a source and a destination. Messages will be dropped if storage is unavailable on intermediate nodes. We also introduce the idea of *storage domain* where all connected nodes act as a single storage unit by sharing the aggregated storage capacity of the nodes in the domain. The storage domains are time-varying as links go up and down and constituent nodes in the domain change. We evaluate the routing algorithm with and without storage domains in an extensive simulation. The most significant simulation result as discussed later shows that routing with storage domain results in better performance even while cutting the storage requirement in half.

The paper is organized as follows: in Section II, we present related work. We discuss the proposed routing framework and the algorithm in Section III and IV. Detailed performance evaluation and simulation results are presented in Sections V and VI with a conclusion in Section VII. We present the pseudo-code for the proposed storage domain algorithm and an example in the Appendix.

II. Related work

DTNs are overlays residing above heterogeneous networks providing network services and interoperability (For architectural details see [2] and [1]).

Authors in [3], [4] propose several routing algorithms specifically for delay tolerant networks that consider intermittent connectivity. They modify Dijkstra's shortest path algorithm by including link weights that take into account the waiting time at nodes because of disconnected links. Different variations of this modified algorithm based on the knowledge of the dynamic network topology are presented. However, none of these variations consider the all important storage constraints on DTN nodes. While they propose an LP formulation that takes into account the storage constraints for the store and forward DTN network, the algorithm itself is heavyweight and impractical. In comparison, our proposed algorithm considers storage constraint within the routing framework as does the LP formulation but has the distinct advantage of being more practical.

In addition to the algorithms in [3], [4], other researchers have developed probabilistic models for describing intermittent link behavior using node mobility. The most well known algorithm using that approach is PROPHET from [5] and variations of it with storage constraints in [6]. The basic idea behind this algorithm is to represent link connectivity behavior using a probabilistic metric called *delivery predictability* at every node in the network topology to each known destination. For instance, nodes that frequently encounter each other will result in high delivery predictability. In [6], the authors combine probabilistic routing from their previous work with various buffer management policies to analyze performance in terms of message delivery, end-to-end delay and overhead. In [7] authors derive a directional link cost for each node using connectivity history which takes into consideration the number of transitions from disconnected state to connected state between pairs of nodes, the duration of disconnected state and so on. We can develop a similar technique in our algorithm to predict link state changes using historical connectivity logs. We will address that in our future work. While the newer version of PROPHET in [6] considers buffer management policies, our proposed storage domain algorithm goes further by incorporating storage as a constraint in routing. Transfers without a complete route with storage to the destination are not initiated at all. The in-network storage is utilized better for that reason.

For mobility based opportunistic link connectivity, other researchers have followed two distinct approaches. The first approach is to model the intrinsic mobility of the nodes using such mobility models as Random Way-Point (RWP) and the second approach uses controlled mobility inspired by the robotic applications. While the first approach has fallen out of favor to some extent in the recent past [8], the controlled mobility approach using special mobile nodes *ferries* is gaining attention as a more realistic approach for mobility to enable link connectivity among stationary nodes in DTNs.

Many of the mobility-based solutions are inspired by the routing problem in sparse ad hoc networks where network partitions similar to DTN can occur. Most of the early algorithms however, rely entirely on node mobility to move data in the event of network partitions such as epidemic routing in [9], or a refinement of the same algorithm in [10]. The authors in [11] propose a routing scheme called Spray and Wait that aims to reduce the overhead of flooding by spraying a limited number of copies into the network and waiting to see if that will suffice to reach the destination node. Li and Rus [12] propose an approach where mobile hosts actively modify their trajectories to deliver messages. Musolesi and others in [13] consider asynchronous communication between nodes in separate clouds by computing delivery probabilities for each host and using the host with the highest delivery probability to actually deliver the data across the cloud. In the same spirit, [14] present five strategies for opportunistic forwarding of messages when two mobile routers are within transmission range but where mobility of vehicles is not controlled. Many of these algorithms are unsuitable for DTNs with stationary nodes.

Zhao and others [15], [16] employ special mobile nodes called message ferries in the deployment area that move in a predictable manner among the stationary nodes to help collect and deliver the data. They address the challenging issue of ferry route design under single or multiple ferries, single or multiple routes, different degrees of interaction between the ferries and nodes. They present algorithms to calculate ferry routes which minimize delivery delay for fixed traffic demand. Using the message ferrying scheme, Chuah and Yang consider buffer management issues to provide differentiated services in [17], [18] so that urgent messages with a guaranteed level of service receive better performance than the regular messages.

In this paper, we abstract the cause of link connection state change and simply consider link up and down times in our routing algorithm. This type of predictable link state knowledge could result from implementing node mobility within the network topology as in message ferrying scheme where the ferry routes are predetermined. Most researchers in this area have been forced to work under either of the two extreme positions – knowing everything about the network topology dynamics or knowing nothing about it. We have assumed that we can assemble topology knowledge and that we have a mechanism to know node and link state changes, for instance, by using an out-of-band, low bandwidth communication mechanism that is different from the data networks needed for a more robust data transfer. Dissemination of control information in a DTN to promote network topology awareness is still an open research area and is not addressed in this paper.

Consideration of storage limitation in DTNs is another important design factor we can not ignore and for that reason, data forwarding mechanisms such as broadcast and flooding are not appropriate. In the recent past, most researchers model storage as a limited resource in the DTN context -- [6], [16], [17], [18] among them. Many of their solutions are however, limited to message drops due to buffer overflow while differentiating and enforcing message priorities. In this paper, we explicitly consider storage constraint as part of the routing problem. In that respect, our paper is similar to some of the LP-based formulations but with one distinct advantage: our algorithm is more practical. In the proposed storage domain routing algorithm, the quickest path with available storage is chosen for each transfer. The proposed algorithm therefore, provides minimum delay, similar to flooding-based approaches, without duplication. The routing solution using storage domains proposed here can easily be adopted using the architectural guidelines from DTNRG such as custody transfer to forward data reliably both within a storage domain and between storage domains. With the proposed algorithm we are pre-computing the routes with storage for each transfer, and therefore, we can easily implement custody transfer. In a related paper in [19], Seligman et al. implement custody transfer policies at *individual* nodes to mitigate storage limitation in DTN. This is appropriate and necessary when complete network topology knowledge is not known and each node has to decide to take custody based only on information on its neighbors.

III. ROUTING FRAMEWORK

Our initial approach for developing a framework for routing in DTN is based on algorithm design and graph theory. We propose to formulate the routing problem using the network topology graph as input with nodes (vertices) having limited buffer, and links (edges) with contact establishment information (when, where, for how long). What makes this formulation different and challenging is the time-varying nature of the underlying topology and the storage constraints on intermediate nodes. Even as we acknowledge that the data being transported is not real-time, the primary emphasis will be on quick delivery – minimizing delay is still an important goal.

Network Connectivity In a DTN environment, disconnections can be long lasting and not generally related to network faults as in traditional networks. The following types of connectivity are possible in a DTN: Predictable or Scheduled, Random or Probabilistic, and Opportunistic. Probabilistic or opportunistic connectivity can be enhanced by node mobility as seen from most of the papers on mobile ad hoc networks mentioned before.

Congestion in DTN will take the form of unavailable storage on DTN nodes for message transfers. Techniques to avoid and control congestion manifest in the routing problem formulation as storage constraints.

A. *Routing using Modified Breadth-First Search (mBFS)*

In this section, we present work that forms the basis for the proposed algorithmic approach which we introduced in [20]. We present a modification to the breadth-first search algorithm to find the *quickest* route between a given source and any destination node in a delay-tolerant network. This is done without flooding the network – at any one time we maintain only one copy of the message in the network. The delay performance of the proposed algorithm and its improved storage domain version is conceptually the same as flooding-based algorithms.

Main assumptions in developing this routing algorithm are: 1. that the link state changes are predictable; 2. that the links are symmetric and, when up, have sufficient bandwidth to carry the messages needed; 3. that intermediate nodes have persistent storage; 4. that network and transmission delays are negligible compared to the delays due to parts of the network being unreachable. Our algorithm determines the path in its entirety at the time of message origination.

The assumption of predictable link state changes is justified and similar to the situation presented in other works in this area. In our model, we do not use an agent to bring about link state changes but leave it as implementation dependent. For instance, Zhao and others in [15] employ special mobile nodes called message ferries in the deployment area that move in a predictable manner among the nodes to help collect and deliver the data. Their main idea is to make the node movement non-random so that data delivery is planned and more efficient. Given a pre-determined ferry route the nodes can either be static or pro-actively move closer to a ferry. With this type of set up, an event list for link state changes of the type we use in our

formulation can be easily generated. Abstracting the link connectivity behavior without using node mobility makes our algorithm applicable to more diverse environments.

Algorithm Description We adapt the breadth-first search (BFS) algorithm for graphs to find the "quickest" route from a single source node to all other nodes in the graph. The pseudo-code for the proposed algorithm is shown in Figure 1.

We assume an undirected graph G = (V, E) where V is a set of vertices (or nodes) and E, edges. We assume an adjacency list representation of G, consisting of an array A of |V| lists, one for each node in V. One of the nodes s Є G is the source node. With delay tolerant networks, any edge (u, v), u, v Є G may be added or deleted at any time, in turn changing G. In general, we call these additions and deletions of edges *events*. We assume that events are predictable, in that we assume that we know in advance which edge will be added to or deleted from the graph and at what time. We refer to fixed edges as static and edges which get added or deleted as dynamic. In our analysis we will assume a starting configuration for G at time t_o. We define a time-ordered set $Evts(u, v, t_e, a)$ to represent the set of events. Each event in the set is represented by a 4-tuple: (u, v) Є G is the edge that is added to or deleted from E at time t_e and *a* denotes the action, which could be either ADD or DELETE. We propose a time limit *T* called the *look ahead time* (LAT) to within which we are to restrict our search. This is to avoid potentially endless event lists where edges are added and deleted regularly. Thus the set *Evts* must contain all events which occur between times t_o and *T*.

```
Modified BFS(G, x, t_discovered)
  1    F ← {x}
  2    While F ≠ Φ
  3    Do u ← head(F)
  4       For each v Є A[u]
  5       Do if color[v] == WHITE
  6          Then color[v] ← GRAY
  7               d[v] ← d[u] + 1
  8               π[v] ← u
  9               d_t[v] ← t_discovered
  10              ENQUEUE(F,v)
  11         DEQUEUE(F)
  12         Color[u] ← BLACK

Single-Source Quickest Delivery (G, s, t_o, T,
Evts)
  1    For each vertex u Є V[G] - {s}
  2    Do color[u] ← WHITE
  3       d[u] ← ∞
  4       Π[u] ← NIL
  5       d_t[u] ← NEVER
  6    Color[s] ← GRAY
  7    d_t[s] ← t_o
  8    Modified BFS(G, s, t_o)
  9    While Evts ≠ Φ
  10   Do Evt ← DEQUEUE(Evts)
  11      u ← u(Evt);
  12      v ← v(Evt);
  13      t_e ← t_e(Evt);
  14      If a(Evt) == DELETE then
  15         E ← E - (u, v)
  16      Else Do E ← E U (u, v)
  17         if color[u] ≠ color[v]
  18         Then do
  19            if color[u] ≠ BLACK
  20              then swap(u, v)
  21              d[v] ← d[u] + 1
  22              π[v] ← u
  23              d_t[v] ← t_e
  24              Color[v] ← GRAY
  25              Modified BFS(G, v, t_e)
```

Figure 1. Pseudo-code for the quickest delivery algorithm (mBFS)

The proposed modified BFS (mBFS) algorithm calculates a route without in-network storage constraint; however, a message is successfully delivered only if there is available storage on all nodes in its path. To calculate a path from a source node *S* to a destination node *D* at a time t_s where t_s is the message origination time, we initially search all nodes reachable immediately from *S* using mBFS. Each node is assigned the time t_s as the node discovery time, $t_{discovered}$. If a destination *D* is reached in the initial search, the shortest path from *D* back to *S* is returned. Otherwise we keep searching other undiscovered nodes to find the *D*. For this, each event in Evts(u, v, t_e, a) is processed from t_s for the duration of the look-ahead-time, *T*. The current topology G is first updated as the event action is ADD or DELETE. If the addition of an edge leads to the discovery of a new node at a certain time *x*, mBFS is called to find other nodes which can be reached through the node at time *x*. The discovered time kept at each node during mBFS search represents the earliest reachable time from a source node *S*. This is because, the discovered time, $t_{discovered}$ is assigned when the node is first discovered by the earliest link up event among events sorted by time. The transfer for a message will not be initiated if a destination node *D* is not discovered even after processing all events between t_s and (t_s + LAT) and will count as a failure. Otherwise, the final route is calculated by following the predecessor of each node from a destination node *D* all the way back to the source node *S*. The computed route is the quickest delivery path from *S* to *D* because each next-hop node from *S* to *D* is reached at the earliest possible time given an events list. During the transfer from *S* to *D*, a message could be dropped due to storage constraint along the path. For a drop policy, we propose that the message with the longest life time in a queue would be dropped when there is no available storage. This conforms with the idea of not transmitting "stale" data.

Analysis In addition to the O(*V*+*E*) time taken for *Modified BFS*, we need to compute the time taken to process the events. Line 17 of our algorithm ensures that only previously unexplored nodes are used as source nodes when calling *Modified BFS* on Line 25. We ignore events where both nodes are the same color, which implies that they are both either discovered or undiscovered. Theerefore, nodes are discovered only once by our algorithm irrespective of the event length or sequence. The running time of the *modified BFS* part of the algorithm therefore is the summation of the running times of BFS on disjoint parts of the graph, or O(*V*+*E*). Since each event is processed once, the running time of the event processing part of the algorithm is O(*Evts*). Therefore, the total running time of the algorithm is O(*V*+*E*+*Evts*).

B. Routing with Storage Constraint

We next introduce the constraint that the amount of storage available at any node is limited. This implies that when we make routing decisions, we must ensure that the message can be stored in its entirety on nodes along the predecessor tree determined by breadth-first search (path). In the basic *mBFS* routing algorithm, storage is considered outside of the routing decision. As the message is transmmitted, it will be stored on a node for next-hop transfer only if the following equation holds true; otherwise the message gets dropped. If S_u is the total storage available on node u, m is the size of the arriving message, and s_u is the amount of storage in use at node u, we must ensure that: $s_u + m \leq S_u$

Drop policy coupled with the proposed algorithm mBFS addresses storage limitations on intermediate nodes. However, this solution does not address mitigating congestion due to unavailable storage on frequently used routes. Our solution is to develop algorithms which use storage from nodes that may not necessarily be on the routing path to the destination. We introduce the idea of *storage domain* as a connected network of nodes, each providing storage on behalf of another node when that node does not have sufficient storage. This idea will help in reducing dropped transfers and result in better performance as we show later in the simulation results. Because of the available connectivity among the nodes in a storage domain, storage on different nodes could be viewed as a single storage. Messages on these nodes could be forwarded back and forth within the domain, thus mitigating the storage limitation on some bottleneck nodes.

IV. Routing using Storage-domains

A. Storage Domains in the Proposed Algorithm

To find the quickest route using storage domains, we must consider routing the message through not just those nodes along the predecessor path, but through other nodes connected to the nodes on the path. It is possible that while storage cannot be found on a node located along the path, storage may be found on nodes which are connected to the congested node while the message is in transit through that node. We must explore all such possibilities. We do this as follows.

If a set of nodes are connected during certain times between times t_o and T, we ignore the routing issues between them (since they can be addressed by traditional routing mechanisms) and assume that we can store the message on any of the connected nodes as convenient. We call such node sets *storage domains*. We thus transform our task from finding routes between nodes to finding routes between storage domains.

We discover storage domains by processing link additions and deletions. A link is redundant or non-redundant based on its effect on the storage domains. When a non-redundant link is added, it combines two storage domains into one; when a non-redundant link is deleted, a storage domain is split into two smaller domains. In Figure 2, all links that are added or deleted are non-redundant links. Since these links alternate between up and down events, the storage domains in the proposed routing algorithm are time-varing as shown in Figure 2. Where links are static, the constituent nodes will always form a storage domain.

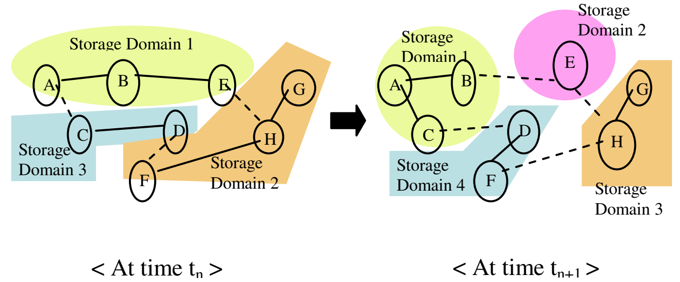

Figure 2. Time-varying Storage Domains

B. Node Re-discovery

While the simplest route between a given source node and any destination would be the one discovered by the algorithm presented earlier, it is possible that storage considerations force a more tortuous route, including possible loops.

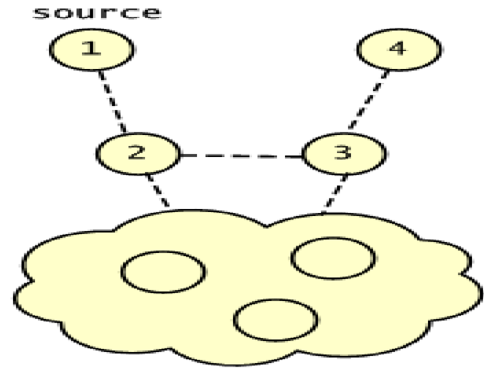

Figure3. Example to illustrate node re-discovery

In the network shown in Figure 3, if a message is to be routed from node 1 to 4, the quickest path has been determined to be 1-2-3-4. It is possible that before the link 3-4 is established, several link state changes occur between 2 and 3. Further, it is possible that as other link state changes occur between 2, 3, and the rest of the network, storage conditions change on 2 and 3, forcing the message to oscillate between 2 and 3. It may be that the link 3-4 cannot be established for a few hours, and that the link 2-3 changes state every few minutes. The conditions are such that storage becomes scarce on 2 at the top of the hour, and on 3 at the bottom of the hour. In this contrived example, the message would have to be transferred back and forth between 2 and 3 hourly until the link 3-4 is eventually established. The proposed algorithm is designed to handle node re-discovery required in situations described above. Unlike traditional routing algorithms, looping within reason is a desirable characteristic for DTN routing. In evaluating different paths

between a source node and a destination node, we must take into account possibilities such as the one above. We do this by allowing nodes to be *re-discovered* as we process events.

Oscillations and loops of unknown length are possible in the proposed storage domain algorithm as discussed above and even have desirable effect for some transfers by offering better storage management. However, their adverse effect is mitigated to the extent that there is no wastage of real resources. This is because for each transfer, the complete route is computed a priori and the transfer initiated only if a successful route with storage is found within a given LAT.

C. Quickest Delivery Algorithm with Storage Domain (Quickest SD)

The pseudo code for the proposed algorithm is presented in the Appendix along with an example. Here we provide a brief description of the essential details of the algorithm. We start at the source node s at time t_o and discover all nodes immediately reachable, marking them with their time of discovery. These nodes form a storage domain. We use three pieces of information – time of formation, time of break-up, and a label (for uniqueness), to identify a storage domain.

We then process the events from the set *Evts*. When a link is added, if one of the nodes is already discovered, we explore all newly reachable nodes and record the time of their discovery t_d, and the predecessor node information. When two previously discovered storage domains that are currently disconnected are subsequently connected by the addition of a link, each domain would be treated as the predecessor of the other. In addition, when a link is added, the associated nodes from both domains form a new storage domain. When a link is deleted, if it leads to partitioning of the storage domain (that is, if the link is not redundant), a message stored on the domain would have to be stored in one of the two new, smaller domains. We process this condition by terminating the large storage domain, starting two new storage domains, and making each of the new domains the predecessor of the other. In our algorithm, as predecessor information is updated on a node, we update all nodes in the storage domain with the same information. This enables us to ignore the routing issues between nodes of a storage domain and treat all nodes in the same domain to belong to a given path. Note that although a node may be part of different storage domains at different times, a node belongs to one and only one storage domain at any given time.

To determine a path between a source and a destination, we process all events and start at the destination node and find the earliest event that led to its discovery, and find its predecessor node. We then repeat the process with the predecessor node, finding its earliest predecessor, and so on, recursively, until we work back to the source node at time t_o. We would now have found one possible path. We then verify whether sufficient storage is available on each of the storage domains during the times the storage is required along the path.

If storage is not found on a domain along the path, we mark its successor node with a flag (to avoid re-trying the same path later) and move on to the next path, by choosing the next earliest predecessor on the successor node, work back to the source node at time t_o along a different path, and check for storage along the new path. We repeat the process of trying new paths by choosing the next predecessors systematically along all predecessor nodes starting from the destination in the order of the discovery time, until we find a path with sufficient storage along it. If no such path can be found, we conclude that the message cannot be delivered within the look-ahead time T.

Implementaion Details In our notation (see pseudo-code in the Appendix), we use a set ***P*** of three-element members to store predecessor information for each node, which include the *time* when the predecessor becomes reachable, the *predecessor node*, and the *flag* that denotes whether delivery along that path has already been attempted, as described above. The flag has value *TRY* initially and is changed to *DONT* when we determine that storage is unavailable on the predecessor storage domain. We use another set ***S*** of three-element members for each node to store storage domain information – the time of formation, time of break-up, and a label. We use the look-ahead time T as default to denote the time of break-up until we have knowledge of when the break-up actually occurs. The label (we use one of the node names as the label) is needed to differentiate between two domains formed as a result of the break-up of a domain. The same information consisting of (time of formation, time of break-up, and the label) on two or more nodes shows that they are part of the same storage domain for that time interval.

Finally, we use a two-dimensional array *Avail*, a V x *Evts* matrix, to update storage availability information on each node between times t_o and T. The example presented in the appendix provides step by step execution of the algorithm including changes to set P, set S on each node and the Avail matrix.

Analysis The routine *mbfss* requires running repeated breadth-first searches. In the worst case, each event would cause breadth-first search to be run on the entire network. Therefore *mbfss* runs in $O(V + E)*Evts$, where $V = |V|$ is the number of nodes, and $|E|$, number of links.

In the recursive routine *FindRoute* for the Quickest SD algorithm (see pseudo-code in Appendix) we note that each predecessor tree is explored at most once. Once it is determined that a sub-tree does not yield a valid custody transfer schedule, the flag enables us to avoid the sub-tree during subsequent searches. The running time of the routine is therefore proportional to the number of predecessor nodes recorded in all nodes which is the same as the running time of the routine *mbfss*, or $O(V + E)*Evts$.

V. Performance Evaluation

A. Simulation Setup

We evaluate the proposed DTN routing algorithm using ns2. Our DTN network topology consists of 15 or 30 nodes with intermittent links between pairs of nodes. There are several

parameters that are applied to affect the basic network topology and evaluate its performance. These are listed below with a description of their effect on the network performance:

Link probability: this parameter is related to topology construction and defines the number of neighbors any node will have in the DTN topology. For instance, 0.1 link probability gives a sparsely connected network than a link probability of 0.5. Therefore, we can expect higher link probabilities resulting in a better delivery ratio.

Disconnection periods – UpTime and DownTime of each link are generated using exponential distributions with a certain mean. For instance, 50/200 sec indicates a uptime mean of 50 sec and downtime mean of 200 sec. The network topology with lower/higher downtime/uptime results in a better delivery ratio.

Storage capacity per node: In the basic mBFS algorithm, we use a drop policy whenever there is no storage on the intermediate node. Higher storage means that we can reduce message drops in intermediate nodes. In the Quickest SD algorithm there are no message drops because the transfer is not initiated if a complete route with storage is not found. However, higher storage will still mean more routes with storage and therefore, higher delivery ratios.

Look Ahead Time (LAT): This parameter is the result of our routing algorithm and the modified BFS. Longer look ahead times mean better delivery ratio and fewer transfers with incomplete routes.

Workload: In our simulation the workload is expressed in terms of messages per second (mps). Each node generates messages with mean exponential interarrival times. The destination for the message is randomly picked. Each data transfer is affected by link disconnections along the path, the LAT, and the storage available on intermediate nodes. Each transfer can result in three different outcomes: 1) failure to find a route because either algorithm failed to discover the destination node within the given LAT and with Quickest SD in particular, failure to find a route with storage; 2) once the quickest path is found between the source and the destination using the mBFS algorithm, the message may be dropped due to unavailable storage on intermediate nodes; 3) the message gets transferred successfully.

Message size: Higher the message sizes, lower is the expected delivery ratio because of storage limitations. In our simulation, we do not consider message fragmentation due to a possible network partition during transmission. We implement a message to be processed in its entirety as it arrives at each intermediate hop. Also in our simulation, we assume that the link bandwidth is unlimited since transmission delays are a negligible part compared to link up and down times.

Table 1 summarizes the various values of each parameter we used for the simulation.

B. *Performance Metrics*

The performance metrics used in the simulation are:

Delivery Ratio (*DR*): is defined as the ratio of successful transfers to number of overall transfers. Overall transfers will include those that result in no routes and message drops in addition to the successful transfers.

$$DR = S / (S + N + D)$$

where S is the number of successful transfers, N is the number of no routes, D is the number of message drops.

Number of successful transfers (*S*): this metric defines the number of complete transfers with storage on intermediate nodes.

Number of No Routes (*N*): this metric defines the number of transfers that result in incomplete paths to the destination because the mBFS fails to find a path within the given LAT.

Number of message drops (*D*): this metric defines the number of transfers failed to complete because of storage unavailability at intermediate nodes. This metric is relevant to mBFS algorithm and the drop policy used is remove the oldest message in the queue.

Table 1. Parameter values used in the simulation

Parameter	Value
Number of nodes	15, 30
Link Probability: Probability of link connection between any two nodes	0.1 (low) 0.2, 0.3 0.25 (medium) 0.4 0.5 (high)
Messages/second: Number of messages generated per second on each source node	0.25, 0.5 0.75, 1
Simulation Time	1000sec.
Look-ahead-time (LAT)	300sec.
Link Downtime/Uptime duration	400/50sec. 350/50sec. 300/50sec. 250/50sec. 200/50sec 150/50sec 100/50sec 50/50sec
Message size	10KB, 20KB, 30KB, 50KB, 70KB, 90KB, 110KB
Storage on each node	100KB, 200KB, 300KB, 400KB, 500KB, 600KB, 700KB

VI. SIMULATION RESULTS

We simulate the network environment using workload parameters with some combinations of values described in Table 1. The proposed routing algorithms are evaluated using flat network topology. The two routing algorithms evaluated are the basic mBFS with message drop policy (labeled DP) and mBFS with storage domain, also called the Quickest SD algorithm (labeled QSD). All results are subjected to 95 percent confidence interval analysis. The intervals themselves are very small and not shown on all graphs. Each experimental result is averaged over 5 trials. Within each trial, a warm-up period is used to eliminate the influence of initial system state.

A. *Effect of Look Ahead Time (LAT)*

Among the simulation parameters mentioned before, the look-ahead-time (LAT) has a critical influence on the performance of the proposed routing algorithm. In order to find a final route, both algorithms first consider time-variant DTN topology within a given look-ahead-time and then the Quickest SD algorithm additionally considers available storage in time-variant storage domain with two kinds of history lists – predecessor list and storage availability list -- calculated within the given LAT. Each algorithm has more information to explore an available route with longer look-ahead-time. However, a long look-ahead-time introduces longer delays in route computation with only marginal improvement in delivery ratio. In Figure 4, we show the results of this experiment for LAT ranging from 100 to 400 seconds. All other parameters are fixed as shown. For the DP algorithm, the unsuccessful messages include messages with no calculated route using the modified BFS function and messages dropped during transfer. For the Quickest SD algorithm, however, it means messages with no route found within LAT considering path and storage simultaneously. The left diagram of Figure 4 shows the number of unsuccessful message transfers in each routing algorithm. Considering the two diagrams in Figure 4, both algorithms get into the stable status at 300 second LAT. We will use this value in all other experiments that follow.

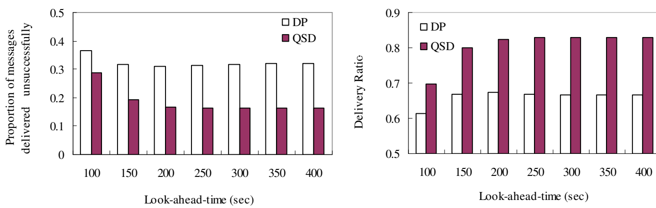

Figure 4. Proportion of messages delivered unsuccessfully (left side) and Delivery ratio (right side) depending on different Look-ahead-time; 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 0.25 messages/sec., 200/50 sec. Link Downtime/Uptime, 300KB storage on each node and 10KB message size

B. *Effect of Traffic*

In Figure 5, we show results of an experiment where we vary traffic, messages per second, injected from each source node in the DTN topology. We show the results for a combination of storage and link probability values both of which have desirable effect on the delivery ratio. Higher link probability indicates a well connected network and higher storage mitigates storage limitation. However, each algorithm shows different rate of increase of delivery ratio. Figure 6 represents the differential in delivery ratio improvement from Quickest SD algorithm over DP algorithm with respect to medium and high link probabilities of DTN topology and different amount of storage on each node. At the most desirable scenario of 0.5LinkProb, 300 KB storage, the advantage of Quickest SD over DP keeps increases even as we increase traffic, where as at the middle of the road scenario of 0.25 LinkProb, 300 KB storage the performance differential is more stable. The third scenario depicting lower link probability (0.25) and lowest storage (150 KB) shows that the Quickest SD algorithm gradually loses advantage over DP as we increase traffic since network connectivity and storage limitation play a dominant role as we increase traffic for both algorithms. It is still significant that the Quickest SD algorithm always performs better than DP over a wide range of traffic situations as well as network topology dynamics as shown in Figures 5 and 6.

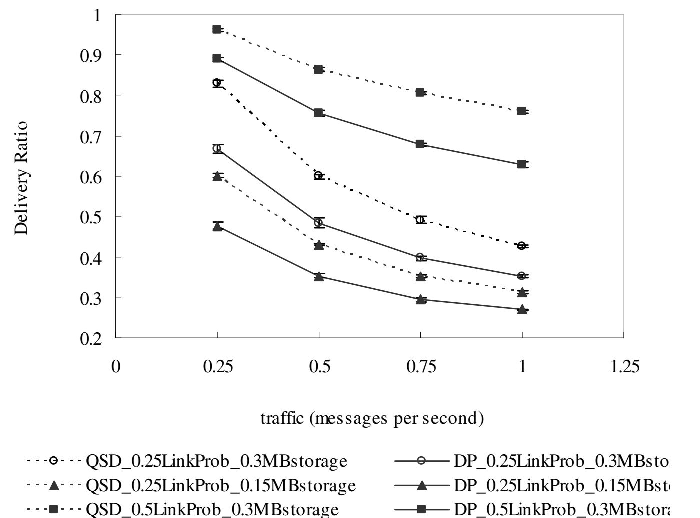

Figure 5. Delivery ratio according to the different amount of traffic in DTN; 15 nodes, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 200/50 sec. Link Downtime/Uptime, and 10KB message size

C. *Effect of Storage*

Figure 7 shows the effect of varying storage from 50KB to 700KB on the delivery ratio. Quickest SD shows better performance over DP with the largest difference occurring at 300 KB storage. The Quickest SD algorithm is likely to exploit available storage on all nodes in DTN to determine a successful route for each message. Another significant result from this experiment is that to achieve the same level of performance from the DP algorithm, we have to double the storage – compare Quickest SD performance at 300KB to DP's performance at

600KB. The proposed Quickest SD algorithm is a routing algorithm that overcomes performance degradation of DTN due to storage constraint. On the other hand, the difference of delivery ratios between the two algorithms becomes very small when the amount of storage on each node is too small or too large like 50 KB and 700 KB in our simulation environment. It is because too small or too large storage means that no smart mechanism is needed to use storage.

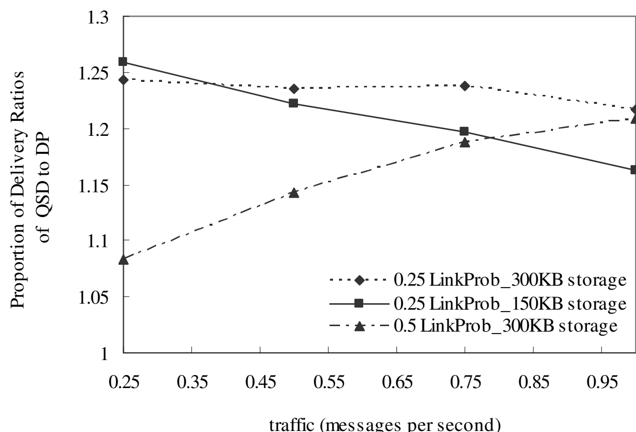

Figure 6. Comparison of delivery ratios of the Quickest SD algorithm and the DP algorithm; 15 nodes, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 200/50 sec. Link Downtime/Uptime, and 10KB message size

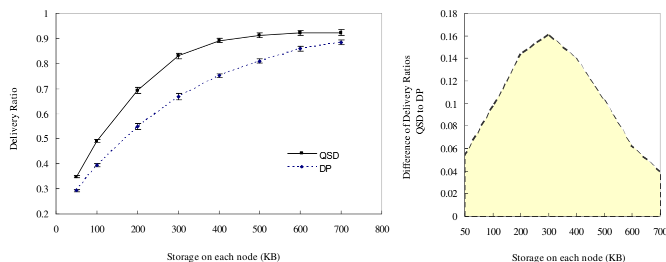

Figure 7. Delivery ratio according to the different amount of storages on each node in DTN (left side) and difference of delivery ratios between the Quickest SD algorithm and the DP algorithm (right side); 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 200/50 sec. Link Downtime/Uptime, and 10KB message size.

D. Effect Link Availability

Figure 8 shows how various link disconnection intervals affect delivery ratio of each algorithm. Both algorithms have higher delivery ratio as link downtime gets lower with the ideal and equal performance at 50/50 downtime/uptime. Notice that the highest performance differential between the two algorithms occurs at 200/50 link availability. If the link downtime is equal to the link uptime as seen in the two leftmost bars, the delivery ratios of both algorithms reaches almost 1.0. This is because the traffic generated during link down time is relatively small to be stored that most messages can be transferred during the next link up interval at intermediate nodes. Storage is not a limitation at this level of link availability. Since a larger link downtime needs more storage with in DTN for both algorithms, delivery ratios decrease as the link downtime increases from 50 to 700. Given the topology dynamics and the LAT used for this experiment, the largest performance differential between the two algorithms occurs at 200/50 link availability – a decrease or increase in downtime from that value makes the performance of both algorithms converge.

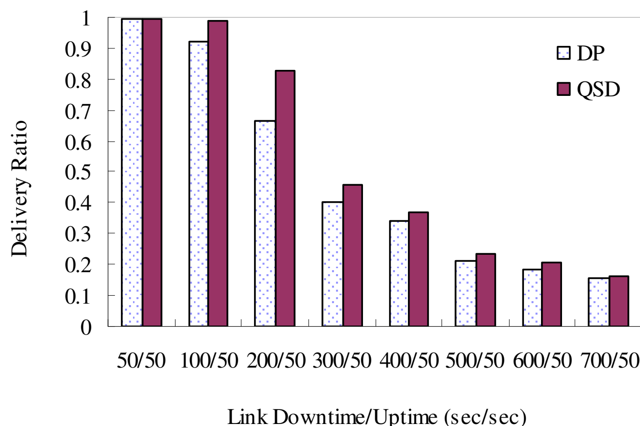

Figure 8. Delivery ratio depending on different link deactivation duration; 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 300KB storage on each node and 10KB message size.

E. Effect of Storage and Link Availability

In this experiment, we analyze the effect of storage with different levels of link availability results of which are shown in Figure 9. We fix the link up time at 50 seconds on average and vary link downtime as 50, 100, 150, 200 and 250 seconds. The top two graphs show that both algorithms approach almost 1.0 delivery ratio when link downtime and uptime are the same as 50 seconds which was the result we observed in Figure 8. No messages are dropped during transfer using the DP algorithm because no link is overflowed. In this experiment all messages to be delivered unsuccessfully are caused by insufficient routes in DTN due to low link probability.

As the link downtime increases, the delivery ratio generally degrades in both routing algorithms. Since the Quickest SD algorithm implements a greedy mechanism to determine a route, the overall network storage is used in a more efficient way than the DP algorithm. Therefore, the former algorithm achieves much higher delivery ratio than the latter even as we increase link downtime. The significant result in this experiment is that the Quickest SD algorithm with 200 second link downtime outperforms the DP algorithm with 150 second link downtime. Also of significance is the performance of the two algorithms at 250/50 second link availability. As we increase storage, notice the diverging performance between the Quickest SD and DP – the reason for this is the improved storage utilization with the Quickest SD algorithm which is required when the downtime is as large as 250 seconds.

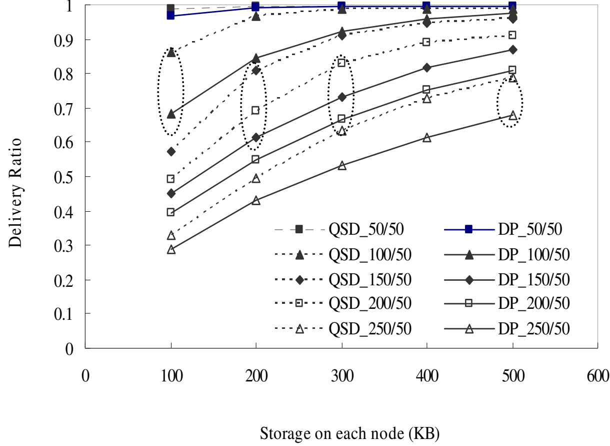

Figure 9. Delivery ratio with different link deactivation duration according to different amounts of storage on each node; 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 message/sec., and 10KB message size

F. Effect of Message Size

Figure 10 represents delivery ratios of the two routing algorithms when different size messages are generated and injected into DTN. In this experiment, each node produces about 250 messages during the entire simulation time using the exponential distribution with 4 seconds mean inter-arrival time. Per node storage is fixed at 300KB storage. With a 10KB message size each source generates 2500 KB of storage demand and each message is needed to be forwarded or stored at the maximum 300KB storage allocated on each node along the successful path.

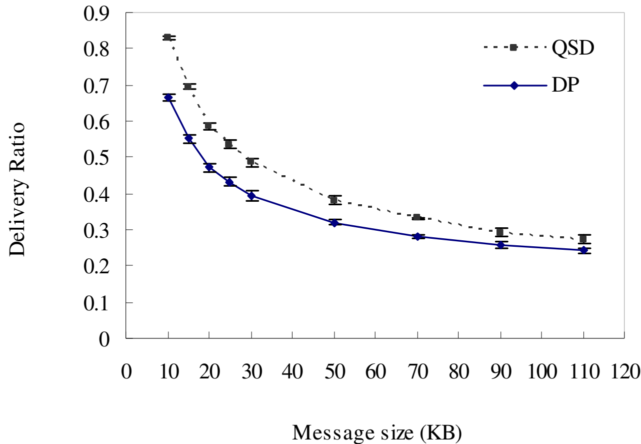

Figure 10. Delivery Ratio depending on different message size; 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 200/50 sec. Link Downtime/Uptime, and 300KB storage on each node

As shown in Table 2, the average number of nodes along the successful path is about 6 and 3 for the Quickest SD and DP algorithms respectively. This result indicates that the Quickest SD algorithm uses twice the storage as the DP algorithm to deliver a message successfully. However, this result points to the disadvantage of the DP algorithm that it does not make use of the available storage effectively. The DP algorithm exploits storages of only three nodes on the path calculated using mBFS to transfer a message. Since the Quickest SD algorithm is, on the other hand, capable of calculating all possible routes considering storage availability in advance before a message is really transferred through DTN, it theoretically exploits storage on all nodes in the network (15 nodes in our simulation).

We present results related to delay and hop count in Table 2 from the same experiment as in Figure 10. Delay refers to the waiting time at intermediate nodes for the links to come up. It is interesting to note that the average delay that each transfer incurs decreases as the message size increases. The reason for this is that there is actually less traffic using network resources as we increase message size. The delivery ratio is degraded but the average delay is improved as the message size increases. Since less traffic uses the overall network resources, messages delivered successfully spend less time across the DTN. This phenomenon is more emphasized in the Quickest SD algorithm because it does not put messages that do not have a complete route (no route) into the network.

Table 2. Average Delay and average hop according to different message size when link probability is 0.25

Message size (KB)	Total Storage Required (KB)	Average Delay (sec)		Average Hop		Average Delay incurred in one hop	
		QSD	DP	QSD	DP	QSD	DP
10	2500	70.91	69.44	6.25	2.87	11.4	24.2
20	5000	69.31	62.46	5.82	2.76	11.9	22.6
30	7500	62.93	54.98	5.39	2.68	11.7	20.5
50	12500	52.78	43.70	4.82	2.54	11.0	17.2
70	17500	49.03	37.40	4.31	2.51	11.4	14.9
90	22500	39.95	32.30	3.98	2.46	10.0	13.1
110	27500	36.10	27.88	3.71	2.41	9.7	11.6

Figure 11 presents delay of each transfer for the length of simulation time from the experiment using 10KB message size in the first row of Table 2. The left side shows delay and hop counts from the DP algorithm and the right side from the Quickest SD algorithm. The delay values in Figure 11 are obtained as the sum of waiting times at each hop due to link unavailability for each transfer. The delay values are bounded by 300 (LAT). The average of the delay values for all transfers over the simulation time are shown in Table 2. The warm-up period used in the simulation gets rid of any undesirable effect from the initial state of the links. It has been verified that these delay values follow the exponential distribution as they should.

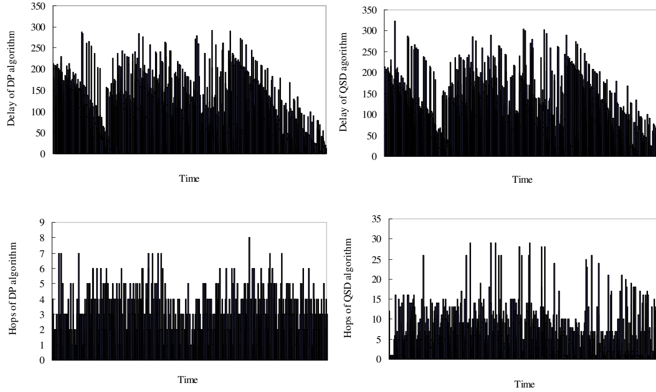

Figure 11. Trace representing Delay values and Hop count for DP (left side) and QSD (right side) algorithms during simulation time; 15 nodes, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 200/50 sec. Link Downtime/Uptime, 300KB storage on each node and 10KB message size

G. *Effect of Link Probability*

The bar graph seen on the left side of Figure 12 shows how delivery ratio is dominated by link probability used to construct DTN topology. Delivery ratios of both routing algorithms degrade as lower link probability is sued to generate DTN topology. It is because both algorithms primarily depend on physical link availability in DTN regardless of storage amount assigned on each node. The link availability is determined by the number of links in DTN calculated by link probability and its dynamic characteristics decided by link downtime and uptime. With fixed link downtime and uptime, a lower link probability means that the number of available routes is small.

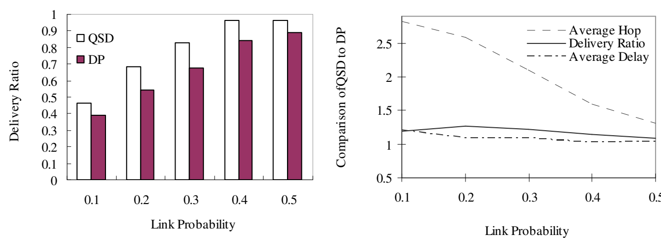

Figure 12. Delivery ratio for different Link Probabilities (left side) and proportions of QSD to DP for Delivery ratio, average hop and average delay between two routing algorithms depending on different Link Probabilities (right side); 15 nodes, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 200/50 sec. Link Downtime/Uptime, 300KB storage on each node and 10KB message size

Table 3 presents the number of neighbors (allocated to a node calculated by link probability), average delay, and average number of hops that a message incurs during transfer in each algorithm. The average delay per transfer for both algorithms goes down as we increase the number of neighbors each node has for the obvious reason that there are more routes available. The average hop count for Quickest SD hovers around 6 and for the DP algorithm around 3. DP shows a slight increase in hop count while the opposite is true for Quickest SD as we increase the number of neighbors. The explanation for DP results is that the algorithm is not optimized in terms of hop count and will pick a route that results in the quickest time even if it is the longer route. For Quickest SD, increasing the number of neighbors means more routes to explore and also spread the storage demand. It will employ shorter routes if the longer quicker route does not have available storage. The hop count for each transfer can have higher variability in Quickest SD because it is designed to explore more number of routes than DP. This result can also be seen in Figure 11. At lower link probability, Quickest SD will use nodes that are not on the routing path for storage and this detour will result in higher hop counts. In the adjoining graph in Figure 12, we see that Quickest SD maintains its superior performance in delivery ratio and average delay where as average hop count converges to a smaller number as explained before.

Table 3. Number of neighbors of each node and average delay and hop incurred during transfer depending on link probability with 15 nodes in DTN

Link Prob.	Number of neighbors of each node (nodes)	Average Delay		Average Hop		Average Delay incurred in one hop	
		QSD	DP	QSD	DP	QSD	DP
0.1	1.5	149.75	123.82	6.86	2.43	21.8	43.9
0.2	3	109.88	100.61	7.11	2.75	15.5	36.6
0.25	3.75	70.91	69.44	6.25	2.87	11.4	24.2
0.3	4.5	74.41	68.65	6.47	3.06	11.5	22.4
0.4	6	37.93	37.23	5.04	3.17	7.5	11.7
0.5	7.5	27.17	25.95	4.13	3.15	6.6	8.2

H. *Scalability and Stability*

Throughout the many experiments we conducted for evaluating the algorithms, we have also addressed issues of scalability and stability of both of them. The DTN topology reflects different degrees of network connectivity as determined by link probability and link up and down interval. In Figure 12, the Quickest SD algorithm shows better performance over DP in a stable pattern for varying link probability. Figure 8 presents results for different values of link downtime/uptime. Figure 7 and Figure 10 show that the Quickest SD algorithm produces stable graphs with better performance as we increase storage and message size. Also the Quickest SD algorithm shows scalability and stability as a function of number of nodes in DTN as seen in Figure 13. Figure 13 shows the behavior of each algorithm when the number of traffic source changes. Each source generates the same amount of traffic during simulation time using the exponential distribution. While the DP algorithm shows high sensitivity in delivery ratio when traffic increases, the Quickest SD algorithm presents a stable performance in this experiment. Since the Quickest SD algorithm utilizes overall network storage capacity, it is not highly sensitive to the change in traffic amount. Figure 14 shows the results for delivery ratio when we double the number of nodes in the DTN topology,

from 15 nodes to 30 nodes. We see that when the topology size is doubled, the performance obtained from Quickest SD is in the acceptable range of 70 to near 100%. The relative performance differential between the two algorithms is still maintained. The performance for 15 node topology corresponds to the bottom two curves in Figure 9.

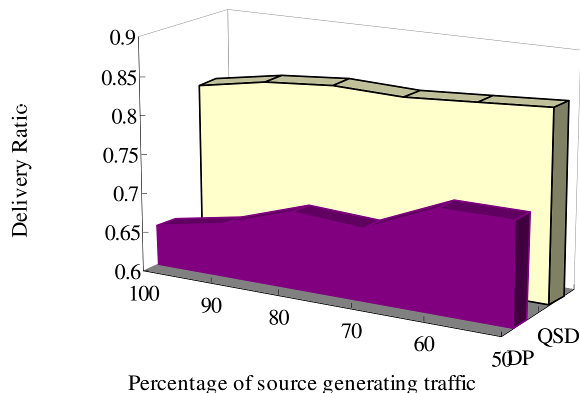

Figure 13. Delivery ratio depending on different percentage of source nodes generating traffic; 15 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 200/50 sec. Link Downtime/Uptime, 300KB storage on each node and 10KB message size

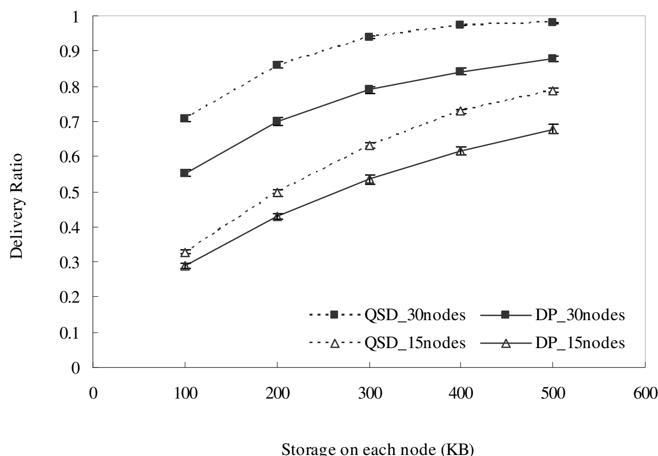

Figure 14. Delivery ratio depending on number of nodes in DTN; 15 or 30 nodes, 0.25 Link Probability, 1000 sec. Simulation Time, 300 sec. Look-ahead-time, 0.25 messages/sec., 250/50 sec. Link Downtime/Uptime, and 10KB message size

VII. CONCLUSION

In this paper, we presented two routing algorithms that result in the quickest delivery time in a DTN environment comparable only to flooding-based algorithms but without the penalty of multiple copies in the network for each transfer. We successfully incorporated the storage constraint into routing in the Quickest SD algorithm. The algorithm is ideally suited for implementing custody transfers between nodes on the path as the routes are initiated only if storage is available. We modified the simplest routing algorithm, namely, the breadth first search (BFS) algorithm to suit DTN environment. We extended the BFS to handle DTN link state change events essential for implementing intermittent connectivity. We assume that these events are predictable but as part of future work, we will model topology dynamics in a more comprehensive manner that includes probabilistic or opportunistic link state changes. In the current paper, the DTN topology dynamics are analyzed by varying: 1) number of nodes generating traffic, 2) link probability, 3) link availability through combinations of downtime/uptime vales, 4) storage per node, 5) message size, and 6) traffic. Most significantly, we show that the results due to the Quickest SD algorithm spread the storage demand across many nodes in the network topology, enabling balanced load and superior network utilization. Summarizing the results, we conclude that:

- Quickest SD always results in better performance than DP for the same network conditions.
- Longer look ahead times generally increase delivery ratio but are limited by the degree of network connectivity and link availability.
- Larger storage will increase delivery ratio for both algorithms. However, too small or too large storage results in only marginal improvement.
- Quickest SD reduces storage requirement in half for the same level of performance with DP.
- In general, higher link availability means higher delivery ratio for both algorithms. However, Quickest SD can tolerate higher link downtime for the same level of performance from DP because of improved storage utilization with the SD algorithm.
- Relative performance advantage is maintained by Quickest SD as message size is increased.
- Both algorithms demonstrate scalability and stability through the many experiments we have shown. However, Quickest SD algorithm shows lower sensitivity and therefore, higher stability to changing network or workload conditions.

ACKNOWLEDGMENT

The authors would like to thank Manohar Malayanur and Yun Teng for fruitful discussions on the algorithms presented in this paper. They would also like to thank the editors and the anonymous reviewers for their insightful comments.

REFERENCES

[1] F. Warthman. Delay-Tolerant Networks (DTNs), A Tutorial. *http://www.dtnrg.org/*

[2] http://www.dtnrg.org/

[3] S. Jain, K. Fall, and R. Patra, “Routing in a delay tolerant network,” in *Proc. ACM SIGCOMM*, 2004.

[4] J. Alonso and K. Fall, “A linear programming formulation of flows over time with piecewise constant capacity and transit times,” *Technical report IRB-TR-03-007, Intel Research Berkeley*, July 2003.

[5] A. Lindgren, A. Doria, and O. Schel´en, “Probabilistic routing in intermittently connected networks,” in *Proc. of the Fourth ACM International Symposium on Mobile Ad Hoc Networking and Computing* (MobiHoc 2003), June 2003.
[6] A. Lindgren and K. S. Phanse, “Evaluation of policies and forwarding strategies for routing in intermittently connected networks,” *IEEE Distributed Systems Online*, August 2006.
[7] D. Thakore and S. Biswas, “Routing with persistent link modeling in intermittently connected wireless networks, “ in *Proc. of MILCOM* 2005.
[8] J. Yoon, M. Liu, and B. Noble, “Random waypoint considered harmful,” in *Proc. of IEEE INFOCOM*, April 2003.
[9] A. Vahdat and D. Becker, “Epidemic routing for partially connected ad hoc networks,” *University of California, San Diego Technical Report CS-2000-06*, July 2000.
[10] X. Chen and A. L. Murphy, “Enabling disconnected transitive communication in mobile ad hoc networks,” in *Proc. of the Workshop on Principles of Mobile Computing* (POMC’01), August 2001.
[11] T. Spyropoulos, K. Psounis, and C. S. Raghavendra, “Spray and Wait: An efficient routing scheme for intermittently connected mobile networks,” in *Proc. Of the Workshop on Delay Tolerant Networks* (WDTN), ACM SIGCOMM Workshops, 2005.
[12] Q. Li and D. Rus, “Communication in disconnected ad hoc networks using message relay,” *Journal of Parallel and Distributed Computing*, 2003.
[13] M. Musolesi, S. Hailes, and C. Mascolo, “Adaptive routing for intermittently connected mobile ad hoc networks,” in *Proc. of the ACM WoWMom*, 2005.
[14] J. LeBrun, C-N Chuah, D. Ghosal, “Knowledge-based opportunistic forwarding in vehicular wireless ad hoc networks,” in *Proc. of IEEE VTC*, Spring 2005.
[15] W. Zhao, M. Ammar, and E. Zegura, “Message ferrying approach for data delivery in sparse mobile ad hoc networks,” in *Proc. of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing* (Mobihoc), 2004.
[16] W. Zhao, M. Ammar, and E. Zegura, “Controlling the mobility of multiple data transport ferries in a delay tolerant network,” in *Proc. of the IEEE INFOCOM* 2005.
[17] M. Chuah and P. Yang, “Message ferrying scheme with differentiated services,” in *Proc. of the IEEE MILCOM* 2005.
[18] R. Viswanathan, J. Li, and M. Chuah, “Message ferrying for constrained scenarios,” in *Proc. of the ACM WoWMoM* 2005.
[19] M. Seligman, K. Fall, and P. Mundur, “Alternative custodians for congestion control in delay tolerant networks,” in *Proc. of the Workshop on Delay Tolerant Networks* (WDTN), ACM SIGCOMM Workshops, 2005.
[20] P. Mundur, S. Lee, and M. Seligman, “Routing in intermittently connected networks,” in *Proc. of the ACM/IEEE MSWiM*, October 2006.

Padma Mundur received a Masters degree in Systems engineering from the University of Virginia, Charlottesville, VA, in 1990 and the Ph.D. degree in information technology from George Mason University, Fairfax, VA, in 2000. She was a faculty member in the Computer Science Department at University of Maryland, Baltimore County from 2000 to 2006. She is currently a research faculty at the Institute for Advanced Computer Studies at University of Maryland, College Park (UMIACS). Her research interests include distributed systems, multimedia networking, DTNs and other challenged networks, and analytical performance modeling. She is on the editorial board of IEEE Communications Surveys and Tutorials Journal; she has been on program committees for ICDCS, ICME and others. She has served as a reviewer for NSF panels, and many IEEE/ACM journals.

Sookyoung Lee received a Masters degree in Computer Science from the Ewha Womans University, Korea, in 1997. She was with LG ELECTRONICS Inc., Electronics and Telecommunications Research Institute, Korea Electrics Technology Institute, and Samsung Electronics Co. LTD, Korea from 1998 to 2004. She is a Ph.D. student in the Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County since Fall 2005. Her primary research interest is in network modeling and performance analysis for dynamic and sparse networks.

Matthew Seligman is a networking researcher at the Laboratory for Telecommunications Sciences (LTS) and is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Stevens Institute of Technology. He received his Masters degree in Electrical Engineering from Stevens Institute of Technology and his Bachelorsdegree in Computer Engineering and Electrical Engineering from Pennsylvania State University. He previously held positions at Coree Networks and Lucent Technologies in New Jersey. His research interests include DTNs, network simulation, and embedded systems. He is a member of ACM and IEEE.

APPENDIX
Quickest SD Algorithm – Pseudo-code

```
mbfss(G, s, Evts, to, T, Avail, m)
 1. Run mbfs with s as source. For each node found:
 2.   P ← P U (to,Φ,TRY)
 3.   S ← S U (to,T,s)
 4. Process Evts. For each event:
 5.   If link (u,v) is added at time te
 6.     If one of the nodes, v, is undiscovered
 7.       Run mbfs with v as source. For each node found:
 8.         P ← P U (te,u,TRY)
 9.         S ← S U (te,T,u)
10.       Run mbfs with u as source. For each node found:
11.         Find the last entry in S; replace T by te
12.         S ← S U (te,T,u)
13.     Else if both nodes are already discovered and link is not redundant:
14.       Run mbfs with v as source. For each node found:
15.         Find the last entry in S; replace T by te
16.         P ← P U (te,u,TRY)
17.         S ← S U (te,T,u)
18.       Run mbfs with u as source. For each node found:
19.         Find the last entry in S; replace T by te
20.         P ← P U (te,v,TRY)
21.         S ← S U (te,T,u)
22.     E ← E U (u,v)
23.   If link (u,v) is deleted at time te:
24.     E ← E - (u,v)
25.     If the link is between previously discovered nodes, and is not redundant:
26.       Run mbfs with u as source. For each node found:
27.         Find the last entry in S; replace T by te
28.         P ← P U (te,v,TRY)
29.         S ← S U (te,T,u)
30.       Run mbfs with v as source. For each node found:
31.         Find the last entry in S; replace T by te
32.         P ← P U (te,u,TRY)
33.         S ← S U (te,T,v)
```

```
findRoute:
 1. Start at the destination node d and find the earliest record u in P.
 2. If predecessor node(u) = Φ then done
 3. Else
 4.   For each predecessor node p in P(u) from earliest to latest:
 5.     Start new linked list L = u
 6.     If FindRecRoute(L,p) succeeds then done
 7.     Else conclude that no route exists
```

```
findRecRoute(L,u)
 1. Enqueue(L,u)
 2. If predecessor node(u) = Φ then
 3.   If custXfer(L) is successful then return success
 4.   Else return failure
 5. Else
 6.   For each predecessor p in P(u) from earliest to latest:
 7.     If FLAG = TRY
 8.       If FindRecRoute (L,p) is successful then return success
 9.       Else
10.         FLAG(u) = DONT
11. Dequeue(L,u)
12. Return failure
```

```
custXfer(L)
 1. For each record u in L
 2.   Find the records in S for the duration of the custody on u
 3.   For each record
 4.     Determine the set of nodes which form the storage domain with u in it
 5.     Check if any node v exists in the domain where su + m ≤ Su
 6.     If it does, note it down
 7.     If not:
 8.       Mark the FLAG of the previous node in L = DONT
 9.       Return failure
10.   For each node noted down:
11.     Reserve the storage of size m for the duration needed (update Avail matrix)
12. Return success
```