A Conent-Based Networking Protocol For Sensor Networks

by

Cyrus P. Hall

B.S.E., University of Colorado, 2002

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2004

This thesis entitled:
A Conent-Based Networking Protocol For Sensor Networks
written by Cyrus P. Hall
has been approved for the Department of Computer Science

_____

_____

Prof. Antonio Carzaniga

_____

Prof. Dirk Grunwald

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Hall, Cyrus P. (M.S., Computer Science)

A Conent-Based Networking Protocol For Sensor Networks

Thesis directed by Prof. Alexander L. Wolf

An ideal sensor network would minimize communication by routing information only to those nodes requiring the information. This thesis explores the use of a content-based network for this purpose, where messages containing sensor readings and associated metadata are relayed from source nodes to destination nodes based solely on the fact that the destination nodes have expressed interest in specific message content. Routing uses a **distance vector** protocol augmented to construct both primary and alternate routes. Forwarding uses content-based matching together with a special process called **dynamic receiver partitioning**. The protocol, called DV/DRP, is specifically designed for wireless sensor networks or other similarly constrained network configurations. We present simulations showing that the protocol scales to large networks while minimizing the resource consumption of individual nodes. It is also shown that the protocol is robust with respect to both transient and permanent node and communication failures. Finally, a preliminary implementation of the protocol on the MOS/Mica2 platform is presented.

Contents

Chapter

1 Introduction 1

2 Routing and Forwarding 6

2.1 Models 6

2.2 Routing 10

2.2.1 Withdrawing Receiver Predicates 12

2.2.2 Recovery 13

2.3 Data Rate Limitation 15

2.4 Forwarding 16

2.4.1 Short-Circuiting Predicate Evaluation 16

2.4.2 Dynamic Receiver Partitioning 18

2.4.3 Forwarding Algorithm 19

3 Evaluation 22

3.1 Simulation Framework 22

3.2 Experimental Setup 23

3.3 Experimental Results 26

3.3.1 Content-Based Delivery 26

3.3.2 Control Traffic 27

3.3.3 Scalability 28

3.3.4 Transient Failures 30
3.3.5 Permanent Failures 32
3.3.6 Prevalence of Route Loops in Alternate Routes 34

4 Implementation 36
4.1 Mantis OS 36
4.1.1 Net Layer 37
4.2 Implementation Design 38
4.2.1 Protocol Implementation 38
4.2.2 Test Applications 39
4.3 Stack Concerns 39
4.4 Implementation Results 40
4.4.1 Resource Usage 42

5 Related Work 44
5.1 Directed Diffusion 44
5.2 Multicast Networking 46
5.3 Content-Based Networking 47

6 Conclusion 48

Bibliography 50

Appendix

A Source Code 53
A.1 dvdrp.h 53
A.2 dvdrp.c 57

Tables

Table

2.1 Predicate Advertisement (a) and Message (b) Packet Formats 9

4.1 Size of various DV/DRP components. 43

Figures

Figure

2.1 Example Predicate 8

2.2 Example Messages 8

2.3 Primary and Alternate Paths 11

2.4 Routing/Forwarding Tables 11

2.5 Dynamic Receiver Partitioning 18

2.6 Sketch of the Forwarding Algorithm 21

3.1 Simulation Framework 24

3.2 25-Node Topology 25

3.3 Functionality: False Negatives and False Positives 26

3.4 Data and Control Traffic 28

3.5 Scalability in Network Size and Number of Receivers, No Failure 29

3.6 Scalability in Network Size and Number of Receivers with Failure 29

3.7 Behavior in the Presence of Transient Failures, Various d-values 31

3.8 Behavior in the Presence of Transient Failures, Proactive vs. Reactive Recovery 31

3.9 Power Drain for Different-Sized Topologies 33

3.10 Tolerance to Permanent Failures 33

3.11 Prevalence of Route Loops 34

4.1 DV/DRP place in the MOS network stack 37

4.2 Multiple hop topology 41

4.3 Multiple base station topology 42

Chapter 1

Introduction

A sensor network is typically composed of a set of resource limited computers, known as **nodes**, that are capable of measuring various types of phenomena. For example, in environmental monitoring the nodes measure physical characteristics, such as temperature, air pressure, and chemical or gas concentrations. The measurements produced by sensor nodes may be transmitted and used either as punctual readings, or they may be used with measurements from other sensors in such a way that the whole network can be utilized as if it were a single "macrosensor"[21].

The kinds of data, as well as the rates at which those data can be usefully consumed, are characteristics of the applications built on top of a sensor network. Sensor nodes themselves are increasingly seen as general-purpose, commodity items that simply produce generic data. Therefore, as others have argued [15], the message traffic in a sensor network should be driven by the dynamic "interests" of the application, rather than by the particular static capabilities or configurations of the sensors. The application might, for example, be interested in only receiving temperature readings from a particular region that exceed a certain threshold or, because of its knowledge of how the data are to be used, it might be interested only in receiving periodic messages containing a temperature reading.

Sensor networks are an ideal application of the **content-based networking** model of communication [8]. In this model, a message is transmitted from a sender

to one or more receivers without the sender having to address the message to any specific receiver. Receivers express interest in the kinds of messages they would like to receive, and the network delivers to the receivers any and only messages matching those interests. Interests are expressed by receivers through **predicate advertisements**. In this receiver-driven style of communication, the network is responsible for efficiently applying predicates to the content of messages so as to minimize the computational and communication costs of the network.

This thesis proposes a protocol that implements the content-based networking model specifically for wireless sensor networks and other similarly resource constrained configurations. Its design is based on the following assumptions:

- **A primitive communication infrastructure:** Nodes in a sensor network have direct access only to their immediate neighbors and communicate with them through a point-to-point or local-broadcast link-layer communication service. There is no multi-hop network service such as unicast, multicast, anycast, or flooding (broadcast) available.

- **Resource-constrained nodes:** Nodes are severely limited in the amount of state they can maintain, in the number of messages they can send and receive, and in the amount of energy they can consume.

- **Many producers and few consumers:** Most of the nodes will act as information producers, while only a few, resource-rich nodes will act as information consumers. The producers (i.e., message senders) are the sensor nodes. The consumers (i.e., message receivers) are commonly referred to as **base stations**, providing functions such as data correlation and gateways to higher-level applications and protocols.

In addition,the protocol currently assumes that the nodes are stationary. Depending

upon the velocity of a given set of mobile nodes, a network could still use the protocol presented herein, however we have not studied the resulting behavior in simulation.

In order to illustrate the kind of sensor network that is consistent with the assumptions listed above, consider one established by dropping wireless nodes from an airplane in a remote geographical region to monitor seismic activity. Because the nodes will need to be inexpensive and expendable, they are limited in both memory and computational power. Similarly, no energy source can be relied upon except that which the nodes bring with them. Some small percentage of the nodes, limited by cost and deployment constraints, will be base stations capable of performing correlation and gateway functions. Once the nodes are on the ground, and after link-layer communication is established, a content-based network will be formed and application-level communication begun. Notice that in this scenario sensor nodes are both entry points for the messages they produce, as well as relay nodes for the messages produced by other sensor nodes; they must be capable of exhibiting both functionalities.

The key features that together distinguish the protocol presented in this thesis from other content-based protocols proposed for sensor networks are as follows:

- **An energy-efficient distance-vector routing protocol that can tolerate both transient and permanent network failures.** For each receiving node r that advertises predicate p_r, the routing protocol creates the forwarding state corresponding to a shortest-paths spanning tree rooted at r. In addition to the best next-hop n_r, the protocol maintains a configurable length array of alternate routes $n_r', n_r'', \ldots$ that are used in case the best route fails. Receivers are informed whenever alternate routes are used, so that they can reactively repair broken routes. This allows the rate at which predicates are periodically readvertised to be reduced or eliminated.

- **A forwarding scheme that efficiently evaluates message content against**

receiver predicates. Predicates are applied to a message once, at the node where the message enters the network. Using a compact bit vector, the message is annotated with an indication of the intended receivers, where each bit represents a different receiver.[1] Relay nodes need only evaluate the bit vector, rather than the full predicate. Since we assume relatively few receivers, the bit vector can be kept small.

- **A routing scheme that avoids loops and redundant paths without requiring the storage and maintenance of message caches or additional routing state.** When a node replicates a message so as to send it down two or more paths (i.e., acting as a path splitter within the topology of the network), each replica is annotated with a partition of the intended receiver space. This simple and local mechanism prevents loops and redundant paths without the need for a costly reverse-path flooding scheme, such as the one used in CBCB [7].

- **A mechanism to limit the rate at which messages are delivered, rather than to limit the rate at which messages are produced.** Following the principle that application interests should drive message traffic, the protocol allows receivers (i.e., base stations) to specify delivery rates as part of predicate advertisements. The protocol spreads and integrates the rate information just as it does the predicates. This allows the network to optimize traffic flow under the given rate limitations by adaptively integrating production rates across the many sensors located around the network. At the same time, rate limitation also serves to limit message congestion close to receivers, which in some cases can save battery life.

We refer to the protocol as **distance vector / dynamic receiver partitioning** (DV/DRP).

[1] Others have proposed annotating messages with receiver information [16], but they record the full address using multiple bytes of information rather than a single bit per receiver.

This thesis presents the design of DV/DRP and gives results of a quantitative evaluation conducted over a number of simulated scenarios. The simulations show that: (1) the service is functional, stable with respect to growing networks, and robust in the face of transient and permanent network failures and (2) the service holds resource consumption to a minimum. For example, a network of 500 sensors with 10 base stations, in the presence of intermittent link failures for 10% of the nodes at any given time, the average level of control traffic remains under 60 packets per second throughout the entire network (0.12 messages a node per second). In terms of power consumption, we estimate that this amounts to only 1.5nA (5400nAh/s) over the entire network (using power figures from [20].

We also present an implementation of the protocol on the MOS/Mica2 platform. The results from the implementation show that: (1) memory resource consumption is acceptable, and (2) the protocol performs as expected in empirical topology studies. Due to constraints in the current implementation of the MOS/Mica2 platform, we were not able to implement DV/DRP's route reliability and recovery code. This remains ongoing work.

In the next chapter, Chapter 2, the details of the DV/DRP protocol are given; in particular, both the routing and forwarding algorithms are discussed. Chapter 3 presents the results of our evaluation studies and Chapter 4 presents an implementation of the scheme on real hardware. Next, Chapter 5 discusses related work, contrasting it to DV/DRP, and we then conclude in Chapter 6 with a discussion of future directions in which the protocol can be taken. Appendix A lists source code for an actual implementation of DV/DRP on the MOS/Mica2 architecture.

Chapter 2

Routing and Forwarding

Before presenting our content-based networking scheme for sensor networks, we provide more details on the envisioned communication service and usage models of the network. At the end of the section we present a case for why sensor networks present a good environment and opportunity for content-based networking.

2.1 Models

Sensor networks typically contain hundreds or thousands of randomly deployed sensor nodes. Each node commonly carries only limited computing resources. For example, the MANTIS Nymph [1] and the Berkeley Mica2 Mote [25] both use the Atmel Atmega 128(L) microcontroller, which can be clocked at a maximum of 16 MHz, but in practice is run at approximately 4 MHz. Memory is an issue as well, with many sensor node designs carrying no more than 128 kilobytes of EEPROM and even less RAM, usually no more than 4 kilobytes.

The key factor limiting the computational resources of sensor nodes is the lack of long-lived power supplies. Many sensor networks cannot be connected to a constant and reliable source of energy and, therefore, must carry their own power in the form of batteries. The amount of battery power found on a sensor node can vary from two small AA batteries to large power packs [20].

Although RAM refreshes and CPU usage provide an obvious source of drain on

the constrained energy budget of a sensor node, the radio of a wireless sensor node overwhelmingly dominates them in terms of total energy consumption. For example, the MANTIS Nymph can draw up to 73 mA while transmitting and 40 mA while receiving, making either operation far more expensive than a fully tasked CPU, which draws only 14 mA. Somewhat counter-intuitively, receiving data often requires more total power than transmitting, as nodes must listen for a period of time looking for the beginning of packets. This listening time can be an order of magnitude longer than the time needed to transmit. Given that a primary task of a sensor network is to move data measuring phenomena from sensor nodes to base stations, the challenge is do so with the minimum amount of communication overhead [2, 13].

Efficient transportation of data is further hampered by the fact that network-level services are usually not available upon deployment. Often the topology of a sensor network is not defined until the sensor nodes are turned on. After the network is powered up, nodes must find valid paths back to the base stations [11]. A given sensor network topology may never stabilize, due to radio interference, node failure, mobility, or malicious interactions from outside. Such instability forces network-level services to perform route recovery and/or maintenance.

The content-based networking model [8] offers a promising approach to structuring energy-efficient, network-level communication services in resource-constrained sensor networks. Instead of using explicitly configured and addressed end points to form communication paths, routes are formed by receivers advertising a message **selection predicate** to the network. A message containing data that match one or more selection predicates is forwarded toward the receivers advertising those predicates.

Consider a base station in a sensor network monitoring wildland fire conditions. The base station may want to be notified (on behalf of some higher-level application) if the temperature or wind speed have reached critical values. The base station might advertise the selection predicate shown in Figure 2.1 so that it is notified whenever

`int` wind_speed >= 30
`int` wind_dir > 0
`int` wind_dir < 160
`int` temperature > 150
`int` humidity <= 5

Figure 2.1: Example Predicate

conditions reach a critical point. Following the simple model of Carzaniga and Wolf [9], a **predicate** is a disjunction of **filters**, which in turn are made up of conjunctions of **constraints**. In the example above, the predicate is formed from two filters, indicated by the horizontal line separating them. Each constraint has a **type** (from a collection of primitive types), a **name**, an **operator**, and a **value**. Names, such as "wind_speed", need not be strings; the actual implementation of DV/DRP uses enumerated integers.

`int` wind_speed = 45
`int` wind_dir = 78
`int` node = 13

`int` wind_speed = 47
`int` wind_dir = 180

Figure 2.2: Example Messages

The sensor nodes in the wild-land fire scenario might produce messages such as the ones represented in Figure 2.2. Again, following the simple model of Carzaniga and Wolf [9], each message is a list of **attributes**. Much like constraints, attributes consist of a type, a name, and a value, but the operator is always equality. Messages are routed through the network toward receivers that have advertised predicates matching the given messages. This form of network-level service model is akin to the application-level communication model known as **publish/subscribe**.

In the context of a multi-hop network protocol, forwarding information is derived, in part, from the selection predicates. Predicates are combined and simplified, and then used in the construction of forwarding tables residing at individual nodes in the network. The forwarding algorithm performs a match against these refined predicates. If a message matches the constraints associated with one or more outbound interfaces,

then the message is forwarded to the next node(s) along the path(s) to the intended receiver(s). In the example above (Figures 2.2 and 2.1), the first message would match the predicate, while the second one would not.

In a resource constrained environment such as a sensor network we must ensure that the routing and forwarding information is small enough to fit within the limited memory available. Moreover, because the network protocol directly influences the total network energy expenditure, we must ensure that paths are optimal and that extraneous communication is minimized. Our hypothesis, substantiated in this work, is that the semantics of content-based networking offer a platform for effectively and efficiently solving many of the problems inherent in sensor-networks, while offering an appropriate interface for applications. Content-based networking allows us to push filtering functionality deep within the network layer, resulting an less energy consumption, while still providing a service that can be made robust and fault-tolerant.

In order to implement the content-based networking service for a sensor network, we have developed specialized routing and forwarding functions in DV/DRP. The routing function is a form of distance-vector routing. The forwarding function uses content-based matching and a special process called **dynamic receiver partitioning**.

Table 2.1 shows the formats of the message and predicate advertisement packets

receiver (16 bits)
next hop (16 bits)
distance (8 bits)
sequence num (8 bits)
bit-vector pos. (5 bits)
min delay (27 bits)
predicate (variable size)

(a)

receiver set (32 bits)
route-failure (1 bits)
message id (31 bits)
message content (variable size)

(b)

Table 2.1: Predicate Advertisement (a) and Message (b) Packet Formats

used in DV/DRP. The various fields, and why each is needed for the operation of routing and forwarding, are now described in the next two sections.

2.2 Routing

DV/DRP uses a straightforward distance-vector routing protocol augmented to maintain an array of alternate next hops. When a receiver node r advertises a predicate p_r, the routing protocol propagates p_r to every node in the network, thereby forming a content-based forwarding tree rooted at r. This can be thought of as the equivalent of a route advertisement. The content-based forwarding tree "attracts" messages matching p_r toward r.

The routing protocol uses a table to represent a distance vector. For convenience, the same table also stores receiver predicates and, therefore, serves as a forwarding table. For each receiver r in the network, the table stores path information as well as path-independent information. The path-independent information consists of the receiver predicate p_r, a sequence number s_r, a bit-vector position b_r, the minimum inter-arrival interval Δ_r, and the time of the last matching message t_r.

Path information consists of an array of distance-vector entries (n_r, l_r), (n'_r, l'_r), (n''_r, l''_r), ... The length of this array is a static parameter of the protocol. The first entry in the array represents the usual distance-vector information. In particular, n_r is the next-hop node on the shortest path toward r, and l_r is the length of that path. The other entries represent the **local second-best path**, the **local third-best path**, and so on, which are used as alternate paths in case of path failure. We define the **local nth-best path** as the nth-best path computed by a traditional distance-vector protocol. In practice, an alternate path goes one hop away from the current node (along a sub-optimal path), and then follows a primary (or, if necessary, alternate) path from there. This definition is exemplified by the graph of Figure 2.3.

Figure 2.4 shows examples of routing/forwarding tables at two points in time. The

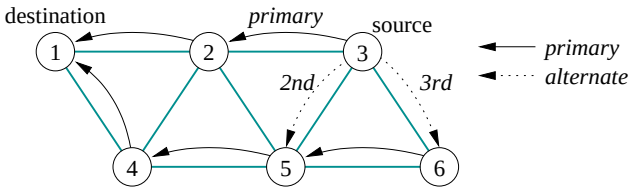

Figure 2.3: Primary and Alternate Paths

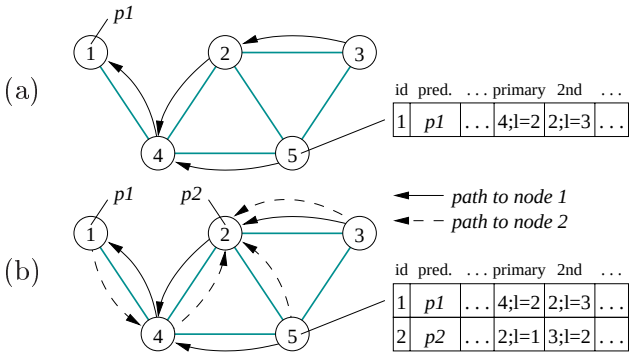

Figure 2.4: Routing/Forwarding Tables

figure shows a network of five nodes, two of which (nodes 1 and 2) become receiver nodes. Initially, node 1 advertises predicate p_1. The predicate propagates through the network, generating a content-based forwarding tree rooted at node 1, represented by the solid arrows in Figure 2.4a. Later, node 2 advertises predicate p_2. This predicate propagates throughout the network, creating the forwarding tree represented in Figure 2.4b with dashed arrows. Also shown are the states of the routing/forwarding table of node 5.

The propagation of predicates follows a simple variation of a traditional distance-vector protocol. A predicate is new when it refers to a new receiver, or when it carries a sequence number greater than the sequence number stored in the table for that receiver. This allows the receiver to change its interests dynamically. In the first case, a new entry is added to the table. In the second case, the existing entry is updated with the new predicate, its primary path is updated, and all the alternate paths are cleared (and then rebuilt). In both cases, the predicate is broadcast to all the neighboring nodes. A predicate is obsolete, and therefore immediately dropped, if its sequence number is lower than the one already stored in the table. If a predicate comes in with the current sequence number for its receiver, it is inserted in the path array according to its distance, and propagated if it is inserted as the primary path. A zero sequence number in the predicate is used to reset the sequence counter in the routing/forwarding table.

2.2.1 Withdrawing Receiver Predicates

A receiver node may also decide to stop receiving messages altogether, effectively withdrawing its advertised predicate. We assume, as with advertisement, that withdrawal is a relatively rare event.

A node withdraws a predicate by issuing a receiver cancellation packet. Receiver cancellations do not need to follow the usual distance-vector protocol used to build up routing/forwarding tables. Therefore, they are simply broadcast to every other node in the network. Upon receiving a cancellation for receiver r, a node simply deletes r from

its routing/forwarding table.

Cancellations are transported on top of a generic broadcast service. This service is implemented by a simple flooding protocol controlled by a per-node packet cache. The route-repair recovery process described below also uses the same broadcast service as a transport layer.

2.2.2 Recovery

Sensor networks are subject to failures that can be modeled as both transient and permanent node failures. For example, wireless links exhibit fading, non-isotropic path propagation, and other types of intermittent interference, which appear as node failures or asymmetric links to neighbors[27]. Also, nodes may be permanently damaged by environmental agents, or they may simply run out of power, causing permanent failures.

DV/DRP deals with failures by detecting errors in data transmissions and by reacting to those errors. At first, DV/DRP tries to route messages around the failed node. If the failure persists, DV/DRP tries to repair the broken paths by reissuing predicate advertisements. In particular, the forwarding algorithm of DV/DRP starts by sending a message to the next-hop node on the primary path to the given receiver. If communication to that node fails, DV/DRP (1) sets a **route-failure** flag in the message, (2) sets the message id field to a random message identifier, (3) inserts that message identifier into a local message cache, (4) tries to send the message along one of the alternate routes, and (5) if all available paths fail, it sends the message as a flood packet. This last stage should be seen as a request to resubscribe by receivers with matching predicates. When a node receives a message that has the route-failure flag set, the node must first check if that message identifier is not in its cache. If the node finds the message identifier in its cache, then it concludes that the packet got trapped in a loop by following one or more alternate routes. In this case, the node falls back to step (5), sending the message as a flood packet. In essence, DV/DRP tries to route a

message around failed nodes by first trying alternate routes, and resorting to flooding only as a last option.

In our simulation topologies, we have observed that usually less than 5% of all route deviations result in loops. We have also studied the prevalence of loops in random topologies of various size. In over 50 500-node topologies studied, we observed loops in at most 15% of all sender/receiver combinations. More details are presented in Section 3.3.6.

If and when the message gets to its final destination, the set route-failure flag will inform the receiver node that one or more branches of its forwarding tree have failed. After seeing a certain number of failures, the receiver node may attempt to repair broken routes by reissuing a predicate advertisement. The policy that controls this reactive re-advertisement process is a configurable parameter of the protocol. In addition to this reactive recovery method, DV/DRP implements a proactive method based on heart-beat re-advertisements. In our experience, reactive recovery is quite effective, so that in many situations the rate of proactive re-advertisements can be reduced to a minimum or completely turned off.

To avoid congestion and to limit the power consumption due to flood packets caused by failed routes, DV/DRP can limit the maximum rate at which nodes are allowed to issue messages as flood packets. Upon initially sending or forwarding a request to resubscribe packet, a node should create a time-stamp entry for each node that has been requested to resubscribe. If another request is made before t_{resub}, the minimal time between re-subscriptions, the new request should be dropped to conserve power. This does not affect the fault tolerance of the network unless the rate is greater than the period between failures. The maximum rate is a configurable protocol parameter.

2.3 Data Rate Limitation

By default, predicate advertisements do not impose a limit on the rate at which messages are delivered. This behavior, however, can be undesirable and even destructive, especially in large networks of sensors producing many messages matching the predicates. In these cases, a receiver and its surrounding relay nodes can be overwhelmed by the message flow.

In order to adapt the content-based service to the needs and capabilities of applications, DV/DRP allows receivers to specify a limit on the message delivery rate. This specification is given as an integral part of a predicate advertisement. A receiver r can advertise a predicate p_r and a minimum inter-arrival interval Δ_r ($\Delta_r = 0$ means no rate limitation). Minimum inter-arrival intervals are then propagated together with the advertisement, and recorded in the routing/forwarding tables. In addition to Δ_r, each entry in the routing/forwarding table records the most recent time t_r that a message has been forwarded to receiver r. This information is used by the forwarding algorithm to filter out messages that would exceed the set delivery rate (see Figure 2.6).

The semantics of this rate limitation service is not intended to uphold fairness across sources. In some circumstances sources may continually overshadow others, which can lead to a non-representative view of the network at the receiver. This problem is not easily solved, and reversing the semantics to instead control the rate of publications per node only shifts the problem to bandwidth and power. A partial solution to the problems of publication-rate semantics would be to track the number of publications from downstream nodes at every node in a given shortest-path tree, balancing between nodes. This in turn creates problems for nodes publishing at differential rates, but may be preferable to doing nothing at all. A better solution may be to introduce a smart aggregation layer, condensing multiple messages into single packets that summarize or transform the data. We feel this is important future work and hope to add some form

of such smart aggregation into the protocol at a latter date.

2.4 Forwarding

The forwarding state produced by the DV/DRP routing protocol is intended to "attract" matching messages toward receiver nodes. A node should proceed to forward an incoming message by matching the message against all the predicates in the routing/forwarding table. The basic idea is that a message m matching predicates $p_1, p_2, \ldots, p_i$ is forwarded to all the next-hop nodes $n_1, n_2, \ldots, n_i$. This matching process is also called **content-based forwarding** [9].

2.4.1 Short-Circuiting Predicate Evaluation

Every node has complete knowledge of all the (small number of) receivers in the network. Therefore, the forwarding algorithm can apply predicate evaluation for a message m, once and for all, at the node where m first enters the network. The results of the evaluation are stored in a header field of the message, called the **receiver set**. At every subsequent hop in the path, the predicate evaluation can be avoided. Instead, all that is necessary is an evaluation of the receiver set against the forwarding information in the routing/forwarding table.

One way to represent a receiver set is as an array of receiver identifiers. This approach is very simple, and can be used directly with existing statically assigned node identifiers, such as MAC addresses. The problem, however, is that it introduces a prohibitive communication overhead, due to the potentially large size of the array. Another approach would be to use Bloom filters [4] to obtain a compact representation of a set of receiver addresses. Unfortunately, Bloom filters offer only a probabilistic membership test and the space needed to increase the precentage change of correctly routing packets is too great for sensor networks.

Our solution is to use a simple fixed-size bit vector, where each receiver is repre-

sented by a dynamically assigned bit position. The obvious advantage of this solution is that it offers a compact representation that incurs no collisions. In principle, the fixed size of the bit vector could be a serious disadvantage, as it imposes an upper bound on the number of active receivers in a network. In practice, such a limitation does not affect most applications in sensor networking, where the vast majority of the nodes are senders (i.e., sensors), and only a few nodes act as receivers (i.e., base stations).

The disadvantage of the bit-vector solution is that it requires nodes to somehow negotiate their bit-vector position. Although static assignment of the position could be performed before deployment, a better approach is to support dynamic assignment. Fortunately, this can be done with a minor extension to the routing protocol, and with a simple local conflict-resolution protocol. Specifically, predicate advertisements are extended to carry the bit position of the receiver. When a node r advertises a predicate p_r for the first time, r randomly chooses its own bit-vector position, b_r, among the ones that are not already in use by other receivers. The node then sends out the predicate advertisement, following the usual distance-vector protocol, with the receiver identifier r, the predicate p_r, and the bit position b_r.

Because it takes some time for predicate advertisements to propagate through the network, it is possible that two or more nodes will pick the same bit-vector position. This rare event is detected by the conflicting nodes as soon as they receive each other's respective advertisements. When a conflict is detected, the node with the lowest node identifier is given priority and, therefore, keeps its chosen bit-vector position. All the other nodes in conflict are forced to choose a different bit-vector position, and resend their advertisements.

This protocol is stable and converges to a conflict-free assignment of bit-vector positions. Other unique values besides the highest node identifier could be used to arbitrate when conflicts in position are found, or explicit meanings could be given when assigning node identifiers. For example, if one base station served as a primary, and an-

other the backup, the primary could take a lower node identifier, assuring itself priority during conflict resolution and, therefore, the best message delivery ratio.

2.4.2 Dynamic Receiver Partitioning

It it easy to see that content-based forwarding delivers messages to interested receivers. However, content-based forwarding alone will also exhibit duplicate deliveries and route loops. As an example, consider the scenario of Figure 2.4b. Assume a message m matching both p_1 and p_2 is sent by node 5. Following both the content-based paths of p_2 (dashed) and p_1 (solid), the message gets to nodes 2 and 4. Then, the copy that went to node 2 is sent again to node 4, and vice versa, thereby creating duplicates. Loops also occur between nodes 2 and 4, and between nodes 1 and 4. The loops can be avoided by a forwarding algorithm that does not send a message back to the node where it came from. However, it is easy to construct examples with loops of three or more nodes, where that simplistic control would not be effective. To avoid duplications and loops, we have designed a forwarding protocol that augments the basic content-based forwarding with a process called **dynamic receiver partitioning**.

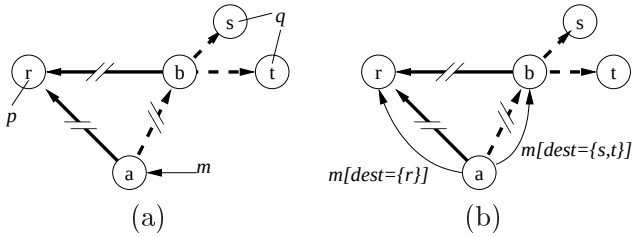

Figure 2.5: Dynamic Receiver Partitioning

We observe that duplicate paths to a receiver r occur when two copies of a message m cross two distinct branches of the content-based spanning tree of r. Figure 2.5a shows a scenario that would cause such a duplication. The figure represents a fragment of a larger network. For generality, we consider content-based paths rather than direct links, as symbolized by broken arrows. The solid and dashed lines represent the content-based

tree of predicates p and q, respectively. Predicate p is advertised by node r, while predicate q is advertised by nodes s and t. The example shows a message m, sent by node a, matching both p and q.

Figure 2.5b illustrates the basic idea of dynamic receiver partitioning. Node a first computes the set of receivers $D = \{r, s, t\}$ interested in m. Then a partitions D according to the next-hop node on the path to each receiver. The result consists of two partitions $D_1 = \{r\}$ and $D_2 = \{s, t\}$, which a attaches to the copies of m going toward r and b, respectively. The idea is that the receiver set attached to a message limits the scope of the forwarding for downstream nodes. In the example, node b receives a packet containing m and $D_2 = \{s, t\}$, which limits the propagation of m to s and t.

2.4.3 Forwarding Algorithm

Figure 2.6 sketches the content-based forwarding algorithm used in DV/DRP. The algorithm makes use of one of two procedures, depending on where within the network the forwarding is to take place. The first procedure, cb_drp_init_forward, takes a message as a parameter, and is executed at the node where the message enters the network. The second procedure, drp_forward, takes a message and a receiver set as parameters, and is executed at each subsequent hop in the routes to the receivers. The procedures are similar in structure, differing only in that the first procedure performs the full predicate evaluation, while the second instead performs only the receiver-set membership test. For a C implementation of this code on the Mantis Operating System (MOS), refer to `dvdrp.c` in Appendix A.2.

The first loop iterates through the entries in the routing/forwarding table. For each entry, the first procedure tests whether the message matches the predicate and whether it does cause the rate limitation for that receiver to be exceeded. For each entry passing the test, it builds a map P that associates receiver sets to next-hop nodes, which is used to represent the receiver partition. An entry that does not pass is said to

be **excluded**.

The second loop is where the message is actually forwarded to next-hop nodes. For each next-hop node n that has downstream interested (i.e., non-excluded) receivers, the algorithm forwards a copy of m and the partition P_n to n.

Rather than performing the content-based match test in nodes downstream from the entry node, the forwarding algorithm executed at such a node uses the second procedure to simply test whether the receiver is in the receiver set attached to the message. If so, it builds the map P and forwards a copy of m and the partition P_n, as in the first procedure.

Note that a node in a sensor network will at the same time act as an entry node for the messages it produces and as relay node for the messages produced elsewhere in the network. Therefore, each node must be capable of executing either function depending on the situation.

```
proc cb_drp_init_forward(message m) {
  map<node, set<node>> P := ∅
  foreach r in fwd_table {
    if time() − t_r > Δ_r and match(m,p_r) {
      P[n_r] := P[n_r] ∪ r
      t_r := time()
    }
  }
  foreach n in P { //for each next-hop node
    send(m,P[n],n)
  }
}
proc drp_forward(message m, set<node> D) {
  map<node, set<node>> P := ∅
  foreach r in fwd_table {
    if time() − t_r > Δ_r and r ∈ D {
      P[n_r] := P[n_r] ∪ r
      t_r := time()
    }
  }
  foreach n in P { //for each next-hop node
    send(m,P[n],n)
  }
}
```

Figure 2.6: Sketch of the Forwarding Algorithm

Chapter 3

Evaluation

We evaluated the DV/DRP protocol by implementing it in simulation and then analyzing its behavior under several scenarios. The primary goals of our analysis were to: (1) assess the ability of DV/DRP to implement the content-based delivery model; (2) profile the costs incurred by DV/DRP, and thereby evaluate its applicability to networks of resource-constrained nodes; and (3) make sure that the protocol is stable and responds gracefully to application demands, as well as to network failures.

3.1 Simulation Framework

Four main Python[24] scripts and the DV/DRP simulator's code make up the bulk of the simulation framework, as shown in Figure 3.1. `create_topology.py` is the first of these, and it creates a number of topologies of a given number of nodes o, forcing each topology created to be fully connected. To keep each topologies average fan-out equivalent, it uses two parameters, μ_r (the mean radio distance) and σ_r (the standard deviation from μ_r), to calculate the appropriate height and width for the given o. This is important, as fan-out has a direct effect on the performance of the protocol, and to directly compare different topologies with different fan-outs would produce useless results.

After topologies are created, `gen_workloads.py` reads workload descriptions and generates n workloads given each set of parameters and m topologies. In this way, each

workload type, defined by the set of parameters, is generated for every size topology, giving a total of nm workloads.

Next, `strip_bs_extras.py` (where `bs` stand for "base station") strips `node_fail`, `node_recover` and `send_message` events for each base station in all workload files. We assume that base stations are stable, with enough power to survive infinite length simulations and that they do not publish sensor data. See below for a longer description of the event types.

Finally, two programs are run on the workloads to generate the final statistics. `calc_loop_metrics.py` runs a heuristic algorithm, detailed in Section 3.3.6, to determine the relative likelihood of a packet encountering a route loop if it is forced to take a secondary path. `dvdrp_sim` generates the main trace files used to calculate false negatives, power consumption, etc.... It rests on top of several content-based simulation libraries that provide basic data structure and workload management functionality. Workloads are loaded and driven by `cbnsim`. Upon initialization, `dvdrp_sim` registers callbacks with `cbnsim`, each one corresponding to an event type in the workload files. Workloads have four event types: (1) `set_predicate` sets a receivers subscription predicate, (2) `send_message` injects a new content-message into the network to be handled by the forwarding function, and (3) `node_fail` and (4) `node_recover`, which respectively switch nodes into and out of a failed state. Events (3) and (4) are used to simulate transient radio interference.

The experimental setup is diagrammed in Figure 3.1.

3.2 Experimental Setup

Our experiments were conducted on random network topologies of 10, 25, 50, 100, 250, and 500 nodes. Topologies were generated by placing nodes randomly within a square target area, and by connecting to each node the other nodes within a given range. For all topologies, we maintained a constant density of nodes so as to obtain a

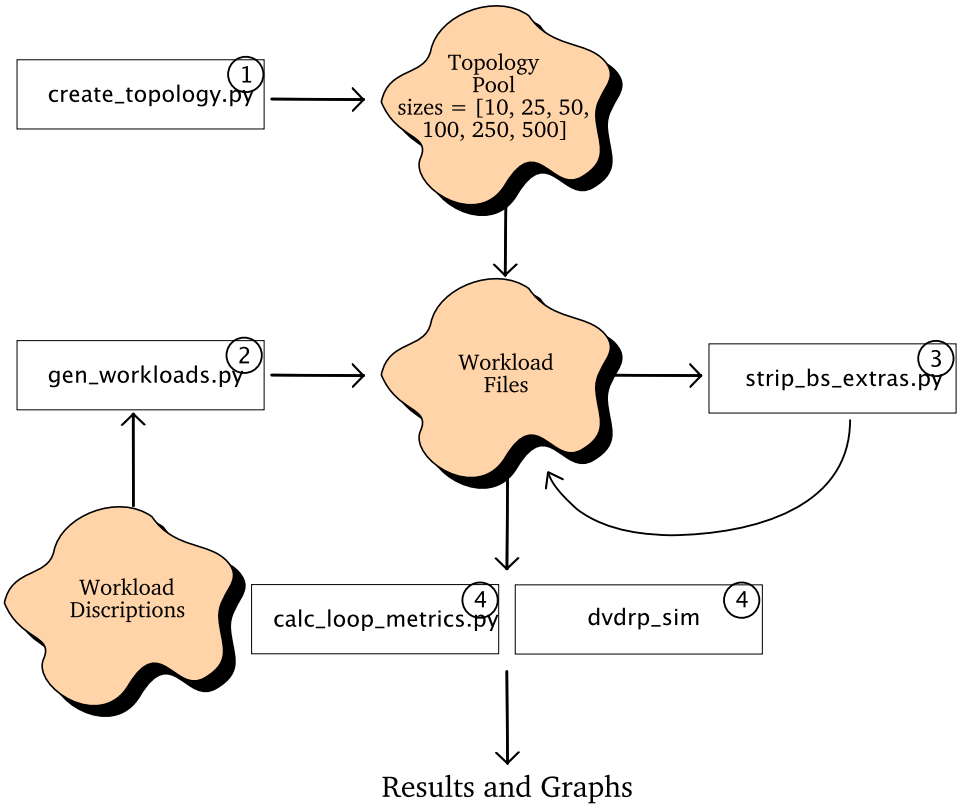

Figure 3.1: Simulation Framework

constant average fan out (edge connectivity) of between five and six neighbors. Finally, we discarded all partitioned topologies. These topology parameters conform to those used in similar simulation studies [22].

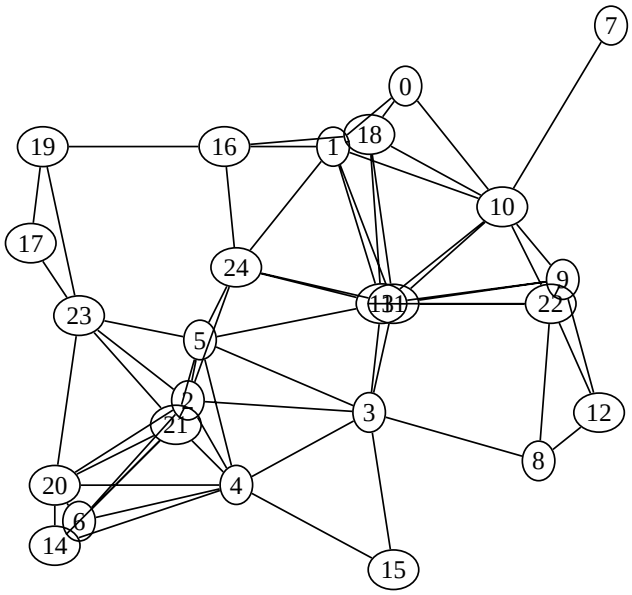

Figure 3.2: 25-Node Topology

As an example, Figure 3.2 shows the 25-node topology that we use in our experiments.

On top of each network topology, we have defined several application scenarios. A scenario is a complete definition of the behavior of each node. In particular, a scenario defines: (1) when nodes send messages; (2) when nodes advertise predicates; and (3) how often nodes are subject to communication failures, and for how long.

The messages and predicates we used are similar to those shown in Figures 2.1 and 2.2. The rate at which senders inject messages is modeled as a Poisson process. Receivers exhibit two types of behavior: some advertise a predicate at the beginning of the simulation, never changing that advertisement, while others change their predicate advertisement periodically throughout the simulation. The parameters that control node failures are the mean time between failures and the durations of those failures,

both modeling a Poisson distribution.

For our simulations, we assumed a CSMA MAC that reports failures to the network protocol. However, DV/DRP can also run on top of "send-and-forget" MACs by using network-level acknowledgments with timeouts. Such "ACKs" incur an extra cost, however, and their use is costly in terms of memory and network usage, leading to extra power usage.

3.3 Experimental Results

3.3.1 Content-Based Delivery

The first experiment we conducted was designed to assess the main functionality of the content-based service implemented by DV/DRP. The scenario that defines this experiment is characterized by a network of 100 nodes, in which each node generates messages at a mean rate of one every 10 seconds. The network contains five receivers that change their predicates every 30 minutes. The scenario has no failures.

The results of this first experiment, plotted in Figure 3.3, are quantified by the percentage of **false negatives** and **false positives** over the total number of messages sent.

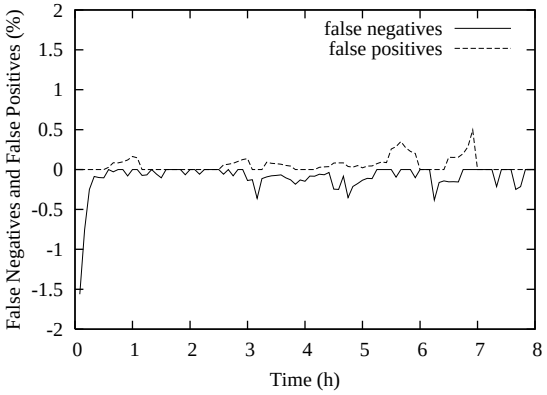

Figure 3.3: Functionality: False Negatives and False Positives

We record a false negative whenever a message does not reach one of its intended

receivers. The false negatives that are observed at the beginning are due to the natural latency of the network as it propagates the initial receiver predicates. During the rest of the simulation, we observe minor occurrences of false negatives due to latencies in propagating changes to receiver predicates.

False positives occur when a message is dropped at one of its final destinations because it does not match the predicate currently advertised by that destination. In practice, false positives are also caused by the latency of the propagation of advertisements.

The primary conclusion we draw from this experiment is that DV/DRP is effective in implementing the content-based service, showing levels of inaccuracy that remain well below 1% of the overall message traffic. The false positives and false negatives that occur are largely unavoidable, due to the natural latency of network communication.

3.3.2 Control Traffic

The experiment above assessed the main functionality of DV/DRP, which is to deliver to receivers all and only the messages matching their predicates. The following experiment analyzed the amount of control traffic incurred by DV/DRP in accomplishing this goal.

Figure 3.4 shows the data and control traffic for the same scenario as Figure 3.3. The values on the Y-axis represent the number of packets of each kind going through the network at any given time. The first curve represents the message (i.e., "data") traffic. The fluctuations are due to basic application behavior: the changing predicates and the variation in message content produced by the sensors.

The second two curves represent the control traffic when using, respectively, the proactive and reactive route recovery methods (see Section 2.2.2). Notice that the two curves are identical in form. This is due to the fact that the scenario contains no failures and the fluctuations in control traffic are caused solely by receivers advertising

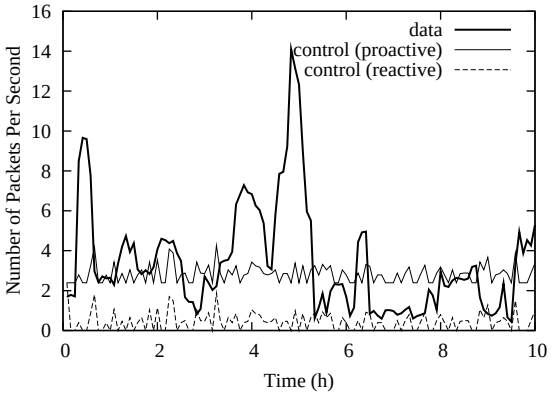

Figure 3.4: Data and Control Traffic

new predicates. However, the proactive method incurs a fixed higher cost than the reactive method due to its use of periodic re-advertisements whose rate is determined by a (fixed) configuration parameter.

3.3.3 Scalability

The purpose of our scalability experiments was to assess the protocol as both the size of the network and the number of receivers increases. We also wanted to verify that DV/DRP remains scalable in the presence of transient failures. Figures 3.5 and 3.6 show experiments conducted over such scenarios. Each node generates messages at a mean rate of one every 12 seconds and each receiver changes its predicate at a mean rate of once every 10 minutes. In practice, receiver predicates are likely to be much more stable, so the experimental rate serves to heavily stress the protocol.

Figure 3.5 shows, in the absence of failures, that the amount of control traffic in the entire network at any given time grows essentially linearly or less with growth in the size of the network. Figure 3.6 shows what happens when 10% of the nodes experience failures at any given time and the network uses the reactive recovery method. Even in the case of a 500-node network with 20 base stations, each node processes only one control packet approximately every five seconds caused by the combination

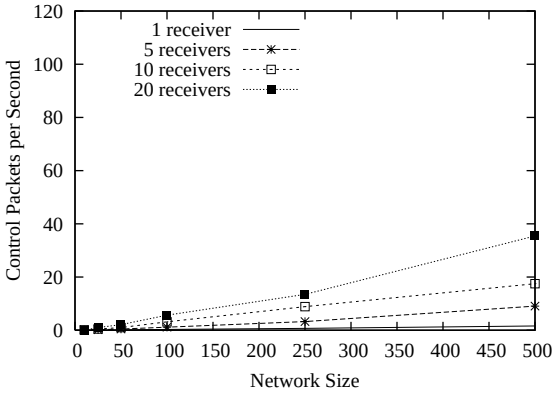

Figure 3.5: Scalability in Network Size and Number of Receivers, No Failure

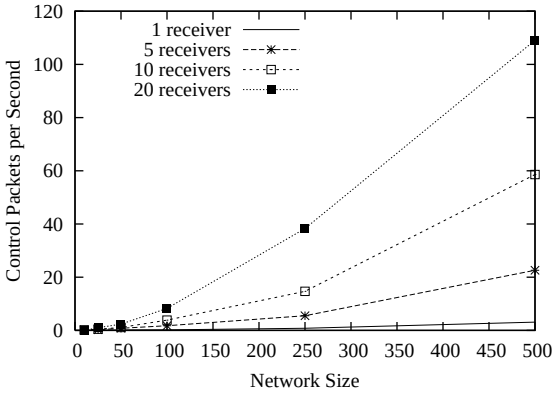

Figure 3.6: Scalability in Network Size and Number of Receivers with Failure

of application-level predicate changes and reactive re-advertisements issued to repair broken paths.

3.3.4 Transient Failures

In this set of experiments we assessed how well DV/DRP recovers from transient failures in the network. The mean time between failure for each node in the network is given as the value d, the failures resolve after a mean time of 60 seconds, and we assume that base stations do not fail. In a real network, failures would be likely to cluster in the same region of the network. However, our simulation does not yet model that scenario. Nevertheless, the parameters we used are realistic enough to get a feel for how DV/DRP performs.

We ran the experiments on the 100-node topology used in sections 3.3.1 and 3.3.2. A mean rate of one message every 30 seconds is generated by sensor nodes, and receiver predicates do not change. Figure 3.7 shows the percentage of false negatives over time, where each curve represents a different mean failure rate. The reactive recovery method was used in all cases. For $d = 300$, where nodes are down approximately 20% of the time, recovery is still quite good, remaining under 40% false negatives for most of the run. In the case of $d > 300$, DV/DRP is able to keep the false negatives below 15% on average.

Figure 3.8 compares the reactive and proactive recovery methods for $d = 600$. The time between re-advertisements for proactive recovery was set so as to create the same average amount of control traffic as reactive recovery. As expected, reactive recovery results in significantly fewer false negatives. This means that reactive recovery is using approximately the same amount of energy to produce a much better result.

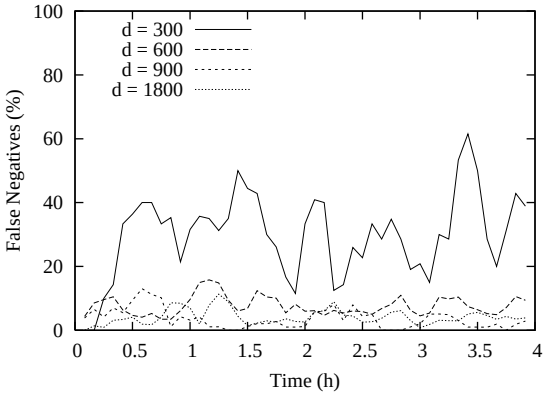

Figure 3.7: Behavior in the Presence of Transient Failures, Various d-values

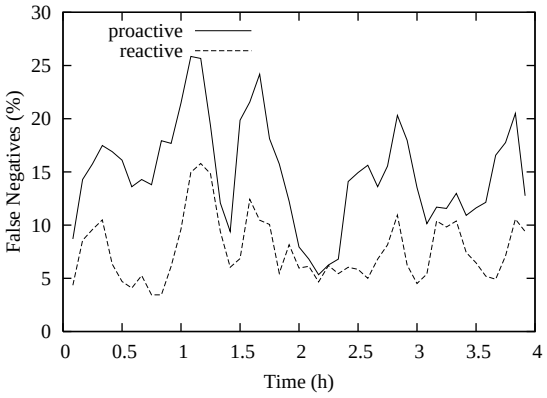

Figure 3.8: Behavior in the Presence of Transient Failures, Proactive vs. Reactive Recovery

3.3.5 Permanent Failures

Protocols such as DV/DRP can exacerbate the problem of power-constrained sensor-network nodes by directing a majority of total network traffic toward a small set of nodes, usually clustered around the base stations. In our next experiments, we assessed the tolerance of DV/DRP during periods when nodes are failing permanently due to power exhaustion. Using a given set of power metrics [1, 20], we calculated the amount of battery charge expended each time a message is sent or received, and each time a sensor value is read. We also accounted for current leakage [20]. Figures 3.9 and 3.10 shows our results.

Each node was given the same, very limited amount of initial charge (100000 nAh), except for base stations, which were given unlimited power. Figure 3.9 shows the percentage of total remaining power in the network for topologies of different sizes. Once the network reaches a critical point, the linear power drain changes slope as the network begins to drain power at a faster rate.

We have traced this critical point to when nodes around the base stations begin to fail. Recall that it is through these neighbor nodes that messages are forwarded to a base station. When a neighbor node fails, another neighbor will be selected (see Section 2.2). However, by the time one node fails, other neighbors of the base station will likely be low on power reserves themselves. There are two reasons for this. First, we assume that the MAC layer is active for all messages, not just those bound for a given node. Second, nodes around the base station have a high likelihood of being the next-hop for a large percentage of traffic in the network and so experience a high level of usage. After the first node fails, a remaining node is chosen, and over time paths become longer, and more power is used, both to deliver messages and maintain routes.

A well-behaved protocol should minimize the rate of undelivered messages until a base station has no neighbors left and has been completely partitioned from the

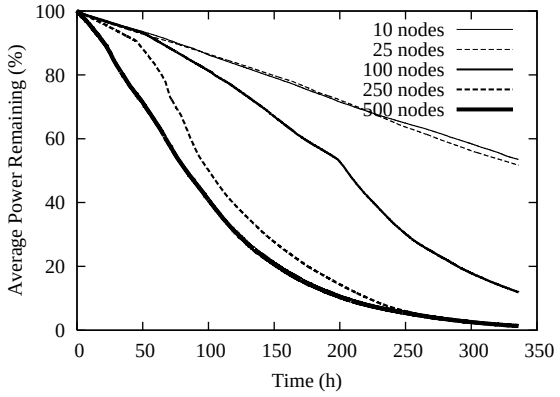

Figure 3.9: Power Drain for Different-Sized Topologies

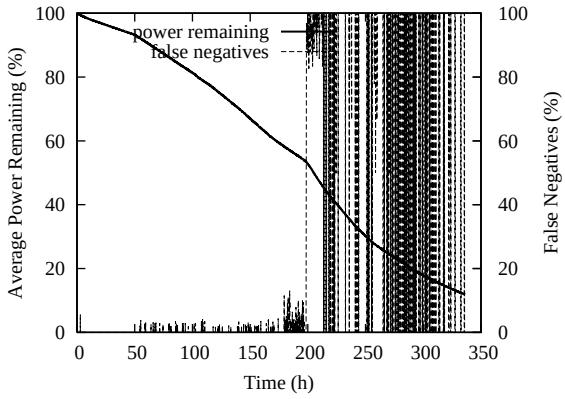

Figure 3.10: Tolerance to Permanent Failures

network. Figure 3.10 shows both the power remaining and false negatives for the 100-node topology. DV/DRP handles the degrading state of the network well, with less than 20% false negatives until the base station is completely isolated after 200 hours. The remainder of the graph is dominated by noise: false negatives fluctuating from 0% to 100% caused by time periods where no messages matching the base station's predicate are generated. If the graph were to continue, this effect would eventually end and false negatives would reach zero after all nodes had expired.

3.3.6 Prevalence of Route Loops in Alternate Routes

DV/DRP uses secondary paths as a first attempt to route messages around failed nodes. Unfortunately, this may lead to route loops, as discussed in Section 2.2.2. Packets that transverse such loops are controlled by DV/DRP using local packet caches. Nevertheless, such loops are undesirable because they cause unnecessary waste of power, and because they may prevent packets from finding an available alternate route to their final destination. Therefore, with the intent to study the effectiveness of DV/DRP in circumventing faults, we have measured the prevalence of route loops in a series of randomly generated topologies. Notice that the existence of these loops is an intrinsic topological property of a network. However, depending on the given scenario, only a small fraction of the existing loops will trap actual packets.

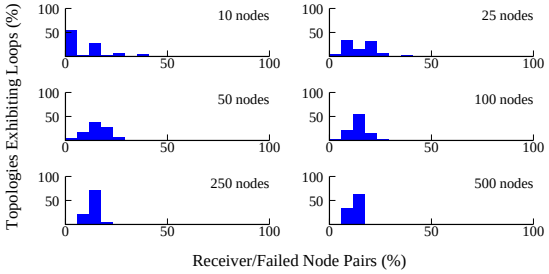

Figure 3.11: Prevalence of Route Loops

Our results are shown in Figure 3.11. The figure shows distribution histograms

for networks of 10, 25, 50, 100, 250, and 500 nodes (50 topologies for each size). These histograms are based on the count of all potential 2-hop loops. In particular, for a receiver node r and a failing node x, we count the number of nodes that would route a message into a loop when trying to avoid x on the way to r. Then we compute the total over all pairs (r, x). The histograms show on the Y-axis the percentage of topologies that exhibit loops for each fraction of the total number of receiver/failed node pairs (X-axis). For example, the lower-right histogram says that, over 50 500-node topologies, we observed loops in at most 15% of all the (receiver/failed) node pairs. Notice that the loops we counted in our study are only potential traps for messages. In our experiments, we observed that usually less than 5% of all route deviations result in loops.

Notice that the loops we counted in our study are only potential traps for messages. In fact, the existence of these loops is an intrinsic topological property of a network. However, the fact that a message is trapped in a loop depends on the position of the sender, the position of the receivers, and the existence and position of a failed node at the time the message is transmitted. In our experiments, we observed that usually less than 5% of all route deviations result in loops.

Chapter 4

Implementation

In order to test the functionality of DV/DRP on real hardware, a basic version of the protocol was implemented on the Mantis OS/Mica2 platform. The primary goals of the implementation where to: (1) confirm that DV/DRP works on real hardware; (2) profile the resources needed on real hardware. Unfortunately, more rigorous tests, such as those performed in Chapter 3, were not possible due to a lack of hardware and software support. Power usage tests could not be performed due to several sleep mode bugs in MOS, creating needless power drain, even when all threads and devices are inactive. The recovery features of DV/DRP were also not implemented due to Mantis OS data structure sharing issues that will be resolved in the future.

4.1 Mantis OS

The Mantis OS (MOS) was designed at the University of Colorado by the Mantis Research Group[1]. It offers a POSIX like API to access kernel functionality, and provides a fully multi-threading/multitasking environment. Unlike other sensor network operating systems, MOS allows users to write applications and kernel code in familiar languages such as C and C++, using standard tools such as GCC. After looking at both MOS and TinyOS[14], we decided that MOS would both be easier and faster to implement on.

Due to ease of which they were acquirable, the Mica2 sensor board from Crossbow

was used. MOS runs on a variety of platforms, including the Mica2. The Mica2 has 4k/bytes RAM, 128k/bytes code space, 128k/bytes flash memory, a CC1000 19.2 kHz radio, and a stackable sensor board interface.

4.1.1 Net Layer

As shown in Figure 4.1, MOS offers a simple **net layer** on which one can easily build new network level protocols. The net layer is split into two sections: sending and receiving. The sending side is called from user threads and uses their context to create, modify and send out messages. This has stack space implications, which are discussed below (see Section 4.3). Packets are received in a dedicated thread, independent from other user and kernel threads.

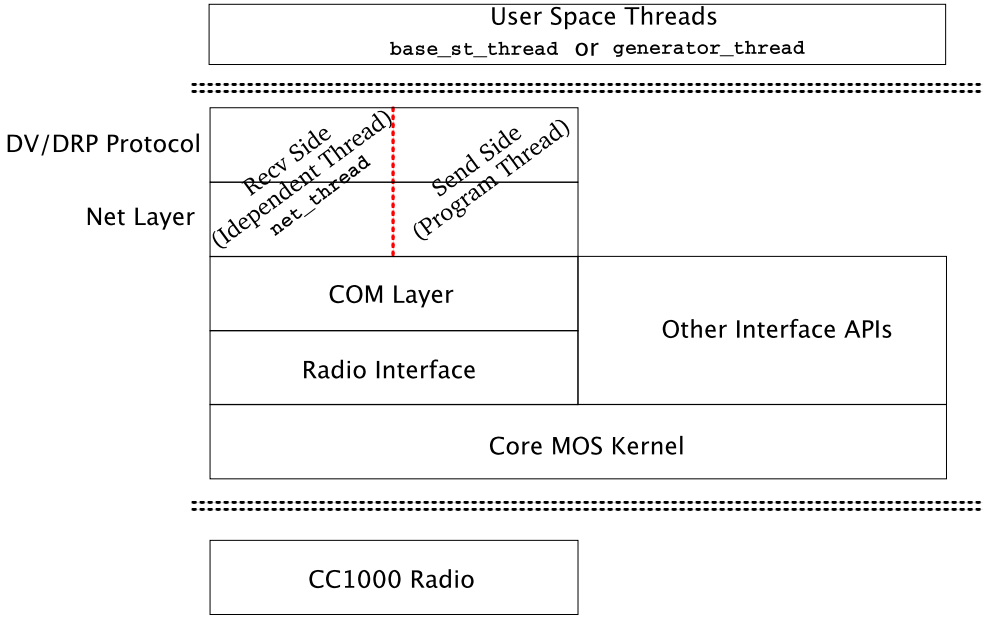

Figure 4.1: DV/DRP place in the MOS network stack

Protocols that sit on top of the net layer are required to implement three functions: (1) `proto_recv(...)` which is called by the `net_thread` whenever a message with the proper protocol ID is received, (2) `proto_send(...)` which is called by `net_send(...)` whenever a user thread wants to send a message, and lastly (3) `proto_ioctl(...)`

which is called by `net_ioctl(...)` whenever a user thread wants to set a protocol specific option. Most protocols will also implement `proto_init` for initialization of internal data structures, although this is not required.

4.2 Implementation Design

There are two main sections of code in the DV/DRP implementation: (1) the protocol implementation, and (2) test applications. The protocol implementation is further broken down into two parts: the protocol stack and the user interface.

4.2.1 Protocol Implementation

The protocol stack interface with the net layer is `dvdrp_recv(...)`, which is called from the net layer whenever a new DV/DRP packet arrives. It proceeds to check the packet type (predicate advertisement, message, repair request) and then call the appropriate handler. The handlers are divided into to main sections of code: the forwarding functions and the route maintenance functions. The forwarding functions are controlled by `dvdrp_mesg_send(...)`, which performs forwarding on a passed in packet.

Route maintenance is coordinated by `dvdrp_route_update(...)`. It performs the work of maintaining the distance vectors in the routing/forwarding table, following the rules set out in Chapter 2.

The user interface to DV/DRP consists of two functions: `dvdrp_ioctl(...)` (called via `net_ioctl(...)`) and `dvdrp_send(...)` (called via `net_send(...)`). Applications call `dvdrp_ioctl(...)`) to set new predicates, or to force a sequence number update, to set failure limits, and to tune other protocols parameters, mostly related to base station functionality. Predicates are built using the protocol memory pools so that the protocol can free passed predicates, if it becomes necessary, without application intervention. This significantly simplifies the memory management model. Applications

call `dvdrp_send(...)` to send new messages containing attributes.

4.2.2 Test Applications

Two test applications were implemented: `base_st_thread` and `generate_thread`. The first sets a given static predicate and then, based on the test scenario, forces sequence number updates. This causes any other nodes in the network to update their route tables and is a temporary solution until recovery can be implemented.

`generate_thread` generates messages at a configurable rate. The attribute names in the messages match those in the predicate set by the base station, however the data values are incremented linearly. This means that as the data values wrap around (they are 8-bit numbers), that messages will start and stop matching the predicates. This is appropriate behavior to test the forwarding code in the protocol stack.

4.3 Stack Concerns

Due to the limited memory constraints on the Mica2, the implementation of DV/DRP had to be carefully planned. Each thread in MOS is allocated its own stack. The stack is statically sized after initialization, forcing the user to carefully watch the number of function calls made, the number of parameters to each call, and the number of local variables. Global variables are stored on the heap, which grows the opposite direction of the stacks. All memory use fits in just 4 k/bytes.

The implementation of DV/DRP was plagued with stack overflows for much of the development period. As new functionality was exposed for the first time, it was often difficult to figure out if a stack overflow or an actual bug was causing misbehavior. As a solution, an OS stack monitor was written to periodically check for overflows. This speed up the development cycle dramatically, as it removed a large portion of the guessing process during initial debugging.

The stack monitor also revealed that the stack sizes that were being used had been

woefully inadequate. Each of the `net_thread`, `base_st_thread`, and `generate_thread` threads were overflowing when certain types/lengths of messages were received. Increasing the stack size solved the problem, but this solution is not optimal due to the new combined size of the stacks, 416 bytes. Normally such a small chunk of memory wouldn't be worth worrying about, but on the Mica2 it is a little over 10% of space available.

There are several possible avenues of opportunity to trim the stack space usage. First, the number of parameters passed, and their size, directly effects stack space usage. One quick way to save space would be to pass 8-bit integer offsets of route table entries instead of pointers. Use of in-line functions could also decrease stack usage, however, only at the cost of function size.

4.4 Implementation Results

Several basic tests were run to confirm that the basic functions of DV/DRP work. Due to the limited number of nodes available, the topologies are simplistic, often with the originating sender of a packet in the same physical broadcast domain as the final receiver. This should not effect the results, as the forwarding code (Appendix A.2) does not differentiate between single-hop and multi-hop final destinations.

All of the test of DV/DRP on actual hardware first call `com_ioctl(IFACE_RADIO, CC1000_TX_POWER, 1)` to limit the CC1000's output power. This allowed us to build multi-node topologies in an area the size of a small desk, greatly simplifying the task of topology setup. Topologies where checked by looking at packet's HTL counter and previous hop ids (in the case of receiver advertisements) at each hop, making sure the values were correct. Only a sparse sampling of message was performed. The Mantis OS CSMA MAC was used in all tests. This MAC does not contain ACK functionality, so DV/DRP operated in best-effort mode.

The first several tests ran were designed to confirm the most basic functionality

of the code. Two nodes were used, one a base station, the other a producer. This test was designed to confirm several basic things: (1) the predicate would be correctly received by the producer, (2) the forwarding functions would then match a produced message against the predicate, forwarding the message to the base station, (3) the base station correct identifies the message as matching a local predicate and delivers it, and (4) delivers the message to the user application (in this case, `base_st_thread`).

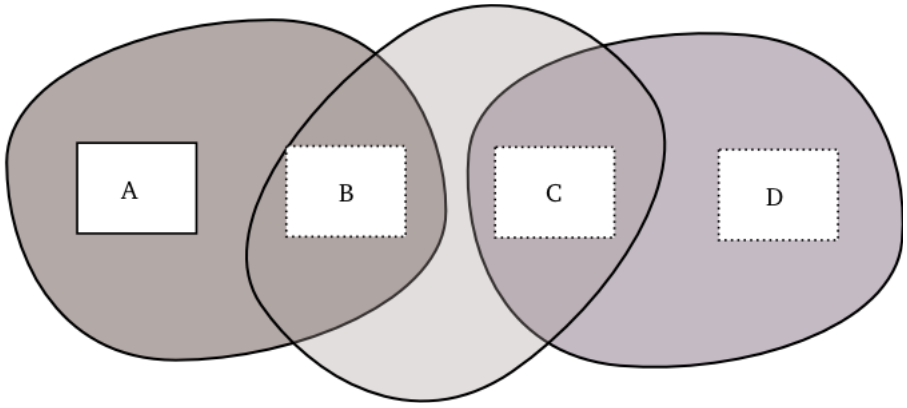

Figure 4.2: Multiple hop topology

After the basic test confirmed the basic functionality of the code and the algorithms implemented therein, multiple hops were tested. This makes sure that the DRP code works correctly in the single receiver case. Figure 4.2 shows the setup used. Base station nodes in the topology graphs are denoted as boxes with solid borders, while message generator nodes have dashed borders. Large, filled, amorphous blobs represent local broadcast domains. The base station subscribed to a simple predicate, asking for all messages generated by the three producers. Message were successfully delivered over multiple hops to node A. The use of a CSMA MAC without ACKs caused some messages to be dropped, and we found the senders to cycle in and out of phase. When in phase, more messages would be dropped due to radio interference, and when out of phase all messages would get through. However, this is a property of the MAC layer, not DV/DRP.

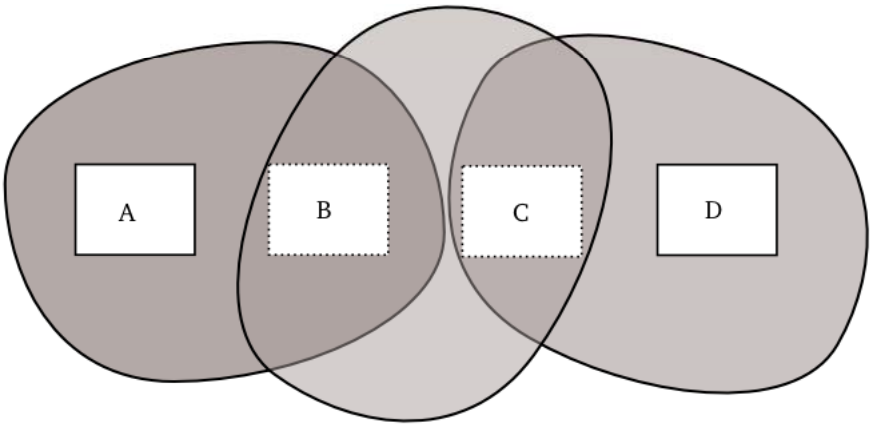

Figure 4.3: Multiple base station topology

The multiple base station receiver test topology is shown in Figure 4.3. Each base-station subscribed to a different predicate, while the generators sent incrementing values. Messages were successfully forwarded across multiple hops to the proper base station(s) by the implementation.

4.4.1 Resource Usage

On average, use of the current DV/DRP implementation on MOS in an application will result in the additional use of 944 bytes of RAM. This includes both heap and stack use and is 23% of all available RAM on the Mica2. This number will go up slightly after the implementation of recovery, but the additional memory buffers needed are already included. Table 4.1 gives the size of each component in the implementation.

The final code size is quite reasonable, taking up only 3% of the code space available. The code size should not grow more than one or two kilobytes with the addition of recovery.

Component	Size (bytes)
`net_thread` stack	192
`base_st_thread` stack	224
`generate_thread` stack	224
`dvdrp.o` heap size	528
`dvdrp.o` code size	3954
Average RAM use	944

Table 4.1: Size of various DV/DRP components.

Chapter 5

Related Work

The work described in this paper builds upon previous work in sensor, ad-hoc, multicast, and content-based networking. In this chapter we present a brief review of those protocols that are most closely related to DV/DRP.

5.1 Directed Diffusion

The sensor-networking community has studied a wide variety of network protocols. A large portion of this work involves techniques for optimizing various aspects of link-layer communications. For example, network data aggregation [17, 18], node clustering techniques [13] for minimizing radio listening time, and energy efficient MAC protocols [23, 26] have all been explored. Many of these techniques can be applied to a network running DV/DRP to further improve its performance.

At a higher level of communication, directed diffusion [15] shares many of our goals. Directed diffusion is a data-driven network service that provides a similar interface to that of DV/DRP. The general routing strategy underlying directed diffusion is based on three processes, not all neccesarily used together: (1) receiver interests are "diffused" throughout a sensor field to establish so-called "gradients"; (2) sensor data are sent toward receivers either in a flooding fashion or by following gradients; and (3) gradients are dynamically modified by applications through reinforcement of paths.

Although originally conceived as a particular routing protocol, directed diffusion

is now presented as a general conceptual framework for a family of protocols [12]. Many fundamental protocol decisions are not defined, but instead delegated to the application (e.g., gradient establishment and reinforcement), giving great flexibility to the protocol designer.

DV/DRP is rather different from directed diffusion in this respect. DV/DRP is a fully specified protocol with an application interface that clearly isolates the application from routing decisions (see Section 2). This architecture was choosen for two main reasons. First, our target is a network of extremely simple and resource-constrained nodes. In such networks, deploying and executing application-specific logic on each node may be either infeasible or undesirable. Second, and perhaps more importantly, we see potential problems in delegating routing decisions to the application developer. In particular, doing so offers little or no guarantees in terms of interoperability, global optimality, or even stability.

In recent work, the general directed diffusion framework has been instantiated as three classes of protocols [22]. Two of these protocol classes, "push" and "two-phase pull" diffusion, are based on a common scheme in which sensor data are flooded throughout the network, and receivers establish preferred routes through positive reinforcement. The primary benefit of this scheme is that by keeping multiple paths alive (i.e., by replicating messages) the network is probabilistically more reliable. There has also been work done on using multipath routing in directed diffusion [10]. However, that solution still replicates messages when there are no failures in the primary path.

DV/DRP adopts a rather different approach to route recovery. In particular, DV/DRP deals with transient failures using **local alternate routes** and permanent route failures using a **reactive repair** mechanism. While a a detailed analysis has not yet been performed, we hypothesize that the overhead involved in continuously inflating traffic on a per-message basis, as done in push and two-phase pull diffusion, is greater than the cost of maintaining local alternate routes and repairing broken paths on a

per-receiver basis.

DV/DRP is most similar to the third directed diffusion protocol class, "one-phase pull". One-phase pull is purely a publish/subscribe scheme, where interests (subscriptions) are flooded to all nodes, and where data (publications) follow backwards the paths established by subscriptions. Algorithms for such a scheme have been extensively studied in the publish/subscribe literature [3, 6] and in the area of content-based networking [5, 7, 9]. While forming the theoretical foundation of DV/DRP, the algorithms in the work described here are substantial adaptations for use in the resource-constrained and primitive environment of a wireless sensor network.

5.2 Multicast Networking

DV/DRP also takes insperation from AODV in both initial route discovery techniques and use of sequence numbers to control forward tables.

The Ad-hoc On Demand Distance Vector (AODV) protocol [19] first performs a route-discovery phase, where queries are sent out looking for specific destinations. As the query packet is flooded, each hop is marked in the packet so that a complete route back to the sender is recorded. Once the packet reaches the destination or another node that already has a route to the destination, it is returned to the sender with the full path list included.

To reduce overhead during the route-discovery phase, packets are dropped at nodes that already have a valid route to the destination with a greater or equal destination sequence number. Paths are cached by nodes, so the route-discovery phase can become quite efficient if there is no mobility in the network. Link failure is reported to senders, which then initiate a new receiver query to find a new route.

The overhead of the on-demand nature of AODV is inappropriate for resource-constrained environments, where battery life is at a premium. The constant network traffic associated with route recovery and continuous global setup is too costly for a

sensor network.

DV/DRP implements what amounts to a multicast delivery, because a message may match the predicates advertised by more than one receiver. Differential destination multicast (DDM) [16] is a protocol used to support multicast groups in mobile ad-hoc networks. In DDM, the destinations for a packet are encoded in a packet header. This is similar to the bit-vector representation in DV/DRP. However, DDM records full addresses, whereas DV/DRP uses only a single bit per receiver. Beyond the common use of a packet header to record information about receivers, the protocols are quite different. For example, DDM is built on top of a unicast service, and nodes keep soft state to facilitate faster communication between mobile multicast group members.

5.3 Content-Based Networking

The CBCB routing protocol for content-based networking [7] utilizes a two-tiered architecture that consists of a content-based forwarding algorithm residing on top of a broadcast networking layer. The forwarding protocol manages a table of predicates that are used to determine whether a message should be retransmitted. The broadcast layer is responsible for actually moving packets across the network in a loop-free manner. Although CBCB exhibits the behavior desired for a content-based network, the overhead involved with setting up the broadcast layer and the storage space necessary to hold the tables of predicates precludes it from being an appropriate solution for sensor networks.

Chapter 6

Conclusion

Work remains to improve and further assess DV/DRP. We have not addressed in-network aggregation or mobility, both of which can be critical capabilities of a sensor network. In particular, aggregation techniques could offer a way to offset switching to a sender based rate limitation scheme, and could add a level of fairness to the current rate-of-delivery scheme. This seems to be an extremely important area to explore if DV/DRP is going to continue to mature.

We also need to explore improved local recovery mechanisms. Others have suggested the use of braided route techniques to create secondary paths. Such paths could offer loop-free secondary paths, possibly eliminating the need for packet caches in DV/DRP. Another option that needs evaluation is the use of dual- or tri-path routes, each route unique and independent except for the end-points. However, such a scheme could force the introduction of an end-to-end reliability scheme, instead of maintaining reliability on a hop-to-hop basis.

To conclude, this work has presented a content-based networking protocol that effectively and efficiently solves many of the problems inherent in sensor-network communication. DV/DRP maximizes proper message delivery and minimizes power consumption through techniques that maintain shortest paths, yet avoid the flooding and loops found in other protocols. Moreover, the techniques are designed to tolerate both transient and permanent network failures. Extensive simulation studies substantiate

our claims.

Acknowledgments: I would like to thank Richard Han who introduced us to sensor networking and helped us to see the need for a new content-based networking protocol thereon. Adam Torgerson and Charles Gruenwald ignored many of their required tasks and duties in order to help me solve and workaround various issues with MOS, and I am in their debt. Antonio Carzaniga, Jeff Rose and Alex Wolf helped to refine many of the ideas presented in this thesis, and the work could not have have happened without them. Brian Shucker read an earlier version of this work and provided valuable feedback.

Bibliography

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for multimodal networks of in-situ sensors. In 2nd ACM International Workshop on Wireless Sensor Networks and Applications, pages 50–59, September 2003.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey of sensor networks. IEEE Communications Magazine, 40(8):102–114, August 2002.

[3] Guruduth Banavar, Tushar D. Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In 19th IEEE International Conference on Distributed Computing Systems, pages 262–272, Austin, Texas, May 1999.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, July 1970.

[5] Fengyun Cao and Jaswinder Pal Singh. Efficient event routing in content-based publish-subscribe service networks. In IEEE INFOCOM 2004, Hong Kong, China, March 2004.

[6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 19(3):332–383, August 2001.

[7] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In IEEE INFOCOM 2004, Hong Kong, China, March 2004.

[8] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In NSF Workshop on an Infrastructure for Mobile and Wireless Systems, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, October 2001. Springer-Verlag.

[9] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In SIGCOMM 2003, pages 163–174, Karlsruhe, Germany, August 2003.

[10] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. Mobile Computing and Communications Review, 2002.

[11] A. Gummalla and J. Limb. Wireless medium access control protocols. IEEE Communications Surveys, pages 2–15, Second Quarter 2000.

[12] John Heidemann, Fabio Silva, and Deborah Estrin. Matching data dissemination algorithms to application requirements. In First International Conference on Embedded Networked Sensor Systems, pages 218–229, Los Angeles, California, November 2003.

[13] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In 33rd Hawaii International Conference on System Sciences. IEEE Computer Society, 2000.

[14] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[15] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. IEEE/ACM Transactions On Networking, 11(1):2–16, February 2003.

[16] Lusheng Ji and M. Scott Corson. Explicit multicasting for mobile ad hoc networks. Mob. Netw. Appl., 8(5):535–549, 2003.

[17] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In 22nd International Conference on Distributed Computing Systems, pages 575–578. IEEE Computer Society, 2002.

[18] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad hoc sensor networks. SIGOPS Operating Systems Review, 36(SI):131–146, 2002.

[19] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In 2nd IEEE Workshop on Mobile Computing Systems and Applications, pages 90–100, New Orleans, Luisiana, February 1999.

[20] Joseph Robert Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California at Berkeley, 2003.

[21] Brian Shucker and John K. Bennett. Scalable control of distributed robotic macrosensors. In Seventh International Symposium on Distributed Autonomous Robotic Systems, June 2004.

[22] Fabio Silva, John Heidemann, Ramesh Govindan, and Deborah Estrin. Directed diffusion. Technical Report ISI-TR-2004-586, USC/Information Sciences Institute, January 2004.

[23] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In First International Conference on Embedded Networked Sensor Systems, pages 171–180, Los Angeles, California, 2003.

[24] Guido van Rossum. The python scriptint language, http://www.python.org.

[25] Alex Woo and David E. Culler. A transmission control scheme for media access in sensor networks. In Seventh Annual International Conference on Mobile Computing and Networking, pages 221–235, July 2003.

[26] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In IEEE INFOCOM 2002, New York, New York, June 2002.

[27] Gang Zhou, Tian He, Sudha Krishnamurthy, and John A. Stankovic. Impact of radio irregularity on wireless sensor networks. In MobiSYS 2004, June 2004.

Appendix A

Source Code

Note that this code is available in its most current form at:

`http://serl.cs.colorado.edu/~hallcp/dvdrp/`.

The base station and generator code used for the implementation tests can be found at the same location under `examples/`. Stable (but less up-to-date) code can also be found in the Mantis OS repository at:

```
svn co svn://mantis.cs.colorado.edu/mantis-src/mantis
```

A.1 dvdrp.h

```
/*
  This file is part of DV/DRP.
  See http://serl.cs.colorado.edu/

  Copyright (C) 2004 University of Colorado, Boulder

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation; either version 2
  of the License, or (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
  the GNU General Public License for more details.

  You should have received a copy of the GNU General Public
  License
  (See http://www.gnu.org/copyleft/gpl.html)
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-
  1307,
```

```
  USA, or send email to mantis-users@cs.colorado.edu.
/

/** @file dvdrp.h
* @brief content-based routing layer
* @author Cyrus Hall  :  hallcp@colorado.edu
* @date 07/20/2004
*
* An implimentation of DV/DRP, a content-based routing and
* forwarding protocol for sensor networks.
*/

#ifndef _DVDRP_H_
#define _DVDRP_H_


#include <stdio.h>
#include <stdarg.h>

#include "mos.h"
#include "net.h"
#include "com.h"

static const uint8_t DVDRP_PROTO_ID = 12;

////////////////////////////////////////////////////////////
// Base types needed for packets, route, and forwarding tables.
////
typedef uint32_t bv_set_t;
typedef uint16_t node_id_t;

typedef struct dvdrp_attribute_t
{
    uint16_t name;
    uint16_t value;
} dvdrp_attrib;  //4 bytes

typedef uint8_t dvdrp_comp_t;
#define DVDRP_COMPARE_LT 1
#define DVDRP_COMPARE_GT 2
#define DVDRP_COMPARE_EQ 3
#define DVDRP_COMPARE_NE 4

typedef struct dvdrp_constraint_t
{
    uint16_t     name;
    uint16_t     value;
    dvdrp_comp_t compare_type;
} dvdrp_constraint;   //5 bytes

typedef struct dvdrp_filter_t {
    uint8_t           num_constraints;
    dvdrp_constraint  constraints[0];
```

```
} dvdrp_filter;

typedef struct dvdrp_predicate_t
{
    uint8_t      num_filters;
    dvdrp_filter filters[0];
} dvdrp_predicate;

////////////////////////////////////////////////////////////
// DVDRP packet types.  Every packet contains a dvdrp_pkt up top.
////
typedef struct dvdrp_pkt_t
{
    uint8_t       pkt_type;
    uint8_t       htl;
} dvdrp_pkt;      //2 bytes

static const uint8_t DVDRP_MAX_MESG_ATTRIBS = 10;
typedef struct dvdrp_mesg_pkt_t
{
    dvdrp_pkt     header;         //2 byte
    bv_set_t      split_bv;      //4 bytes (6)
    node_id_t     immed_dest;    //2 bytes (8)
    uint8_t       is_secondary;  //1 byte  (9)
    uint8_t       num_attribs;   //1 byte  (10)
    dvdrp_attrib  attributes[0]; //varaible
} __attribute__((packed)) dvdrp_mesg_pkt;

static const uint8_t DVDRP_MAX_ADVERT_CONSTRAINTS = 8;
static const uint8_t DVDRP_MAX_ADVERT_FILTERS = 4;
typedef struct dvdrp_advertisement_pkt_t
{
    dvdrp_pkt        header;      //2 byte
    node_id_t        receiver_id; //2 bytes (4)
    node_id_t        prev_hop;    //2 bytes (6)
    uint8_t          seq_num;     //1 byte  (7)
    uint8_t          bv_pos;      //1 byte  (8)
    uint32_t         min_delay;   //4 bytes (12)
    dvdrp_predicate  pred;        //variable
} __attribute__((packed)) dvdrp_advert_pkt;

typedef struct dvdrp_req_repair_pkt_t
{
    dvdrp_pkt header;      //2 bytes
    node_id_t receiver_id; //2 bytes (4)
} dvdrp_req_repair_pkt;

static const uint8_t DVDRP_MESG_PKT   = 0x1;
static const uint8_t DVDRP_ADVERT_PKT = 0x2;
static const uint8_t DVDRP_REPAIR_PKT = 0x3;

////////////////////////////////////////
// Other DVDRP constants
```

```
////
#define DVDRP_RT_MAX_PREDICATES 5
#define DVDRP_RT_MAX_PATHS 2

//#define DVDRP_MAX_RECEIVERS 32
#define DVDRP_MAX_HTL 20
#define DVDRP_INVALID_NODE_ID 255
#define DVDRP_INVALID_BV_ID 32

//ioctl types
enum dvdrp_ioctl_t {
    DVDRP_IOCTL_SETPRED = 1,
    DVDRP_IOCTL_SETFAILURE_LIMIT,   //not impl
    DVDRP_IOCTL_RESET_PROTO         //not impl
};

////////////////////////////////////////
// Memory pools for route tables
////
typedef struct mem_pool_t {
    uint8_t  num_elems;
    uint8_t  type_size;
    uint32_t free_vec;
    void    *pool;
} mem_pool;

typedef struct dvdrp_constraint_ptr_t {
    uint16_t                        name;
    uint16_t                        value;
    dvdrp_comp_t                    compare_type;
    struct dvdrp_constraint_ptr_t *next;
} dvdrp_constraint_list;

typedef struct dvdrp_filter_ptr_t {
    dvdrp_constraint_list     *constraints;
    struct dvdrp_filter_ptr_t *next;
} dvdrp_filter_list;

#define DVDRP_MEM_CONSTRAINTS 20
#define DVDRP_MEM_FILTERS 10
#define DVDRP_MEM_COMBUFS 4
extern mem_pool constraint_pool;
extern mem_pool filter_pool;
extern mem_pool combuf_pool;

void dvdrp_init_mem_pools();
void *pool_malloc(mem_pool *pool);
void pool_free(mem_pool *pool, void *mem);

////////////////////////////////////////////
// Public function prototypes
////
void dvdrp_init();
```

```
int8_t dvdrp_send(comBuf *pkt, va_list args);
bool dvdrp_recv(comBuf *pkt, uint8_t **footer);
int8_t dvdrp_ioctl(uint8_t request, va_list args);

////////////////////////////////
// Some basic bit operations.
////

#define mask_set(X, Y) (X |= ((uint32_t)0x0001 << (uint32_t)Y))
#define mask_unset(X, Y) (X &= ~((uint32_t)0x0001 << (uint32_t)Y)
                  )
#define mask_query(X, Y) (1 == (((uint32_t)X >>
                                    (uint32_t)Y) & (uint32_t)0x0001))
#define mask_clear(X) (X = 0)

#endif
```

A.2 dvdrp.c

```
/*
  This file is part of DV/DRP.
  See http://serl.cs.colorado.edu/

  Copyright (C) 2004 University of Colorado, Boulder

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation; either version 2
  of the License, or (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
  the GNU General Public License for more details.

  You should have received a copy of the GNU General Public
  License
  (See http://www.gnu.org/copyleft/gpl.html)
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston,
  MA 02111-1307, USA, or send email to
  mantis-users@cs.colorado.edu.
/

/** @file dvdrp.c
* @brief A DV/DRP implementation for the mos net layer.
* @author Cyrus Hall
* @date 07/20/2004
*/
```

```
#include <stdlib.h>
#include <string.h>

#include "led.h"
#include "node_id.h"

#include "dvdrp.h"

////////////////////////////////////////
// Cache realted defines and globals
////
#define CACHE_SIZE   10
static uint16_t cache_ptr[CACHE_SIZE];
static uint8_t  cache_curpos;

int8_t cache_check(dvdrp_pkt *pkt);

////////////////////////////////////////////////
// Forward and routing table related variables
////
typedef struct dvdrp_path_t {
    node_id_t  next_hop;
    uint8_t    distance;
    uint8_t    fail_cnt;
} dvdrp_path;

typedef struct dvdrp_route_t {
    //main route/forwading information
    dvdrp_filter_list *pred;
    node_id_t          receiver_id;
    uint8_t            bv_pos;
    dvdrp_path         paths[DVDRP_RT_MAX_PATHS];

    //extra routing information
    uint8_t            seq;

    //forwarding timing
    uint32_t           last_pkt;
    uint32_t           min_delay;
} dvdrp_route;

//the main route table - this, along with the memory pools it
//uses, is the most hefty piece of memory usage in the code.
//If you're running low on memory (easy to do with only 4/8k),
//try trimming the pool sizes below to the minimum you need
//and then reducing DVDRP_RT_MAX_PREDICATES (see dvdrp.h)
static dvdrp_route route_tbl[DVDRP_RT_MAX_PREDICATES];

//forwarding functions
static void dvdrp_get_splitbv(dvdrp_mesg_pkt *pkt, uint32_t *bv);
static uint8_t dvdrp_mesg_send(dvdrp_mesg_pkt *dvdrp_ptr,
                               comBuf *cbuf);
static uint8_t dvdrp_match(dvdrp_mesg_pkt *, dvdrp_route *);
```

```
static uint8_t dvdrp_attr_match(dvdrp_attrib *attr,
                                dvdrp_constraint_list *constraint);

//route update/maintance
static uint8_t dvdrp_get_free_bv(uint8_t additional_bv);
static uint8_t dvdrp_send_route_advert(dvdrp_route *route,
                                       uint8_t htl);
static uint8_t dvdrp_route_update(dvdrp_route *route,
                                  dvdrp_advert_pkt *pkt);
static void dvdrp_route_init(dvdrp_route *route,
                             dvdrp_advert_pkt *pkt);
static uint8_t dvdrp_route_local_update(dvdrp_route *route,
                                        dvdrp_filter_list *pred);
static uint8_t dvdrp_route_resolve_bv_conflicts(dvdrp_advert_pkt
                                                *pkt);

static dvdrp_route *dvdrp_route_find(node_id_t receiver_id);
static dvdrp_route *dvdrp_route_find_unused();

static void dvdrp_route_copy_pkt_pred(dvdrp_route *route,
                                      dvdrp_advert_pkt *pkt);
static void dvdrp_route_free_pred(dvdrp_filter_list *pred);
static void dvdrp_path_copy(dvdrp_path *dest, dvdrp_path *src);

////////////////////////////
// Memory pools
////
dvdrp_constraint_list constraint_mem[DVDRP_MEM_CONSTRAINTS];
mem_pool              constraint_pool;
dvdrp_filter_list     filter_mem[DVDRP_MEM_FILTERS];
mem_pool              filter_pool;
comBuf                combuf_mem[DVDRP_MEM_COMBUFS];
mem_pool              combuf_pool;

////////////////////////////////////
// Private DV/DRP variables
////
static uint8_t   node_seq_num;
static uint8_t   node_bv;  //stored as a decimal offset
static node_id_t node_id;
static uint32_t  node_delay;

//for protocol packets to use
static comBuf    local_pkt;

////////////////////////////////////
// Public DV/DRP interface
////
/* Setup the DV/DRP protocol and register with the net layer.*/
void dvdrp_init() {
    uint8_t i;

    dvdrp_init_mem_pools();
```

```
    for(i = 0; i <  DVDRP_RT_MAX_PREDICATES; ++i) {
        route_tbl[i].receiver_id = DVDRP_INVALID_NODE_ID;
    }

    memset(cache_ptr, 0, sizeof(uint16_t) * CACHE_SIZE);
    cache_curpos = 0;

    node_seq_num = 0;
    node_id = mos_node_id_get();
    node_bv = DVDRP_INVALID_BV_ID;
    node_delay = 0;

    srand(node_id);

    net_proto_register(DVDRP_PROTO_ID, dvdrp_send, dvdrp_recv,
                       dvdrp_ioctl);
}

int8_t dvdrp_send(comBuf *pkt, va_list args) {
    uint8_t i, orig_size;
    uint8_t num_attribs = va_arg(args, int);
    dvdrp_mesg_pkt *dvdrp_ptr = (dvdrp_mesg_pkt*)
        &(pkt->data[pkt->size]);
    dvdrp_attrib *attribs = va_arg(args, dvdrp_attrib *);

    orig_size = pkt->size;
    if(pkt->size + sizeof(dvdrp_mesg_pkt) + (num_attribs << 2)
       > COM_DATA_SIZE)
    {
        return NET_BUF_FULL;
    }

    dvdrp_ptr->header.pkt_type = DVDRP_MESG_PKT;
    dvdrp_ptr->header.htl = DVDRP_MAX_HTL;
    dvdrp_ptr->is_secondary = FALSE;
    dvdrp_ptr->num_attribs = num_attribs;

    pkt->size += 10;

    for(i = 0; i < num_attribs; ++i) {
        dvdrp_ptr->attributes[i].name = attribs[i].name;
        dvdrp_ptr->attributes[i].value = attribs[i].value;
        pkt->size += sizeof(dvdrp_attrib);
    }

    pkt->data[pkt->size++] = pkt->size - orig_size;

    dvdrp_get_splitbv(dvdrp_ptr, &dvdrp_ptr->split_bv);
    if(dvdrp_ptr->split_bv != 0) {
        dvdrp_mesg_send(dvdrp_ptr, pkt);
        i = pkt->size;
    } else {
```

```
        //no valid destinations, drop packet, don't do work
        i = 0;
    }

    return i;
}

bool dvdrp_recv(comBuf *pkt, uint8_t **footer) {
    uint8_t hdr_size = pkt->data[pkt->size-1];
    uint8_t matches_local = FALSE;
    dvdrp_pkt *hdr = (dvdrp_pkt*)&
        pkt->data[pkt->size - hdr_size - 1];

    hdr->htl--;
    if(hdr->pkt_type == DVDRP_ADVERT_PKT) {
        dvdrp_route_update(
            dvdrp_route_find(((dvdrp_advert_pkt *)hdr)->receiver_id),
            (dvdrp_advert_pkt *)hdr);
    } else if(hdr->pkt_type == DVDRP_MESG_PKT) {
        if(((dvdrp_mesg_pkt *)hdr)->immed_dest == node_id) {
            matches_local = dvdrp_mesg_send((dvdrp_mesg_pkt *)hdr, pkt);
        }
    }

    // Pull our header off.
    pkt->size -= hdr_size + 1;

    //Set the footer if we need to deliver info to the
    //application
    *footer = (uint8_t *)hdr;

    if(matches_local) {
        return TRUE;
    }

    return FALSE;
}

int8_t dvdrp_ioctl(uint8_t request, va_list args) {
    uint8_t ret = TRUE;
    dvdrp_route *route;
    dvdrp_filter_list *pred;

    switch(request) {
    case DVDRP_IOCTL_SETPRED: // Change subscription predicate
        pred = va_arg(args, dvdrp_filter_list *);
        node_delay = va_arg(args, uint32_t);

        route = dvdrp_route_find(node_id);
        if(!route) {
            ret = dvdrp_route_local_update(NULL, pred);
        } else {
            ret = dvdrp_route_local_update(route, pred);
```

```
        }

        break;
    case DVDRP_IOCTL_SETFAILURE_LIMIT:
    case DVDRP_IOCTL_RESET_PROTO:
    default:
        break;
    }

    return ret;
}

static uint8_t dvdrp_send_route_advert(dvdrp_route *route,
                                       uint8_t htl)
{
    dvdrp_advert_pkt *advert_pkt = (dvdrp_advert_pkt*)
        &(local_pkt.data[0]);
    dvdrp_filter_list *f_list;
    dvdrp_filter *cur_f;
    dvdrp_constraint_list *c_list;
    uint8_t num_c, num_f, constraint_size, orig_size;

    orig_size = local_pkt.size = 0;

    advert_pkt->header.pkt_type = DVDRP_ADVERT_PKT;
    advert_pkt->header.htl = htl;
    advert_pkt->receiver_id = route->receiver_id;
    advert_pkt->prev_hop = node_id;
    advert_pkt->seq_num = route->seq;
    advert_pkt->bv_pos = route->bv_pos;
    advert_pkt->min_delay = route->min_delay;

    local_pkt.size = 13;   //bytes up to this point

    f_list = route->pred;
    num_f = 0;
    cur_f = &advert_pkt->pred.filters[0];
    while(f_list) {
        c_list = f_list->constraints;
        constraint_size = num_c = 0;
        while(c_list) {
            cur_f->constraints[num_c].name = c_list->name;
            cur_f->constraints[num_c].value = c_list->value;
            cur_f->constraints[num_c++].compare_type = c_list->
                                                compare_type;
            c_list = c_list->next;
            constraint_size += sizeof(dvdrp_constraint);
        }
        cur_f->num_constraints = num_c;
        constraint_size += sizeof(uint8_t);

        cur_f += constraint_size;
        local_pkt.size += constraint_size;
```

```
        f_list = f_list->next;
        num_f++;
    }
    advert_pkt->pred.num_filters = num_f;

    local_pkt.data[local_pkt.size++] = local_pkt.size -
                                       orig_size;
    // Add the proto ID to the end of the packet.
    local_pkt.data[local_pkt.size++] = DVDRP_PROTO_ID;

    com_send(IFACE_RADIO, &local_pkt);

    return local_pkt.size;
}

/** Functions private to this file. **/
////////////////////////////////////
// Foward/Route table maintance.
////
//forwarding funcs
static void dvdrp_get_splitbv(dvdrp_mesg_pkt *pkt, uint32_t *bv)
{
    uint8_t i;

    *bv = 0;
    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(dvdrp_match(pkt, &route_tbl[i])) {
            mask_set(*bv, route_tbl[i].bv_pos);
        }
    }
}

static uint8_t dvdrp_mesg_send(dvdrp_mesg_pkt *dvdrp_ptr,
                               comBuf *pkt)
{
    node_id_t cur_next_hop = 0;  //to make build clean
    uint32_t  global, local_bv;
    uint8_t   i, ret;

    // Add the proto ID to the end of the packet.
    pkt->data[pkt->size++] = DVDRP_PROTO_ID;

    // This is a rather rough function to write without
    // dynamicly sized DSs, so here it is unoptimized:
    // 1) create a global bit vector marked with matching routes
    // 2) take the first marked route and record it's next_hop
    // 3) unmark that route from the global bv
    // 4) mark in the local bv
    // 5) iterate through the rest of the marked routes
    //   a) if next_hop is the same, repeat 3 & 4 for route
    // 6) local bv is packets bvs; assign and send out
    // 7) goto 2
```

```
//
// This strategy, while slow, saves on the need for larger
// arrays or other memory hungary data structures.  We can
// pay for smaller memory size with a little bit of energy.

// 1)
global = 0;
ret = FALSE;
for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
    if(mask_query(dvdrp_ptr->split_bv, route_tbl[i].bv_pos)) {
        mask_set(global, i);
    }
}

// 2)
while(1) {
    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(mask_query(global, i)) {
            cur_next_hop = route_tbl[i].paths[0].next_hop;
            // 3)
            mask_unset(global, i);
            // 4)
            mask_set(local_bv, route_tbl[i].bv_pos);
            break;
        }
    }

    if(i == DVDRP_RT_MAX_PREDICATES)
        break;

    // 5)
    local_bv = 0;
    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        // 5a)
        if(mask_query(global, i) &&
           cur_next_hop == route_tbl[i].paths[0].next_hop)
        {
            // 5a - 3)
            mask_unset(global, i);
            // 5a - 4)
            mask_set(local_bv, route_tbl[i].bv_pos);
        }
    }

    //check for local delivery
    if(cur_next_hop == node_id) {
        ret = TRUE;
    } else {
        dvdrp_ptr->immed_dest = cur_next_hop;
        com_send(IFACE_RADIO, pkt);
    }
}
```

```
    return ret;
}

static uint8_t dvdrp_match(dvdrp_mesg_pkt *dvdrp_ptr,
                           dvdrp_route* r)
{
    dvdrp_filter_list     *f_list;
    dvdrp_constraint_list *c_list;
    uint8_t counter, filter_size, i;

    //for each filter in the predicate
    for(f_list = r->pred; f_list; f_list = f_list->next) {
        //for each constraint in the filter
        counter = filter_size = 0;   //reset size of filter
        for(c_list = f_list->constraints; c_list; c_list = c_list->next)
        {
            filter_size++;
            //for each attribute in the message
            for(i = 0; i < dvdrp_ptr->num_attribs; ++i) {
                //check to see if attribute is covered by the constraint
                if(dvdrp_attr_match(&dvdrp_ptr->attributes[i], c_list))
                {
                    counter++;
                }
            }
        }

        //if all contraints are covered, packet needs to filtered down
        //interface
        if(counter == filter_size) {
            //if this filter matches, skip the rest of the predicate
            return TRUE;
        }
    }

    return FALSE;
}

static uint8_t dvdrp_attr_match(dvdrp_attrib *attr,
                                dvdrp_constraint_list *constraint)
{
    uint8_t ret = FALSE;

    if(attr->name == constraint->name) {
        switch (constraint->compare_type) {
        case DVDRP_COMPARE_EQ:
            ret = (attr->value == constraint->value);
            break;
        case DVDRP_COMPARE_GT:
            ret = (attr->value > constraint->value);
            break;
        case DVDRP_COMPARE_LT:
            ret = (attr->value < constraint->value);
```

```
            break;
        case DVDRP_COMPARE_NE:
            ret = (attr->value != constraint->value);
            break;
        }
    }

    return ret;
}

////////////////////////
//Route table funcs
//

//get free bv id for a new route
//returns 32 (DVDRP_INVALID_BV_ID) if no free bv
static uint8_t dvdrp_get_free_bv(uint8_t additional) {
    uint32_t bv = 0;
    uint8_t i, j;

    //first marked all used bits
    if(additional != DVDRP_INVALID_BV_ID) {
        mask_set(bv, additional);
    }

    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(route_tbl[i].pred) {
            mask_set(bv, route_tbl[i].bv_pos);
        }
    }

    //now find free bv
    j = 0;
    if(bv == 0xFFFF) {  //we're full, so sorry
        i = DVDRP_INVALID_BV_ID;
    } else {
        i = rand() % 32;

        for( ; j < 32; ++j) {
            if(!mask_set(bv, (i + j) % 32)) {
                break;
            }
        }
    }

    return (i + j) % 32;
}

//pass NULL for local init
static void dvdrp_route_init(dvdrp_route *route,
                             dvdrp_advert_pkt *pkt)
{
    uint8_t i;
```

```
    if(!route)
        return;

    if(pkt) {
        route->receiver_id = pkt->receiver_id;
        route->bv_pos = pkt->bv_pos;
        route->paths[0].next_hop = pkt->prev_hop;
        route->paths[0].distance = DVDRP_MAX_HTL - pkt->header.htl;
        route->seq = pkt->seq_num;
        route->min_delay = pkt->min_delay;
    } else {
        route->receiver_id = node_id;
        route->bv_pos = node_bv;
        route->paths[0].next_hop = node_id;
        route->paths[0].distance = 0;
        route->seq = node_seq_num;
        route->min_delay = node_delay;
    }

    route->paths[0].fail_cnt = 0;
    route->last_pkt = 0;
    for(i = 1; i < DVDRP_RT_MAX_PATHS; ++i) {
        route->paths[i].next_hop = DVDRP_INVALID_NODE_ID;
        route->paths[i].distance = DVDRP_MAX_HTL;
        route->paths[i].fail_cnt = 0;
    }
}

static uint8_t dvdrp_route_local_update(dvdrp_route *route,
                                        dvdrp_filter_list *pred)
{
    if(!route) {
        //the local node has not yet set a route, lets do so
        route = dvdrp_route_find_unused();
        node_bv = dvdrp_get_free_bv(DVDRP_INVALID_BV_ID);

        if(!route || node_bv == DVDRP_INVALID_BV_ID)  {
            //no unused routes
            return FALSE;
        }

        dvdrp_route_init(route, NULL);
    }

    //predicate is localy allocated by the application - use
    //their version of the pred to save space and force
    //ioctl for updates
    route->pred = pred;
    route->seq = ++node_seq_num;

    return dvdrp_send_route_advert(route, DVDRP_MAX_HTL);
}
```

```
static uint8_t dvdrp_route_update(dvdrp_route *route,
                                  dvdrp_advert_pkt *pkt)
{
    uint8_t pkt_dist;
    uint8_t i, j;
    uint8_t ret = FALSE;

    pkt_dist = DVDRP_MAX_HTL - pkt->header.htl;
    if(!route) {
        //first we need to check for and resolve any BV conflicts
        //also, make sure there is an available route
        if(!dvdrp_route_resolve_bv_conflicts(pkt) ||    //bv conflicts?
           !(route = dvdrp_route_find_unused()))        //free route?
        {
            return FALSE;
        }

        dvdrp_route_init(route, pkt);

        dvdrp_route_copy_pkt_pred(route, pkt);
        dvdrp_send_route_advert(route, pkt->header.htl);

        return TRUE;
    }

    for(i = 0; i < DVDRP_RT_MAX_PATHS; ++i) {
        if(pkt_dist < route->paths[i].distance ||
           pkt->seq_num > route->seq)
        {
            j = DVDRP_RT_MAX_PATHS - 1;
            while(j > i) {
                dvdrp_path_copy(&route->paths[j], &route->paths[j-1]);
                --j;
            }

            route->paths[i].next_hop = pkt->prev_hop;
            route->paths[i].distance = pkt_dist;

            if(pkt->seq_num > route->seq) {  //note: i will always be 0
                for(j = 1; j < DVDRP_RT_MAX_PATHS; ++j) {
                    route->paths[j].next_hop = DVDRP_INVALID_NODE_ID;
                    route->paths[j].distance = DVDRP_MAX_HTL;
                }

                if(pkt->pred.num_filters == 0) {
                    //we need to kill this route
                    route->receiver_id = DVDRP_INVALID_NODE_ID;
                    dvdrp_route_free_pred(route->pred);  //free old pred
                    route_tbl[i].pred = NULL;
                } else {
                    dvdrp_route_free_pred(route->pred);
                    dvdrp_route_copy_pkt_pred(route, pkt);
```

```
                    route->seq = pkt->seq_num;
                    route->min_delay = pkt->min_delay;

                    //there are timing situations in bv arbitration
                    //that force this to be updated here
                    route->bv_pos = pkt->bv_pos;

                }
            }

            //we only want to propigate the subscription if we update
            //the PRIMARY - a second broadcast only adds noise to an
            //already received route.
            if(i == 0) {
                dvdrp_send_route_advert(route, pkt->header.htl);
            }

            ret = TRUE;
            break;
        }

        //protect against a next_hop being listed twice
        if(route->paths[i].next_hop == pkt->prev_hop) {
            break;
        }

    }

    //make sure that there arn't multiple paths using the same
    node
    for(j = i+1; j < DVDRP_RT_MAX_PATHS; ++j) {
        if(route->paths[j].next_hop == route->paths[i].next_hop) {
            route->paths[j].next_hop = DVDRP_INVALID_NODE_ID;
            route->paths[j].distance = DVDRP_MAX_HTL;
        }
    }

    return ret;
}

static uint8_t dvdrp_route_resolve_bv_conflicts(dvdrp_advert_pkt
                                                *pkt)
{
    uint8_t i, ret = TRUE;

    // Here's the layout:
    // 1) Check for any other route with the same bv
    //   a) if found, check receiver id;
    //   b) if new_mesg.recv_id < old_bv
    //     i) check if *we* were the loser - readvertise with
    //        new bv
    //    ii) else kill the old route - they need to readvertise
    //   c) else return FALSE, as this new advert is the loser
```

```
    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(route_tbl[i].bv_pos == pkt->bv_pos) {
            if(pkt->receiver_id < route_tbl[i].receiver_id) {
                if(route_tbl[i].receiver_id == node_id) {
                    //get a new node_bv
                    node_bv = dvdrp_get_free_bv(pkt->bv_pos);
                    route_tbl[i].bv_pos = node_bv;

                    //send replacement advertisement
                    dvdrp_send_route_advert(&route_tbl[i], DVDRP_MAX_HTL);
                } else {
                    //kill old route
                    route_tbl[i].receiver_id = DVDRP_INVALID_NODE_ID;
                    dvdrp_route_free_pred(route_tbl[i].pred);
                    route_tbl[i].pred = NULL;
                }
                break;
            } else {
                ret = FALSE;
                break;
            }
        }
    }

    return ret;
}

static dvdrp_route *dvdrp_route_find(node_id_t receiver_id) {
    uint8_t i;

    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(route_tbl[i].receiver_id == receiver_id) {
            return &route_tbl[i];
        }
    }

    return NULL;
}

static dvdrp_route *dvdrp_route_find_unused() {
    uint8_t i;

    for(i = 0; i < DVDRP_RT_MAX_PREDICATES; ++i) {
        if(route_tbl[i].receiver_id == DVDRP_INVALID_NODE_ID) {
            return &route_tbl[i];
        }
    }

    return NULL;
}

static void dvdrp_route_copy_pkt_pred(dvdrp_route *route,
```

```
                                        dvdrp_advert_pkt *pkt)
{
    dvdrp_filter_list     **f_list;
    dvdrp_filter           *cur_filter;
    dvdrp_constraint_list **c_list;
    uint8_t                 i, j, size;

    f_list = &route->pred;
    cur_filter = &(pkt->pred.filters[0]);
    for(i = 0; i < pkt->pred.num_filters; ++i) {
        *f_list = pool_malloc(&filter_pool);
        size = sizeof(uint8_t);
        c_list = &((*f_list)->constraints);

         for(j = 0; j < cur_filter->num_constraints; ++j) {
            *c_list = pool_malloc(&constraint_pool);
            (*c_list)->name = cur_filter->constraints[j].name;
            (*c_list)->value = cur_filter->constraints[j].value;
            (*c_list)->compare_type = cur_filter->constraints[j].
      compare_type;

            c_list = &((*c_list)->next);
            size += sizeof(dvdrp_constraint);
        }

        *c_list = NULL;
        cur_filter += size;
        f_list = &(*f_list)->next;
    }

    *f_list = NULL;
}

static void dvdrp_route_free_pred(dvdrp_filter_list *pred) {
    dvdrp_filter_list *f_list;
    dvdrp_constraint_list *c_list;
    void *temp_mem;

    f_list = pred;
    while(f_list) {
        c_list = f_list->constraints;
        while(c_list) {
            temp_mem = c_list;
            c_list = c_list->next;
            pool_free(&constraint_pool, temp_mem);
        }
        temp_mem = f_list;
        f_list = f_list->next;
        pool_free(&filter_pool, temp_mem);
    }
}

static void dvdrp_path_copy(dvdrp_path *dest, dvdrp_path *src) {
```

```
    dest->next_hop = src->next_hop;
    dest->distance = src->distance;
    dest->fail_cnt = src->fail_cnt;
}

////////////////////////////////////
// Pool maintance functions.
////
void dvdrp_init_mem_pools() {
    uint8_t i;

    constraint_pool.num_elems = DVDRP_MEM_CONSTRAINTS;
    constraint_pool.type_size = sizeof(dvdrp_constraint_list);
    for(i = 0; i < DVDRP_MEM_CONSTRAINTS; ++i) {
        memset(&constraint_pool.free_vec, 0xFF, sizeof(uint32_t));
    }
    constraint_pool.pool = &constraint_mem[0];

    filter_pool.num_elems = DVDRP_MEM_FILTERS;
    filter_pool.type_size = sizeof(dvdrp_filter_list);
    for(i = 0; i < DVDRP_MEM_FILTERS; ++i) {
        memset(&filter_pool.free_vec, 0xFF, sizeof(uint32_t));
    }
    filter_pool.pool = &filter_mem[0];

    combuf_pool.num_elems = DVDRP_MEM_COMBUFS;
    combuf_pool.type_size = sizeof(comBuf);
    for(i = 0; i < DVDRP_MEM_COMBUFS; ++i) {
        memset(&combuf_pool.free_vec, 0xFF, sizeof(uint32_t));
    }
    combuf_pool.pool = &combuf_mem[0];
}

void *pool_malloc(mem_pool *pool) {
    uint8_t i;

    for(i = 0; i < pool->num_elems; ++i) {
        if(mask_query(pool->free_vec, i)) {
            mask_unset(pool->free_vec, i);
            return (pool->pool + (i * pool->type_size));
        }
    }

    return NULL;
}

void pool_free(mem_pool *pool, void *mem) {
    uint8_t bit;

    bit = (mem - pool->pool) / pool->type_size;
    mask_set(pool->free_vec, bit);
}
```