Troubleshooting Wireless Mesh Networks

Lili Qiu
Univ. of Texas at Austin
lili@cs.utexas.edu

Paramvir Bahl
Microsoft Research
bahl@microsoft.com

Ananth Rao
UC Berkely
ananthar@cs.berkeley.edu

Lidong Zhou
Microsoft Research
lidongz@microsoft.com

ABSTRACT

Effective network troubleshooting is critical for maintaining efficient and reliable network operation. Troubleshooting is especially challenging in multi-hop wireless networks because the behavior of such networks depends on complicated interactions between many factors such as RF noise, signal propagation, node interference, and traffic flows. In this paper we propose a new direction for research on fault diagnosis in wireless mesh networks. Specifically, we present a diagnostic system that employs trace-driven simulations to detect faults and perform root cause analysis. We apply this approach to diagnose performance problems caused by packet dropping, link congestion, external noise, and MAC misbehavior. In a 25 node mesh network, we are able to diagnose over 10 simultaneous faults of multiple types with more than 80% coverage.

Categories and Subject Descriptors

C.2.3 [**Computer-Communications Networks**]: Network Operations—*network management*; C.2.1 [**Computer-Communications Networks**]: Network Architecture and Design—*Wireless Communication*

General Terms

Management, Measurement, Performance, Reliability

Keywords

Mesh networks, Fault Diagnosis, Simulation

1. INTRODUCTION

There is widespread grassroots interest in community and rural-area wireless mesh networks. Mesh networks grow organically as users buy and install equipment, but they often lack centralized network management. Therefore, self-management and self-healing capabilities are key to the long-term survival of these networks. It is this vision that inspires us to research network troubleshooting in multihop wireless networks.

Troubleshooting a network, may it be wired or wireless, is a difficult problem. This is because a network is a complex system with many inter-dependent factors that affect its behavior. The factors include networking protocols, traffic flows, hardware, software, different faults, and most importantly the interactions between them. Troubleshooting a multihop wireless network is even more difficult due to unreliable physical medium, fluctuating environmental conditions, complicated wireless interference, and limited network resources. We know of no heuristic or theoretical technique that captures these interactions and explains the behavior of such networks.

To address these challenges, we propose a novel troubleshooting framework that integrates a network simulator into the management system for detecting and diagnosing faults occurring in an operational network. We collect traces, post-process them to remove inconsistency, and then use them to recreate in the simulator the events that took place inside the real network. Note that while simulation has been applied to network management, such as performance tuning and what-if analysis for route simulation, to our knowledge there is no previous work that uses simulation to detect and identify network faults.

For our system to work, we must solve two problems: (i) accurately reproduce inside the simulator what just happened in the operational network; and (ii) use the simulator to perform fault detection and diagnoses.

We address the first problem by taking an existing network simulator (e.g., Qualnet [16], a commercially available packet-level network simulator) and identify the traces to drive it with. (Note: although we use Qualnet in our study, our technique is equally applicable to other network simulators, such as NS-2, OPNET etc.). We concentrate on physical and link layer traces, including received signal strength, and packet transmission and retransmission counts. We replace the lower two networking layers in the simulator with these traces to remove the dependency on generic theoretical models that do not capture the nuances of the hardware, software, and radio frequency (RF) environment.

We address the second problem with a new fault diagnosis scheme that works as follows: the performance data emitted by the trace-driven simulator is considered to be the expected baseline ("normal") behavior of the network and any significant deviation indicates a potential fault. When a network problem is reported/suspected, we selectively inject a set of possible faults into the simulator and observe their effect. The fault diagnosis problem is therefore reduced to efficiently searching for the set of faults which, when injected into the simulator, produce network performance that matches the observed performance. This approach is significantly different from the traditional signature based fault detection schemes.

Our system has the following three benefits. First, it is flexible. Since the simulator is customizable, we can apply our fault detection and diagnosis methodology to a large class of networks operating under different environments. Second, it is robust. It can capture complicated interactions within the network and between the network and the environment, as well as among the different faults. This allows us to systematically diagnose a wide range and combination of faults. Third, it is extensible. New faults are handled independently of the other faults as the interaction between the faults is captured implicitly by the simulator.

We have applied our system to detect and diagnose performance problems that arise from the following four faults:

- Packet dropping. This may be intentional or may occur because of hardware and/or software failure in the networked nodes. We care about persistent packet dropping.
- Link congestion. If performance degradation is due to too much traffic on the link, we want to be able to identify this.
- External noise sources. RF devices may disrupt on-going network communications. We concern ourselves with noise

sources that cause sustained and/or frequent performance degradation.

- MAC misbehavior. This may occur because of hardware or firmware bugs in the network adapter. Alternatively, it may be due to malicious behavior where a node deliberately tries to use more than its share of the wireless medium.

The above faults are more difficult to detect than fail-stop errors (e.g., a node turns itself off due to power or battery outage), and they have relatively long lasting impact on performance. Moreover, while we focus on multihop wireless networks, our techniques may be applicable for wireless LANs, as well.

We demonstrate our system's ability to detect random packet dropping and link congestion in a small multihop IEEE 802.11a network. We demonstrate detection of external noise and MAC misbehavior via simulations because injecting these faults into the testbed in a controllable manner is difficult. In a 25 node multihop network, we find that our troubleshooting system can diagnose over 10 simultaneous faults of multiple types with more than 80% coverage and very few false positives.

To summarize, the primary contribution of our paper is to show that a trace-driven simulator can be used as an analytical tool in a network management system for detecting, isolating, and diagnosing faults in an operational multihop wireless network. In the context of this system, we make the following three contributions:

- We identify traces that allow a simulator to mimic the multihop wireless network being diagnosed.
- We present a generic technique to eliminate erroneous trace data.
- We describe an efficient search algorithm and demonstrate its effectiveness in diagnosing multiple network faults.

The rest of this paper is organized as follows. We describe the motivation for this research and give a high-level description of our system in Section 2. We discuss system design rationale in Section 3. We show the feasibility of using a simulator as a diagnostic tool in Section 4. In Section 5, we present fault diagnosis. In Section 6, we describe the prototype of our network monitoring and management system. We evaluate the overhead and effectiveness of our approach in Section 7, and discuss its limitations and future research challenges in Section 8. We survey related work in Section 9, and conclude in Section 10.

2. SYSTEM OVERVIEW

Our management system consists of two distinct software modules. An *agent* that runs on every node, gathers information from various protocol layers and the wireless network card. It reports this information to a management server, called *manager*. The manager analyzes the data and takes appropriate actions. The manager may run on a single node (centralized architecture), or may run on a set of nodes (decentralized architecture) [17].

Our troubleshooting involves three steps: data collection, data cleaning, and root cause analysis. During the data collection step, agents continuously collecting and transmitting their (local) view of the network's behavior to the manager(s). Examples of the information sent include traffic statistics, received packet signal strength on various links, and re-transmission counts on each link.

It is possible that the data the manager receives from the various agents results in an inconsistent view of the network. Such inconsistencies could be the result of topological and environmental changes, measurement errors, or misbehaving nodes. The *Data Cleaning* module of the manager resolves inconsistencies before engaging the analysis model.

After the inconsistencies have been resolved, the cleaned trace data is fed into the root-cause analysis module which contains a modified network simulator (see Figure 1). The analysis module drives the simulator with the cleaned trace data and establishes the expected normal performance for the given network configuration and traffic patterns. Faults are detected when the expected performance does not match the observed performance. Root cause for the discrepancy is determined by efficiently searching for the set of faults that results in the best match between the simulated and observed network performance.

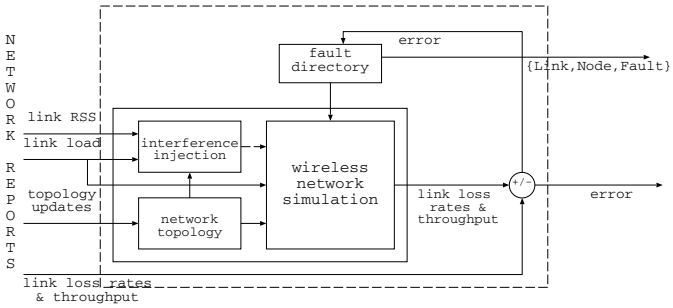

Figure 1: Root cause analysis module

3. DESIGN RATIONALE

A wireless network is a complex system with many inter-dependent factors that affect its behavior. We know of no heuristic or theoretical technique that captures these interactions and explains the behavior of such networks. In contrast, a high quality simulator provides valuable insights on what is happening inside the network.

As an example, consider a 7 * 3 grid topology network shown in Figure 2. Assume there are 5 long-lived flows F_1, F_2, F_3, F_4 and F_5 in the network, each with the same amount of traffic to communicate. All adjacent nodes can hear one another and the interference range is twice the communication range. F1 interferes with F2, and similarly F5 interferes with F4. However, neither F1 nor F5 interferes with F3. Table 1 shows the throughput of the flows when each flow sends CBR traffic at a rate of 11 Mbps. As we can see, the flow F_3 receives much higher throughput than the flows F_2 and F_4.

A simple heuristic may lead the manager to conclude that flow F_3 is unduly getting a larger share of the bandwidth, whereas an on-line trace-driven simulation will conclude that this is normal behavior. This is because the simulation takes into account the traffic flows and link quality, and based on the reported noise level it determines that flows F_1 and F_5 are interfering with flows F_2 and F_4, therefore allowing F_3 a open channel more often. Thus, the fact that F_3 is getting a greater share of the bandwidth will not be flagged as a fault by the simulator.

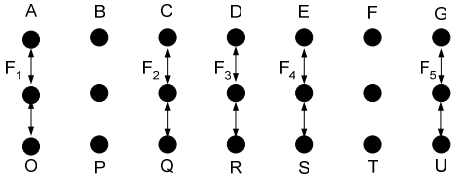

Figure 2: The flow F_3 gets a much higher share of the bandwidth than the flows F_2 and F_4, even though all the flows have the same application-level sending rate. A simple heuristic may conclude that nodes D and R are misbehaving, whereas simulation can correctly determine the observed throughput is expected.

Consequently, a good simulator is able to advise the manager on

F_1	F_2	F_3	F_4	F_5
2.50 Mbps	0.23 Mbps	2.09 Mbps	0.17 Mbps	2.55 Mbps

Table 1: Throughput of 5 competing flows in Figure 2

what constitutes normal behavior. When the observed behavior is different from what is determined to be normal, the manager can invoke the fault search algorithms to determine the reasons for the deviation.

In addition, while it might be possible to apply traditional signature-based or rule-based fault diagnosis approach to a particular type of network under a specific environment and configuration, simple signatures or rules do not capture the intrinsic complexity of fault diagnosis in general settings. In contrast, a simulator is customizable and with appropriate parameter settings, it can be applied to a large class of networks under different environments. Fault diagnosis built on top of such a simulator inherits its generality.

Finally, recent advances in simulators for multihop wireless networks, as evidenced in products such as Qualnet, have made the use of a simulator for real-time on-line analysis a reality. For example, Qualnet [16] can simulate 250 nodes in less than real-time on 3.2 GHz Pentium IV processor with 2 GB of RAM. Such simulation speed is sufficient for the size of a network, up to a few hundred nodes, we intend to manage.

4. SIMULATOR ACCURACY

We now turn our attention to the following question: "Can we build a fault diagnosis system using on-line simulations as the core tool?" The answer to the question cuts to the heart of our work. The viability of our system hinges on the accuracy with which the simulator can reproduce observed network behavior. To answer this question, we quantify the challenge in matching the behavior of the network link layer and RF propagation. We then evaluate the accuracy of trace-driven simulation.

4.1 Physical Layer Discrepancies

Factors such as variability of hardware performance, RF environmental conditions, and presence of obstacles make it difficult for simulators to model wireless networks accurately [9]. We further confirm this using measurement. Our results in [15] show that the theoretical model does not estimate the RSS accurately, because it fails to take into account obstacle and multipath. Accurate modeling and prediction of wireless conditions is a hard problem to solve in its full generality, but by replacing theoretical models with data obtained from the network we are able to significantly improve network performance estimation as we will show later.

4.2 Baseline Comparison

Next we compare the performance of a real network to that of a simulator for a few simple baseline cases. We design a set of experiments to quantify the accuracy of simulating the overhead of the protocol stack as well as the effect of RF interference. The experiments are for the following scenarios:

1. A single one-hop UDP flow (1-hop flow)
2. Two UDP flows within communication range (2 flows - CR)
3. Two UDP flows within interference range (2 flows - IR)
4. One UDP flow with 2 hops where the source and destination are within communication range. We enforce the 2-hop route using static routing. (2-hop flow -CR)
5. One UDP flow with 2 hops where the source and destination are within interference range but not within communication range. (2-hop flow -IR)

All the throughput measurements are done using Netgear WAG511 cards and Figure 3 summarizes the results. Interestingly, in all cases the throughput from simulations are close to the real measurements. Case (1) shows that Qualnet simulator models the overheads of the protocol stack, such as parity bits, MAC-layer back-off, IEEE 802.11 inter-frame spacing and ACK, and headers accurately. The other scenarios show that the simulator accurately takes into account contention from flows within the interference and communication ranges.

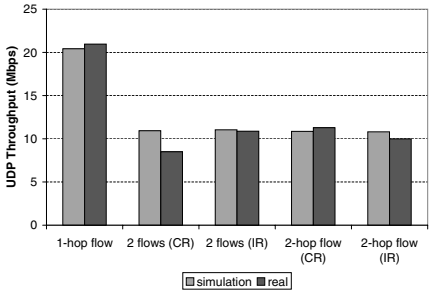

Figure 3: Estimated throughput from the simulator matches measured throughput in a real network when the RF condition of the links is good.

In the scenarios above, data are sent on high-quality wireless links, and almost never gets lost due to low signal strength. In our next experiment, we study how RSS affects throughput. We vary the number of walls between the sender and receiver, and plot the UDP throughput for varying packet sizes in Figure 4.

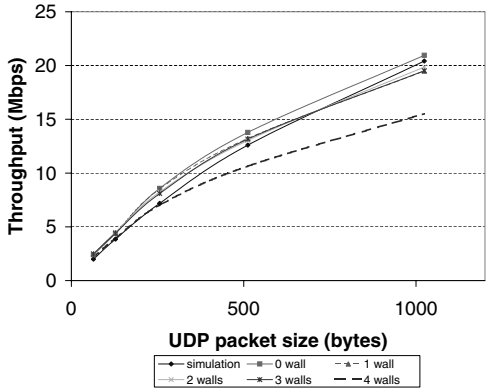

Figure 4: Estimated throughput matches with measured throughput when the RF condition of the links is good, and deviates when the RF condition of the links is poor (1-hop connection).

When the signal quality is good (e.g., when there are fewer than 4 walls in between), the throughput measured matches closely with the estimate from the simulator.

When the signal strength is poor, e.g., when 4 or more walls separate the two laptops, the throughput estimated by the simulator deviates from real measurements. The deviation occurs because the simulator does not take into account the following two factors:

- Accurate packet loss as a function of packet-size, RSS, and ambient noise. This function depends on the signal processing hardware within the wireless card.
- Accurate auto-rate control. On observing a large number of packet retransmissions at the MAC layer, many WLAN cards adjust their sending rate to something that is more appropriate for the conditions. The exact details of the algorithm used to determine a good sending rate differ from card to card.

Of the two factors mentioned above, the latter can be taken care of as follows. If auto-rate is in use, we again employ trace-driven

# walls	loss rate	measured throughput	simulated throughput	difference
4	11.0%	15.52 Mbps	15.94 Mbps	2.63%
5	7.01%	12.56 Mbps	14.01 Mbps	11.54%
6	3.42%	12.97 Mbps	11.55 Mbps	-10.95%

Table 2: Estimated and measured throughput match, when we compensate the loss rates due to poor RF in the real measurements by seeding the corresponding link in a simulator with an equivalent loss rate.

simulation as follows: we monitor the rate at which the wireless card is operating, and provide it to the simulator (instead of having the simulator adapt in its own way). When auto-rate is not used (e.g., researchers have shown that auto-rate is undesirable when considering aggregate performance and therefore it should be turned off), the data rate is known.

The first issue is much harder to address because it may not be possible to accurately simulate the physical layer. One possible way to address this issue is through offline analysis. We calibrate the wireless cards under different scenarios and create a database to associate environmental factors with expected performance. For example, we carry out real measurements under different signal strengths and noise levels to create a mapping from signal strength and noise to loss rate. Using such a table in simulations allows us to distinguish between losses caused by collisions from losses caused by poor RF conditions. We evaluate the feasibility of this approach by computing the correlation coefficient between RSS and loss rates when the sending rate remains the same. We find the correlation coefficient ranges from -0.95 to -0.8. The high correlation suggests that it is feasible to estimate loss caused by poor RF conditions.

Based on this idea, in our experiment we collect another set of traces in which we slowly send out packets so that most losses are caused by poor signal (instead of congestion). We also place packet sniffer near both the sender and receiver, and derive the loss rate from the packet-level trace. Then we take into account the loss rate due to poor signal by seeding the wireless link in a simulator with a Bernoulli loss rate that matches the loss rate in real traces.

We find that after considering the impact of poor signal, the throughput from simulation matches closely with real measurements as shown in Table 2. Note that the loss rate and measured throughput do not monotonically decrease with the signal strength due to the effect of auto-rate, i.e., when the data rate decreases as a result of poor signal strength, the loss rate improves. (The driver of the wireless cards we use does not allow us to disable auto-rate.) Note that even though the match is not perfect, we do not expect this to be a problem in practice because several routing protocols try to avoid the use of poor quality links by employing some appropriate routing metrics (e.g., ETX [8]).

4.3 Remarks

In this section, we have shown that even though simulating a wireless network accurately is a hard problem, for the purpose of fault diagnosis, we can use trace-based simulations to reproduce what happened in the real network, after the fact. To substantiate this claim, we look at several simple scenarios and show that the throughput obtained from the simulator matches reality after taking into account information from real traces.

5. FAULT ISOLATION AND DIAGNOSIS

We now present our simulation-based diagnosis approach. Our high-level idea is to re-create the environment that resembles the real network inside a simulator. To find the root cause, we *search over a fault space* to determine which fault or set of faults can reproduce performance similar to what has been observed in the real network.

In Section 5.1 we extend the trace-driven simulation ideas presented in Section 4 to reproduce network topology and traffic pattern observed in the real network.

Using trace-driven simulation as a building block, we then develop a diagnosis algorithm to find root-causes for faults. The algorithm first establishes the expected performance (e.g., loss rate and throughput) under a given set of faults. Then based on the difference between the expected and observed performance, it efficiently searches over the fault space to re-produce the observed symptoms. This algorithm can not only diagnose multiple faults of the same type, but also perform well in the presence of multiple types of faults.

Finally we address the issue of how to diagnose faults when the trace data used to drive simulation contains errors. This is a practical problem since data in the real world is never perfect for a variety of reasons, such as measurement errors, nodes supplying false information, and software/hardware errors. To this end, we develop a technique to effectively eliminate erroneous data from the trace so that we can use good quality trace data to drive simulation-based fault diagnosis.

5.1 Trace-Driven Simulation

Taking advantage of trace data enables us to accurately capture the current environment and examine the effects of a given set of faults in the current network.

5.1.1 Trace Data Collection

We collect the following sets of data as input to a simulator for fault diagnosis:

- Network topology: Each node reports its neighbors. To be efficient, only changes in the set of neighbors are reported.
- Traffic statistics: Each node maintains counters for the volume of traffic sent to and received from its immediate neighbors. This data drives traffic simulation described in Section 5.1.2.
- Physical medium: Each node reports its noise level and the signal strength of the wireless links from its neighbors.
- Network performance: To detect anomalies, we compare the observed network performance with the expected performance from simulation. Network performance includes both link performance and end-to-end performance, both of which can be measured through a variety of metrics, such as packet loss rate, delay, and throughput. In this work, we focus on link level performance.

Data collection consists of two steps: collecting raw performance data at a local node and distributing the data to collection points for analysis. For local data collection, we can use a variety of tools, such as SNMP [6], packet sniffers (e.g., Airopeek [4]), WRAPI [20], and Native 802.11 [12].

Distributing the data to a manager introduces overhead. In Section 7.1, we quantify this overhead, and show it is low and has little impact on the data traffic in the network. Moreover, it is possible to further reduce the overhead using compression, delta encoding, multicast, and adaptive changes of the time scale and spatial scope of distribution.

5.1.2 Simulation Methodology

We classify the characteristics of the network that need to be matched in the simulator into the following three categories: (i) traffic load, (ii) wireless signal, and (iii) faults. Below we describe how to simulate each of these components.

Traffic Load Simulation: A key step in replicating the real network inside a simulator is to re-create the same traffic pattern. One

approach is to simulate end-to-end application demands. However, there can be potentially $N * N$ demands for an N-node network. Moreover, given the heterogeneity of application demands and the use of different transport protocols, such as TCP, UDP, and RTP, it is challenging to obtain end-to-end demands.

For scalability and to avoid the need for obtaining end-to-end demands and routing information, we use link-based traffic simulation. Our high-level idea is to adjust application-level sending rate at each link to match the observed link-level traffic counts. Doing this abstracts away higher layers such as the transport and the application layer, and allows us to concentrate only on packet size and traffic rate. However, matching the sending rate on a per-link basis in the simulator is non-trivial because we can only adjust the application-level sending rate, and have to obey the medium access control (MAC) protocol. This implies that we cannot directly control sending rate on a link. For example, when we set the application sending rate of a link to be 1 Mbps, the actual sending rate (on the air) can be lower due to back-off at the MAC layer, or higher due to MAC level retransmission. The issue is further complicated by interference, which introduces inter-dependency between sending rates on different links.

To address this issue, we use the following *iterative search* to determine the sending rate at each link. There are at least two search strategies: (i) multiplicative increase and multiplicative decrease, and (ii) additive increase and additive decrease. As shown in Figure 5, each link individually tries to reduce the difference between the current sending rate in the simulator and the actual sending rate in the real network. The process iterates until either the rate becomes close enough to the target rate (denoted as $targetMacSent$) or the maximum number of iterations is reached. We introduce a parameter α, where $\alpha \leq 1$, to dampen oscillation. In our evaluation, we use $\alpha = 0.5$ for $i \leq 20$, and $\frac{1}{i}$ for $i > 20$. This satisfies $\sum_i \alpha_i \rightarrow \infty$, and $\alpha_i \rightarrow 0$ as $i \rightarrow \infty$, and ensures convergence. Our evaluation uses multiplicative increase and multiplicative decrease, and we plan to compare it with additive increase and additive decrease in the future.

```
while (not converged and i < maxIterations)
    i = i + 1;
    if (option == multiplicative)
      foreach link(j)
        prevRatio = targetMacSent(j)/simMacSent(j);
        currRatio = (1 − α) + α * prevRatio;
        simAppSent(j) = prevAppSent(j) * currRatio;
    else // additive
      foreach link(j)
        diff = targetMacSent(j) − prevMacSent(j);
        simAppSent(j) = prevAppSent(j) + α * diff;
    run simulation using simAppSent as input
    determine simMacSent for all links from simulation results
    converged = isConverge(simMacSent, targetMacSent)
```

Figure 5: Searching for the application-level sending rate using either multiplicative increase, multiplicative decrease or additive increase additive decrease.

Wireless Signal: Signal strength has a very important impact on wireless network performance. As discussed in Section 4, due to variations across different wireless cards and environments, it is hard to come up with a general propagation model to capture all the factors. To address this issue, we drive simulation using the real measurement of signal strength and noise, which can be easily obtained using newer generation wireless cards (e.g., Native WiFi [12] cards).

Fault Injection: To examine the impact of faults on the network, we implement the ability to inject different types of faults into the simulator, namely (i) packet dropping at hosts, (ii) external noise sources, and (iii) MAC misbehavior [14].

- Packet dropping at hosts: a misbehaving node drops some traffic from one or more neighbors. This can occur due to hardware/software errors, buffer overflow, and/or malicious drops. The ability to detect such end-host packet dropping is useful, since it allows us to differentiate losses caused by end hosts from losses caused by the network.
- External noise sources: we support the ability to inject external noise sources in the network.
- MAC misbehavior: a faulty node does not follow the MAC etiquette and obtains an unfair share of the channel bandwidth. For example, in IEEE 802.11 [14], a faulty node can choose a smaller contention window (CW) to send traffic more aggressively [10].

In addition, we also generate link congestion by putting a high load on the network. Unlike the other types of faults, link congestion is implicitly captured by the traffic statistics gathered from each node. Therefore trace-driven simulation can directly assess the impact of link congestion. For the other three types of faults, we apply the algorithm described in Section 5.2 to diagnose them.

5.2 Fault Diagnosis Algorithm

We now describe an algorithm to systematically diagnose root causes for failures and performance problems.

General approach: Applying simulations to fault diagnosis enables us to reduce the original diagnosis problem to the problem of searching for a set of faults such that their injection results in an expected performance that matches well with observed performance. More formally, given a network settings, NS, our goal is to find $FaultSet$ such that $SimPerf(NS, FaultSet) \approx ObservedPerf$, where the performance is a function value, which can be quantified using different metrics. It is clear that the search space is high-dimensional due to many combinations of faults. To make the search efficient, we take advantage of the fact that different types of faults often change one or few metrics. For example, packet dropping at hosts only affects link loss rate, but not the other metrics. Therefore we can use the metrics in which the observed and expected performance have significant difference to guide our search. Below we introduce our algorithm.

Initial diagnosis: We start by considering a simple case where all faults are of the same type, and the faults do not have strong interactions. We will later extend the algorithm to handle more general cases, where we have multiple types of faults, or faults that interact with each other.

For ease of description, we use the following three types of faults as examples: packet dropping at hosts, external noise, and MAC misbehavior, but the same methodology can be extended to handle other types of faults once the symptoms of the fault are identified.

As shown in Figure 6, we use trace-driven simulation, fed with current network settings, to establish the expected performance. Based on the difference between the expected performance and observed performance, we first determine the type of faults using a decision tree as shown in Figure 7. Due to many factors, simulated performance is unlikely to be identical with the observed performance even in the absence of faults. Therefore we conclude that there are anomalies only when the difference exceeds a threshold. The fault classification scheme takes advantage of the fact that different faults exhibit different behaviors. While their behaviors are not completely non-overlapping (e.g., both noise sources and packet dropping at hosts increase loss rates; lowering CW increases the traffic and hence increases noise caused by interference), we can categorize the faults by checking the differentiating component first. For example, external noise sources increase noise experienced by its neighboring nodes, but do not increase the sending rates of any node, and therefore can be differentiated from MAC misbehavior and packet dropping at hosts.

After the fault type is determined, we then locate the faults by finding the set of nodes and links that have large differences between the observed and expected performance. The fault type determines what metric is used to quantify the performance difference. For instance, we identify packet dropping by finding links with large difference between the expected and observed loss rates. We determine the magnitude of the fault using a function $g()$, which maps the impact of a fault into its magnitude. For example, under the end-host packet dropping, $g()$ function is the identity function, since the difference in a link's loss rate can be directly mapped to a change in dropping rate on a link (fault's magnitude); under the external noise fault, $g()$ is a propagation function of a noise signal.

1) Let NS denote the network settings
(i.e., signal strength, traffic statistics, network topology)
Let $RealPerf$ denote the real network performance
2) $FaultSet = \{\}$
3) Predict $SimPerf$ by running simulation with input $(NS, FaultSet)$
4) if $|Diff(SimPerf, RealPerf)| > threshold$
determine the fault type ft using the decision tree shown in Fig. 7
for each link or node i
if $(|Diff_{ft}(SimPerf(i), RealPerf(i))| > threshold)$
add $fault(ft, i)$ with
$magnitude(i) = g(Diff_{ft}(SimPerf(i), RealPerf(i))$

Figure 6: Initial diagnosis: one pass diagnosis algorithm

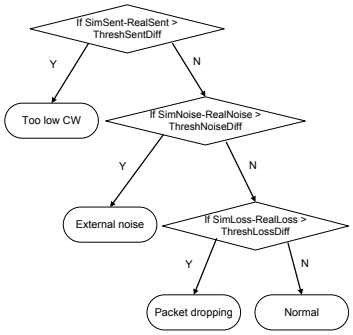

Figure 7: An algorithm to determine the type of faults

The algorithm: In general, we may have multiple faults interacting with each other. Such interactions may make the above one pass diagnosis insufficient. To address these challenges, we develop an iterative diagnosis algorithm, as shown in Figure 8, to find root causes.

The algorithm consists of two stages: (i) initial diagnosis stage, and (ii) iterative refinements. During the initial diagnosis stage, we apply the one-pass diagnosis algorithm described above to come up with the initial set of faults; then during the second stage, we iteratively refine the fault set by (i) adjusting the magnitude of the faults that have been already inserted into the fault set, and (ii) adding a new fault to the set if necessary. We iterate the process until the change in fault set is negligible (i.e., the fault types and locations do not change, and the magnitudes of the faults change very little).

We use an iterative approach to search for the magnitudes of the faults. At a high level, the approach is similar to the link-based simulation, described in Section 5.1.2, where we use the difference between the target and current values as a feedback to progressively move towards the target. During each iteration, we first estimate the expected network performance under the existing fault set. Then we compute the difference between simulated network performance (under the existing fault set) and real performance. Next we translate the difference in performance into change in faults' magnitudes using the function $g()$. After updating the faults with new magnitudes, we remove the faults whose magnitudes are too small.

In addition to searching for the correct magnitudes of the faults, we also iteratively refine the fault set by finding new faults that can best explain the difference between expected and observed performance. To control false positives, during each iteration we only add the fault that can explain the largest mismatch.

1) Let NS denote the network settings
(i.e., signal strength, traffic statistics, and network topology)
Let $RealPerf$ denote the real network performance
2) $FaultSet = \{\}$
3) Predict $SimPerf$ by running simulation with input $(NS, FaultSet)$
4) if $|Diff(SimPerf, RealPerf)| > threshold$
go to 5)
else
go to 7)
5) Initial diagnosis:
initialize $FaultSet$ by applying the algorithm in Fig. 6
6) while (not converged)
a) adjusting fault magnitude
for each fault type ft in $FaultSet$ (according to decision tree in Fig. 7)
for each fault i in $(FaultSet, ft)$
$magnitude(i) -= g(Diff_{ft}(SimPerf(i), RealPerf(i)))$
if $(|magnitude(i)| < threshold)$
delete the fault (ft, i)
b) adding new candidate faults if necessary
foreach fault type ft (in the order of decision tree in Fig. 7)
i) find a fault i s.t. it is not in $FaultSet$
and has the largest $|Diff_{ft}(SimPerf(i), RealPerf(i))|$
ii) if $(|Diff_{ft}(SimPerf(i), RealPerf(i))| > threshold)$
add (ft, i) to $FaultSet$ with
$magnitude(i) = g(Diff_{ft}(SimPerf(i), RealPerf(i))$
c) simulate
7) Report $FaultSet$

Figure 8: A complete diagnosis algorithm for possibly multiple fault types

5.3 Handling Imperfect Data

In the previous sections, we describe how to diagnose faults by using trace data to drive online simulation. In practice, the raw trace data collected may contain errors for various reasons as mentioned earlier. Therefore we need to clean the raw data before feeding it to a simulator for fault diagnosis.

To facilitate the data cleaning process, we introduce *neighbor monitoring*, in which each node reports performance and traffic statistics not only for its incoming/outgoing links, but also for other links within its communication range. Such information is available when a node is in the promiscuous mode, which is achievable using Native 802.11 [12].

Due to neighborhood monitoring, multiple reports from different nodes are likely to be submitted for each link. The redundant reports can be used to detect inconsistency. Assuming that the number of misbehaving nodes is small, our scheme identifies the misbehaving nodes as the minimum set of nodes that can explain the discrepancy in the reports. Based on the insight, we develop the following scheme.

For a link from node i to node j, a sender i reports the number of packets sent and the number of MAC-level acknowledgements received for a directed link l as $(sent_i(l), ack_i(l))$; a receiver j reports the number of packets received on the link as $recv_j(l)$; in addition, a sender or receiver's immediate neighbor k also reports the number of packets and MAC-level acknowledgement it observes sent or received on the link as $(sent_k(l), recv_k(l), ack_k(l))$. An inconsistency in the reports is defined as one of the following cases, where t is a given threshold used to mask the discrepancies caused by nodes unsynchronously sending their reports.

1. The number of packets received on a link, as reported by the destination, is noticeably larger than the number of packets sent on the same link, as reported by the source. That is, $recv_j(l) - sent_i(l) > t$.
2. The number of MAC-level acknowledgments on a link, as reported by the source, does not match the number of packets

received on that link, as reported by the destination. That is, $\mid ack_i(l) - recv_j(l) \mid > t$.

3. The number of packets received on a link, as reported by the destination's neighbor, is noticeably larger than the number of packets sent on the same link, as reported by the source. That is, $recv_k(l) - sent_i(l) > t$.
4. The number of packets sent on a link, as reported by the source's neighbor, is noticeably larger than the number of packets sent on the same link, as reported by the source. That is, $sent_k(l) - sent_i(l) > t$.

Note that, without inconsistent reports, the above constraints cannot be violated as a result of lossy links.

We then construct an *inconsistency graph* as follows. For each pair of nodes whose reports are identified as inconsistent, we add them to the inconsistency graph, if they are not already in the graph; we add an edge between the two nodes to reflect the inconsistency. Based on the assumption that most nodes send reliable reports, our goal is to find the smallest set of nodes that can explain all the inconsistency observed. This can be achieved by finding the smallest set of vertices that covers the graph, where the identified vertices represent the misbehaving nodes.

This is essentially the minimum vertex cover problem, which is known to be NP-hard. We apply a greedy algorithm, which iteratively picks and removes the node with the highest degree and its incident edges from the current inconsistency graph until no edges are left.

History of traffic reports can be used to further improve the accuracy of inconsistency detection. For example, we can continuously update the inconsistency graph with new reports without deleting previous information, and then apply the same greedy algorithm to identify misbehaving nodes.

6. SYSTEM IMPLEMENTATION

We have implemented a prototype of network monitoring and management module on the Windows XP platform. Our prototype consists of two separate components: *agents* and *managers*. An agent runs on every wireless node, and reports local information periodically or on-demand. A manager collects relevant information from agents and analyzes the information.

The two design principles we follow are: simplicity and extensibility. The information gathered and propagated for monitoring and management is cast into performance counters supported on Windows. Performance counters are essentially (name, value) pairs grouped by categories. This framework is easily extensible. We support both read-only and writable performance counters. The latter offer a way for an authorized manager to change the values and influence the behavior of a node in order to fix problems or initiate experiments remotely.

Each manager is also equipped with a graphical user interface (GUI) to allow an administrator to visualize the network as well as to issue management requests.

The manager is also connected to the back-end simulator. The information collected is processed and then converted into a script that drives the simulation producing fault diagnosis results. The capability of the network monitoring and management depends heavily on the information available for collection. We have seen welcoming trends in both wireless NICs and the standardization efforts to expose performance data and control at the physical and MAC layers, e.g., Native 802.11 NICs [12].

7. EVALUATION

In this section, we present evaluation results. We begin by quantifying the network overhead introduced by data collection and show its impact on the overall performance. Next, we evaluate the effectiveness of our diagnosis techniques and inconsistency detection scheme. We use simulations in some of our evaluation because this enables us to inject different types of faults in a controlled and repeatable manner. When evaluating in simulation, we diagnose traces collected from simulation runs that have injected faults. Finally we report our experience of applying the approach to a small-scale testbed. Even though the results from the testbed are limited by our inability to inject some types of faults (external noise and MAC misbehavior) in a controlled fashion, they demonstrate the feasibility of on-line simulations in a real system. Unless stated differently, the results from simulations are based on IEEE 802.11b and DSR routing protocol. The testbed uses IEEE 802.11a (to avoid interference with IEEE 802.11b infrastructure wireless network) and DSR-like routing protocol.

7.1 Data Collection Overhead

We evaluate the overhead of collecting traces, which will be used as inputs to diagnose faults in a network. We show that the data collection overhead is low and has little effect on application traffic in the network. Therefore it is feasible to use trace-driven simulation for fault diagnosis. We refer the readers to our technical report [15] for the detailed results.

7.2 Evaluation of Fault Diagnosis through Simulations

In this section, we evaluate our fault diagnosis approach through simulations in Qualnet.

Our general methodology of using simulation to evaluate fault diagnosis is as follows. We artificially inject a set of faults into a network, and obtain the traces of network topology and link load under faults. We then feed these traces into the fault diagnosis module to derive expected network performance (under normal network condition), which includes sending rate, noise, and loss rate as shown in Figure 7. Based on the difference between expected and actual performance, we infer the root cause. We quantify the diagnosis accuracy by comparing the inferred fault set with the fault set originally injected.

We use both grid topologies and random topologies for our evaluation. In a grid topology, only nodes horizontally or vertically adjacent can directly communicate with each other, whereas in random topologies, nodes are randomly placed in a region. We randomly select a varying number of nodes to exhibit one or more faults of the following types: packet dropping at hosts, external noise, and MAC misbehavior. To challenge our diagnosis scheme, we put a high load on the network by randomly picking 25 pairs of nodes to send one-way constant bit rate (CBR) traffic at a rate of 1 Mbps. Under such load, there is significant congestion loss, which makes it harder to identify losses caused by packet dropping at end-hosts. Correct identification of dropping links also implies reasonable assessment of congestion loss.

For a given number of faults and its composition, we conduct three random runs, with different traffic patterns and fault locations. We evaluate how accurate our fault diagnosis algorithm, described in Section 5.2, can locate the faults. The time that the diagnosis process takes depends on the size of topologies, the number of faults, and duration of the faulty traces. For example, diagnosing faults in 25-node topologies takes several minutes. Such diagnosis time scale is acceptable for diagnosing long-term performance problems. Moreover, the efficiency can be significantly improved through code optimization and caching previous simulation results.

We use *coverage* and *false positive* to quantify the accuracy of fault detection, where coverage represents the percentage of faulty locations that are correctly identified, and false positive is the number of (non-faulty) locations incorrectly identified as faulty divided

by the total number of true faults. We consider a fault is correctly identified when both its type and its location are correct. For packet dropping and external noise sources, we also compare the inferred faults' magnitudes with their true magnitudes.

Detecting packet dropping at hosts: First we evaluate the ability to detect packet dropping. We select a varying number of nodes to intentionally drop packets, where the dropping rate is chosen randomly between 0 and 100%. The number of such packet-dropping nodes is varied from 1 to 6.

We apply the diagnosis algorithm, which first uses trace-driven simulation to estimate the expected performance (i.e., noise level, throughput, and loss rates) in the current network. Since we observe a significant difference in loss rates, but not in the other two metrics, we suspect that there is packet dropping on these links. We locate such links by finding links whose loss rates are significantly higher than their expected loss rates. We use 15% as a threshold so that links whose difference between expected and observed loss rates exceed 15% are considered as packet dropping links. (Our empirical results suggest that 15% difference gives good balance between coverage and false positive in our evaluation, though a small change in threshold does not significantly change the diagnosis results.) We then inject packet dropping faults at such links with the dropping rates being the measured loss rate minus the simulated loss rate. We find that the injected faults reduce the difference between the simulated and observed performance.

Figure 9 shows the results for a 5×5 grid topology. Note that some of the faulty links do not carry enough traffic to meaningfully compute loss rates. In our evaluation, we use 250 packets as a threshold so that only for the links that send over 250 packets, loss rates are computed. We consider a faulty link sending less than a threshold number of packets as a *no-effect fault* since it drops only a small number of packets.[1] As Figure 9 shows the coverage (i.e., the fraction of detected faults and no effect faults) is over 80%, except in one case the coverage is lower, where one of the two faults is not identified. The false positive (not shown) is 0 except for two cases in which one link is misidentified as faulty. Moreover the accuracy does not degrade with the increasing number of faults. We also compare the difference between the inferred and true dropping rates. The inference error, computed as $\sum_i |infer_i - true_i| / \sum_i true_i$, is within 25%. This error is related to the threshold used to determine if the iteration has converged. In our simulations, we consider an iteration converges when changes in loss rates are all within 15%. We can further reduce the inference error by using a smaller threshold at a cost of longer running time. Also, in many cases it suffices to know where packet dropping occurs without knowing precise dropping rates.

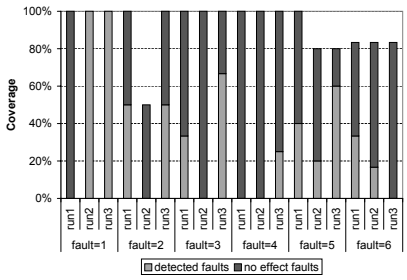

Figure 9: Accuracy of detecting packet dropping in a 5×5 grid topology

[1]These faulty links may have impact on route selection. That is, due to its high dropping rate, it is not selected to route much traffic. In this paper, we focus on diagnosing faults on data paths. As part of our future work, we plan to investigate how to diagnose faults on control paths.

Detecting external noise sources: Next we evaluate the accuracy of detecting external noise sources. We randomly select a varying number of nodes to generate ambient noise at 1.1e-8 mW. We again use the trace-driven simulation to estimate the expected performance under the current traffic and network topology when there is no noise source. Note that simulation is necessary to determine the expected noise level, because the noise experienced by a node consists of both ambient noise and noise due to interfering traffic; accurate simulation of network traffic is needed to determine the amount of noise contributed by interfering traffic. The diagnosis algorithm detects a significant difference (e.g., over 5e-9mW) in noise level at some nodes, and conjectures that these nodes generate extra noise. It then injects noise at these nodes with magnitude derived from the difference between expected and observed noise level to the simulator. After noise injection, it sees a close match between the observed and expected performance, and hence concludes that the network has the above faults.

Figure 10 shows the accuracy of detecting noise generating sources in a 5×5 grid topology. As we can see, in all cases noise sources are correctly identified with at most one false positive link. We also compare the inferred magnitudes of noises with their true magnitudes, and find the inference error, computed as $\sum_i |infer_i - true_i| / \sum_i true_i$, is within 2%.

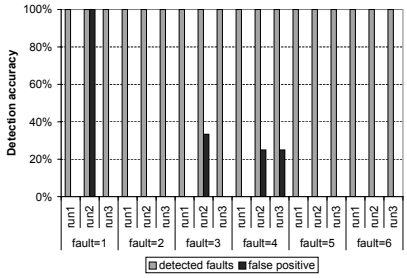

Figure 10: Accuracy of detecting external noise sources in a 5×5 grid topology

Detecting MAC misbehavior: Now we evaluate the accuracy of detecting MAC misbehavior. In our evaluation, we consider one implementation of MAC misbehavior. But since our diagnosis scheme is to detect unusually aggressive senders, it is general enough to detect other implementations of MAC misbehavior that exhibit similar symptoms. In our implementation, a faulty node alters its minimum and maximum MAC contention window in 802.11 (CWMin and CWMax) to be only half of the normal values. The faulty node continues to obey the CW updating rules (i.e., when transmission is successful, CW = CWMin, and when a node has to retransmit, CW = min((CW+1)*2-1, CWMax)). However since its CWMin and CWMax are both half of the normal, its CW is usually around half of the other nodes'. As a result, it transmits more aggressively than the other nodes. As one would expect, the advantage of using a lower CW is significant when network load is high. Hence we evaluate our detection scheme under a high load.

In our diagnosis, we use the trace-driven simulation to estimate the expected performance under the current traffic and network topology, and detect a significant discrepancy in throughput (e.g., the ratio between observed and expected throughput exceeds 1.25) on certain links. Therefore we suspect the corresponding senders have altered their CW. After injecting the CW faults at the suspected senders, we see a close match between the simulated and observed performance. Figure 11 shows the diagnosis accuracy in a 5×5 topology. We observe the coverage is mostly around 70% or higher. The false positive (not shown) is zero in most cases; the only case in which it is non-zero is when there is only one link

Topology	# Faults	4	6	8	10	12	14
25-node random	Coverage	100%	100%	75%	90%	75%	93%
	False positive	25%	0	0	0	0	7%
7×7 grid	Coverage	100%	83%	100%	70%	67%	71%
	False positive	0	0	0	0	8%	0

Table 3: Accuracy of detecting combinations of packet dropping, MAC-misbehavior, and external noises in other topologies

misidentified as faulty.

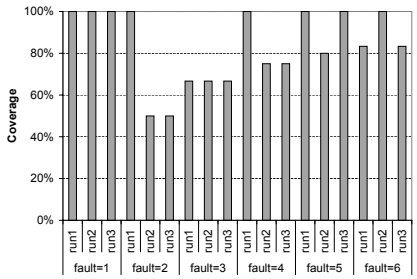

Figure 11: Accuracy of detecting MAC misbehavior in a 5×5 grid topology

Detecting mixtures of all three fault types: Finally we evaluate the diagnosis algorithm under mixtures of all three fault types as follows. As in the previous evaluation, we choose pairs of nodes adjacent to each other with one node randomly dropping one of its neighbors' traffic and the other node using an unusually small CW. In addition, we randomly select two nodes to generate external noise. Figure 12 summarizes the accuracy of fault diagnosis in a 5×5 topology. As it shows, the coverage is above 80%. The false positive (not shown) is close to 0. The accuracy remains high even when the number of faults in the network exceeds 10. The inference errors in links' dropping rate and noise level are within 15% and 3%, respectively.

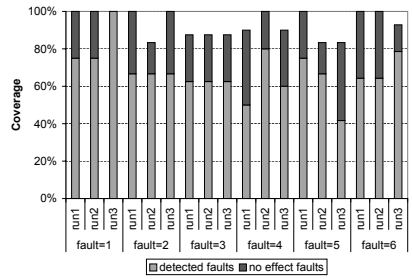

Figure 12: Accuracy of detecting combinations of packet dropping, MAC misbehavior, and external noises in a 5×5 grid topology

To test sensitivity of our results to the network size and type of topology, we then evaluate the accuracy of the diagnosis algorithm using a 7×7 grid topology and 25-node random topologies. In both cases, we randomly choose 25 pairs of nodes to send CBR traffic at 1 Mbps rate. Table 3 summarizes results of one random run. As it shows, we can identify most faults with few false positives.

Summary: To summarize, we have evaluated the fault diagnosis approach using a variety of scenarios, and shown it yields fairly accurate results.

7.3 Data Cleaning

To deal with data imperfectness, we process the raw data by applying the inconsistency detection scheme described in Section 5.3 before feeding them to the diagnosis module. We evaluate the effectiveness of this scheme using different network topologies, traffic patterns, and degrees of inconsistency. We again use coverage and false positive to quantify the accuracy, where coverage denotes the fraction of misbehaving nodes that are correctly identified, and false positive denotes the ratio between the number of nodes that are incorrectly identified as misbehaving and the number of true misbehaving nodes. We summarize our evaluation results as follow. Refer to our technical report [15] for the detailed results.

- The accuracy of detecting misbehaving nodes is high, with coverage above 80% and false positive below 15%. This is true even when 40% nodes in the system are misbehaving.
- The minimum node degree in a network topology is important to detection accuracy. As a result, topologies with high node density have high detection accuracy.
- Incorporating history information further improves the detection accuracy.

7.4 Evaluation of Fault Diagnosis in a Testbed

In this section, we evaluate our approach using experiments in a testbed. Our testbed consists of 4 laptops, each equipped with a Netgear WAG511 card operating in 802.11a mode. The laptops are located in the same office with good received signal strength. The traffic statistics on all links are periodically collected using the monitor tool, described in Section 6. We randomly pick a node to drop packets from one of its neighbors, and see if we can detect it. To resolve inconsistencies in traffic reports if any, we also run Airopeek [4] on another laptop. (Ideally we would like to have nodes monitor traffic in the promiscuous mode, e.g., using Native 802.11 [12], but since we currently do not have such cards, we use Airopeek to resolve inconsistencies.)

First, we run experiments under low traffic load, where each node sends CBR traffic at a rate varying from 1 Mbps to 4 Mbps to another node. We find the collected reports are consistent with what has been observed from Airopeek. Then we feed the traces to the simulator (also running in the 802.11a mode), and apply the diagnosis algorithm in Figure 8. Since one node is instructed to drop one of its neighbor's traffic at a rate varying from 20% to 50%, the diagnosis algorithm detects that there is a significant discrepancy between the expected and observed loss rates on one link, and correctly locates the dropping link.

Next we re-run the experiments under a high network load, where each node sends 8-Mbps CBR traffic. In this case, we observe that the traffic reports often deviate from the numbers seen in Airopeek. The deviation is caused by the fact that the NDIS driver for the NIC sometimes indicates sending success without actually attempting to send the packet to the air [13]. The new generation of wireless cards, such as Native 802.11 [12], will expose more detailed information about the packets, and enable more accurate accounting of traffic statistics. The inaccurate traffic reports observed in the current experiments also highlight the importance of cleaning the data before using them for diagnosis. In our experiment, we clean the data using Airopeek's reports, which capture almost all the packets in the air, and feed the cleaned data to the simulator to estimate the expected performance. Applying the same diagnosis scheme, we derive the expected congestion loss, based on which we correctly identify the dropping link.

8. DISCUSSION

To the best of our knowledge, ours is the first system that applies a network simulator to troubleshooting an operational multihop wireless network. The results are promising.

Our diagnosis system is not limited to the four types of faults discussed in this paper. Other faults such as routing misbehavior may also be diagnosed. Since routing misbehavior has been the

subject of much previous workwe focus on diagnosing faults on the data path, which have not received much attention. In general, the fault to be diagnosed determines the traces to collect and the level of simulation.

We focus on faults resulting from misbehaving but non-malicious nodes. Malicious faults are generally hard to detect as they can be disguised as benign faults. It would be interesting to study how security mechanisms (e.g., cryptographic schemes for authentication and integrity) and counter-measures such as secure traceroute can be incorporated into our system.

Our current system works with a fairly complete knowledge of the RF condition, traffic statistics, and link performance. Obtaining such complete information is sometimes difficult. It would be useful to investigate techniques that can work with incomplete data, i.e. data obtained from a subset of the network. This would improve the scalability of the troubleshooting system.

Finally, there is room for improvement in our core system as well. Our system depends on the accuracy and efficiency of the simulator, the quality of the trace data, and the fault search algorithm. Improvement in any one of these will result in better diagnosis. For example, our system could benefit from fast network simulation techniques developed by other researchers recently. Further, while we use network simulator to perform what-if analysis, we could potentially replace network simulator with a scalable and accurate network model if one is available. This may significantly speed up fault diagnosis. Moreover, it would be useful to understand the effects of heterogeneous devices and how they affect data collection, aggregation, and simulation. In addition, Bayesian inference techniques could be helpful for diagnosing faults that exhibit similar faulty behavior.

We are continuing our research and are in the process of enhancing the system to take corrective actions once the faults have been detected and diagnosed. We are also extending our implementation to manage a 50 node multihop wireless testbed. We intend to evaluate its performance when some of these nodes are mobile.

9. RELATED WORK

Many researchers have worked on problems that are related to network management in wireless networks. We broadly classify their work into three areas: (1) protocols for network management; (2) mechanisms for detecting and correcting routing and MAC misbehavior, and (3) general fault management.

In the area of network management protocols, Chen *et al.* [7] present Ad Hoc Network Management Protocol (ANMP), which uses hierarchical clustering to reduce the number of message exchanges between the manager and agents. Shen et al. [17] describe a distributed network management architecture with SNMP agents residing on every node. Our work differs from these two pieces of work in that we do not focus on the protocol for distributing management information, but instead on algorithms for identifying and diagnosing faults. Consequently, our work is complimentary to both [7] and [17].

A seminal piece of work in the area of detecting routing misbehavior is by Marti, Giullu, Lai, and Baker [11]. The authors address network unreliability problems stemming from selfish intent of individual nodes. The authors propose a *watchdog* and *pathrater* agent framework for mitigating routing misbehavior and improving reliability.

Different from a simple watch-dog mechanism, which considers end points as misbehaving when its adjacent links incur high loss rate, our diagnostic methodology takes into account current network configuration and traffic patterns to determine if the observed high loss rates are expected, and determines the root causes for the loss (e.g., whether it is due to RF interference, or congestion, or misbehaving nodes). In addition, we use multiple neighbors and take historical evidence into account to derive more accurate link loss rates.

In the area of wireless network fault management, there exist a number of commercial products in the market, such as AirWave [5], AirDefense [3], IBM's Wireless Security Auditor (WSA) [21], Computer Associate's UniCenter [1], Symbol's Wireless Network Management System (WNMS) [18], and Wibhu's SpectraMon [19]. Due to the proprietary nature of the products, their technical details are not available. Recently, Adya et al. [2] present architecture and techniques for diagnosing faults in IEEE 802.11 infrastructure networks. Our work differs from the above work in that the latter target wireless infrastructure networks, whereas multihop wireless networks are significantly different.

10. CONCLUSION

Troubleshooting a multihop wireless network is challenging due to the unpredictable physical medium, the distributed nature of the network, the complex interactions between various protocols, environmental factors, and potentially multiple faults. To address these challenges, we propose online trace-driven simulation as a troubleshooting tool.

We evaluate our system in different scenarios and show that it can detect and diagnose over 10 simultaneous faults of different types in a 25-node multihop wireless network. This result suggests that our approach is promising. An important property of our system is that it is flexible and can be extended to diagnose additional faults. We hope that this paper will inspire other researchers to further investigate trace-driven simulation as a tool to diagnose and manage complex wireless and wireline networks.

11. REFERENCES

[1] The future of wireless enterprise management. http://www3.ca.com/.

[2] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and techniques for diagnosing faults in IEEE 802.11 infrastructure networks. In *In Proc. of ACM MOBICOM*, Sept. 2004.

[3] AirDefense: Wireless LAN security and operational support. http://www.airdefense.net/.

[4] Wildpackets Airopeek. http://www.wildpackets.com/products/airopeek.

[5] Airwave, a wireless network management solution. http://www.airwave.com/.

[6] J. Case, M. Fedor, M. Schoffstall, and J. Darvin. A simple network management protocol (SNMP). In *Internet Engineering Task Force, RFC 1098*, May 1990.

[7] W. Chen, N. Jain, and S. Singh. ANMP: Ad hoc network management protocol. In *IEEE JSAC*, volume 17 (8), Aug. 1999.

[8] D. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proc. of ACM MOBICOM*, Sept. 2003.

[9] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dartmouth College, Computer Science, Hanover, NH, Jul. 2003.

[10] P. Kyasanur and N. Vaidya. Detection and handling of MAC layer misbehavior in wireless networks. In *Dependable Computing and Communications Symposium (DCC) at the International Conference on Dependable Systems and Networks (DSN)*, pages 173–182, San Francisco, CA, Jun. 2003.

[11] S. Marti, T. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of ACM MOBICOM*, Boston, MA, Aug. 2000.

[12] Native 802.11 framework for IEEE 802.11 networks. Windows Platform Design Notes, Mar. 2003. http://www.microsoft.com/whdc/hwdev/tech/network/802x/Native80211.mspx.

[13] Network devices and protocols: Windows DDK. NDIS library functions.

[14] L. M. S. C. of the IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Standard 802.11*, 1999.

[15] L. Qiu, P. Bahl, A. Rao, and L. Zhou. Fault detection, isolation, and diagnosis in multi-hop wireless networks. In *Microsoft Technical Report TR-2004-11*, 2004.

[16] The Qualnet simulator from Scalable Networks Inc. http://www.scalable-networks.com/.

[17] C.-C. Shen, C. Jaikaeo, C. Srisathapornphat, and Z. Huang. The Guerrilla management architecture for ad hoc networks. In *Proc. of IEEE MILCOM*, Anaheim, CA, Oct. 2002.

[18] SpectrumSoft: Wireless network management system, Symbol Technolgies Inc. http://www.symbol.com/.

[19] SpectraMon, Wibhu Technologies Inc. http://www.wibhu.com/.

[20] Wireless research API. http://ramp.ucsd.edu/pawn/wrapi/.

[21] Wireless security auditor (WSA). http://www.research.ibm.com/gsal/wsa/.