FINAL REPORT

April 30, 2010

DOT

Dual-ordered Object Transport

DARPA DTN
Disruption-Tolerant Networks

Principal Investigators:
Ignacio Solis - *isolis@parc.com*
JJ Garcia-Luna-Aceves - *jjgla@parc.com*
Palo Alto Research Center

Contents

1 Summary

This is the final report for RIDE (Robust Internetworking in Disruptive Environments). A Content Centric infrastructure built for DARPA's DTN program Phase III. The report has two major themes, the protocol simulation and the prototype infrastructure. They both work towards the goal of delivering content in these harsh environments. First by studying the performance under complex simulated networks and second by working with a real system.

Content dissemination in disrupted networks poses a big challenge, given that the current routing architectures of ad hoc networks assume the need to establish routes from sources to destinations before content can disseminated between them. In ad hoc networks subject to disruption, lack of reliable connectivity between producers and consumers of information makes most routing protocols perform very poorly or not work at all. We present DOT (Dual-ordered Object Transport), which is a content dissemination approach for ad hoc networks that exploits in-network storage and the hop-by-hop dissemination of named information objects. It is a continuation of our work from phase II, DIRECT (DIsruption RESilient Content Transport) Extensive simulation experiments illustrate that DOT provides a high degree of reliability while maintaining acceptable levels of delivery latencies and signaling and data overhead compared to epidemic routing.

We have also implemented a prototype of the DOT protocol as a router plug-in module for the DTNRG's DTN2 Routing Daemon. This prototype has allowed us to evaluate DOT in a more realistic environment and has shown us many of the areas that need attention to bring such a technology to the real world.

This document should meet the following deliverable requirements:

- CDRL A003 - *CLIN 0004AA* - Content-Based Opportunistic Routing Algorithm Report. (See sections 2, 3, 4 and 6)
- CDRL A004 - *CLIN 0005AA* - Final Report with DTN experimentation evaluation (See sections 5 and 7)

2 The Algorithm: DOT

Two key assumptions in the design of the routing protocols for the IP Internet and wireless multi-hop ad hoc networks (MANETs) have been that: (a) physical connectivity exists on an end-to-end basis between sources and destinations for extended periods, and (b) routes are established to reach the locations of the intended destinations. These assumptions have had profound implications on how communication bandwidth is shared, how routing is accomplished, and how data are disseminated. In particular, routing in packet-switching networks has been based on routing tables derived entirely from topology (or connectivity) information that represents only a snapshot of the state and characteristics of network links at particular instants. Furthermore, while the assumption of routing information to specific addresses renders adequate delivery rates in connected networks with stable topologies, it can lead to very poor resource utilization in disrupted networks when physical paths to the original locations of content are lost for long periods of time, or never exist.

The cost, energy consumption, and form factors of computing devices have enabled embedded computing and networking devices that can be used in environments in which end-to-end connectivity may be intermittent at best. These new application environments range from interplanetary research to wearable computers. Example applications include monitoring of wild life and disruptive phenomena (e.g., wild fires), object tracking, emergency response, vehicular or interpersonal networking, and the tactical communication in the battlefield. Networks in which end-to-end connectivity is not guaranteed have been called delay-tolerant, disruption-tolerant, intermittently-connected, episodically-connected, or highly-partitioned. In this paper, we use the term *disruption-tolerant networks*, or DTNs. Clearly, routing in a DTN cannot be accomplished in the same way as routing in a network in which end-to-end connectivity is assumed to exist except for extraordinary circumstances. As the seminal work by Fall [12] and others has shown, routing in a DTN must be accomplished differently than in traditional networks. Routing in the Internet and well-connected ad hoc networks has no temporal dimension, because it is based on a distributed or local search of paths obtained from snapshots of the network topology. In contrast, routing in a DTN must be a function of space and time, because physical links exist only temporarily, and paths from sources to destinations can be considered to exist only as functions of connectivity (links) occurring over time.

Section 4 summarizes the prior work on routing for DTNs, which started with the work of the Interplanetary Internet Research Group (IPNRG) [5] within the

IRTF (Internet Research Task Force). This survey indicates that prior solutions to the DTN routing problem either have focused on establishing spatio-temporal routes to specific destinations, rather than to content, or have modified epidemic dissemination of content. These solutions have relied on knowing the entire network topology, the ability to control some nodes, or the ability to duplicate data freely in the network.

Section 2 presents DOT (Dual-ordered Object Transport), which disseminates content by name on-demand to sites with interest in content from the nearest sites hosting the content. DOT is based on the previous work for Phase II, DIRECT and is similar to Directed Diffusion [**?**], in that named content is disseminated on the basis of statements of interest. However, DOT accommodates disruptions in network connectivity of arbitrary duration. To accomplish this, DOT disseminates interest in content persistently across connected components, so that content can be replicated opportunistically as nodes move across such components, but only towards those nodes that have expressed interest.

Section 3 presents the results of simulation experiments used to compare DOT against epidemic content dissemination. The results indicate that DOT provides the best of both approaches. DOT is able to transition from an epidemic-like approach to a connected network approach and drastically reduce overhead.

2.1 DOT

DOT is based on the dissemination of interest in and copies of information objects. Every object is has a name and a publisher. The name and publisher are encoded as strings. To retrieve an object from the network, a node expresses an interest in the object name and publisher. This contrasts with the current delay-Tolerant infrastructure proposed by the DTNRG[10] and much of the prior work on disruption tolerant networks, where reliable information dissemination is still based on the identification of endpoints reminiscent of TCP connections.

DOT is the continuation of our work on DIRECT. DIRECT was based on simple query ordering to distribute interests. DOT on the other hand uses a dual ordering mechanism (hence the name Dual-ordering Object Transport). They do share part of the architecture, specifically the API presented to applications. An application designed for DIRECT can work under DOT.

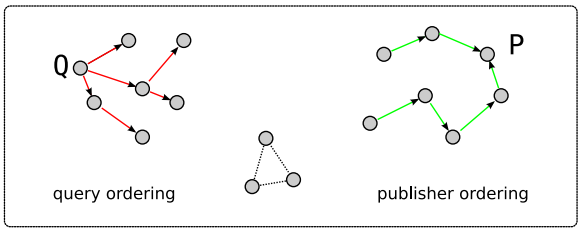


Figure 1: Dual-ordering

2.2 A Content Centric Interface to Applications

DOT defines three high level primitives: Publish, Get and Handle. These primitives give the network more flexibility in managing data and provides more functionality to applications.

- PUBLISH: When a node wants to make some information available to other nodes it *publishes* an object. Publishing an object involves giving the object a name and some attributes. The local network daemon then takes possession of the object. This is equivalent to the node becoming a server for that piece of static data. When other nodes require or request the data it will be transmitted.

- GET: If a node wants some specific piece of data it asks the network to *get* that object. The node needs to specify the name and attributes of the object it wishes to receive. This specification can include expressions to allow for complex queries. The current instantiation of DOT allows for prefix and postfix matching in the name and publisher of an object. This can effectively be used for hierarchical naming, which DOT exploits for routing.

- HANDLE: Not all objects need to be created before they are accessible, some objects will be created as replies to specific queries or gets. This is the case of dynamic data, for example. A program can request to handle a subset of the naming space. In this situation, the network will tell the program when it has received a get for a name and the program can generate the object on the fly and issue a publish.

Communication happens when an interest(get) matches a published object. When a node wishes to make some data available it will publish the data object. That object will stay at the publishing node until an interest is received. When a

node wants to retrieve some data, it will issue a get. If matching data exists it will be returned to the requesting node. If matching data does not exist but a matching handle exists the application that has issued the handle will receive the interest. It can the publish data to match the interest and the normal retrieval method will return the data to the requester. If no matching data exists the query will eventually time out.

Nodes in a DOT network are configured with identifiers. These identifiers act as hostnames, aliases and group identifiers, and they are used as the publisher identifier of an object. A single node can have multiple of these but an object can only have one publisher.

Each node maintains four main tables, the neighbor table, the object table, the query table and forwarding table. The neighbor table keeps track of all the neighbors that the node encounters and their status. The object table holds all the information the node has about objects and their (possible) location. The query table contains all the queries the node has received or created.

2.3 Algorithm

Publishing is pretty straight forward. When a program wants to publish an object it gives the object to the network layer with its *absolute name*. We assume that, in the majority of cases, the publisher of the object is one of the node identifiers. The publisher is tied to a cryptographic signature, which basically assumes that the node has permission to publish under that name (by having the right key). It is possible for a node to "publish" an object created by another entity (for example, a pre-signed object), but in this case, the node is merely adding the object to the cache, not performing a publish operation.

Once an object is published, it is the responsibility of the network layer and it is added to the object table. A locally published object is never discarded, unless explicitly told to do so by an application.

To retrieve an object from the network, a program creates a query to issue a get. The application fills in a query data structure. This includes the name of the object, the publisher, the type and the count. Wildcards can be used in the name, publisher and type. The count specifies how many objects the application would like to receive.

When the network receives the get from the application, it has to decide what to do with the query. In the simplest case, we can match the whole query to objects that are stored locally. In this case they would just be returned to the application. This can be done via a callback or through polling by the application.

If we can't answer the query locally we will have to look for the objects in the network. Under default operation DOT will do this in 4 phases. First it will try to ask the publisher, then a proxy. If those two fail it will issue a scoped flood and finally a full flood. All of these parameters can be configured depending on the situation. For each phase there is a timer that monitors if we get a reply. If the timer expires DOT moves on to the next phase.

In the initial phase DOT will try to get the data directly from the publisher. This basically makes DOT work like a connected routing protocol. We will check our forwarding table. If an entry exists for the publisher we will route the query in that direction. DOT will populate the forwarding table as it receives objects from different publishers. However, if DOT is working under a hybrid environment it can take advantage of the existing connected routing to populate the routing table. This is assuming there is some way to map from standard routing identifiers (ip addresses) to publishers. If there is no entry in the forwarding table for the publisher this phase is skipped.

If there is no match for the publisher in the forwarding table then DOT will check if there is a proxy setup for this publisher. While logically this can be done with a separate proxy table, DOT currently achieves this by adding proxy entries to the forwarding table. Engineering proxy locations for specific publishers is out of the scope of this paper but is being studied. If there is no proxy entry in the forwarding table for this publisher this phase is skipped.

If neither of these approaches work, DOT continues by trying to find the content in the local neighborhood using scoped flooding. This basically tries to see if any of our immediately connected neighbors have the content cached. The query is basically issued with a small hop count limit (TTL). If there is no reply to the scoped flood and policy allows for network flooding then the protocol continues by flooding the network to find the publisher.

When a new query is received by a node, it is added to query table along with the previous hop information. The query is then compared to the objects in the object table. If a match is made, the object is returned to the previous hop. If the query has a count greater than 1 then the process is repeated, decreasing the count for every match found. Nodes keep a cache of transmissions so they don't re-send objects to the same node. If the query has not been fulfilled with local matches, that is, the count is greater than zero, then we need to check if it needs to be forwarded and where. This depends on the phase the query is in (flooding for example), TTL, etc.

If a query reaches a count of 0, it becomes inactive. This query is basically fulfilled and doesn't need any more replies. If other nodes don't know that the

query has been fulfilled and are looking for more data, we will tell them via a "silence" query.

Upon receiving an object, a node adds it to its object table. It then checks whether the object matches any of the active queries. If it does, the object is forwarded to the respective previous hops or up to the application. The count on the queries is decremented and marked as inactive if it reaches 0.

If after a certain delay the query has not been fulfilled, the requesting node will increase the sequence number assigned to the query and issue the query again. This will basically refresh the query state on the network. This refreshed query can potentially have a lower count.

Periodically, nodes broadcast hello messages. These messages are used to maintain the neighbor table. When a node hears from a new node, it adds the node as an active neighbor. If a node does not hear from a neighbor for a period of time, the node is marked as remote. If such a node returns to the neighborhood, it is again marked as active. When a node is marked as active, it triggers a new-neighbor event. When a new-neighbor event occurs, a node exchanges active query information with the new neighbor. This effectively spreads queries that have not been answered yet.

If a node marked as inactive is a previous hop for a query, the query is marked as disconnected. Queries marked as disconnected spread differently that connected queries and react to update rules differently. Disconnected nodes cannot update connected nodes.

Pseudo-code for some of the event handling functions is presented in figure 2. These functions get called when processing queries and objects.

2.4 Content

We call a named piece of content an object. The content inside a normal data object is a blob of bits that is not interpreted by the network in any way. Objects are characterized by multiple attributes, including a name, a publisher, a timestamp, a type, etc. We call this set of attributes the *absolute name*. Our current design is simple and does not address arbitrary attributes, tags or associations. Such extensions is the subject of future work.

The name and publisher fields are the basis for matches and queries. If we were to map (in a very simple way) URLs to them, the hostname would be the publisher and the path section would be the name. On a more complete system, the publisher would actually be a security association between the entity doing

the publishing and the mapping between the *absolute name* and the data (using signatures and encryption if necessary).

Field	Definition
Name	Object name
Publisher	Object publisher
Pub Dist	Publisher distance
Type	Type of object
Size	Object size
Time Stamp	Object creation time
Version	Object version
Lifetime	How long the object will live in the network
State	Local object state (local)
Last Hop	Who sent the object/announcement (local)

Table 1: Object Table Entry

Information about objects is kept in a table at every node. Table 1 shows the structure of the table entries. Information about an object is entered into the table when the object or an object announcement is received . The fields match directly to the *absolute name* and expanded with local information (labeled in the table) about the state of the object and who sent us the information.

Currently DOT uses PUBLISHED, CACHED and NAME ONLY as object states. A PUBLISHED object was generated locally and is not deleted until an explicit delete request is received. A CACHED object is one that is flowing through the network and has been cached locally. A NAME ONLY object is one we have heard of but don't have they payload for.

2.5 Queries

When a program wants to retrieve an object, it issues a Get command. This translates directly to a network query. Matching a query to an object is basically a comparison between the query fields and the *absolute names*. A query contains very similar fields to an *absolute name* (name, publisher, etc). The name and publisher can be encoded to do prefix or exact match. This simple addition adds a great deal of flexibility to queries and name structures.

Queries are kept in a query table. Table 2 describes the field format. The table entries are composed of the fields of the query plus local information (labeled in

Field	Definition
Name	Object name being queried
Publisher	Object publisher being queried
Type	Object Type being queried
Version	Minimum object version
Order	Order number
Mode	Query Mode (Phase)
Count	Query Count
State	Query State
Time Stamp	Query time stamp
Source	Query originator
TTL	Hops to travel
Last Hop	Who forwarded this query to us (local)

Table 2: Query Table Entry

the table). Each entry contains information about the object for which a query is issued. Name and Publisher can contain wildcards for prefix matching. Type can be a specific object type or a wildcard. The Time Stamp defines when this query was originated and is used for comparisons, it works as a query version. The Source is used to know who originated the query.

Count is used by the query to request a certain number of replies. If the requesting node wants just 1 reply it can set this to 1. But if it wants multiple answers it can use a higher number. This can be used for flowcontrol. Mode stores the protocol phase, whether we are going to the publisher, proxy or doing a flood. State tells us if this query is connected, disconnected or inactive. Finally, Last Hop is used to calculate who the previous hop in the tree is. If we match this query we will try to contact this node with the object.

2.6 Neighbors, Transmissions and Forwarding

Neighbor information is kept in the neighbor table. This table keeps track of the last time we talked to neighbors. It is used to maintain the connected state of the queries.

DOT also keeps a cache of the transmissions it has made or received. In the case of sent date it is used to suppress unnecessary re-transmissions of queries and objects. Query sending is throttled to avoid overhead. Before an object is sent as

a reply we check that we haven't sent it (successfully) to that neighbor before. If that is the case we will look for another match.

For objects received, we keep track of the last objects received per query. This is used to populate the forwarding table. The route to a publisher is inferred from received objects using the last hop.

```
function process-query(q)
  add/update q to local query table
  reply to previous hop up to q->count new objects
  if q has not been fulfilled
    forward query according to q->mode

function handle-query(q)
  if(q is new)
    process-query(q)
    return
  lq = local version of query
  if lq is inactive and q is not, send silence
  if lq version is higher, discard q
  if lq is older, update and process-query(q)
  if q is a silence, set lq to inactive
  if lq has been fulfilled
    if we are connected
      if q is disconnected
        return
      if q is not lower order
        return
    process-query(q)
    return
  if (q is from our last hop
      or q has lower order)
    process-query(q)

function handle-object(object)
  for every query that matches object
    if query has been fulfilled
      return
    if query is local
      deliver local
    else
      if previous hop for query is an active neighbor
        send object to previous hop
    decrease query->count
    if query->count is 0
      set query to inactive
      (query has been fulfilled)
```

Figure 2: Simplified pseudo code

3 Simulation

3.1 Evaluation

To evaluate DOT we decide to compare it to Epidemic routing. It is not easy to compare a content-based routing algorithm to a traditional routing algorithm. Even in the case of late-binding endpoint based routing protocols like Prophet [**?**] we would need to design an algorithm to provide the same semantics and this would not make the comparison useful. We hope that by using epidemic as a baseline we can normalize comparisons. In the future we do plan to explore direct comparisons between algorithms.

We simulated DOT under Qualnet 4.5. We implemented the DOT routing algorithm as well as an event system for object publishing and querying based on names. We selected a 2Mbps 802.11b channel. All data objects used are 10000 bytes. All simulations are run for 5 random seeds.

3.2 DOT

DOT is using the default setup parameters. It tries source, proxy, scoped flood and finally full flood. For these simulations we didn't use any proxies, so that phase is effectively skipped by the protocol. When a query is received the source (publisher) is tried first. If there is no route or if there is no reply in 2 seconds the protocol moves to the next phase. Scope flooding spreads for 6 hops and times out in 6 seconds. Unless otherwise specified all queries are for a count of 1. Even though UDP is used for neighbor discovery, the data objects are transmitted from neighbor to neighbor using TCP. If multiple neighbors exist they will get the object via multiple TCP connections.

3.3 Epidemic

Epidemic routing uses the same infrastructure used for DOT but with epidemic rules. Basically queries are flooded everywhere and any match triggers a reply. There is no notion of connectedness or transmission caching. Just like DOT, epidemic uses TCP to transmit the object and UDP for neighbor discovery.

3.4 Scenarios

We evaluated the performance of DOT versus epidemic using 2 scenarios.

The first is a simple grid type scenario which can be seen in figure 3. In this scenario we divide the area into an 8 by 8 grid and place a node on each grid square for a total of 64 nodes. Each node is then given random direction mobility inside the square. A node selects a random direction and a random (valid) point in that direction. It then travels towards that location at a random speed in a range of 1-5m/s. The node then pauses for a short period (random 0-10s) and chooses a new direction. This scenario was chosen to focus our evaluation on one of the core functions of protocols for DTN environments, namely, how they perform when the network is not connected. As nodes move in a random direction model inside their grid squares, this creates periods of disconnection among any two nodes placed in neighboring squares.

We evaluated the performance for a variety of loads and grid square sizes. By varying the area covered by our grid the nodes become more or less likely to be disconnected. At the low end, a size of 200m (that is 200m x 200m) keeps the network connected. At the opposite end, 600m, nodes are disconnected most of the time.

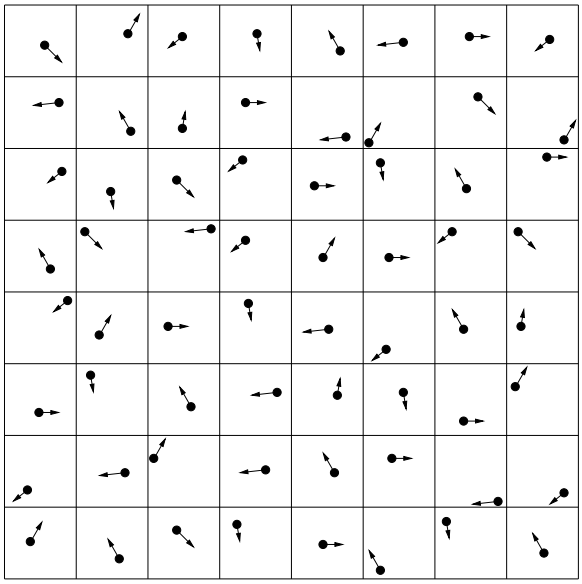


Figure 3: Grid mobility

For every load we choose 5 random nodes to act as publishers. In the first 30 seconds objects are created and placed at a random publisher. We vary the objects created from 5 to 100. We then generate 100 queries and distribute those amongst

the nodes randomly. Queries are randomly spread over the next 20 minutes. Each query is for a random object. A node will not query for an object if it is the publisher or if it already has a query for that object. By varying the number of objects we change the amount of duplication. If only 5 objects are published then lots of nodes will request the same object. If 100 objects are published then many objects will be requested only once. The scenario is run for 30 minutes, allowing queries to be fulfilled.

Our second scenario tries to create a more realistic situation. Figure 4 depicts the scenario. We have an area of 3000x3000m. In the center we have a static base with 10 nodes randomly spread within 100m of each other. We then have 2 groups of 5 nodes patrolling opposite halves of the area. The patrols have internal group mobility and follow 4 waypoints. They move at a speed of 2m/s (fast walking).

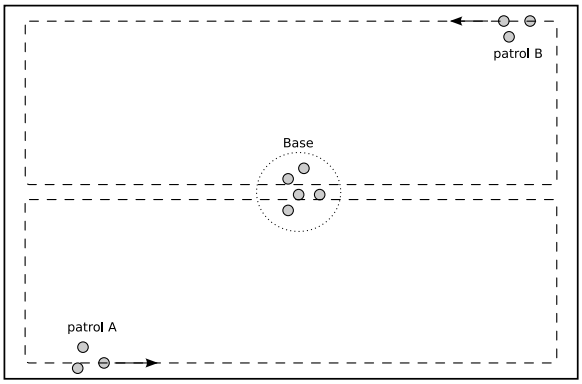


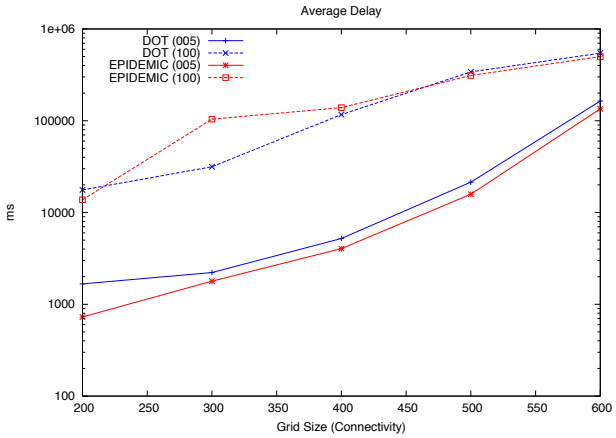
Figure 4: Patrol mobility

We have 3 configurations for the patrol scenario. The first is data gathering, where the patrols generate data. One of the base nodes has an active query that matches all published data by the patrols. We vary the number of objects published from 10 to 100. Each object is published at a random patrol node between 30 and 5000 seconds. Our second load scenario is data distribution. The base nodes will publish 10 to 100 data objects. The patrolling nodes will generate 100 requests between 30 and 5000 seconds for a random object. The requests will be spread randomly amongst the patrol nodes. The lower the number of published objects, the higher the duplication. Finally, we have a mixed load scenario. This is basically adding the load of the two previous scenarios. So for 10 objects published in the mix scenario, 10 objects are published by the base and 10 by the patrols.

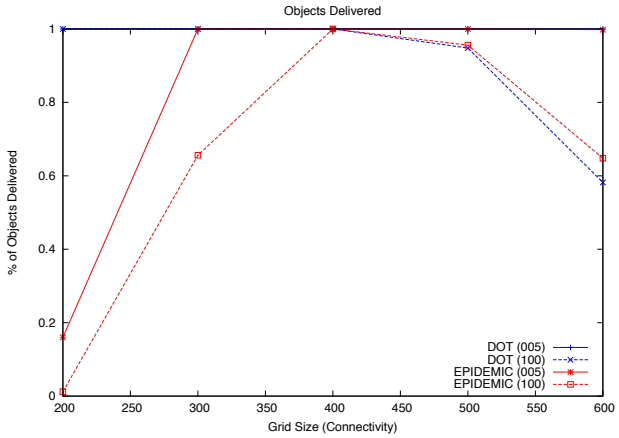
The patrol scenario runs for 200 minutes. At 2m/s a patrol will take 60 minutes to complete a round. Each patrol starts a opposite corners and they will meet in the middle at the base.

3.5 Results

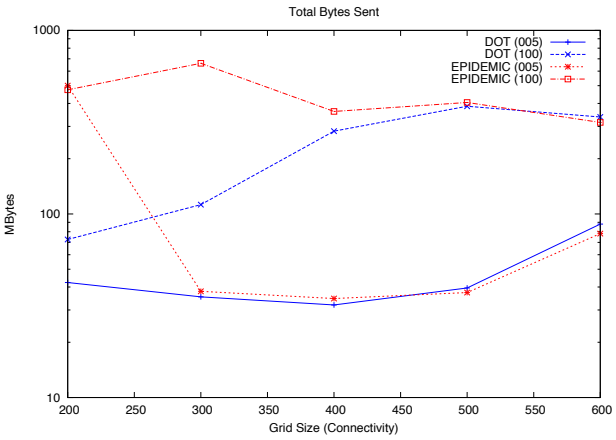
Our simulations generated a large amount of data that we have summarized in the following graphs. We present the most relevant scenarios.



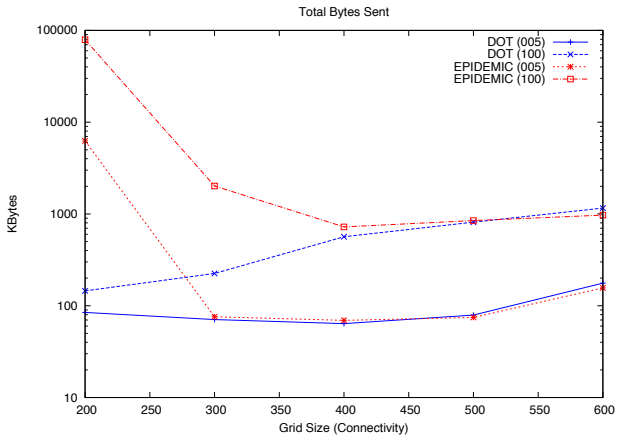
a) Average delay



b) Average objects delivered



c) Total bytes transmitted (by IP)



d) Total bytes transmitted per object (by IP)

Figure 5: Results by grid square size

Figure 5 has the results for the synthetic grid scenario. Two sets of results

are graphed. High duplication is represented by the results when there are only 5 objects in the network (005). Low duplication is represented by results when there are possible objects in the network (100). Remember that there is always the same amount of demand. So, if there are only 5 choices then repetitions will happen very frequently. Conversely, with 100 possible objects in the network it is not common for an object to be requested twice.

Looking at delay, we see that both algorithms perform very similarly. For the low duplication experiments (100) the delay is much higher. This is to be expected since it's not usual for a node to find a nearby cached object if there is no duplication. The other trend we see is that as the system gets more disconnected (grid size increases) the delay increases. This is expected too since nodes will have to wait longer for an object to make it's way to the requesting node as they travel one hop at a time and wait then wait for the node to move. The final thing to note is that when duplication is high, epidemic has lower delay than DOT. This is due to the fact that epidemic is optimized for high duplication scenarios where everybody wants the same object. Epidemic effectively pre-fetches the object for many nodes.

The remaining 3 graphs show something very interesting. Epidemic performs very badly for the high connectivity scenario. It delivers very few objects and uses a lot of bandwidth to do it. The reason for this is partially due to the MAC layer. When connectivity is high, the traffic demand by epidemic makes the network collapse. We are using an 802.11b MAC and like any contention based network it crashes on heavy load.

In summary, DOT performs as well as epidemic for disconnected scenarios and better than epidemic for highly connected scenarios where epidemic's traffic makes the network collapse.

Figure 6 has the results for the Patrol scenario. There are 3 loads used, gather (base requests from patrols), distribute (base distributes to patrol) and mixed (sum of gather and distribute query loads).

The first thing to notice in this delay graph is the scale. DOT is taking about 1000 seconds to deliver an object. The reason it's taking so long is the scenario itself. In the patrol scenario objects can be generated when the patrol is not in communication range. This means the object has to wait until the patrol returns to base to deliver the object. The same applies to both DOT and epidemic.

DOT delivers data very evenly, epidemic has some trouble. The main reason for the trouble is that when the patrol is in range of the base every node is trying to epidemically send data to every other node. This creates too much load for the wireless network to handle and some objects don't make it through. The graph

measures average delay of the actual objects received, it does not take into account undelivered objects. This gives the appearance that delay decreases as the load increases for the epidemic scenarios involving data distribution.

The remaining 3 graphs for objects delivered, total data and data per object show how epidemic has a very hard time delivering objects. DOT is able to deliver all objects while epidemic falls to 60% in the best case scenario, with distribution falling under 15%. In terms of data per object this makes epidemic have 100 times more overhead than DOT.

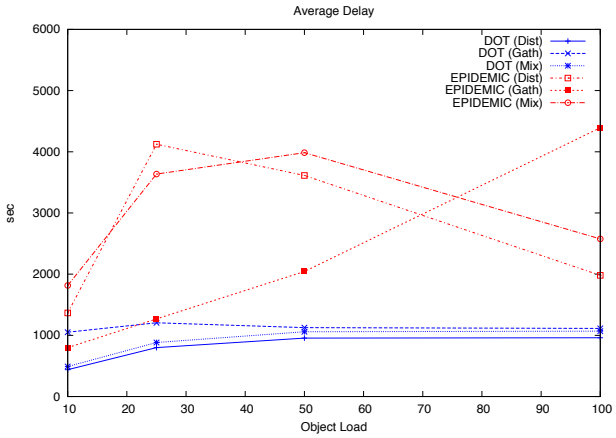
To get a complete picture of what is happening in delay and arrivals we plot the data showing the cumulative arrival by delay. Figure 6 has the results for the grid experiment at a resolution of 300 and the patrol experiment for the mixed data scenario. On the vertical axis we have the cumulative arrival, that is, the percentage of objects received. On the horizontal axis we have the delay an object took to get to the requesting node. For example in the grid scenario using DOT 84% of the objects arrived with a delay of 1000 ms or less.

For the grid experiment we see that for the high duplication scenario (005) epidemic received 90% of the objects with a delay of 100ms or less. This indicates that most of these objects were already cached. Arrivals with 100ms delay or less are highly likely cached objects. For the same situation DOT had only received 66%. DOT eventually catches up to epidemic but in this high duplication scenario epidemic is better.

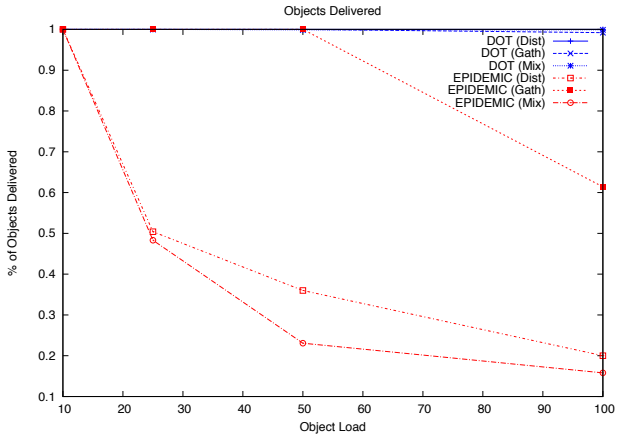
For the low duplication scenario (100) epidemic starts with 20% of the objects already cached or in neighboring nodes. DOT starts with 7% of the objects in that situation but quickly catches up, eventually delivering all objects.

For the patrol scenario we have a more complex graph. In the case of epidemic, when the load is low (and duplication high) it performs relatively well, caching a number of objects. As the load increases performance tanks and almost nothing is delivered.

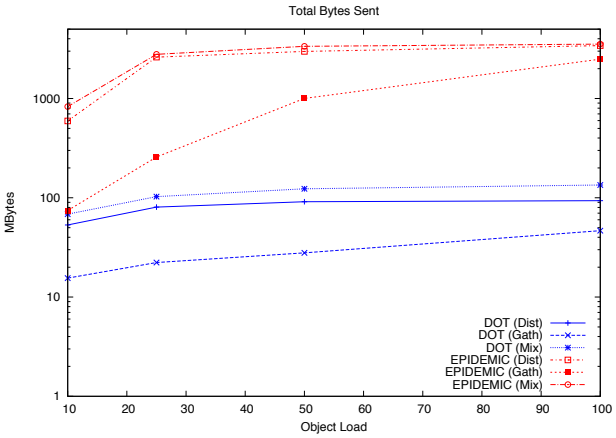
For DOT we have a stepped graph. The initial jump of arrivals from 100ms to 1000ms are objects retrieved from the local neighborhood, mostly the other patrol nodes. There is a big jump starting at 1000 seconds, this correlates to the average time it takes for a patrol to make a round and come back to base. Basically, things get cached until we are in communication range and then objects are delivered. DOT is able to deliver all the objects at all loads, epidemic on the other hand can only deliver all objects when the load is low.



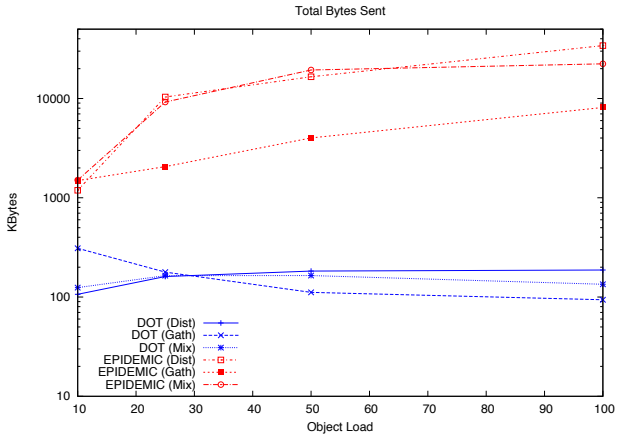
a) Average delay



b) Average objects delivered

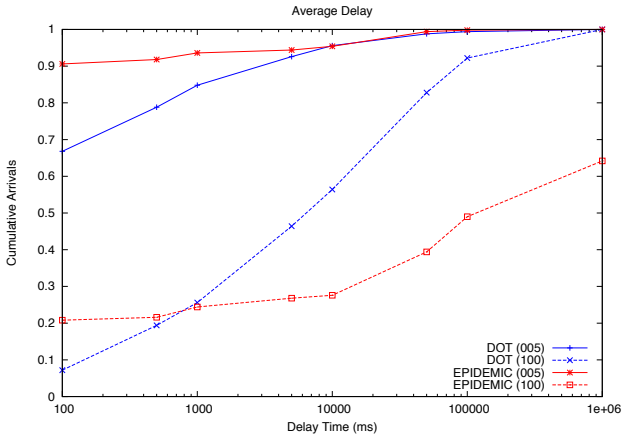


c) Total bytes transmitted (by IP)

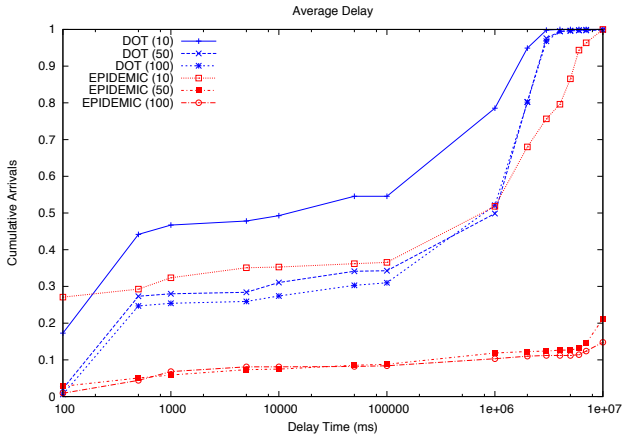


d) Total bytes transmitted per object (by IP)

Figure 6: Results by Query Load



a) 300 Grid



b) Mix traffic Patrol

Figure 7: Cumulative Arrival by Delay

4 Related Work

The IRTF's *DTNRG* (Delay Tolerant Networking Research Group) [10] introduced the *bundle* architecture [12] which groups messages into bundles that encompass entire sessions, performs *store-carry-and-forward* of bundles, and employs *custody* [13] transfer for reliability. The DTNRG also designed addressing and naming schemes [5] for DTNs.

The routing schemes that have been proposed for DTNs thus far have focused on establishing spatio-temporal routes to nodes or groups of nodes, given that connectivity information (links or contacts) is enforced, scheduled, or random.

Routing schemes based on enforced contacts typically employ specialized nodes such as robots with controlled mobility. Such specialized nodes (called mules and message ferries in many schemes) [25, 31, 17, 32, 19] are such that their mobility can be controlled to provide connectivity to other nodes in networks as needed. Much of the work has focused on route scheduling of the specialized nodes and synchronization between their routes. It has also been shown that such data mules or ferries can be used as an energy saving device for other nodes in the network; if there are no ferries nearby, nodes can be turned off to conserve energy.

Several approaches [21] and [16] take advantage of the periodicity inherent to some mobility patterns and assume global knowledge of node schedules. They use variants of *space-time* routing tables, and employ, among other methods, a modified Dijkstra's algorithm to determine shortest paths over time in these structures. Some other approaches address the case of predictable or scheduled mobility (e.g., buses and trains). Some approaches [29, 7] rely on past mobility and topology knowledge to predict future behavior. *MaxProp* [2], showed better performance in a deployed network of buses than an oracle with perfect schedule knowledge. The approach described in [29] tries to predict the future topology of a network by determining *how long* nodes that are connected will remain connected.

Location information has been shown to be very useful for routing in disrupted networks (e.g., MobySpace [18] and MV routing [4]). These methods require an external localization mechanism, such as GPS, and assume that a node who has visited a particular location is likely to revisit it, and therefore is a good candidate to carry messages to that location.

Some approaches have also proposed the deployment of static nodes to increase contact opportunities in DTNs. This is the case of *Throwboxes* [33] which are stationary nodes that typically have greater wireless communication, storage, and power capabilities. *Throwboxes* act as static relays that can receive and forward messages as nodes come in range. If node schedules are known or can be

predicted, these static relays can be placed such as to optimize contact opportunities.

When connectivity information cannot be enforced or predicted, routing is done opportunistically. The simplest scheme is *epidemic routing* [30], with which a copy of a message is forwarded to all nodes encountered by the message. The limitation of epidemic routing is that its transmission requirements can be prohibitively expensive. As a result, several schemes [14, 20, 26] have been proposed to improve on epidemic routing by controlling the way in which message flooding occurs. In *spray and wait* [27], only the source can replicate a message; after 'spraying' several copies of a message, the host 'waits' until one is delivered. In *Spray and Focus* [28], after the source "sprays" a message similarly to spray and wait, each copy is forwarded according to a utility-based function aimed at finding better opportunities to relay messages. The CAR [22] algorithm uses adaptive weights on several node attributes as well as Kalman filters to maximize the probability of a node to deliver messages to a destination. Work on *pocket switched* networking [15, 6] models the distribution of contact and inter-contact times in order to better design forwarding strategies, and a number of mechanisms are proposed to extend opportunistic forwarding protocols.

Several schemes have been proposed to extend routing algorithms originally designed for mobile ad hoc networks (MANET), in particular on-demand routing, to operate in scenarios in which end-to-end connectivity does not exist. Disconnected transitive communication [7] is a proposal to enable communication across clusters in MANETs using utility values to decide which node in the cluster is best suited to transmit a message to the destination. The goal of the efficient route discovery mechanism proposed by Dubois-Ferriere et al. [11] is to decrease the amount of route discovery overhead by forwarding route requests in the direction of the destination. The proposal by Ott et al. [23] augments AODV with the ability to identify DTN-capable routers. The source then decides whether to use AODV– (if available) or DTN routes. An underlying DTN routing mechanism is assumed to be in place. The Space-Content-adaptive-Time Routing (SCaTR) protocol [1] also extends AODV to accommodate partitions; however, SCaTR does not assume the existence of an underlying DTN routing fabric nor does it leave the routing decisions to the sources.

In *Island Hopping* [9], nodes run a distributed algorithm to find a set of *concentration points* (CPs), which are the set of connected subgraphs, or "connectivity islands"; and a set of edges representing possible node movements between CPs. Nodes learn the entire graph by gossiping. Given that edges are based on prior node movement, a certain degree of message replication is employed to en-

sure that at least one is delivered to the final destination.

From the above discussion of prior work on routing for DTNs, it appears that the approaches proposed so far have made one of three assumptions. One assumption is global topology knowledge, which has proved to be useful in determining future connectivity. Another assumption is the existence of controllable nodes, which can significantly aid delivery in sparse networks. The third assumption is that data can be duplicated freely among nodes, which leads to increased. Our work is an attempt to do away with these assumptions, yielding a more general framework applicable to any network scenario where any one of these assumptions may be unrealistic.

Prior approaches for routing in disrupted environments do not route named content objects directly. Our approach, DOT, is inspired in Directed Diffusion and consists of routing named objects based on interest statements that percolate across connected components of a disrupted network, so that some nodes can choose to copy and carry the objects of interest back to the nodes that originated the interest in the objects.

5 Prototype

5.1 Overview

As part of this program we implemented part of our Content Centric Network (CCN) infrastructure on top of the DTNRG router code. With this, we are able to build CCN Applications. We call this prototype the CCN-DTN Architecture.

CCN applications transfer data in terms of Content Objects. These content objects are given to the network to publish. Applications that want to get the content then query the network to get these objects. There needs to be a way for applications to identify content, this is done via Content Names.

5.2 Functions

The CCN-DTN architecture provides 2 functions to applications:

- *Publish*

  An application gives data to the CCN-DTN. This data is treated like a content object and passed as a bundle. To identify the data we assign a name to it. This name is passed as part of the Endpoint ID.

- *Subscribe*

  To request data, an application needs to tell the CCN-DTN that it wants to subscribe to a specific Content Name. This is done by registering an Endpoint ID which the CCN-DTN will interpret as a subscription request. The name of the data is embedded in the Endpoint ID. We allow wildcards so that a whole range of content objects can be returned to a single subscription.

5.3 Naming

For the naming of objects and subscriptions (queries) we decided to overload the Endpoint IDs. We took the base URI format and embedded the name of the object.

Content Object names are treated like a basic string. For clarity purposes we use slashes (/) as separators, but this is not required. In this way we can have a name that looks like this:

```
/maps/cities/newyork
```

Using slashes as separators is intuitive since it's the same way we deal with files and web pages (URIs). And given that the DTN code has some primitives for decoding URIs it makes sense.

In fact, we use URIs because that is what the DTNRG code uses. It also allows us to naturally include the identity of the sender, or, as we call it, the publisher. The publisher does not need to be the node name specifically, it can be an arbitrary role or string. Due to the encoding in the URI it has to meet the same requirements as the host portion of the URI. What we end up seeing are publishers that look like fully qualified host names. An example publisher could be:

```
army.convoy-alpha.scout.camera
```

When data needs to be published, we need to send this information to the CCN-DTN system. We construct the destination of the bundle by joining the destination of the CCN Router with the name. The CCN Router destination should be a known local Endpoint ID. We are using the standard destination for external routers as the CCN Router destination and then adding a function identifier. The result is an Endpoint ID that looks like:

```
dtn://[localhost]/ext-rtr/publish/p=/my.publisher.com
/ot=/0/n=/maps/cities/newyork
```

[localhost] will be replaced by the local DTNRG address. This can be easily obtained from the DTNRG API. */ext-rtr/* is the External Router Endpoint Identifier and *publish* is the function we are requesting. We then add a couple of parameters. The identity of the publisher is added after the */p=/*, in this case *my.publisher.com*. The type of object is added as */ot=/*, in this case, 0. Finally we concatenate the name after the */n=/* marker. For this example, /maps/cities/newyork.

To establish a subscription, we need to tell the CCN-DTN system what we are looking for. This is done via a registration. Once again, we are overloading the Endpoint IDs. The method we currently use is to register at a specific ID. This tells the CCN Router what you are interested in receiving. In a similar fashion to publishing we use a well known endpoint prefix, in this case *direct/get*. So if we had an application interested in receiving all the city maps we would register the following Endpoint ID:

```
dtn://[localhost]/direct/get/c=/1/ot=/0/qt=/2
/p=/my.publisher.com/n=/maps/cities/*
```

Like in the example before, we overload the Endpoint Identifier to pass parameters to the CCN-DTN router. */c=/* specifies the count, or how many objects to

retrieve with this query. /*ot*=/ specifies the object count. /*qt*=/ specifies the query typer, /*p*=/ the publisher and finally /*n*=/ the name.

5.4 Architecture

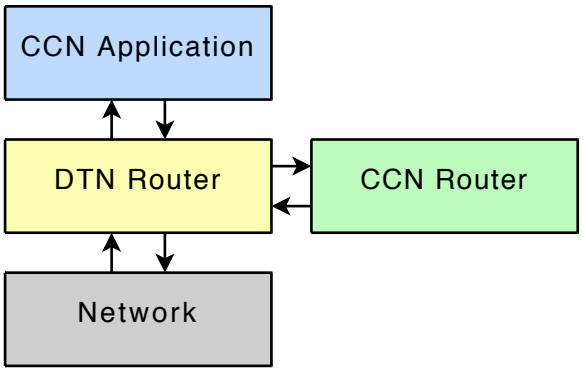


Figure 8: Overall architecture

The main goal for this project was to add Content Centric capabilities to the DTNRG infrastructure. For this purpose we chose to pursue an architecture that was unobtrusive. The final design can be seen in figure 8.

CCN applications will use the normal DTN API to communicate to the CCN Router. There are no API modifications necessary and no extensions are used. This basic approach allowed us to do some fast prototyping and application development. We decided to use the Endpoint Identifiers as a way to communicate to the CCN Router what content was being published or what content was being queried for.

Content routing will happen at the CCN Router level. When content or queries move from node to node all the communication is done at the CCN Router level. For architecture purposes the actual data sending and receiving from the CCN Router to the core of the DTN Router uses the same functionality an application would use. Even though the BBN External Router plug-in interface is used, the CCN Router registers like any other application. This means that to send data from a node to a neighbor the DTN router treats it as a normal bundle send with a destination of a one hop neighbor. Logistically, this means that data has to cross the whole stack. From the CCN Router down to the DTN Router then to the network, and back up at the other end. Figure 9 have a simple depiction of this.

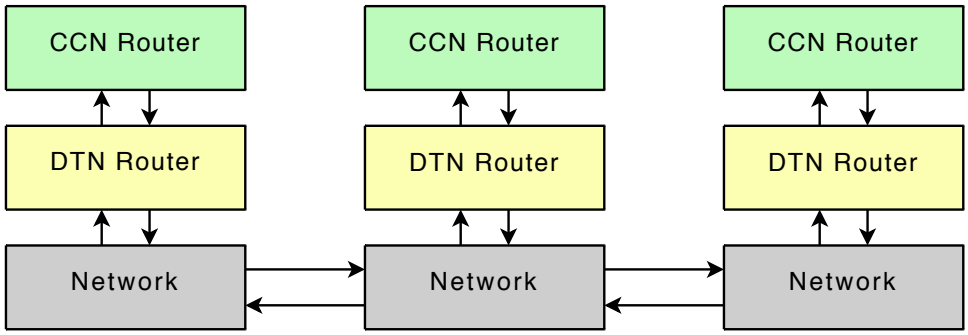


Figure 9: Hop by hop CCN Router Communication

When a query is created and it needs to be propagated throughout the network. The CCN Router will exchange query information with the peer CCN Routers. When data is published or data matches a query, the CCN Routers will exchange information on a hop by hop basis. All intermediary CCN Routers will process queries and content objects.

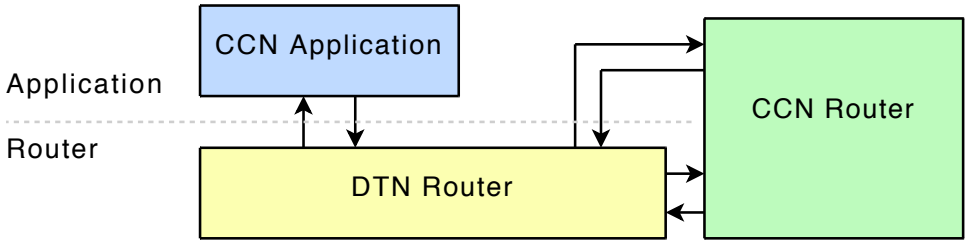


Figure 10: Internal logical architecture view

The internal architecture used is shown in figure 10. Content Centric applications communicate to the DTN Router using the DTN API. The concept of communication from one application to another does not exist at this level. The CCN Router will register as the router module, so it will get all the routing requests, link events and registrations. At the same time, the it registers as an application endpoint. On a more advanced system this will allow the CCN Router to co-exist with the standard DTN routing algorithms and deal exclusively with the Content Centric traffic.

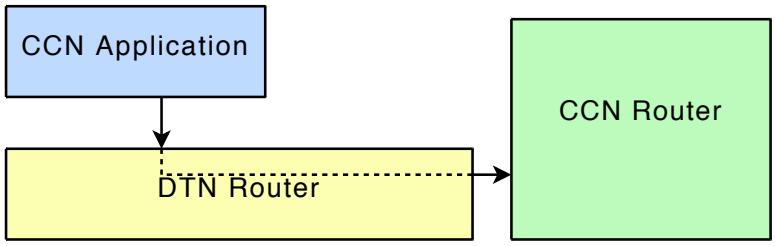


Figure 11: Application Registration / Query

When an application wants to query for a piece of data, it registers at the DTN level. It uses a specifically named Endpoint ID that implies a CCN registration. Given that all registrations are sent to the router module, the CCN Router receives this registration. It stores it, parses it and converts it to a content query. Figure 11 shows the data flow for a registration. The CCN Router will keep a table of local registrations. When it receives content objects it will check to see if they match any of the outstanding queries and if so, delivers the object to the application.

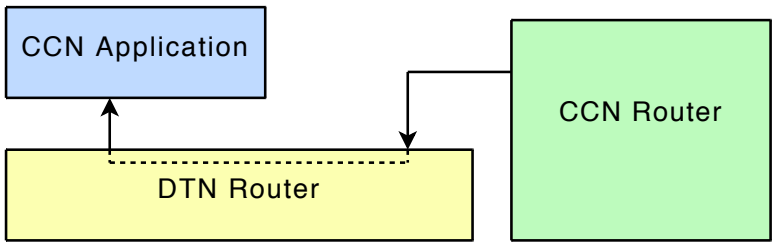


Figure 12: Data delivery

A registration can be active for a extended period of time. Over this time multiple objects can be received. The CCN Router will deliver those objects to the application using the standard DTN API. Shown in figure 12 is the data path for delivery data to the applications. This requires no extra modifications to the DTN Router since it's a simple bundle delivery. The source Endpoint ID of the bundle is constructed to give the application extra information about this Content

Object.

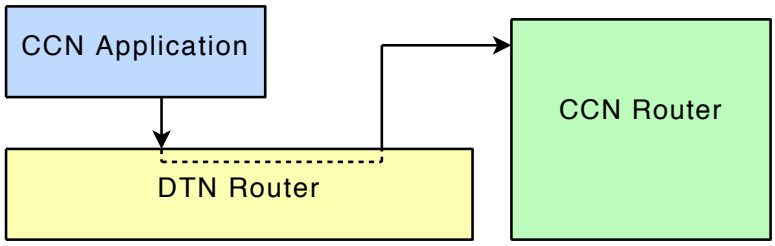


Figure 13: Content Publishing

When an application wants to publish a content object it sends it to the local CCN Router. To do this it registers an Endpoint ID that will function as the source of the data, and sends the data to the local CCN Router Endpoint ID. This destination is formated such that the CCN Router will understand what the content is named. Figure 13 illustrates how the communication takes place. Once the Content Object is sent the application does not need to worry about it any more. The CCN Router will store it. This also means that the DTN Router does not need to store the bundles, since the DTN Router is the destination and the bundle has been delivered. With this approach, storage is controlled by the CCN Router and can employ advanced content replacement techniques.

The reason why we developed the CCN Router as a partial DTN Application was to overcome the limitations we had as a DTN External Router plug-in. The flexibility provided by plug-ins comes at a price, namely, communication overhead.

6 Implementation Notes

6.1 Naming

Throughout the course of the DTN program, the RIDE project has gone over various names. These are names that were given to different protocols and systems and were not reconciled. Some of the names have stayed in the codebase and as such make appearances in this report. Most of the time they can be treated interchangeably. In this section we'll try to go over the different names for clarity.

RIDE - This was the original name of the project and is used mostly in titles. It stands for Robust Internetworking in Disruptive Environments.

PARC DTN - This name was used in the code to refer to the work done in Qualnet. It was mostly used in Phase II and should not be present in the current codebase. This was discontinued due to the general nature of the name.

CCN - Content-Centric Network. We used this name in Phase II to refer to some of the components used for the Qualnet simulation as well as the prototype. This name was discontinued due to a conflict with an internal PARC project working on Content-Centric networks that was not related to the DTN program. The name is no longer used in the code but some references remain in the prototype documentation.

DIRECT - DIsruption REsilient Content Transport. This was the final name give to the algorithm produced for Phase II. It is a simple content routing algorithm. It is the name most commonly seen in the code, both for prototyping and for simulation.

DOT - Dual-ordered Object Transport. This is the name of the current protocol, the one worked on for Phase III. It was originally named DIRECT2 but we decided against this because it could create confusion and adopted the new more meaningful name. DOT is not used in the code but it's used in this report. In most places where we refer to DIRECT you can assume this means DOT.

6.2 Portability

Working on Phase II of the DTN project taught us many things about implementing protocols and prototypes DTN systems. This had a great effect on design decisions for the current phase.

We had decided to port DIRECT to the DTN implementation by BBN for the WNaN program. For this we had to create a portable framework for our protocol that would make it work under different scenarios.

All of the core code of DIRECT is written to be portable and use a system abstraction layer. Under this framework we were able to link our code with Qualnet for simulations, with DTN2 to create an external router and with BBNs BPA (under BBN's PLF system) as a content router.

The modifications required to port DIRECT to a platform are small. A helpers.c file is in charge of providing system dependant functions like malloc, send, etc. Another file is needed to translate from the system specific calls to the DIRECT calls. This file is normally called direct_¡architecture¿.c, for example direct_qualnet.c.

7 Simulation Implementation

To evaluate the performance of some of our Content Centric work we conducted simulations in Qualnet 4.5 [24]. This section explains in a little bit more detail the architecture and code used in these simulations. For exact specifications please refer to the actual code.

The qualnet code produced by this project is distributed as a patch file to Qualnet 4.5. To use it you'll need to patch and recompile Qualnet. From now on when we refer to qualnet we will be referring specifically to Qualnet 4.5 with the PARC DIRECT patches.

7.1 Code structure

Most of the changes to the qualnet code take place in the form of a new application, DIRECT. Users can now enable DIRECT in the qualnet simulation and then tell qualnet to simulate the publishing of objects and the request of objects. The code is concentrated in *contributed/direct*. In other places the modifications are marked by *ADDON_DIRECT*.

The file *main/application.cpp* was modified for DIRECT. The *Init* and *Finalize* functions are called from here at the beginning and end of the simulation respectively. The demultiplexor for applications also calls the main *Layer* function when the message is of type *APP_DIRECT*. This file is also in charge of implementing the read mechanism for the DIRECT events, the publish and subscribe.

7.2 Qualnet DIRECT architecture

Qualnet is a discrete event driven simulator. Events are implemented on top of a message passing infrastructure. Way when an event or a message is generated, the dispatcher will send it to the relevant module. Every module needs to have a main function in charge of handling events and messages. Events have types to indicate what kind of event or message it is. We will now describe this function as well as the other important functions:

- `AppDirectProcessEvent()`

  This is the main event/message function. It is called when a message is received. This message is an internal simulator message and does not necessarily correspond to a "network" message. The main purpose of the function is to act as a demultiplexor for messages and call the correct handler.

- `AppDirectInit()`

  Initialize the DIRECT system. Create the node structure and set debug levels. It registers the APP with qualnet. It calls the architecture independent directInit().

- `AppDirectFinalize()`

  This function is called by Qualnet at the end of the simulation to terminate the DIRECT subsystem. This function tells us we need to collect all data and output statistics for analysis.

- `AppDirectParseEvent()`

  This function gets called when Qualnet is parsing the event files (app files). Whenever a DIRECT event is found it gets called. We then parse the event line and internally create events. Events can be data queries and publishes.

7.3 Simulation Setup

The setup of a DIRECT simulation happens just like any other qualnet simulation. There is a main config file, which tells qualnet what parameters to use. In the case of DIRECT the main parameter that needs to be set is:

- `DIRECT-DEBUG n` Set a debug level. Where *n* is a value from 0 to 6. With 0 being no debugging and 6 being max debugging.
- `DIRECT-SIMPLE-FLOOD n` Tells DIRECT how many hops to use for simple flood.
- `DIRECT-SCOPE-FLOOD n` Tells DIRECT how many hops to use for scoped flood.
- `DIRECT-FULL-FLOOD n` Tells DIRECT how many hops to use for full flood.
- `DIRECT-STATUS-LOGGING n` Tells DIRECT if we want to do status logging.

DIRECT uses the application config file (.app file) to read the simulation setup from the user. That is, publishes and gets.

The format for a publish request is:

`DIRECT <node> PUBLISH <time> <oname> <pname> <size> <type>`

The format for a get request is:

`DIRECT <node> GET <time> <oname> <pname> <num> <otype> <qtype>`

where: *oname* is the object name, *pname* is the publisher name, *num* is the number of objects we want for this query, *otype* is the type of the object and *qtype* is the type of the query. Please refer to the code distribution for example files.

8 Prototype Implementation

8.1 Architecture

Most of our architecture implementation decisions were constrained by the system we were working under. In this case, the DTNRG codebase. There were a few specific requirements that we wanted to meet. We wanted to make our code work with the plug-in architecture created by BBN,t his way we could modify and compile our code without any modification the the core DTNRG code. There is some benefit in having the CCN Router code be developed as an internal module to DTNRG, for example, access to all the storage functions as well as events and data structures.

To balance these two approaches we decided to make a plug-in, in such a way that the code developed could be shared between the plug-in and the internal system. What we ended up doing was cloning the DTNRG code, stripping unneeded code and adding an endpoint for the plug-in connection. This is what we called the ERD or External Routing Daemon. Figure 14 shows an overview of the system using the ERD.

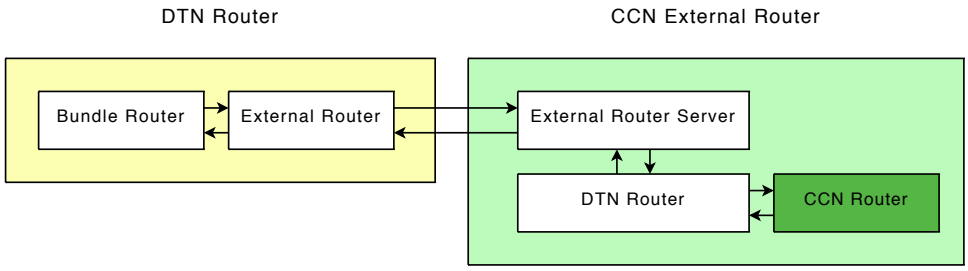


Figure 14: Overall architecture

Alternatively, you could use the code to compile the CCN Router inside the normal DTNRG code. This approach would look more like figure 15. In this situation the modifications needed for the CCN Router (or any other router) to work are minimal if any.

8.2 DTNRG

From the start we decided to create our prototype in such a way that the changes required on the DTNRG code were minimized. The main idea was that as a plug-

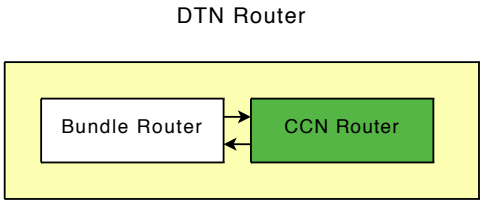


Figure 15: Overall architecture

in we could work without any changes to the standard codebase.

Given the dynamic nature of open source development, the active DTN community and the other program participants it was a challenge to keep our code in a working condition while maintaining compatibility with the SVN head branch. Because of this we decided to work with only one version of the DTNRG code, namely DTN2 2.4. This code already incorporated the changes made by BBN to add a plug-in infrastructure. We haven't tested our code with different releases and can't assure that everything will work correctly. As part of the code deliverable we are including the code which we used as the base for the DTN2 daemon. Changes to what you can get out of the official SVN repository for DTN2 2.4 should only be in the build environment configuration and not any actual functionality.

To communicate with an external router plug-in the DTN2 daemon uses an External Router module. One of the functions of this module is to register an endpoint that can be used to communicate with the external router as an application. We uses this endpoint to communicate with the CCN router.

8.3 External Routing Daemon - ERD

The External Routing Daemon works as a plug-in to the standard DTN2 daemon. It's purpose is to act as a base for building routing plug-ins that work both inside and outside the standard DTN2 code.

Since we wanted the plug-ins to work on both situations, we had to create an environment where the code would have the same functions and features available to it. Basically, a shell around the actual router code. The easiest way to achieve this was to copy the DTN2 code and strip the parts, for example, the convergence layers. This would allow us to provide the routing code an environment similar

enough to the real DTN2 codebase such that the router code could be portable.

8.3.1 External Router Server

In the standard DTN2 code the *ExternalRouter* object is used to interface with external routers. It is in charge of sending events to and from the internal bus to the external plug-in. To do this it uses an XML communication channel.

The *ExternalRouterServer* has the same job, except at the opposite end of the conversation. The documentation from BBN and the DTNRG details the API available via this XML infrastructure. Some of the messages that get converted back from XML to the internal bus are:

- *Link Created Event*
- *Link Deleted Event*
- *Contact Up Event*
- *Contact Down Event*
- *Registration Added Event*
- *Registration Removed Event*
- *Bundle Received Event*

Some of the messages that get created from the internal ERD bus and sent to the main DTN2 Router are:

- *Send Bundle Request*
- *Open Link Request*
- *Inject Bundle Request*

The final goal is for the External Router server to communicate with the DTN2 daemon in such a way that the rest of the ERD code doesn't need to know that it is working as a plug-in. This is not completely possible due to the limitation of the External Router plug-in API.

8.4 DIRECT Router

A DIRECT Router code is divided into 3 main components. The first component is in charge of the router logic and interfacing with the rest of the ERD. The other two components deal with objects and queries.

When a bundle is delivered to the DIRECT Router it is classified as a query or object. Then it is delivered to the relevant handling code. When a new registration is added it is parsed to find if it's a new local query. When a new object or query are created they are matched with possible queries/objects. If a match occurs a response will be sent out or delivered locally.

Lets go over some of the main functions of the DIRECT Router:

- `handle_registration_added()`

  A registration added event means an application has just registered an endpoint. We need to parse the Endpoint ID use by the application and check to see if we need to create a query. The current query registration endpoint is */ccn/get/object_name*, where object_name is the name of the object we wish to query for.

- `handle_contact_up()`

  When we get a contact up event it means a new neighbor has been detected. We need to exchange queries to update state.

- `inject_bundle(String name)`

  When we want to send a bundle from the DIRECT Router, we need to do a series of calls. First we prepare the bundle we need to send, filling in the Endpoint IDs. We then inject the bundle, which sends it to the DTN2 daemon. This will trigger a bundle_injected event.

- `handle_bundle_injected()`

  Once a bundle is injected in the system, the router needs to tell the DTN2 daemon that it needs to be sent.

- `handle_bundle_expired()`

  This function is not normally called. In the DIRECT-DTN infrastructure bundles are sent on a hop by hop basis or locally. So it is rare to see a bundle expire. If this does happen it will be due to a neighbor node going away while a send was in place. This does not affect the protocol since it keeps soft state.

- *handle_bundle_delivery(String name)*

  When a bundle needs to be delivered to the DIRECT Router a bundle_delivery event is generated. This function is actually just a wrapper for the *deliver_bundle()* function.

  deliver_bundle() needs to check what kind of bundle was just received. It can be an object or it can be a query. If the object is new it will be added to the ObjectStore database. If the object is an updated object then we will update the local object. If it is an old object it will be discarded. A similar path is taken for queries. If the query is new it gets added and triggers a query broadcast.

9 Final Thoughts

For RIDE, Phase III of the DARPA DTN program was a partial success. The reason we can't say that Phase III was a full success is that we were hoping to get DIRECT integrated with the new DTN work done by the other participants, namely, BBN and MITRE. However, the code from BBN and MITRE was not available to us. The decision to move to a closed source distribution of the DTN project limited our involvement in developing code for it. Due to this, we were forced to implement DIRECT in the old DTN2 framework, which has been left behind by the DARPA community.

Other than the integration issues with the close-source DTN core the project was very fruitful. We were able to extend our work in Content-Base networks by developing DIRECT.

We implemented DIRECT, a content centric dissemination protocol in Qualnet. Through simulation we have shown that is a powerful mechanism for communication. DIRECT was based on the work we did for phase II. DIRECT made a tradeoff to perform better (less overhead) than an epidemic protocol by reducing the flexibility in searching. An epidemic protocol will not scale on any large network and this tradeoff needs to be made. Since the generic search is not feasible there needs to be a carefully designed namespace and application framework to interact with the name-based routing.

Working with the DTN2 code was a challenge. We learned the hard way that changing the paradigm of how people perceive and use communication tools is hard. The DTN project has already been dealing with this by utilizing the concept of bundles. We want to move things further and make people conceptualize data as content objects.

Finally, talking with potential users of the military systems such a technology might be involved in showed great confusion. A new way to conceptualize communication poses a hurdle that is not only technical but also organizational. We need to be careful how we promote this new technology and have to selectively pick the right applications and problems to tackle. A content centric infrastructure can't solve every problem and trying to address something it is not suited for might be counter productive.

We still believe that applications will be the driving force for content-based networks. We are working hard to deliver prototypes that take advantage of content-based paradigms to show the real benefits provided by such a system.

References

[1] J. Boice, J.J. Garcia-Luna-Aceves, and K. Obraczka. On-demand routing in disruptive environments. In *In proceedings of the IFIP Networking 2007, 2007*, 2007.

[2] J. Burgess, B. Gallagher, D. Jensen, and B.N. Levine. Maxprop: Routing for vehicle-based disruption-tolerant networking. In *Proceedings of IEEE Infocom*, 2006.

[3] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, R. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: An approach to interplanetary research.

[4] B. Burns, O. Brock, and B. N. Levine. Mv routing and capacity building in disruption tolerant networks. In *Proceedings of IEEE INFOCOM*, 2005.

[5] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-tolerant network architecture, 2005.

[6] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of human mobility on the design of opportunistic forwarding algorithms. In *Proceedings of IEEE Infocom*, 2006.

[7] X. Chen and A.L. Murphy. Enabling disconnected transitive communication in mobile ad hoc networks. In *Workshop on Principles of Mobile Computing, colocated with PODC'01*, pages 21–27, 2001.

[8] J.A. Davis, A.H. Fagg, and B.N. Levine. Wearable computers as packet transport mechanisms in highly-partitioned ad-hoc networks. In *Proceedings of the International Symposium on Wearable Computing, Zurich, Switzerland, October 2001*, pages 141–148, 2001.

[9] N. S. Djukic, M. Piorkowski, and M. Grossglauser. Island hopping: Efficient mobility-assisted forwarding. In *In proceedings of the IEEE SECON 2006, 2006*, 2006.

[10] DTNRG. Delay tolerant networking research group.

[11] H. Dubois-Ferriere, M. Grossglauser, and M. Vetterli. Age matters: Efficient route discovery in mobile ad hoc networks using encounter ages. In *Proceedings of Mobihoc*, 2003.

[12] K. Fall. A delay tolerant networking architecture for challenged internets. In *Proceedings of SIGCOMM*, 2003.

[13] K. Fall, W. Hong, and S. Madden. Custody transfer for reliable delivery in delay tolerant networks.

[14] K. Harras, K. Almeroth, and E. Belding-Royer. Delay tolerant mobile networks (dtmns): Controlled flooding schemes in sparse mobile networks. In *IFIP Networking*, 2005.

[15] P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket switched networks and the consequences of human mobility in conference environments. In *Proceedings of ACM Sigcomm (DTN Workshop)*, 2005.

[16] S. Jain, K. Fall, and S. Patra. Routing in a delay tolerant network. In *Proceedings of ACM SIGCOMM*, 2004.

[17] H. Jun, W. Zhao, M. Ammar, C. Lee, and E. Zegura. Trading latency for energy in wireless ad hoc networks using message ferrying. In *Third IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'05)*, 2005.

[18] J. Leguay, T. Friedman, and V. Conan. Dtn routing in a mobility pattern space. In *Proceedings of SIGCOMM 2005*, 2005.

[19] Q. Li and D. Rus. Communication in disconnected ad hoc networks using message relay. In *Journal of Parallel and Distributed Computing*, volume 63. Academic Press, Inc., 2003.

[20] A. Lindgren, A. Doria, and O. Schelen. Probabilistic routing in intermittently connected networks. In *The First International Workshop on Service Assurance with Partial and Intermittent Resources (SAPIR 2004)*, 2004.

[21] S. Merugu, M. Ammar, and E. Zegura. Routing in space and time in networks with predictable mobility.

[22] M. Musolesi, S. Hailes, and C. Mascolo. Adaptive routing for intermittently connected mobile ad hoc networks. In *Proceedings of SIGCOMM 2005*, 2005.

[23] J. Ott, D. Kutsher, and C. Dwertmann. Integrating dtn and manet routing. In *In proceedings of the ACM SIGCOMM Workshop on Challenged Networks (CHANTS), 2006*, 2006.

[24] Scalable-network technologies: Qualnet 3.9, 2007.

[25] R. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: Modeling a three-tier architecture for sparse sensor networks.

[26] T. Small and Z. J. Haas. Resource and performance tradeoffs in delay-tolerant wireless networks. In *Proceedings of ACM SIGCOMM*, 2005.

[27] T. Spyropoulos, K. Psounis, and C. Raghavendra. Spray and wait: An efficient routing scheme for intermittently connected mobile networks. In *Proceedings of SIGCOMM 2005*, 2005.

[28] T. Spyropoulos, K. Psounis, and C. Raghavendra. Spray and focus: Efficient mobility-assisted routing for heterogeneous correlated mobility. In *In proceedings of IEEE PERCOM, on the International Workshop on Intermittently Connected Mobile Ad hoc Networks (ICMAN), 2007.*, 2007.

[29] W. Su, S. Lee, and M. Gerla. Mobility prediction and routing in ad hoc wireless networks, 2000.

[30] A. Vahdat and D. Becker. Epidemic routing for partially connected ad-hoc networks, 2000.

[31] W. Zhao, M. Ammar, and E. Zegura. A message ferrying approach for data delivery in sparse mobile ad hoc networks. In *Proceedings of ACM Mobihoc*, 2004.

[32] W. Zhao, M. Ammar, and E. Zegura. Controlling the mobility of multiple data transport ferries in a delay tolerant network. In *Proceedings of IEEE Infocom*, 2005.

[33] W. Zhao, Y. Chen, M. Ammar, M. Corner, B. Levine, and E. Zegura. Capacity enhancement using throwboxes in dtns. In *In proceedings of the IEEE Intl. Conf. on Mobile Ad hoc and Sensor Systems (MASS), 2006*, 2006.