Directed Flood-Routing Framework for Wireless Sensor Networks

Miklós Maróti

Institute for Software Integrated Systems (ISIS), Vanderbilt University,
Box 1829, Station B, Nashville, TN 37235, USA
miklos.maroti@vanderbilt.edu

Abstract. The directed flood-routing framework (DFRF) for wireless sensor networks is introduced in this paper that allows the modeling and rapid development of application specific routing protocols based on directed flooding. Flood-routing protocols are probabilistic methods that make only the best effort to route data packets. The presented family of protocols can route regular sized data packets via broadcast messages according to customizable, state machine based routing policies that govern the way intermediate nodes rebroadcast messages. The framework supports automatic data packet aggregation, and allows in-network data packet filtering and alteration.

1 Introduction

Routing protocols for wireless sensor networks are proliferating. Unlike wired networks, where the TCP/IP is dominant, wireless sensor networks have no prevailing routing protocol. Even well designed and tested routing protocols can exhibit subpar performance under a different application load, in a certain deployment scenario, or on a new hardware platform. We argue that this is unlikely to change in the near future, and current research shall focus on developing and classifying broad families of routing protocols that are easily adaptable to a wide variety of real word applications.

The connectivity and topology of the wireless network, as well as the characteristics of the medium access control (MAC) of the operating system, fundamentally influence the design of any routing protocol. Separating the essential part, called the policy, of a routing protocol from the implementation techniques common to a family of protocols, and expressing it in a compact representation reap substantial benefits. First, protocols become easier to understand. Second, automatic optimization techniques can be utilized to find the best policy that adapts the protocol to the target network topology and to the particular implementation of the MAC. Third, the engine that coordinates and executes these routing policies becomes a general middleware service that bridges the gap between the application requirements and the characteristics of the underlying networking services.

We have identified a rich family of routing protocols based on directed flooding that can be parameterized by policies, as described above. Flood-routing protocols are probabilistic methods that make only the best effort to route data packets. On the other hand, they are particularly resistant to node and link failures because data packets can reach their destination through different routes.

In an acoustic shooter localization application [5] we have successfully used several flood-routing protocols to reconfigure the nodes of the network and to gather

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 99–114, 2004.
© IFIP International Federation for Information Processing 2004

sensor readings, the time of muzzle blast and shock wave events. Designing a routing protocol that can handle this load is especially challenging, since a large subset of nodes detects these acoustic events approximately at the same time and has to report back to the base station under real time constraints. The routing protocols that achieved the requirements of the acoustic shooter localization application were developed using the proposed directed flood-routing framework (DFRF).

The framework consists of an engine and several flooding policies. The engine stores and manages data packets enroute to their destination, while routing policies define state machines that describe which packets need to be rebroadcasted by intermediate nodes and when. The framework supports automatic data packet aggregation, and allows in-network data packet filtering and alteration.

In the next section we introduce the targeted hardware platform, and then survey the available routing protocols on this hardware. Next, we formally define the directed flood-routing framework. Finally, we present a set of selected flooding policies that can be used to build robust wireless sensor network applications. Among these a novel spanning tree based convergecast routing policy is introduced that routes messages in a "*lane*" consisting of the nodes at most one hop away from the shortest path from the sender to the root. The resulting routing policy is (1) robust to node and link failures, (2) energy efficient as the maximum number of routing messages increases only linearly with the distance of the sender, and (3) has no data or message overhead compared to simple spanning tree based routing.

2 The Target Platform

The directed flood-routing framework was evaluated on the Berkeley Mica motes [1], the most widely used platform for researching ad-hoc wireless sensor networks with limited resources. This platform exemplifies the class of resource constrained platforms having a broadcast medium which the proposed framework was designed for. We highlight the main characteristics of this platform now. The second generation Mica2 version features a 7.3 MHz microcontroller, 4 KB of RAM, 128 KB of flash memory, and a 433 MHz wireless radio transceiver. The motes are powered by two AA batteries, which last for a few days under continuous operation. A wide variety of pluggable sensor boards containing light, temperature, magnetic and other sensors are available. The Mica motes run a small, embedded, open source operating system, called TinyOS, specifically designed for resource limited sensor networks [2]. TinyOS applications are statically linked graphs of event-driven operating system and application components written in the nesC language, a variant of C [4].

The characteristics of the radio transceiver and the radio stack of the target platform are of special importance to the performance of any multi-hop communication protocol. The radio chip (CC1000) of the Mica2 mote utilizes a single radio channel, has 38.4 Kbps transfer rate and maximum 500-foot communication range in open space. Close to or on the ground the range drops dramatically to tens of feet. TinyOS employs up to 36-byte long radio messages. Seven bytes are reserved by the OS to store the length, the cyclic redundancy check (CRC) and other parameters of the message, leaving only 29 bytes for application data at most. The MAC is based on the carrier sense multiple access (CSMA) technique with random backoff [3]. The Mica2 mote can transmit or receive up to 30 messages per second provided no radio colli-

sions occur. Due to manufacturing differences and fading effects, there are more "polite" motes that will not transmit at all if nearby motes are constantly occupying the radio channel. Others are more prone to start transmitting messages in the middle of other transmissions causing radio collisions.

3 Existing Approaches

Conventional routing protocols are insufficient for ad-hoc wireless sensor networks because of their routing related communication overhead. Examples of a few proposed protocols are: dynamic source routing (DSR) [6], ad-hoc on demand distance vector routing (AODV) [7], temporally ordered routing algorithm (TORA) [8], and the zone routing protocol (ZRP) [9]. On the other hand, routing protocols for sensor networks can exploit the physical properties of the environment where the network is deployed. For example, the location of nodes and their sensor readings in these networks are typically more important than their node IDs.

Existing research mostly focused on location-aware routing protocols allowing routers to be nearly stateless: each node needs to know only the position of its neighbors to make the right forwarding decisions. The greedy perimeter stateless routing protocol (GPSR) use perimeter forwarding to get around voids [10]. Location-aided routing (LAR) improves the efficiency of on-demand route-discovery algorithms by restricting routing-packet flooding to "request zones" [11]. This particular protocol could be developed in the proposed DFRF. The Stateless protocol for real-time communication (SPEED) [12] provides soft real-time communication based on feed-back control.

There are several other routing protocols in the literature relevant to the DFRF. The gradient broadcast (GRAB) protocol builds and maintains a gradient field on a particular subgraph of the network describing the direction sensory data is forwarded to a sink [13]. The gossip routing protocol performs a reliable network broadcasts, probabilistically [14]. These two protocols fit precisely the proposed DFRF. Flooding policies achieving similar functionalities will be presented in Sections 5.1 and 5.2. The rumor routing protocol is a combination of two flooding algorithms: query and event flooding. This protocol utilizes available power resources well [15]. A similar algorithm can possibly developed in DFRF. Constraint based routing (CBR) is another directed flood-routing protocol [16]. TinyDiffusion [17] is another flooding based routing protocol with a publish-subscribe interface that utilizes route reinforcement. Broadcasting protocols are compared in [18] and [19]. The AODV, GPSR, SPEED, CBR and TinyDiffusion protocols are already implemented in TinyOS, and many others are in development [20].

A wide range of other middleware services related to routing were proposed for wireless sensor networks. A new group management middleware and distributed programming paradigm was introduced EnviroTrack [21]. Database management middleware services for wireless sensor networks, such as TinyDB [22] and COUGAR [23], are also actively researched and can offer greater abstraction than traditional routing middleware services. Most existing services for sensor networks however are not readily reconfigurable to meet application requirements and to fully exploit the capabilities of the underlying hardware and networking services, which distinguishes the proposed directed flood-routing framework from other middleware services.

4 The DFRF Algorithm

The directed flood-routing framework is built around a flood-routing engine middleware service that manages the routing messages on all nodes in the network. Application components using the flood-routing engine can send and receive regular sized data packets according to a flooding policy. Flooding policies specify the "direction" of flooding and how intermediate nodes rebroadcast messages. The DFRF engine keeps the recently received data packets in a table together with their *priority* (which is managed by the policy), periodically selects the packets with the highest priority, packs them into a single radio message and broadcasts it to the neighboring nodes. In the rest of this section we will describe this algorithm.

4.1 System Architecture

The modules that implement the DFRF algorithm on each node and their interactions are depicted in Fig. 1. Each node has a single routing engine module that can serve several application modules. Application modules register data packet types and corresponding flooding policies with the engine. The same policy can be registered for several data packet types. The modules interact with each other through method invocations, which are depicted as arrows originating from the caller. These methods will be covered in detail in the following sections.

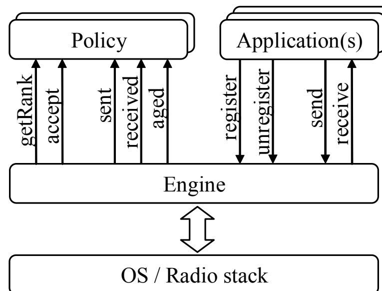

Fig. 1. The architecture of the directed flood-routing framework.

4.2 The Data Packet

Most applications of wireless sensor networks must send and receive different types of data packets, but each of these data types has a well defined internal structure. Typically, data packets of the same type have the same size, as well. For the ease of implementation, and to maximize the available radio message space, we made this a requirement. The lack of variable length data packets allows the DFRF algorithm to aggregate many very small (e.g. 2-byte) data packets into a single radio message. The DFRF engine does not need to know the internal structure of each data packet, only its length.

In directed flooding the same data packet originating from a single node can reach its destination through different routes. This necessitates that the final node, as well as

intermediate nodes, be able to uniquely identify the same data packet in order to discard multiple messages. Most routing protocols append a globally unique identifier to each data packet for this purpose, which adds extra data overhead, typically 2-3 bytes. However, this is not necessary for some applications where the data packet is either already globally unique, or the source of the data packet is unimportant. For example, if nodes send time-stamped sensor readings to a base station, then the node ID (or the 3D coordinates of the sensor) together with the time stamp can serve as a unique identifier of the data packet. Or in a multi-hop network reprogramming application, which uploads a new executable image to each node in the network, the missing capsule ID can be used as the unique identifier of the missing capsule message sent to the base station, since it is unimportant which node did not receive a particular capsule of the image.

Because of these considerations, the DFRF engine does not generate globally unique identifiers but requires the user of the algorithm to employ data packets that can be uniquely identified by their first few bytes. This requirement does not put a lot of burden on the user, as generating unique IDs where necessary is trivial. The number of bytes used to uniquely identify data packets is called the *unique length* of the data packet. We say that two data packets are *analogous* if their unique parts (the first unique length of bytes) are identical. Note that analogous data packets are not necessarily identical, as for example, intermediate nodes can modify data packets enroute to the destination.

Each node in the network must know about all packet types used in a particular wireless sensor application. The packet types are identified by a type ID, and they define the length and the unique length of the packet. The type ID is transmitted with each radio message (which can contain several data packets of the same type), and used by the engine to slice the radio message to the appropriate type of data packets, identify the data packets by their unique part, and notify the corresponding application component.

4.3 The Node Rank

When the DFRF engine (re)broadcasts a radio message, it does not include the node ID of the sender in the message, instead a policy dependent value, called the *rank* of the node, is inserted. The rank describes the progress of a data packet in a policy dependent way, and is used to determine what to do with incoming data packets. For the DFRF engine, the rank is simply a (possibly empty) array of bytes, which is passed to the flooding policy when a data packet arrived. The broad possibilities of what the rank can describe are best illustrated through examples.

For the converge-cast policy that routes along a gradient vector to a base station, the node rank is the hop-count distance from the root. In this policy, when the rank of the sender is smaller than that of the receiver, the receiver simply drops the data packet because it comes from a node closer to the root than itself. For the network-wide broadcast policy, the rank is an empty array. It does not matter where the data packet was received from, it will be rebroadcasted if this is the first time this node has received it. For the spanning tree policy that routes message along a spanning tree to a base station, the node rank can be the node ID of the parent node. Here the parent does not care which of its children sent the data packet. There is a robust variation of this policy where the rank is the node ID of the grandparent, which will be covered

later. For a geographic routing protocol the rank of the node can be the coordinates of the node. A data packet is sent further if the receiving node is closer to the final destination (which is contained in the data packet) than the sender.

It is important to note that the rank does not depend on the data packet, thus the single rank value is used for multiple aggregated data packets of the same type. On the other hand, it is allowed for the rank of a node to change over time. For example, the gradient vector can change if the root of the converge-cast is mobile. It is even possible to provide flow control through ingenious use of ranks. For example, the rank of a node can include a flag indicating that temporally the node cannot store further data packets for retransmission. Neighboring nodes can detect this and delay transmitting new data packets.

The flooding policy has to implement two methods (or commands in the nesC language) that are used by the DFRF engine. First, the *getRank* method has to return the current rank of the node in this flooding policy. This method is invoked for each transmitted radio message. Second, for each received radio message the policy is consulted via the *accept* method whether the message should be processed at all based on the rank of the sender.

4.4 The Priority

Apart from defining the rank of nodes, the flooding policy has the primary role to govern the life-cycle of data packets at each node. Typically, analogous data packets are received multiple times at each node because radio messages are always broadcasted. An intermediate node first receives the data packet from a node further from the destination, next it rebroadcasts it, and then it will normally hear the packet from a node closer to the destination. This shows that each data packet (or more precisely, the family of analogous data packets) has a life cycle at each node. This life-cycle is governed by the flooding policy.

The life-cycle of a particular data packet is described by a finite state machine. There are states in which the data packet is eligible for retransmission, and there are states in which the data packet must be remembered but should not be retransmitted. For example, if an intermediate node A retransmits a data packet D and then hears the same packet from a node closer to the target than A, then it should remember D for some time, but not retransmit it again to prevent receiving and consequently retransmitting an analogous data packet from some node further from the target than A.

The DFRF engine periodically selects, packs and sends data packets from its internal table to the neighbors. Since nodes have very limited memory, an existing data packet from the table might have to be evicted when a new data packet arrives. The flooding policy directs these two selection processes in the following way. The life-cycle states are numbered, typically from 0 to 255, and these numbers are regarded as the *priority* of data packets. The DFRF engine selects data packets for sending or evicting based on their priority.

We have said that in a subset of states data packets are not retransmitted. It can be very important to keep and remember a data packet on a node even if we do not want to retransmit it immediately. The priorities of these data packets must be high, to avoid eviction, and marked as non-transmittable. To have a dense set of non-transmittable states, we selected the odd number priorities for this purpose.

The DFRF engine holds a table of data packets together with their priority or state in which they are currently in. It selects the data packet with the highest even priority (the smallest number) for sending, and with the lowest priority (the largest number) for evicting. There are two special priorities, the smallest and largest values. The value 0 is the initial state of the state machine, while the value 255 is considered the terminal state. If the DFRF engine has a data packet in the terminal state then the packet is considered invalid and the corresponding slot empty.

4.5 The Policy Actions

The flooding policy defines the transitions of the finite state machine that describes the life-cycle of data packets. There are three events: *sent, received* and *aged*. The first is fired when a data packet has been (successfully) broadcasted, the second when a new or an analogous data packet has been received, and the third is fired at regular time intervals. The flooding policy implements three corresponding methods: *sent*, *received* and *aged* that compute the new state of a data packet based on the old state (and on the rank of the sender for the *received* event).

When the method *sent* is invoked, the corresponding data packet has been already successfully broadcasted. Note that the data packet had to pass the selection criteria for it to be sent, that is, it had to have one of the highest even priorities. However, by the time this method is called, it might not have the same (or even an even numbered) priority since other actions could have modified it between the two events. As radio links are naturally unreliable due to collisions and fading, flooding policies typically retransmit the same data packet a few times by stepping through even numbered priorities in increasing order, e.g. from 0 to 2, then to 4, etc. This way, the same data packet gets gradually lower priorities and could become evicted if the engine is short on memory.

The *received* method is called for each incoming data packet. If this is the first time this data packet is received at this node, then priority 0, otherwise the priority of the existing analogous data packet is passed as an argument to this method. The rank of the sender is also available on which the flooding policy can base its action. Normally, the rank of the sender and that of the current node is compared, and if the flooding policy determines that the packet was heard from a node "closer" to the target than the current node, then it either drops or remembers the packet, but it will never become eligible for retransmission. The packet can be dropped by entering state 255 that makes the corresponding slot free. It can be remembered by walking through a high valued chain of odd priorities, e.g. 201, 203, etc., incremented in the *aged* method.

The *aged* method is invoked periodically for all valid (with priority other than 255) data packets. Typically, policy implementations should decrease the priority of the packet by increasing this number and eventually drop the packet by entering priority 255.

4.6 Message Layout

Each radio message contains one or more data packets of the same type. The layout of the message is as follows. The first field is the type ID (1 byte) followed by the rank

of the sender node. The rank is stored in zero or more bytes depending on the flooding policy that corresponds to the type ID. After these two fields come the data packets. On the selected platform the number of data packets is not included in the message, because it can be calculated from the length of the radio message and the type ID. The priority of the data packet is not transmitted, as it is maintained locally and separately by each of the nodes that participate in the routing. This compact representation keeps the number of extra bytes at the absolute minimum, which allows several data packets to be aggregated into a single radio message. The message layout on the TinyOS platform is depicted in Fig. 2.

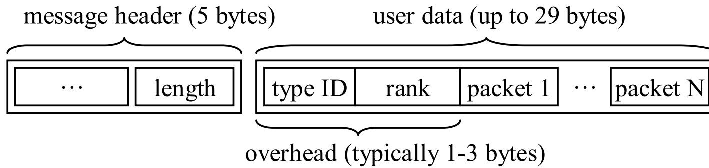

Fig. 2. The message layout on TinyOS. The overhead imposed by the DFRF is typically 1-3 bytes depending on the choice of flooding policy.

4.7 The Data Packet Table

The DFRF engine maintains a table for each type of data packets. This table includes the data packets and their priorities. This table holds at most one data packet from any family of analogous packets at any given time. Currently, this table is held in a fixed size array, but a hash table based implementation (based on the unique first bytes of data packets) is also possible. If a data packet has priority 255, it is considered invalid and the corresponding slot free. The engine has three basic activities: broadcasting and receiving radio messages, and aging data packets in the table.

When a message has been sent, the engine invokes the *sent* method to calculate the new state for each data packet contained in the message. Then it selects the next batch of data packets. It looks for the highest (lowest number) even priority data packets and packs them into a radio message buffer until it is full. Then it obtains the current rank of the node from the flooding policy and passes the radio message buffer to the radio stack for transmission. The engine stops sending messages if there are no more even numbered data packets in any of the tables.

When a new radio message is received, the engine first identifies the data type of the packets contained in the message, then invokes the *accept* method of the corresponding flooding policy to determine if further processing is necessary. If so, it unpacks each data packet contained in the message. For each packet, it locates an analogous data packet in the table. If there is no match, then the user of the flooding algorithm is notified of the newly arrived data packet via the *receive* method. The engine then finds a place for this packet by evicting an existing packet with the lowest priority from the table. Note that this selection includes available free slots as their priority is 255, the lowest. This evicted packet is overwritten by the newly arrived data packet with priority 0. Once the packet (or an analogous packet) is in the table, the *received* method of the flooding policy is invoked to calculate the new state of the packet, and the next packet in the message is considered.

Finally, the DFRF engine periodically ages all valid data packets in the table by invoking the *aged* method of the flooding policy.

4.8 Initialization

Since the type description and the corresponding policy of data packets are not passed around in radio messages, all nodes in the network (or that part of the network that routes a particular type of data packet) must initialize the DFRF engine with the same configuration for each data type. This configuration consists of the type ID, the length and the unique length of the data packet, and the selected flooding policy. Given that the target platform does not support dynamic memory allocation, the configuration includes the address and length of a user provided memory buffer where the engine will store the data packets. The engine keeps track of all registered data types and it can route data packets of different types concurrently. Typically, the types of data packets do not change during the lifetime of the application. Nevertheless, it is possible to register and unregister configurations dynamically.

4.9 Sending and Receiving Data Packets

The user of the directed flood-routing protocol interacts with the DFRF engine. When the user wants to send a data packet it simply passes it to the *send* method of the DFRF engine. Similarly to the case of data packets received via the radio, the engine first checks if an analogous data packet is already in the data table. If yes, then it simply returns (with an error code) because this packet is already being transmitted. If it is not in the table, then it evicts an already existing packet with the lowest priority from the table, as described before, and inserts the new data packet with priority 0. The actual transmission and life-cycle management is taken care of by the engine.

The *receive* event is fired by the DRFR engine to notify the user of the arrival of a new data packet. This event is fired exactly once for each family of analogous data packets, at the time when the packet was inserted into the table. Unlike in other routing algorithms, the *receive* event is fired at each intermediate node towards the target. This allows the user to modify or even drop the data packet enroute to the destination, a critical feature used in smart data aggregation protocols. We will present examples exploiting both of these features in the following sections. Note that this notification scheme does not complicate the use of the routing protocol, as the user can easily consult the particular routing policy at each node to check if it is the true destination of the packet.

The application component implementing the *receive* method gets a pointer to the data packet as a parameter and returns a boolean value. The received pointer can be used to read the data and possibly update its content (other than the first unique length bytes that must not be changed). If the *receive* method returns false, the engine drops the newly arrived data packet by not inserting it into its table. Otherwise, the data packet enters its life-cycle on this node, as described in Section 4.7.

5 Flooding Policies

Flooding policies have two central functions. First, they define the meaning and compute the value of the node rank. Second, they implement the state machine that gov-

erns the life-cycle of individual packets on every node. Flooding policies can be classified by either of these two traits. We can speak of, for example, broadcast policies where the node rank is vacuous (an empty array), or energy-aware policies where the actions of the state machine depend on the available power of the node and its neighbors. We grouped our selection of routing policies according to their definition of rank.

5.1 Broadcast Policy

The broadcast policy is used to route data packets from an arbitrary source node to all nodes in the network. A data packet is rebroadcasted one or more times at every node until all nodes received it with a high probability. There are several variations where the target area is limited in an application specific manner.

The node rank in the broadcast policy is void. There are several ways intermediate nodes can retransmit data packets. First, we present the simplest version where each intermediate node retransmits the data packet exactly once, as soon as possible. The nodes remember each data packet as long as possible to avoid receiving the old packet and classifying it as new. The corresponding state machine is depicted in Fig. 3.

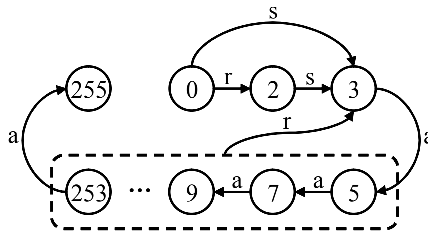

Fig. 3. The state machine of the broadcast policy.

Each circle represents a state. The states are numbered by their unique priority, from 0 to 255, but possibly not all of them are used. State 0 and state 255 are always the initial and terminal states, respectively. The arrows represent state transitions. The label of the arrow describes the corresponding type of event: ‘s’ for *sent*, ‘r’ for *received*, and ‘a’ for *aged*. State transitions that do not change the state are not shown. For example, the *aged* event does not change the state of the machine in states 0 and 2. Arrows originating from a composite state, a dashed rounded rectangle, represent transitions from each of the contained states. Recall that a data packet is eligible for transmission only in even numbered states.

A data packet always starts its life-cycle in state 0, either because the packet originates from this node (the user called the *send* method of the engine), or when it is received for the first time by this node. If it is the latter, then its state is immediately changed to state 2 by the flooding policy, because we want packets originating from this node to have higher priority (i.e. 0) than those that we received from another node. Once the packet is in either state 0 or 2, we wait until it gets selected and transmitted by the engine. After transmission, we enter state 3. The sequence of states, starting from 3 up to 255, is used to remember the same packet for 126 aging actions (63 seconds in the current implementation) before dropping it. If during this period the node receives the same packet again, we start counting again from state 3. Note

that in general this procedure does not prevent a data packet getting into an infinite cycle in a large dynamic network. However, the user can terminate this broadcast when handling the *receive* event.

As an application of the broadcast policy, we outline how to measure the hop-count distance from a root node to all other nodes in the network. The data packet shall contain a field for the "current" hop-count, and possibly others for the node ID of the root, etc. The unique part of the packet should not include the hop-count field. When the root initiates the network-wide broadcast, it fills in 0 for the hop-count in the packet. Upon receiving a data packet of this type, the application code should increment the current hop-count value in the *receive* event. The DFRF engine will not change this value, even if it later receives an analogous message with a different hop-count value, and will retransmit it with the incremented value. To get a more valuable estimate of the hop-count distance, the application should measure the hop-count distance from the root several times and the nodes should use the average of the measured values.

The range of the broadcast can also be limited in a similar way. For example, the root enters the required maximum number of hops into the hop-count field of the original message. Upon receiving the message, the hop-count fields needs to be decremented. If it reaches zero, then the *receive* method should return false, which will terminate the retransmission of the packet.

We found this basic policy to work very well on the Mica2 platform for planar networks with average degree of five or higher. This can be attributed to the sensible connectivity of the network and to the excellent radio collision avoidance of the radio stack. However, the same policy does not perform well on linear networks or on platforms with erratic radio collision avoidance. Nevertheless, this can be overcome by retransmitting each data packet two or more times on each node, with random delay in between. One particular implementation of this modified broadcast policy is shown in Fig. 4.

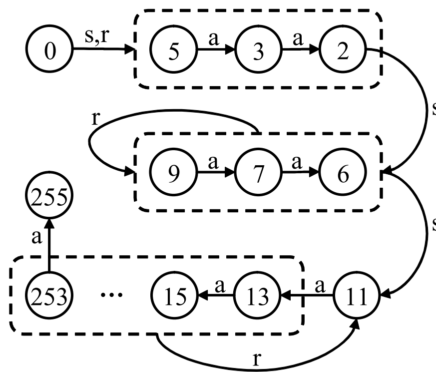

Fig. 4. The state machine of the reliable broadcast policy.

Arrows pointing to a composite state stand for transitions that enter one of the contained states based on a random choice. There are several subtle design choices that make this broadcast policy more robust than the one pictured in Fig. 3. First, note that the composite states (2,3,5) and (6,7,9) facilitate the random delay via the aging event. Not only does it wait for a random number of aging events, but also the aging

events are executed asynchronously in the network. Second, the priority value is decreased inside these composite states, because if the engine is short of memory, we want to keep those packets that we can retransmit sooner. What is more, the ‘r’ self-loop at the composite state (6,7,9) implements a random backoff functionality. Observe that the ‘s’ arrow to state 11 does not come from state 6, the only even numbered state in (6,7,9) allowing retransmission, but from the whole composite state. The reason is that the engine can select the packet in state 6 for transmission, pass the radio message buffer to the OS, receive an analogous message that restarts the backoff delay, and only then does the OS complete the transmission of the previously packed message. As a final point, the source node of the broadcast transmits the packet three times in contrast to relaying nodes, which transmit every packet only twice.

5.2 Gradient Convergecast

Convergecast policies are used to route data packets from all nodes of the network to a selected node, called the root. Intermediate nodes rebroadcast a data packet zero, one or more times until it is received from a node “closer” to the root than the current node. In the gradient convergecast policy, being closer means that the hop-count distance from the root is smaller. Thus, the rank of each node is the hop-count distance from the root, and the hop-count distances of the sender and receiver are compared. The same data packet can reach the root through several different paths, always descending in the gradient field. This guarantees robustness and fast message delivery at the expense of higher communication overhead. The data packet typically arrives at the root first through unreliable “long” links, then through more reliable “short” links.

The hop-count distance can be calculated, for example, by an application of the broadcast policy, as described in Section 5.1 above. The gradient convergecast policy implements this functionality and allows the user to set and query the current root in the network. Data packets of several types can share the same gradient field, or different gradient fields can be computed if there are multiple roots in the network. The overall cost of calculating the gradient field is rather large; possibly several network-wide broadcasts. However, once the field is calculated, it takes very little memory space, 1 or 2 bytes, to store it.

Fig. 5 depicts the state machine of the gradient convergecast policy. The receive action has been split into two separate actions: ‘r^-’ and ‘r^+’ for messages received from nodes closer to and further from the root than the current node, respectively. Note that nodes with the same rank have been explicitly excluded from this list, because we want to direct the flooding as mush as possible by preventing the same data packet to spread among nodes having the same hop-count distance. The policy avoids this case by returning false in the *accept* method for radio messages with the same rank as of the receiving node (see Section 4.7).

Each node retransmits a data packet up to three times, with two and one aging actions in between. The delay between the first and second transmissions is relatively long but it leaves the nodes receiving the first transmission enough time and radio channel bandwidth to retransmit the packet. The transition labeled by ‘r^-’ on the left hand side in Fig. 5 implements implicit acknowledgment in the following way. If node A sends a packet that is received by node B that is closer to the root than A, and then B rebroadcast this packet, which is received, among others, by A, then the state

of the packet on A becomes 7 and A will not retransmit the packet again. The policy remembers each data packet for a certain time period since the last time it was received from a node further from the root. This is enough, because even if the node receives an analogous packet from a node closer to the root later, it will immediately enter state 7 again.

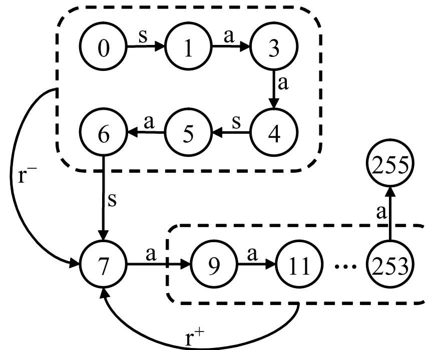

Fig. 5. The state machine of the gradient convergecast policy.

Clearly, this policy does not guarantee message delivery, but best effort only. This is not a serious limitation for most wireless sensor network applications because they have to prepare for message loss as the result of failing nodes and unreliable links. However, a variation of this policy can guarantee message delivery in connected networks provided the hop-count distance gradient field remains accurate. This variation retransmits the packet on each node other than the root until it is received from a node closer to the root.

The gradient convergecast policy yields a very fast and robust routing protocol to deliver messages to a root node, but at the expense of significant message overhead. Depending on the topology of the network, the number of transmissions during the routing of a single data packet can grow as the square of the distance between the sender and the root.

5.3 Fat Spanning Tree Convergecast

The major shortcoming of the gradient convergecast is its message overhead. The optimal solution, with respect to the number of messages, would be to route the data packet along a spanning tree towards the root. However, this algorithm is inherently fragile: the radio links are not reliable causing message loss in any fixed path. Moreover, a single node failure close to the root can cut off a large portion of the network from the root. The speed and robustness of the gradient convergecast and the low message overhead of the spanning tree routing protocol can be combined in the following way. Instead of utilizing a single path starting from the source node towards the root, define a small neighborhood of this path and flood the data packet in this "*lane*". The lane can be defined as all nodes one hop away in the spanning tree from the nodes of the path. The resulting routing policy is called the fat spanning tree con-

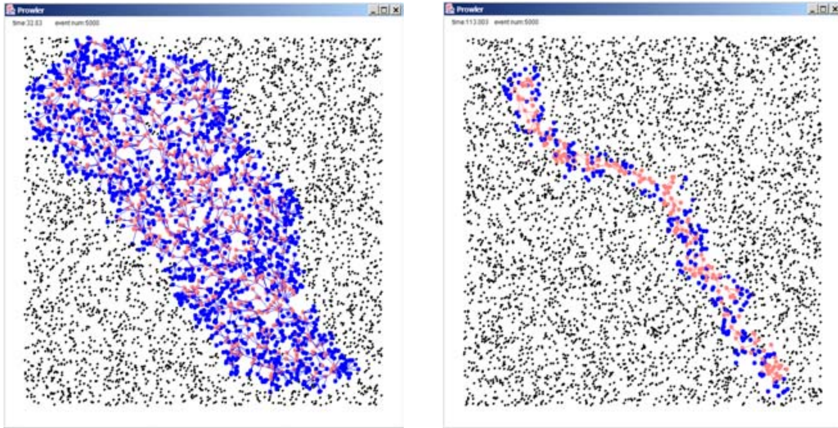

Fig. 6. The message overhead of the gradient and fat spanning tree converge cast policies in a 5000-node network. Dark blue and light red colors indicate nodes that received or received and retransmitted the routed message, respectively.

vergecast. The message overhead of the gradient and fat spanning tree policies is illustrated in Fig. 6, where a single data packet is routed from a node in the bottom right corner to the root in the top left corner. Red nodes retransmitted the packet, while the blue ones received it but did not retransmit it.

This particular definition of the lane allows a strikingly simple implementation of directed flood-routing in the lane with minimal storage requirement. Each node has to know the node IDs of its parent, grandparent, great-grandparent and great-great-grandparent. The node rank is simply the node ID of the grandparent. The relationship between the sender and the receiver of a radio message can be computed by the receiver from the rank of the sender, which is stored in the message, as follows:

(1) If the rank of the sender is the node ID of the receiver or its parent, then the sender is further from the root than the receiver. The corresponding event will be denoted by ‘r^+’.
(2) If the rank of the sender is the node ID of the grandparent of the receiver, then the sender is at the same distance from the root as the receiver. These types of message are also denoted by ‘r^+’.
(3) If the rank of the sender is the node ID of the great-grandparent or its parent of the receiver, then the sender is closer to the root than the receiver. The corresponding event will be denoted by ‘r^-’.
(4) If the rank of the sender is none of the above, then the receiver is either not in the lane of the source, or more than two steps away from the sender. In both cases we ignore the message by returning false in the *accept* method of the policy.

The spanning tree can be constructed and the node IDs of the four ancestors found by a simple network-wide broadcast, or by other methods. Finding the spanning tree that best supports directed flood-routing is possibly a challenging problem and is not addressed here.

Once the spanning tree is formed and the ‘r^+’ and ‘r^-’ receive events defined, we can reuse the state machine of the gradient convergecast policy (see Fig. 5) for the

spanning tree convergecast policy. The performance of the spanning tree convergecast for arbitrary networks will be similar to that of the gradient convergecast for essentially linear networks. In particular, the number of messages required to route a data packet from the source to the root is proportional to the hop-count distance of the source from the root.

6 Conclusion

We have introduced the directed flood-routing framework for wireless sensor networks. We demonstrated that the state machine based language describing routing policies is rich enough to capture a wide variety of existing flood-routing protocols. The supporting engine and flooding policies were implemented for TinyOS and extensively tested on the Mica and Mica2 platforms. The gradient convergecast policy was used in an acoustic shooter localization application to route acoustic events back to a base station. A network of 60 motes covering a 100 by 40 meter urban area with diameter of 10 hops was used to evaluate the performance of both the routing and shooter localization algorithms. Typically, 25-30 motes were triggered by a shot, half of them managed to report their events in the first second, and the other half in the next second.

There are several research opportunities in directed flood-routing in general and flooding policies in particular. For example, it seems possible to design convergecast flooding policies that implement flow control by delaying retransmission of data packets if nodes closer to the root are overloaded. Another challenging research area is the study of topology changes with respect to convergecast policies. For example, is it possible to dynamically update the gradient field or the spanning tree if the root node is mobile?

The state machines of flooding policies can clearly be optimized for different hardware platforms and network configurations, as well as for speed, reliability and power consumption. Since these state machines have a limited number of actions and are relatively small, it seems possible that they can be mechanically optimized utilizing a simulator to compute the fitness of policies.

Acknowledgment

The DARPA/IXO NEST program (F33615-01-C-1903) has supported, in part, the activities described in this paper.

References

1. J. Hill and D. Culler, “Mica: A Wireless Platform for Deeply Embedded Networks,” IEEE Micro., vol 22(6), Nov/Dec 2002, pp 12–24.
2. J. Hill, R. Szewczyk, A. Woo, S. Hollar and D. C. K. Pister, “System architecture directions for networked sensors,” ASPLOS, November 2000.
3. A. Woo and D. Culler, “A Transmission Control Scheme for Media Access in Sensor Networks,” Mobicom 2001, July 2001, Rome.

4. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler, “The nesC Language: A Holistic Approach to Networked Embedded Systems,” Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.
5. Gy. Simon, M. Maroti, A. Ledeczi, Gy. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai and K. Frampton, “Sensor network-based countersniper system,” accepted to SenSys 2004.
6. D. B. Johnson and D.A. Maltz, “Dynamic Source Routing in Ad Hoc Wireless Networks.” In Mobile Computing, Chapter 5, pages 153–181, Kluwer Academic Publishers, 1996.
7. C. E. Perkins and E. M. Royer, “Ad-hoc On Demand Distance Vector Routing.” In WMCSA'99, February 1999.
8. V. D. Park and M.S. Corson, “A highly adaptive distributed routing algorithm for mobile wireless networks.” In Proceedings of IEEE INFOCOM, April 1997.
9. M. R. Pearlman and Z.J. Haas, "Determining the Optimal Configuration for the Zone Routing Protocol," IEEE JSAC, special issue on Ad-Hoc Networks, Vol. 17, No. 8, August 1999.
10. B. Karp and H. T. Kung. “GPSR: Greedy Perimeter Stateless Routing for Wireless Networks.” In IEEE MobiCom, August 2000.
11. Y. B. Ko and N. H. Vaidya. “Location-Aided Routing (LAR) in Mobile Ad Hoc Networks.” In IEEE MobiCom 1998, October 1998.
12. T. He, J. A. Stankovic, C. Lu and T. F. Abdelzaher, “SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks”, In International Conference on Distributed Computing Systems (ICDCS 2003), Providence, RI, May 2003.
13. F. Ye, G. Zhong, S. Lu and L. Zhang, “GRAdient Broadcast: A Robust Data Delivery Protocol for Large Scale Sensor Networks,” accepted by ACM WINET (Wireless Networks).
14. M. Lin, K. Marzullo and S. Masini, “Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small networks.” UCSD Technical Report TR CS99-0637.
15. D. Braginsky and D. Estrin, “Rumor Routing Algorithm for Sensor Networks,” ACM WSNA, 2002.
16. Yi Shang, M. P. J. Fromherz, Y. Zhang and L. S. Crawford: “Constraint-based Routing for Ad-hoc Networks.” IEEE Int. Conf. on Information Technology: Research and Education (ITRE 2003), Newark, NJ, USA, Aug. 2003, pp. 306–310.
17. Osterweil, E. and Estrin, D, “Tiny Diffusion in the Extensible Sensing System at the James Reserve,” May 2003; see www.cens.ucla.edu/~eoster/tinydiff/.
18. Jie Wu and Fei Dai: “Broadcasting in Ad Hoc Networks Based on Self-Pruning.” IEEE Conf. on Computer and Communications Societies (INFOCOM), 2003.
19. B. Williams and T. Camp: “Comparison of broadcasting techniques for mobile ad hoc networks.” Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing (MOBIHOC) June 2002, pp. 194–205.
20. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer and D. Culler, “The Emergence of Networking Abstractions and Techniques in TinyOS,” NSDI, 2004.
21. T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru and A. Wood, “EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks,” International Conference on Distributed Computing Systems, Tokyo, Japan, March 2004.
22. S. Madden, J. Hellerstein, and W. Hong, “TinyDB: In-Network Query Processing in TimyOS,” Intel Research, IRB-TR-02-014, October 2002.
23. P. Bonnet, J. Gehrke, and P. Seshardi. Towards sensor database systems. In 2nd International Conference on Mobile Data Management, pp. 3–14, Hong Kong, January 2001.