Visibility: A New Metric For Protocol Design

Megan Wachs[†], Jung Il Choi[†], Jung Woo Lee[†], Kannan Srinivasan[†],
Zhe Chen[⋆], Mayank Jain[†], and Philip Levis[†]
[†]Computer Systems Laboratory, Stanford University, Stanford, CA
[⋆] Columbia University, New York, NY

Abstract

This paper proposes a new sensornet protocol design goal: visibility. Visibility into behaviors at the network level will simplify debugging and ease the development process. We argue that increasing visibility is the responsibility of the network protocols themselves, and not solely the responsibility of existing debugging tools. We describe a quantitative visibility metric to evaluate and compare protocols, where visibility is defined as the energy cost of diagnosing the cause of a behavior in a protocol. The design and evaluation of Pull Collection Protocol, a novel multi-hop collection protocol, is an example of how to design for visibility without sacrificing throughput or node-level fairness. We also describe our optimizations for an existing protocol, Deluge, to increase its visibility and efficiency.

1 Introduction

Traditional metrics for sensornet protocol design focus directly on performance. Developers often design for high fairness, low latency, and high throughput. However, real world sensornet deployments often fail to meet these goals, even after extensive simulation and testbed experiments.

In order to gain a better understanding of the difficulties faced by sensornet developers, we surveyed the literature on existing sensornet deployments to identify the leading causes of sensornet failure. Overwhelmingly, the authors could not concretely identify the causes of failure in real-world deployments. Developers need greater visibility into the causes of behaviors in network protocols, in order to isolate the causes of failure and address them. If we could trace all packet transmissions and receptions and internal memory state, it would be easy to identify the causes and fix them. However, providing visibility is tough for sensornets due to limited storage and energy on nodes. Limited storage limits the amount of state we can store locally at nodes. Limited energy at nodes restricts nodes' computation and communication. Many tools and methods have been developed to address this need, which we survey in Section 2. These tools are extremely useful and we do not wish to discourage their use. However, these tools generally require extracting information from the network, requiring energy. We argue that it is the job of the protocols themselves to help these tools without adding layers to an obfuscated network. The protocols should provide high visibility at a low cost.

We propose a new design metric for sensornet protocols. The idea behind this visibility metric is that protocols should aim to

> "Minimize the energy cost of diagnosing the cause of a failure or behavior."

Since we argue that protocols should be designed for high visibility, we must define a quantifiable metric for the visibility of a protocol. Section 3 derives an equation for the visibility cost of a protocol. This equation requires an estimate of the probability distribution of failures, which may vary with many factors. We describe one way of measuring the probability distribution in a controlled environment to get an initial idea, and explain how this can be extended to a real-world deployment. With this metric, protocol designers can make design choices based on the effect on the visibility of their protocol.

One way to improve the visibility of a protocol is to identify and eliminate possible causes of failure. A reconsideration of common protocol practices reveals that their design introduces causes of failure that are not inherent to their purpose. For example, a collection tree protocol may drop packets after a fixed number of retransmissions, though all packets have the same destination. At the root, this adds an additional cause of packet loss that may be hard to distinguish from other causes. In Section 4, we reconsider the assumptions in a collection tree protocol and design a new collection protocol, Pull Collection Protocol (PCP). PCP uses a novel technique to eliminate ingress drops and collect data fairly from the nodes in the network without maintaining per-child state. PCP is designed for high visibility, yet it shows that that visibility is not always orthogonal to high performance, as it achieves high throughput and node-level fairness.

Visibility comes at a cost. An extreme example is when we design a collection protocol that sends no packets. While such a protocol has very high visibility because we know that the failure was because no packets were sent, it has zero throughput. Surprisingly, in the examples we show in later sections, improving visibility can also lead to an improvement in performance. In Section 5, we describe how a dissemination protocol, Deluge [9], achieves better efficiency and lower latency by a consideration of its visibility.

Section 6 describes future directions for our work and concludes.

2 Prior Work

Sensornet deployments often perform much worse than expected, even after extensive simulation. To deal with this issue, many tools have been designed to provide information about what is happening in a sensor network. To gain a better understanding the causes of real-world network failures, we reviewed the literature on existing deployments and surveyed sensornet developers to understand the difficulties and failures encountered in deploying a successful sensornet.

2.1 Real-World Deployments

The LOFAR-agro deployment [13] encountered failures on many levels. One main cause of failure was the conflicting goals of protocols that were used. MintRoute [27] expected the nodes to be in a promiscuous mode so that link qualities could be available for routing, while T-MAC [23] avoided snooping to save energy. The developers also reported that a malfunctioning Deluge [9] quickly exhausted the batteries on the motes. A system level failure prevented logging of data, so many other causes were unidentifiable as very little data was logged. The authors reported an overall data yield of only 2%.

A surveillance application [1], "A Line In The Sand", also showed the detrimental effects of misbehaving nodes. A few nodes that constantly detected false events were a serious problem, as they affected the entire network. As a result, their batteries were also exhausted sooner than expected. Sensor desensitization and extreme environmental conditions also led to problems with their network.

In "Unwired Wine", a vineyard monitoring application [2], the authors identified the failure of the backbone network as the cause of most of the packet losses. This failure was due to unknown causes. While their lab deployment had 92% data yield the actual deployment, even with 5x redundancy, had only 77% data yield.

In an industrial monitoring deployment in the North Sea [12], the authors found the failures to be highly spatially correlated and attributed the failures to a node with a hardware problem.

In the Great Duck Island deployment [20] the developers were able to analyze the data collected from their application to diagnose some causes of packet loss. By using the sequence numbers of the received packets, the authors were able to determine that clock drift caused their nodes to become unsynchronized. As a result, nodes tried to transmit at the same time and a lot of packets were lost to collisions. They also correlated humidity readings with node failures – another cause of packet loss. They could not identify any other causes for packet failure.

Although the developers from the above deployments were able to identify some causes of failure, they could not identify all of them with certainty. While other deployments we surveyed [3, 7, 11, 19, 22, 24, 26] found similar causes of failure as above, many developers reported the causes of failure of their deployment to be unknown or hypothetical. This leads us to the need for visibility into sensornet systems.

2.2 Visibility Tools

Visibility at the system level is not a new concept, and the operating systems field has addressed it well. Dtrace [4] is a tool designed to dynamically troubleshoot applications and the operating system on which they run. This dynamic tool provides real-time, system-level visibility on demand.

There are also numerous efforts to improve visibility for sensor network systems. EmStar [6] is designed to provide visibility on a single node by using user-level device callbacks to interact between different modules in a system. This allows the same code to be run in simulation or in emulation, and gives the user read/write access to the state of the emulated network. EmStar assumes a back-channel so that all states of a node can be reliably sent to a centralized node. Thus, EmStar provides visibility into the workings of a software system, similar to DTrace. Marionette [26] allows user-friendly lightweight RPC in a multihop network. Users can peek and poke all variables on motes as well as execute functions remotely.

While EmStar is in general a pre-deployment debugging tool, Sympathy [17] and SNMS [21] are pre/post-deployment tools. They do not require a wired back-channel to operate. SNMS is a management system for sensornets which runs in parallel with, and independent of, an application. It requires the nodes to passively store state and then actively communicate only when queried. Sympathy's design reduces the amount of states a node has to store compared to SNMS. Sympathy identifies and localizes failures by making the sink actively and passively collect statistics. A Sympathy module is present at the sink and at all nodes that interact with local components. The statistics thus collected are reported to the sink Sympathy module which performs diagnostics for the cause of failure, if any. The sink Sympathy module performs this diagnosis using a decision tree approach, an approach we adopt in our methodology.

Dtrace and EmStar provide system-level visibility. However, when failures occur in a network of systems, it is important to know which system to examine to identify the cause of failure. Such network visibility may be achieved by with a tool like Sympathy, which has a root node that collects statistics from the network and diagnoses the cause. In this paper, however, we discuss how to design protocols such that they enhance network visibility. Such a design does not preclude the use of a tool like Sympathy, though protocols designed for visibility can make Sympathy-like tools much simpler, more accurate, and less expensive to use in terms of energy. In the next section, we describe a metric for determining how visible a protocol is.

3 Quantifying Visibility

In order to compare how well protocols follow the visibility principle, we must have a quantifiable metric. The metric will define the energy cost of diagnosing the cause of a behavior. In this section we derive an equation for the visibility cost of a protocol, then calculate the visibility of an actual protocol, MultihopLQI.

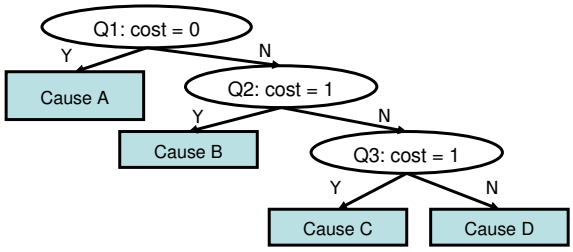


Figure 1. Example decision tree for calculating visibility

3.1 The Visibility Metric

To calculate incurred energy for visibility, we apply Sympathy's decision tree approach: energy for diagnosing the cause of a behavior of interest can be quantified by measuring the energy cost of traversing the decision tree for that behavior. Any diagnosis tool may be used in addition to the protocol to send diagnostic queries and responses. At each node of the decision tree, the diagnosis tool must answer a question, where each question may incur a different energy cost. We define the cost of traversing the decision tree as the sum of the cost of answering each question q, weighted by the probability that q will have to be answered.

Figure 1 shows an example decision tree. In the example, the questions have arbitrary costs. In actuality, these costs are due to energy required to transmit or store bits of information to allow traversal of the tree. The energy required for answering a question is a factor of many variables: which node the question is about, how many bits are involved in answering the question, and how many radio transmissions are required to get those bits to a diagnosing node. Additionally, some questions may be answered with queries, while others may be answered by piggybacking information on data packets to the root. For simplicity, we assume only queries, and divide questions into those that can be answered locally at a diagnosing node, and those that require queries or additional information. We use C to represent the cost of answering a question that requires transmissions external to the protocol.

This gives us a very simple equation for the energy required to answer a question q:

$$QuestionEnergy_q = C \cdot I_q \tag{1}$$

I_q is an indicator function that is 0 when the question q can be answered in the diagnosing node and 1 when it requires queries into the network.

Since Equation 1 gives the energy for answering a question q, the energy for diagnosing a specific cause k can be given as:

$$DiagnosisEnergy_k = C \sum_{q^{th}\text{ question}} I_q Q_{qk} \tag{2}$$

Q_{qk} is an indicator of whether question q must be asked to diagnose cause k. In calculating the final visibility cost, we should weight the energy required to diagnose each cause by the cause's likelihood. Ideally, this would incorporate a probability distribution of causes for each node in the network. This is difficult, as the probability distribution would differ depending on the location of a node; for example, in a tree protocol, a node close to the root will experience more ingress drops (incoming packet drops due to queue overflow) than leaf nodes, who will experience no ingress drops. However, if we calculate the probability distribution as an average over the entire network, then using a single value of p_k for all nodes i is a reasonable simplification.

The final visibility cost would be given by the sum of the cost of diagnosing each cause weighted by its probability, as in Equation 3:

$$VisibilityCost = C \sum_{k^{th}\text{cause}} (p_k \sum_{q^{th}\text{question}} I_q Q_{qk}) \tag{3}$$

Equation 3 is equivalent to summing the cost of answering each question, weighted by the probability that the question will need to be answered.

To minimize the visibility cost, it is desirable to create an optimal decision tree, based on the energy cost of the questions and the probability of the causes. We would like to rule out likely causes before spending energy diagnosing unlikely causes. While by no means a trivial problem, the construction of decision trees is a well-studied problem [16].

As a simple example, consider the decision tree in Figure 1. The four causes (A, B, C, and D), are equally probable. Distinguishing cause A has no cost, while distinguishing each other cause requires a query into the network. The cost of traversing this tree is $C((0.25)0+(0.25)1+(0.25)2+(0.25)2)$, or $1.25C$.

As time passes, the probability distribution of causes will most likely change. Thus, it is possible for the visibility of a protocol to vary over time, just as other sensornet metrics vary with time and network configuration.

Equation 3 gives insights into how to increase visibility. The terms which vary across protocols are I_q and p_k. In other words, the probability distribution of causes that can not be diagnosed locally determines the visibility of protocols. If a protocol designer tries to minimize these types of causes, they will increase their protocol visibility. The following section describes ways to calculate these metrics in the design phase of a protocol.

3.2 Measuring Visibility

The previous section explained that we need to know the causes of failure in a protocol. In addition, having knowledge about the probability distribution of those failures can help us shape the decision tree in an optimal way. For example, we would prefer to ask inexpensive questions that rule out the most likely causes before pursuing less likely causes. In this section, we calculate the visibility of a multihop protocol. We explain the intermediate steps required to get an estimation of the probability distribution.

3.2.1 Experiment with Back-channels

The first step in calculating the visibility metric is to find a probability distribution of causes of a behavior. In a real world deployment, this may not be possible, as getting this

Message Type	Reason For Sending
Packet generated	A source generates a packet and submits it to the network successfully.
Packet forwarded and acked	A mote forwards a packet which is acked
Packet forwarded and not-acked	A mote forwards a packet and hears no ack
Reboot	A mote reboots
Duplicate suppression	There is duplicate suppression due to a match in the forwarding queue or cache
Ingress drop	An incoming packet is dropped because a queue is full
Egress drop	The retransmit threshold is reached.

Table 1. Messages sent by motes in experiments designed to measure the probability distribution of causes of packet loss in MultihopLQI.

information from the network may be as hard as getting data from the network. Thus, the probability distribution is a good thing to measure in simulation and in testbeds, where wired back-channels are available. Of course, the probability distribution will vary with the environment and over time. However, this methodology gives an estimate for visibility when designing protocols.

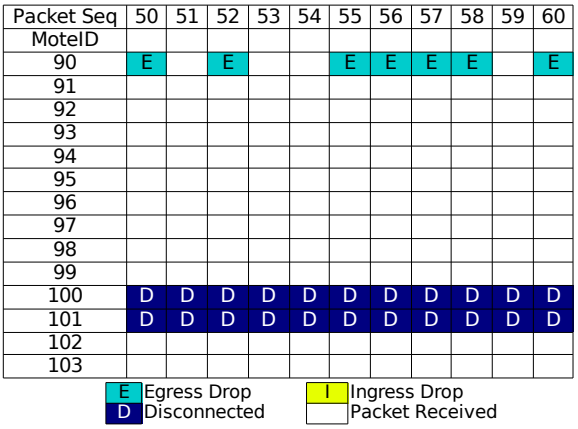
To get as clear a view as possible into the network, we used a wired back-channel to log network events such as packet reception and transmission, as well as possible failure causes such as ingress and egress drops. We wrote a simple analysis program to process the collected information to account for every missing packet. The data collected over the UART may itself have been lossy, so the analysis tool inferred the missing information from what it did receive.

We ran MultihopLQI, an updated version of the MintRoute protocol [27] on the Motelab testbed at the Harvard University [25]. MultihopLQI is a component library in TinyOS 2.x [8]. We programmed 31 motes to periodically generate packets. The payload of the packet is a 4-byte internal sequence number, which is different from the network level sequence number. The network level sequence number can wrap around, but our analysis tool requires that two packets be uniquely distinguishable. Each mote sends notification messages for the events listed in Table 1. We repeated the test with different packet generation rates to study the effect of sending rates on the distribution of causes.

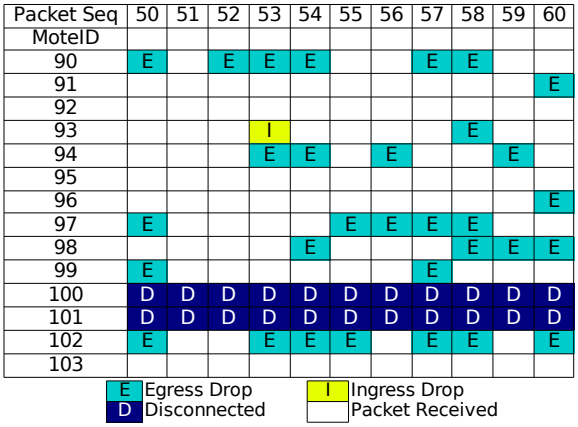
3.2.2 MultihopLQI Visibility

Figure 2 shows a sample of the output of the analysis tool for MultihopLQI running at different data generation rates. At low data rates (Figure 2(a)), most nodes transmit their packets successfully, but node 90 suffers from many egress drops. When we stress the network with a higher data rate (Figure 2(b)), we observe more ingress and egress drops.

The analysis program could not identify the cause of about 1% of the packet drops (depending on the sending rate). A possible cause in such cases was lower-level failures. For example, we observed that a sender received an ack from a receiver, but the receiver did not confirm the reception of the packet at its network layer. It is possible that the radio received the packet and sent an ack, but the operating



(a) Packet Generation Rate = 0.5 pps



(b) Packet Generation Rate = 20 pps

Figure 2. An example of causes of packet loss MultihopLQI protocol during a 40-node testbed experiment. Each row corresponds to a node in the network, and each column corresponds to a packet in the sequence. The reasons for packet loss vary as the sending rate increases from 0.5 pps (a) to 20 pps (b).

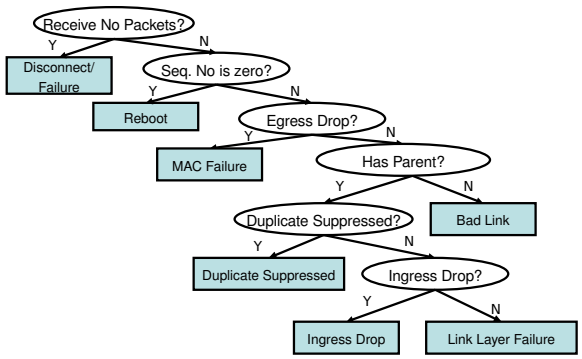


Figure 3. MultihopLQI Decision Tree, optimized with knowledge of the probability distribution

Cause	0.5 pps	1 pps	2 pps	10 pps	20 pps
Disconnected	0.154	0.135	0.138	0.114	0.082
Reboot	0	0	0	0	0
Egress Drop	0.523	0.596	0.512	0.587	0.634
Ingress Drop	0	0	0	0.011	0.008
Duplicate Suppression	0	0	0.005	0.003	0
Bad Link	0.308	0.269	0.276	0.227	0.164
Lower Level Failure	0.015	0	0.069	0.061	0.112

Table 2. MultihopLQI Decision Tree Probability Distribution for experiments with different packet generation rates

system discarded the packet because the receive buffer was corrupted. In addition, UART messages could be dropped or corrupted at high data rates.

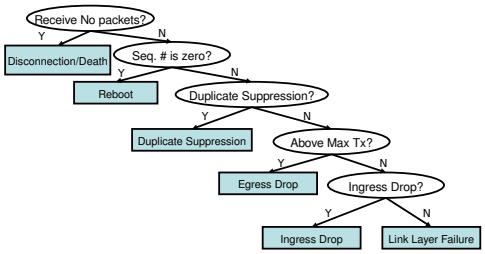
Table 2 shows the probability distributions of failure causes for different packet generation rates. Figure 3 shows a diagnosis tree for determining the cause of packet loss. The back-channel experiments enable the identification of most causes, but we cannot distinguish among collision, self-interference, and interference from external sources, which we classify as MAC-layer failures. In the decision tree in Figure 3, the first two questions each require no queries into the network (I_q is zero). The remainder of the questions require a query into the network (I_q is one). From Equation 3, the probabilities in Table 2, and the decision tree in Figure 3, the visibility energy cost for MultihopLQI sending at 2pps is $1.355C$. In the following section we will compare these values with a collection protocol designed for visibility, namely, Pull Collection Protocol.

4 Visibility as a First Principle

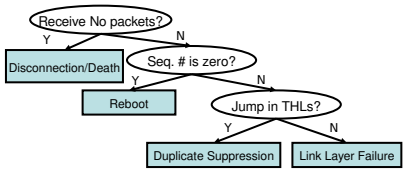
In this section, we examine how one might design a protocol based on the visibility principle. As an example, we consider the most common type of sensornet protocol: tree-based collection. We examine the possible failures for a tree-based collection protocol and see that many are not inherent and can be eliminated. With this idea, we describe the design of Pull Collection Protocol (PCP), a collection tree protocol designed for visibility from the beginning. We evaluate the resulting protocol in terms of visibility, as well as throughput, reliability, and node-level fairness, and compare it with protocols optimized for both throughput and fairness.

Disconnection	Temporarily or permanently broken link.
Destruction	Depleted batteries or permanent hardware failure.
Reboot	Software failure loses packets in RAM.
Egress drop	Retransmit threshold is reached.
Ingress drop	A packet is received when the queue is full.
Suppression	Temporary loops cause nodes to mistake looped packets as duplicates and drop them.
Link Layer Failure	A packet is thought to be transmitted successfully by the link layer but it is not.

Table 3. Causes of packet loss for collection tree protocols. Temporary disconnections can introduce huge latencies, which may or may not actually drop packets. For example, if a disconnected node thinks it has no parents, it will not encounter egress drops, but if it erroneously thinks it has parents, it will.



(a) Traditional Routing Protocol Decision Tree



(b) PCP Decision Tree

Figure 4. Decision trees for identifying causes of packet loss.

4.1 Pull Collection Protocol

As discussed in section 3, a key question for designers of collection tree protocols is why the network drops packets. Table 3 lists common causes of packet drop for traditional collection protocols. We can construct a decision tree to identify the cause of a lost packet, shown in Figure 4(a).

Our goal is to minimize the cost of traversing this decision tree. In designing a new collection protocol, PCP, we want to eliminate or efficiently diagnose the causes of packet drops. Our design eliminates or diagnoses the causes listed in Table 3 as follows:

Ingress drops: Ingress drops occur when parents receive and acknowledge packets, but have no buffer space to store the packets. This is a direct result of child nodes sending at a higher aggregate rate than the parent can drain its buffers. PCP uses a novel approach to limit ingress drops. Traditional push-based protocols (Figure 5) push data from sources up towards the sink. Rate-limiting information must be propagated to the children from parents before they can limit their

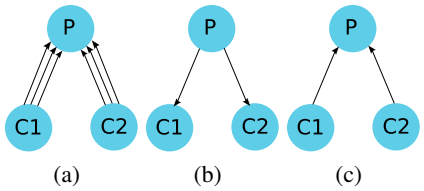


Figure 5. Traditional Push-Based Method for Rate Limiting: a) Children send data to parent at high rate. b) Parent sends rate-limiting information. c) Children send data to parent at reduced rate.

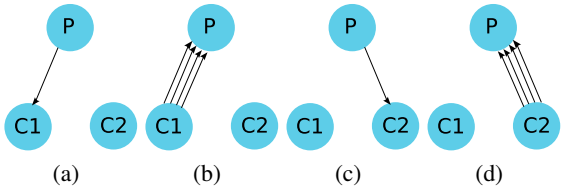


Figure 6. Pull-Based Method for Rate Limiting: a) Parent sends grant-to-send to one child. b) Child sends a burst of packets to the parent. c) Parent sends grant-to-send to another child. d) Child sends a burst of packets to the parent.

rate. For example, IFRC [18] sends information about queue lengths and sending rates to children and potential interferers. PCP takes a different approach, where sinks pull data from the network (Figure 6). When parents have space in their buffers, they send a request to a specific child for a burst of packets. Children, in turn, keep their buffers full by requesting from their own descendants. This pulling mechanism is described in detail in Section 4.2.2.

Egress Drops: Egress drops occur when a node attempts to transmit a packet too many times, and drops it in order to transmit another. This makes sense in routing protocols where packets may have many destinations, as a subset of destinations may be unreachable, but others are not. However, in a collection tree protocol, all packets have the same destination: the parent in the tree. Therefore, there is no reason to penalize one packet in favor of another. Because of this, PCP allows infinite retransmissions of a packet, thereby eliminating egress drops as a cause of packet loss. Low-quality links will no longer cause the protocol to drop packets, rather, the packets will only be delayed.

Reboot: PCP packets have a sequence number used by the protocol itself to identify and suppress duplicates. This sequence number is set to zero on reboot. Thus, if we unexpectedly start receiving packets from a node with sequence numbers starting at zero, we can identify that the node has been rebooted. Any time-correlated packet loss was likely due to the node dropping the packets stored in its buffers. This sequence number does wrap, so a sequence number of 0 indicates a reboot only when it was unexpected.

Disconnection/Death: If we receive no packets from a node, we can deduce that the node is dead or disconnected. The two differ only in that a disconnected node may eventually become connected as conditions change.

False Duplicate Suppression: Collection protocols often attempt to suppress duplicates, in order to prevent exponential growth of the number of packets in the network. To do this, they often use an origin-sequence number pair to identify packets. If a packet arrives with the same origin-sequence pair as one in a node's send cache, the node will drop the packet. This is problematic if a temporary routing loop occurs and a packet is legitimately sent to a node multiple times. To prevent this, we include a Time-Has-Lived field in the packet header. Thus, each packet is identified by a origin-sequence number-THL tuple. For the most part, this is enough to prevent false duplicate suppression, but the THL field may wrap around if a loop is long-lived. In that case, we will still observe packet loss. This loss will be accompanied by unusually high THL values in packets that have been around the loop many times, so we can identify this cause of loss at the root.

Thus, PCP shortens the decision tree into the one shown in Figure 4(b). PCP is able to traverse this tree without sending additional queries into the network.

The key concept for increasing visibility in PCP is the elimination of ingress drops. In the following section, we describe in detail the implementation of its pulling mechanism for eliminating ingress drops.

4.2 PCP Design

Figure 6 shows a simple example of pulling in a situation when a parent has two children, with no descendants. In reality, PCP nodes seek to pull fairly from their children based on their subtree size. This section describes the mechanisms PCP uses to do this.

4.2.1 Counting Descendants

Maintaining counts of subtree size is crucial to achieving fairness. For example, a child with no descendants should be asked to send less data than a child with many descendants. However, achieving accurate counts also requires per-node state which might be problematic. Instead of keeping accurate counts, PCP nodes use synopsis diffusion [15, 5] to approximate the size of their subtrees.

Each node maintains its own synopsis and a synopsis version number. The synopses are piggybacked on non-data packets and propagate quickly. It is important to consider only one synopsis from each node. To ensure this, packets also include a synopsis version number. When a node hears a synopsis from its child with the correct version number, it includes it in its synopsis calculation. Periodically, the nodes increment the version number and create a new synopsis. Nodes which hear a higher version number create a new synopsis and update their own version number, synchronizing to the highest. In our implementation of PCP, each subtree count is determined from seven 8-bit synopsis values. Nodes update their synopsis version number once per minute.

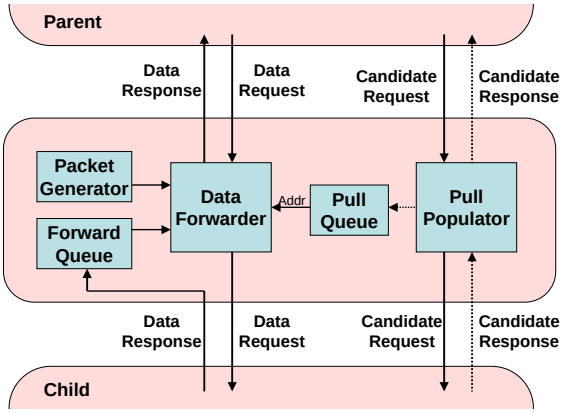


Figure 7. Component diagram of PCP and its control and data flows. Each node can act concurrently as a parent and a child, using the forwarding queue as a shared resource.

4.2.2 Pulling Mechanism

Earlier we explained how PCP can use a pull-based approach to eliminate ingress drops as a cause of packet loss. However, Figure 6 is an overly simple depiction of this technique. It shows the parent nodes sending a request for a burst of packets to one specific child. This implies that the parent knows of all its children and can decide which child to pull from next. This in turn implies that the parent is storing and updating state for each of its children. However, the number of children may exceed the allocated space for the state. This will cause unexpected behavior which can cause the protocol to fail. Furthermore, maintaining state requires memory and also requires constant monitoring of topology changes. Thus, PCP adds an additional step to the pulling mechanism shown in Figure 6. This step populates a pulling queue that stores node IDs to indicate which node the parent should pull from next. The process of populating this pulling queue is central to PCP's operation and is described in detail below.

Figure 7 shows the data flow of PCP. PCP maintains two queues: the forwarding queue and the pulling queue. The forwarding queue contains data packets waiting to be sent up the tree. The pulling queue is used to store the node IDs of children to which requests for packets will be sent. There are four types of packets involved in the data flow:

Candidate Request (Parent → Broadcast) The parent sends out a request on the broadcast address, requesting information about nodes which may wish to send it packets.

Candidate Response (Children → Parent) Children who receive the Candidate Request who have packets to send may send a candidate response to the parent. The parent uses the Candidate Responses to populate its pulling queue.

Data Request (Parent → Child) The parent takes the next node ID off its pulling queue and sends a data request to the corresponding child. In the request, it indicates both the maximum number of packets which it can hold in its buffer, and a time limit in which the child has to send them.

Data Response (Child → Parent) The child sends a burst of packets to the parent. Note that this burst is extremely efficient, as the child is not expecting the parent to retransmit the packets immediately. Thus, it does not need to allow time for the packets to clear the interference range, and can send the next immediately.

The size of the pulling queue has direct impact on the performance of PCP. When the size is 1, PCP works as 4-way handshake data transmission because it must issue a candidate request before any data requests. When the size is large, the data transmission approaches the two-way mechanism shown in Figure 6. Since each entry in the pulling queue is only a two-byte node ID, we can set the size to be large. In our implementation, the pulling queue has 20 entries.

The first step in pulling is to decide which children to pull from, and how much data to pull from each. This is done with a candidate request and response handshake.

When there is space in the pulling queue, the parent broadcasts a candidate request packet. The candidate request packet contains the quantity of the parent's immediate children and a response probability. Children first examine their forwarding queue to see if they have enough packets to send a burst. If so, they flip a coin using the probability embedded in the candidate request packet to decide whether to respond or to be silent. If they decide to respond, they introduce a random jitter first to avoid the broadcast storm problem. Since the number of candidate responses can be estimated from the number of immediate children and the response probability, nodes control the range of the jitter accordingly.

Ignoring link quality for the moment, the parent receives the candidate responses with equal probability across its children. Therefore, upon receiving the candidate responses, the parent should control the admission to the pulling queue to fairly distribute the bandwidth. The ideal bandwidth for each child is proportional to the subtree size of the child. All nodes, however deep in the tree, should receive the same share. This proportional bandwidth is not trivial to achieve. Section 4.2.3 explains the details of the fairness algorithm.

Since the range of the jitter is also known to the parent, the parent sets a timer waiting for candidate responses. When this timer fires, it examines the size of the pulling queue. If there is still a lot of space in the queue it increases the response probability. When the pulling queue becomes too full, the parent decreases the response probability. Thus the parent adaptively maintains a reasonable number of candidate responses for the amount of space it has to store them.

When the pulling queue has an entry and the forwarding queue has enough space for a burst of packets, the parent unicast a request packet to the first child node in the pulling queue. The request packet indicates a granted time for the child during which the child can send a burst of data packets. During this time, the parent cannot send another data request since multiple bursts would result in heavy collisions around the parent. Child nodes are allowed to transmit data packets only at this time to prevent ingress drops at the parent. The grant time is calculated from the number of packets in a burst and the packet time. Note that the packet time is not a constant value, due to the probabilistic CSMA back-

off. Thus, fixing the grant time alone does not control the number of data packets in a burst. Instead, nodes are only allowed to send up to the predefined number of packets per request. This may cause a portion of grant times to be wasted, but setting the packet time conservatively can minimize the wasted grant times. In our implementation, the burst size is 10 packets. To prevent egress drops, there is no limit on retransmissions of data packets.

When the pulling queue has enough free space, the parent broadcasts candidate request packets to repopulate the queue. The depletion of the queue results in throughput loss, thus maintaining an appropriate level is important.

4.2.3 Fairness in Pulling

The policy of admission to the pulling queue determines the node-level fairness of the protocol. PCP uses a centralized fairness measure, where parents have the control, rather than distributive, where children have the control. A distributive algorithm would have children control their own response probability according to their subtree size. However, children cannot calculate the optimal response probability without the knowledge of the subtree sizes of the other children. For PCP, the pulling mechanism is inherently centralized. Thus we can benefit from a centralized fairness measure without penalty.

Since each candidate response contains the child's estimated size of its subtree, perfect node fairness can be achieved if the each entry of the pulling queue is populated with probability $N(s)/N(S)$, where $N(s)$ is the subtree count for the child s and $N(S)$ is the count for the parent S. Therefore, upon receiving the candidate response from node s, the parent flips a coin with probability $N(s)/N(S)$ Q times, where Q is the number of available pulling queue entries. The parent then allocates pulling queue entries to the child according to the number of successful coin flips. To neutralize the effect of the order of responses, the parent notes Q when the candidate request is sent, and flips the coin for the same number of times for all incoming responses.

The above algorithm assumes identical link quality. If one of the children has a worse link than others, the child will hear the candidate request less often, the parent will hear its candidate responses less often. This node will have less of a chance of being admitted to the pulling queue. Also, since the data transmission time per request is fixed instead of number of successful packets, the child with bad link will not be able to send as many data packets as other children. Therefore, a compensation measure is required for nodes with bad links. When children send candidate responses, they embed a compensation value on the packet which is given as the reciprocal of (Candidate Response Success Ratio $\times$ Data Realization Ratio). To calculate the Candidate Response Success Ratio, the candidate request packets contain a sequence number field, which enables the children to keep track of the number of candidate requests which are successfully received. Even if a candidate request is successfully received, it is counted as a failure if the corresponding candidate response is not acknowledged. The Data Realization Ratio is the ratio of the number of successfully transmitted data packets to the number of packets requested by the parent. The queue entry probability is given by $Compensation \times N(s)/N(S)$.

This compensation makes the queue entry probability greater than one. Since this penalizes nodes whose responses arrive late, parents multiply the pulling queue entry probability by a scaling factor. The role of the scaling factor is to maintain the pulling queue size at the desired level. When an entry cannot enter the pulling queue the parent decreases the scaling factor, and when the pulling queue is not too empty after a candidate request the parent increases the scaling factor. The response probability serves the same purpose as the scaling factor. However, adjusting the response probability does not perform well for few children. The scaling factor helps finely adjust the number of entries in the pulling queue.

In summary, the probability of the coin flip on node S for each pulling queue entry for a child s is as follows:

$$P(s) = \frac{N(s)}{N(S)} \times \text{Compensation} \times \text{ScalingFactor}, \quad (4)$$

where $N(x)$ is the subtree size of node x.

Finally, nodes must generate their own packets at the appropriate rate. Since a node knows its subtree size, it adds its own packets to the forwarding queue only when it has forwarded enough packets from its children.

4.2.4 Link Estimation

PCP uses MultihopLQI's link estimation to form a tree topology with a single root. It uses the Link Quality Indicator (LQI) given by the CC2420 radio hardware to choose good links. Using LQI, rather than the expected number of transmissions to the root (ETX), has several benefits. First, it requires only one packet to estimate the link quality, reducing the control overhead. Second, unlike ETX, LQI does not fluctuate much, so the topology is more stable. Finally, LQI does not depend on data transmissions for estimation.

PCP changes MultihopLQI's routing algorithm slightly. When a node is disconnected from the network, MultihopLQI sets its parent to be NULL to prevent meaningless transmissions. However, PCP requires a parent's request to transmit. Thus, PCP does nothing when the node is disconnected and hopes it reconnects. Also, the threshold values are set higher so that PCP nodes change their parents less often. MultihopLQI aims to send to the best parent whenever possible, but frequent parent changes in PCP make it difficult to pull fairly from each child.

4.3 Experimental Setup

We tested PCP on the Motelab testbed with 52 motes, and also ran MultihopLQI on the same network for comparison. A snapshot of the network topology is shown in Figure 8 For PCP, nodes generated messages at a much higher rate than the network could handle, and buffered them at the origin node. This is to simulate a situation in which nodes wish to send data as fast as possible. For MultihopLQI, we experimented with various sending rates since it does not have

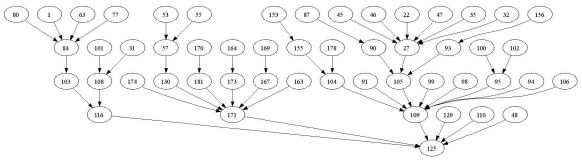


Figure 8. Motelab Topology for PCP and MultihopLQI Experiments. This is a representative snapshot, as the topology was not fixed in the experiments.

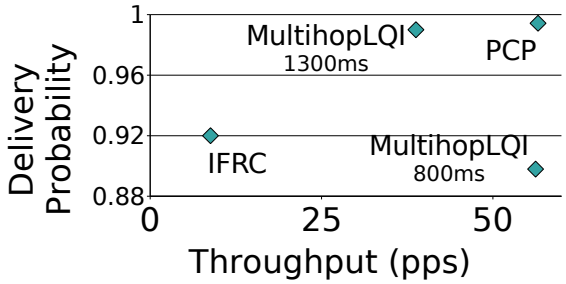


Figure 9. Comparison of PCP, MultihopLQI, and IFRC in terms of throughput and reliability on the Motelab testbed with 52 nodes. MultihopLQI is shown at two packet generation rates: one packet/800ms/node and one packet/1300ms/node. The first has the same reliability as PCP, the second rate has the same goodput as PCP.

a rate-limiting mechanism. We evaluate PCP against MultihopLQI in terms of visibility, delivery reliability, throughput, and node-level fairness. We also compare against IFRC, which is optimized for fairness. Since IFRC required parameters tuning, the IFRC results are from a 40-node testbed experiment described in [18].

4.4 Results

Figure 9 shows the goodput and the end-to-end delivery reliability for the protocols. Since PCP eliminates causes of packet loss, it is optimized for delivery reliability. PCP lost only 0.6% of the generated packets, where all losses were due to link-layer failures. Since the link-layer CRC check is not a perfect measure, packets can be acknowledged by the hardware and discarded in higher layers. More dominantly, we observed many false acknowledgments. False acknowledgments are a known problem with current radio scheme, where the sender thinks the packet is acknowledged but the receiver has not received the packet. Of all data packets, the link-layer failure rate per transmission was 0.16%.

MultihopLQI achieves the same goodput as PCP when each node generates a packet every 800ms, meaning the network generates 63.75 pps. The delivery reliability for this case was 90.4% due to ingress drops and egress drops. The loss decreases as the traffic load decreases, achieving 99.0% at 1300ms generation interval, or network generation rate of 39.2 pps. For IFRC, since IFRC is optimized for node-level fairness, it achieves throughput of 8.8 pps with a delivery reliability of 92%.

PCP performs better than MultihopLQI in terms of goodput and delivery reliability due to the rate-limiting nature of the pulling mechanism. For PCP, only one child per parent can transmit at a time, reducing the probability of collision at the parent. Limiting the number of transmitting nodes is more powerful than limiting the transmissions of each node, since the hidden terminal problem is avoided more effectively. An experiment showed MultihopLQI requires an average of 0.66 retransmissions per successful transmission, and while PCP requires only 0.16. Considering that retransmissions can occur even if only acknowledgment packets are lost, the low retransmissions for PCP indicates the reliability gain comes more from the pulling mechanism rather than infinite retransmissions. In contrast, MultihopLQI lacks a rate-limiting mechanism. Thus, when the same traffic load as PCP was used for MultihopLQI, it achieved 66.0 pps throughput but its delivery reliability was only 13.0%.

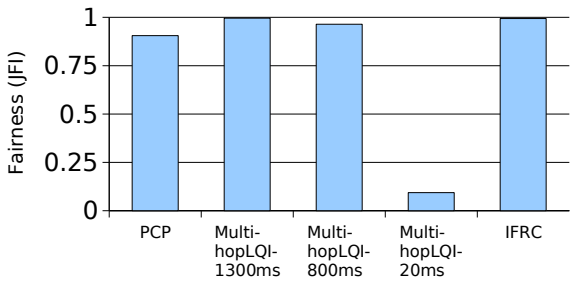


Figure 10. Comparison of PCP, MultihopLQI, and IFRC in terms of node-level fairness on Motelab testbed with 52 nodes. IFRC results are from a 40-node testbed in [18]. PCP provides less fairness than MultihopLQI at lower sending rates, due to its approximated subtree sizes.

Figure 10 shows the node-level fairness in JFI [10]. For MultihopLQI, since the generation rate was identical for all nodes, it achieves perfect fairness when all packets are delivered. Thus MultihopLQI with a 1300ms generation interval achieves a very high fairness of 0.996. When the generation interval decreases to 800ms, the fairness drops to 0.97. When the traffic load increases further, nodes adjacent to the root cannot receive packets due to channel saturation, disconnecting the route for nodes that are deeper in the tree. When each node generates a packet every 20ms, the fairness drops to 0.089. Meanwhile, IFRC, although the reliability and the goodput are not favorable, is optimized for fairness thus achieving near perfect fairness of 0.9994 regardless of the traffic load.

PCP achieves a fairness of 0.905. This is relatively high considering the traffic load was at maximum, but it is lower than MultihopLQI with the same goodput. The main reason for PCP's reduced fairnes is inaccurate estimates of the subtree size. As we mentioned in Section 4.2.3, the size of the subtree is the main metric which decides how frequently the parent pulls from each child. While synopsis diffusion enables fixed-state estimates, its inaccuracy directly affects the fairness for PCP. In addition, when a child changes its parent, it takes the parent time to realize the change in subtree size since the estimates are calculated periodically.

	MultihopLQI		PCP	
	p_k	$Energy_k(C)$	p_k	$Energy_k(C)$
Disconnection	0	0	0	0
Death	0	0	0	0
Reboot	0	1	0	0
Ingress Drop	0.134	4	0	1
Egress Drop	0.795	1	0	1
False Duplicate Suppression	0	3	0	0
Bad Link	0	2	0	0
Unknown (Link Layer Error)	0.071	4	1.0	0
$\sum_{k^{th}\ \text{cause}} Energy_k p_k$	1.615		0	

Table 4. Visibility calculation for MultihopLQI and PCP in a controlled testbed environment. $Energy_k$ is calculated using equation 2.

4.5 Evaluation of PCP's Visibility

Although in designing PCP we considered node-level fairness as well as reasonable throughput, our key design point was to make it easier to diagnose packet losses. We wish to use equation 3 to calculate the visibility of PCP compared with MultihopLQI. This requires knowing the p_k and I_q terms for PCP. In our experiments, we used the methodology described in Section 3 to gather the probability distribution of packet losses for both protocols. We set MultihopLQI's sending rate so that its throughput would be comparable with PCP's, which meant sending packets at an 800ms interval from each node. As mentioned earlier, all PCP's packet losses were due to link layer errors. We determined the $Energy_k$ terms by considering the decision tree shown in Figure 4(b) for PCP and Figure 3 for MultihopLQI. With our measured distributions, we are able to calculate the visibility cost of each protocol as shown in Table 4. In this environment, at this sending rate, MultihopLQI has a visibility cost of $1.615C$ and PCP has a cost of 0. In this deployment the probability distribution for MultihopLQI would lead us to reoptimize the decision tree, by considering ingress drops earlier. This would reduce the visibility cost of MultihopLQI slightly, to $1.34C$.

The above analysis indicated PCP has a visibility cost of 0. However, eliminating ingress drops required additional control packets, in the form of candidate request/responses and data requests. In order to determine whether this actually reduced the cost of the protocol, we consider the energy overhead as well. In our implementation, the burst size is 10 packets. In order to send these 10 packets, there must be a candidate request, a candidate response, and a data request. This would imply that PCP incurs roughly a 30% control overhead (ignoring link quality estimation, which is identical to MultihopLQI). If we also consider retransmissions, our experiments show that at similar throughputs, PCP requires 1.6 retransmissions per 10 packets, while MultihopLQI requires 6.6 retransmissions per 10 packets. This means that PCP has a 46% overhead, while MultihopLQI has a 66% overhead. Therefore, even though PCP adds control overhead, the reduction in collisions outweighs this cost. PCP can further reduce the control overhead by increasing the data burst size.

Another serious question to consider in evaluating PCP's visibility is whether we have simply changed the question. It is easy to diagnose why PCP does not drop packets, but it is more difficult to diagnose why a node never sends any packets. It may not be hearing candidate requests, or its parent may not be receiving its candidate responses. It may miscalculate its subtree size and send only from its children, or its sibling in may erroneously claim to have a massive subtree. In designing PCP we attempted to deal with each of these failure causes, by adding redundancy to the synopses and considering link quality when populating the pulling queue. However, we acknowledge that PCP may not be as visible when nodes are expected to send at a pre-specified rate.

In its entirety, PCP is 29.3 KB of program memory. The forwarding engine, which handles all sending and receiving of data and pull control packets, requires 6.3K of program memory. The pulling queue requires 40 bytes of state, and each node needs 27 bytes of data for fairness. Each node maintains 22 bytes of synopsis information. The size of the forwarding queue is 25 packets (MultihopLQI's forwarding queue was increased for fair comparison).

5 Improving Existing Protocols

Although designing visible protocols from the scratch is powerful, it is neither always possible nor efficient. In this section, we walk through a procedure to improve visibility with existing network protocols. In particular, we chose Deluge [9], a dissemination protocol for large-sized binaries, to complement our visible collection protocol. We briefly introduce Deluge and its current causes of failure, and discuss ways to eliminate them. We describe V-Deluge, our version of Deluge which is augmented for greater visibility. We compare V-Deluge to the original in terms of efficiency and throughput, as well as visibility.

5.1 Deluge As It Is

Deluge is a dissemination protocol which uses Trickle [14] to politely spread the latest binary using advertisement packets. Deluge defines a dynamic trickle period, in which a node sends a single advertisement packet unless it hears another node transmitting the same version. When the network is stable, nodes increase their trickle periods exponentially to reduce transmissions. When a node hears advertisements with a different version, it resets its trickle period to the minimum to expedite the process of updating its code. Out-of-date nodes send requests for data transmissions to nodes sending newer advertisements.

Nodes that have out-of-date code do not send a request packet immediately after an advertisement. Instead, they decrease their trickle timer to the minimum and wait for four trickle periods. During these periods, they record the RSSI values of advertisements that they hear. After this period, the requester unicasts a request packet to the node with the highest RSSI to minimize packet losses. It uses hard timeouts after requests to send another request to different nodes if it does not hear data packets with the version that it needs.

In response to the request packet, the destination node becomes a transmitter and initiates a burst of data transmission.

Deluge is a protocol intended to disseminate a large amount of data, thus a unit of a burst is a page size of the flash memory, a few tens of packets with the default TinyOS packet size. When nodes are exchanging data packets, they cannot initiate another data transmission. Therefore, when a node is transmitting or receiving data packets, it defers sending advertisement packets.

5.2 Possible Failures in Deluge

In a collection protocol, the key question to answer is why the network loses packets. In a dissemination protocol, nodes keep a copy of the disseminated binary, so lost data is not an issue. In addition, Deluge is a single-hop protocol, thus it is rare that it takes infinite time for a node to acquire the correct binary unless it is disconnected. However, it may take a long time for some nodes to have the correct binary, and the deployment developer may wish to diagnose the causes for this behavior. Thus, for dissemination, the question that we would like to be able to answer easily is, "Why does a node still have an out-of-date binary?"

One possible answer to this question is suppression. To reduce network traffic, Deluge suppresses transmissions when it overhears similar packets. Nodes suppress their advertisements if they overhear an identical advertisement in a trickle period. Similarly, they cancel their request packets if they overhear an identical request. Data transmitters cancel their transmission when they hear a data transmission of a older version.

While suppression enhances the scalability and efficiency of the protocol, it can be a cause of failures. If a faulty node sends continuous advertisement packets, the neighbors cannot send advertisements although they can hear advertisements with the older version. Similarly, if a node sends continuous request or data packets, Deluge grinds to a halt.

While suppression can prevent dissemination, collision can delay it. Although transmitters will cancel their transmissions if they overhear another, multiple hidden terminals can become transmitters, destroying an entire packet burst at the receiver. More generally, any traffic from neighbors will collide with the data packet burst. When a data packet is corrupted by collision, nodes must restart their request process. For this reason, collision can be a significant source of dissemination delay.

In the next section, we describe the visibility of Deluge as it is currently implemented, and describe how one would identify why a node still has an out-of-date binary.

5.3 Deluge Visibility

In addition to the causes that we identify in Section 5.2, Deluge has inherent causes of failure such as disconnection, reboot, or bad links. It is also possible that the question is simply asked too soon when Deluge is normally disseminating binaries, or that the delayed dissemination is caused by ascendants. The decision tree shown in Figure 11 diagnoses which cause is responsible for the delay in dissemination.

If the root cannot communicate with the node in question at all, it implies that the node is disconnected. If we

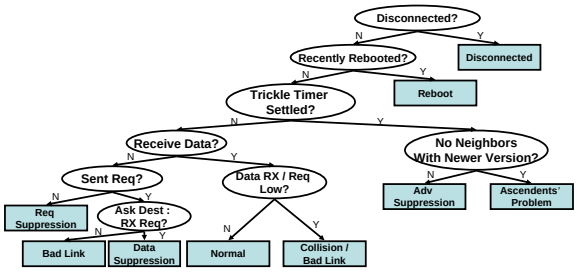


Figure 11. Decision trees for Deluge to answer the question, "Why does a node still have an out-of-date binary?" Some questions cannot be answered from local information. This decision tree cannot distinguish between collisions and a bad link.

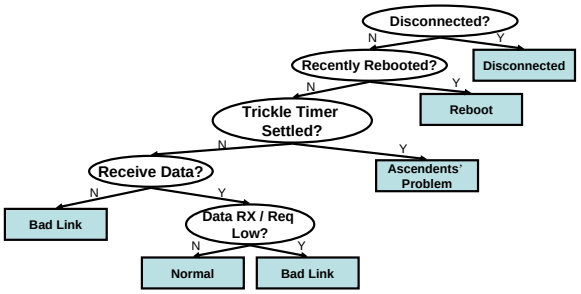


Figure 12. Decision trees for V-Deluge to answer the same question as Figure 11. All questions are local and collision is minimized.

can communicate with the node, we can diagnose reboot by adding a simple timer that starts at boot-up. We can examine the trickle timer period to see if the timer has settled. If it has, then it has not heard the advertisements for the newer version. This means that either its neighbors do not have the new version either, or they are unable to send advertisements. If the neighbors do have the new version, then their advertisements are being suppressed by a faulty node.

If, on the other hand, the trickle timer has not settled, it means that the node is hearing advertisements for the new binary. If the node is receiving large amounts of data for each request it sends, then Deluge is operating normally, and the question has simply been asked too soon. If the node is not receiving much data for each request it sends, then the delay is due to packet loss. We do not currently have a good way to distinguish collisions from bad links, thus the cause of delay is uncertain. If the node is receiving no data at all, yet we know it is not disconnected, then the remaining possibilities are loss of request packets or request / data suppression. It may be that the requester is being suppressed by a faulty node, or the request may not be received at all.

The problem with the current decision tree is that some questions are not local. In order to answer them, we must query neighboring nodes to determine if they have sent or received packets. These questions increase the cost of traversing the energy tree significantly, because they require sending additional queries or updates.

We now show an example of calculating the visibility metric for Deluge. First, for fair comparison, we assume

	p_k	$Energy_k(C)$
Disconnection	0.13	0
Reboot	0.27	0
Advertisement Suppression	0.01	1
Ascendant's Problem	0	1
Request Suppression	0.01	0
Bad Link (Inhibiting Rx of Requests)	0.20	1
Data Suppression	0.01	1
Normal (Question asked too early)	0	0
Bad Link (Inhibiting Rx of Data)	0.37	0
$\Sigma_{k^{th}\ \text{cause}} Energy_k p_k$	0.02	

Table 5. Visibility calculation for Deluge in a hypothetical environment. Note that the calculations assumes that the out-of-date node is diagnosing itself. An additional C term should be included to query the out-of-date node.

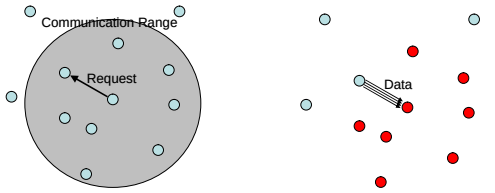


Figure 13. V-Deluge's collision avoidance mechanism. a) A requester unicasts a request packet to a node with a newer version. The other nodes in the communication range overhear this packet. b) The overhearing nodes and requester(in red) cannot send any packets during the time specified by the request packet. No packets from nodes with symmetric links will collide at the requester while it receives its burst of data packets.

that the question is not asked too soon or to irrelevant nodes. With a testbed experiment, we could not verify nodes with suppression failures, which is natural because it would typically happen when operating on battery power, when nodes can begin to behave erratically. Here, since we don't have prior knowledge, we conjecture the actual probability of suppression failure to be 1% each (for data, request, and advertisement suppression) for the purposes of this example. For disconnection and reboot, we assume that 27% of failures are due to reboot and 13% are due to disconnection. This leaves the probability of a bad link as 57%. Since a testbed experiment had 1.88 times more data packets than request packets, the probability for the bad link inhibiting data reception is estimated to be 20% and bad link preventing request reception to be 37%. In calculating the visibility for Deluge, we consider C as the energy required to query the out-of-date node. If the node cannot diagnose itself, it must query its neighbors. In our example this will incur an additional cost of $0.02C$. So the visibility cost of Deluge is $1.02C$.

In the next section we describe our modifications to Deluge to minimize the amount of non-local querying that is required to diagnose Deluge's behavior. We call this modified version of Deluge V-Deluge.

5.4 Eliminating Causes of Failure

The first cause of failure that we would like to prevent is suppression of advertisements. In V-Deluge, nodes keep a few bytes of state to monitor the dominant source of advertisements. A node maintains a counter which it resets to zero whenever it sends an advertisement. The node increments the counter whenever it overhears an advertisement from the currently dominant node. If it hears an advertisement from a different source, it decrements the counter. If this counter ever exceeds a threshold, the dominant node is assumed to be faulty. Its advertisements are ignored and no longer act to suppress other nodes' advertisements.

V-Deluge prevents request suppression with the same algorithm, but this algorithm can not be used to prevent suppression of data packets. This is because a node at the edge of the network may receive all data packets from a single source. Instead, a node prevents data suppression failures by counting how many times it repeatedly hears the same page. If this value exceeds a threshold, the node ignores the data transmission and allows its own data transmissions.

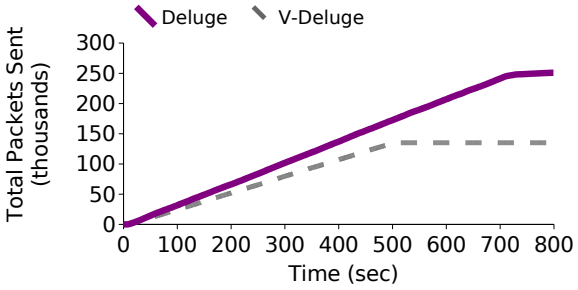
Collision is mainly an issue when nodes in the vicinity of a requester interfere with the binary being sent to it. To prevent this, V-Deluge enforces silence on nodes that overhear or send request packets. Only the receiver of the request packet can transmit. As shown in Figure 13, nodes within reception range of the requester will remain silent, with the exception of the node providing the data burst.

This change means that the data burst from the recipient of a request may be delayed significantly, since the recipient may need to remain silent while a neighbor's data burst completes. This means that hard timeouts for responses to requests no longer make sense. Instead, requesters use packet acknowledgments to hypothesize whether the destination node has received its request. Even if the request is acknowledged, the requester returns to its initial state and listens for other sources of data. It may decide to send a request to a different destination. If the previous destination overhears this second request, it should cancel its pending data transmission to prevent conflicts at the requester.

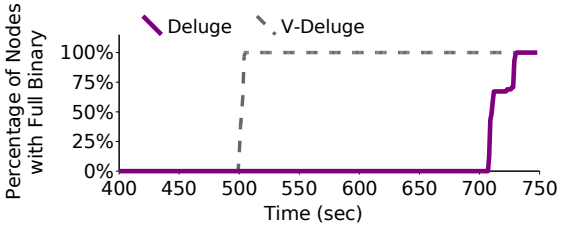
Note that this initial solution can actually cause further failure, if a node continuously sends request packets. In this case, the neighboring nodes will be silent indefinitely. To avoid this situation, nodes filter faulty requests by checking to see if a request packet is sent from a node which should be silent (because it sent an earlier request packet, suppressing itself). In this way, V-Deluge limits the collision problem without introducing additional failures.

These mechanisms do not guarantee that collision will never occur. However, as we will show in Section 5.5, they significantly reduce packet collision. Thus, we claim that collision is no longer one of the major causes of delay.

Figure 12 shows the decision tree once we have introduced suppression and collision avoidance. Now the decision tree only contains inherent causes of failure. Furthermore, all questions can be answered with only local information. Thus, the visibility cost for V-Deluge is C. The visibility cost saving in the hypothetical example shown in Table 5 is $0.02C$, and the visibility cost for V-Deluge is al-



(a)



(b)

Figure 14. Performance comparisons between Deluge and V-Deluge. Both disseminate 200 pages of 30 packets each on a 61-node testbed. a)Total Number of Packets sent by the network over time. V-Deluge sends significantly fewer packets. b) Percentage of nodes which have received the full binary over time. V-Deluge disseminates the binary faster than the original Deluge.

ways lower than for Deluge. However, this gain in visibility does not come without costs. The code has additional complexity and state. In V-Deluge, failure prevention required an additional 2458 bytes of ROM and 116 bytes of RAM on telosb motes. We evaluate the effect on performance in the following section.

5.5 Performance Comparison

We compare Deluge and V-Deluge in an experiment using 61 nodes on the Motelab testbed. We disseminate a binary of 200 pages, 30 packets each, to all of the nodes. Figure 14 shows the latency and the number of transmitted packets until each node receives the full binary. Comparing to Deluge, V-Deluge requires only 69.2% of the time to distribute the binary and transmits only 54.4% as many packets. This gain is due mostly to the collision avoidance mechanisms, which increase both throughput and efficiency.

Our results show that visibility does not always come at the expense of performance. Rather, eliminating causes of failure not only simplifies diagnosis, but helps performance, because failure affects performance.

6 Discussion and Conclusion

In this paper, we argued that visibility should be a major consideration when designing a sensornet protocol. We need more than a network management system such as Sympathy; the network itself should be more visible. We proposed a metric to quantify visibility, which translates into the energy required to diagnose a cause of a behavior of interest.

Since this is the first attempt at quantifying visibility, it allows significant space for future work. First, calculating visibility requires a priori knowledge of the probability distribution of causes of a behavior. Although we can reasonably conjecture the distribution, the distribution can vary significantly in the real world. Moreover, the distribution changes with time, which makes it even more difficult to calculate the visibility of a protocol. Adaptive measures should be employed to dynamically monitor the distribution of causes. As the distribution changes, the optimal decision tree should also change. For example, if disconnection gradually becomes the prominent cause of failure, the optimal tree may first determine whether a behavior is due to disconnection.

In this paper, we calculate visibility assuming all questions are to be answered by sending queries. However, it is also possible to piggyback some needed information on data packets, as part of normal operation. The cost of piggybacked data should be included in the visibility cost metric. As the network is dynamic, the optimal way to get an answer for a question may change between querying and piggybacking. Since the visibility of a protocol should measure the minimum cost, the trade-off between querying and piggybacking should be explored.

Our analysis in this paper assumes that all sources of failure are within the protocol itself. This is not true for typical sensornet systems, which usually contain several network protocols. Protocols can affect each other and cause failures in a different network protocol. Thus, for a truly visible system, we should ensure that a failure in a protocol is caused only by itself. This implies the necessity of a mechanism to provide isolation between network protocols.

Our work on PCP was an exercise in pursuing visibility above all other metrics. The result was a rather extreme protocol which introduced significant control overhead, and early versions of PCP made sending packets even more complicated. When improving visibility for one behavior it is important to keep in mind how it will affect the visibility for other behaviors, or introduce new ones. While such extreme approaches to visibility are probably going a bit far, our work with V-Deluge showed that pursuing visibility can have additional benefits, like lower latency, without introducing overheads.

This visibility principle in designing network protocols is a significant departure from existing ideas. Traditional figures of merit of network protocols are throughput, efficiency, or goodput. However, the fact that there has not been a satisfactory large-scale sensornet system deployment even with a decade of research suggests that we should begin to think about new directions. Due to the inherent energy constraints of sensor networks, visibility is not as easy to achieve as in traditional systems. We argue that visibility should be considered as much as other metrics, because visibility will inherently speed progress in the others.

7 Acknowledgements

This work was supported by generous gifts from Intel Research, DoCoMo Capital, Foundation Capital, the National Science Foundation under grant #0615308 ("CSR-EHS"), scholarships from the Samsung Lee Kun Hee Scholarship Foundation, a Stanford Graduate Fellowship, and a Stanford Terman Fellowship. We would like to thank those who maintain Mirage and MoteLab for providing community testbeds. Finally, we would like to thank the reviewers for their comments and Jie Liu for his help in improving our paper.

References

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection. *Computer Networks (Elsevier)*, 46, 2004.

[2] R. Beckwith, D. Teibel, and P. Bowen. Unwired wine: Sensor networks in vineyards. In *Proceedings of IEEE Sensors*, 2004.

[3] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. Task: Sensor network in a box. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[5] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[6] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference On Embedded Networked Sensor Systems (SenSys)*, 2004.

[7] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks (TOSN)*, 2006.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[9] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[10] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, September 1984.

[11] P. Kimelman. personal communication, 2006.

[12] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Tarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *Proceedings of the Third ACM Conference On Embedded Networked Sensor Systems (SenSys)*, 2005.

[13] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *the Fourteenth Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[15] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the Second ACM Conference On Embedded Networked Sensor Systems (SenSys)*, 2004.

[16] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[17] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the Third ACM Conference On Embedded Networked Sensor Systems (SenSys)*, 2005.

[18] S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-aware fair rate control in wireless sensor networks. In *Proceedings of the ACM SIGCOMM*, 2006.

[19] T. Schmid, H. Dubois-Ferriere, and M. Vetterli. Sensorscope: Experiences with a wireless building monitoring sensor network. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2005.

[20] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[21] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks, 2005.

[22] V. Turau, C. Renner, M. Venzke, S. Waschik, C. Weyer, and M. Witt. The heathland experiment: Results and experiences. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2005.

[23] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*, Los Angeles, CA, Nov. 2003.

[24] G. Werner-Allen, K. Lorincz, J. Johnson, J. Leess, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[25] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedigns of the 4th International Symposium on Information Processing in Sensor Networks*, 2005.

[26] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks: Special Track on Sensor Platform, Tools, and Design Methods for Network Embedded Systems (IPSN/SPOTS)*, 2006.

[27] A. Woo and T. Tong. Tinyos mintroute collection protocol. tinyos-1.x/lib/MintRoute.