FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks

Jef Maerien, Pieter Agten, Christophe Huygens, and Wouter Joosen

IBBT-DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
B-3001, Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. Given the limited resources of wireless sensor network infrastructure, knowledge of the traffic generated by each node and service can be of great value. Yet, accounting and conveying this monitoring information in this low-resource infrastructure is challenging. Current monitoring solutions for wireless sensor networks either use passive monitoring, requiring a separate network, or focus only on detecting and debugging failures in the sensor network. This paper presents FAMoS: a flexible active monitoring service to collect wireless sensor network traffic volume and distribution data. The service is provided with limited overhead and is applicable in many contexts. Each node locally collects data about network traffic and then periodically transmits the data to a back-end for further analysis and processing. A prototype for the Contiki operating system is presented and evaluated in terms of runtime, memory and communication overhead.

Keywords: WSN, monitoring framework

1 Introduction

Wireless sensor networks (WSN) consist of embedded devices, connected by an ad-hoc wireless network. Characteristics include large scale, resource constraints, unreliable networking and a need for energy efficiency. These characteristics make application development particularly difficult. Care must be taken to achieve minimal energy consumption in order to maximize the sensor node lifetime. An in-depth understanding of the network traffic and its distribution over different nodes and applications can aid development. For example by comparing the amount of application traffic to the total amount of network traffic, one can estimate the need for further application optimization. Post-deployment situations can benefit from this information as well, for instance as an approach for assessing wireless network health.

This monitoring information can be gathered either (a) actively: the nodes themselves collect the data and transmit it to the back-end, or (b) passively: a secondary network of nodes is deployed which log the transmissions in the primary network and transmit it to the back-end using a separate channel.

Existing active monitoring approaches such as Sympathy [**?**] focus primarily on detecting and debugging failures and less on providing an overview of the traffic distribution in the WSN. Passive monitoring solutions, such as SNIF [**?**], Pimoto [**?**] and SNDS [**?**], can provide a very detailed view on WSN traffic but inherently require the use of a separate network for transporting the monitoring data. This leads to increased overhead cost and limits their use to smaller scale and controlled environments. Passive monitoring may also contradict security.

Our contribution to the monitoring field is a generic and efficient active monitoring framework for collecting information on the network traffic volume and distribution over different nodes and applications of a WSN. Architecturally, each node locally counts network traffic. This data is periodically sent to a back-end server for further processing and visualization. A prototype tested in a simulation environment showed a worst case transmission delay of 3 percent, ROM overhead of 9 percent, RAM overhead of 6 percent and communication overhead of 9 percent.

The rest of this paper proceeds as follows: section 2 describes three use cases and lists the requirements for efficient monitoring. Section 3 introduces the architecture of the monitoring service. Section 4 presents an implementation for the Contiki operating system. Section 5 evaluates this implementation in terms of runtime, memory and communication overhead. Section 6 compares our solution to related work. Section 7 describes some topics for future research on this subject. Finally, section 8 presents our conclusions.

2 Context and Requirements

2.1 Use Cases

This section lists the three motivating cases : (1) WSN Research, (2) Shared Sensor Networks, and (3) Network Administration.

WSN Research Research tasks such as the development and validation of WSN applications and protocols can benefit greatly from having detailed information about the distribution of network traffic. Depending on the particular problem at hand, a researcher might be interested in different views of the WSN traffic. Some problems require a very detailed view of the traffic passing through a particular part of a WSN test setup. Other situations need an overview of how traffic is distributed over the different sensor nodes, applications or protocols. In general, information from all layers of the network stack, in as much detail as possible, is useful in a research scenario.

Shared Sensor Networks In a shared wireless sensor network, applications of different stakeholders run on the network [**?**]. A *platform owner* operates a wireless sensor platform, consisting of sensor nodes and border routers, for use by several *application owners.* The platform owner wants to monitor the amount of traffic generated by each application so he can charge each application owner for the traffic volume consumed. In contrast to the research use case, the required information is limited: the platform owner mainly wants information about the traffic volume per application.

Network administration and security Network monitoring can be used for network administration and security purposes such as load balancing, problem detection and intrusion detection: (a) **load balancing**: load balancing can be done by monitoring the traffic sent by each node and assigning new tasks to the least burdened nodes; (b) **problem detection**: a change in route maintenance traffic in a certain part of the network could indicate problems such as as node failure or loss of connectivity; (c) **intrusion detection**: some attacks cause a significant change in the traffic patterns and can be detected by monitoring in combination with an anomaly-based intrusion detection system. For these three operational scenarios, it is sufficient to have a general overview of the network traffic divided into general categories such as application traffic, routing traffic, forwarded traffic and specific protocol traffic.

2.2 Requirements

This section lists both the functional and the non-functional requirements.

Functional FAMoS must be able to monitor both in-network transmission and transmissions from the nodes to the back-end infrastructure. It must provide data about many different categories of network traffic. The data collected by a single traffic category can be:

- **Per-application data**: Traffic generated by a node-application pair.
- **In-depth protocol data**: Traffic generated by network protocol messages.
- **Summarized data**: Traffic aggregated into general traffic categories such as application, relaying, routing, monitoring, etc.
- **Failed traffic data**: Traffic that is not sent due to errors in the stack.

Users must be able to (1) compare time aggregated volumes of a single traffic category between different nodes, (2) compare time aggregated volumes of different traffic categories for a single node, (3) view the time evolution of the traffic volume of a single category on a single node.

Non-functional The monitoring system must also satisfy a number of non-functional concerns, the most important of which are:

- **Flexible monitoring**: the data collection process should be customizable to a specific use case.
- **Low resource consumption**: Due to limited resources, the amount of network traffic and processing power needed must be minimized.
- **Transparency**: The monitoring service should require no modifications to existing applications.
- **Scalability**: The monitoring service must be applicable in a realistic, large-scale, industrial context.

3 Architecture

3.1 Overview

FAMoS uses active monitoring to gather the required data because the alternative monitoring methods cannot meet the requirements. Passive monitoring scales very poorly and is unable to detect failed traffic. Monitoring performed in the gateways or back-end infrastructure also is unable to detect failure, and cannot detect traffic flowing between nodes in the network.

Incorporating active monitoring into a system causes side effects, most importantly an influence on the WSN traffic as well as increased resource consumption. To prevent this potential inaccuracy, FAMoS is able to measure the traffic volume generated by itself.

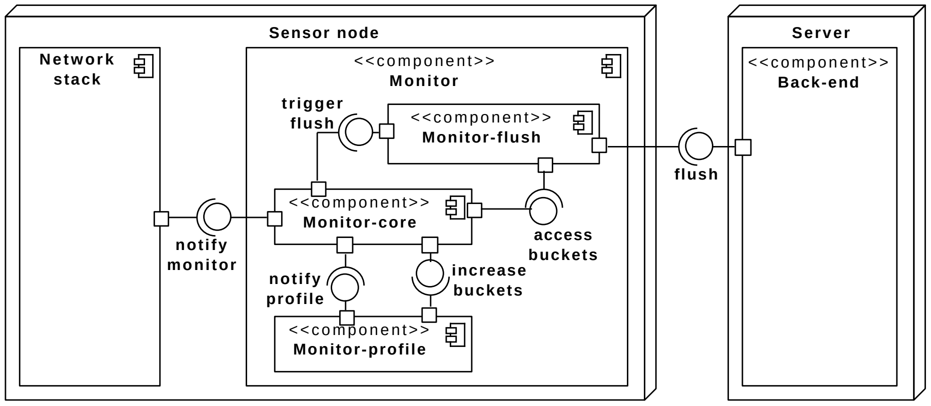

Fig. 1. The component and deployment model of FAMoS

Figure 1 illustrates the architecture of FAMoS. The architecture consists of two main components: a `monitoring` component on each sensor node and a `back-end` component deployed on a server. Code hooks are placed throughout each node's network stack. These hooks extract metadata about each transmitted packet and notify the `monitor-core` subcomponent just before a packet is transmitted. The `monitor-core` subcomponent delegates these notifications to the `monitor-profile` subcomponent, which uses the metadata to increase zero or more *buckets*. Buckets are byte counters which counts the number of bytes transmitted for a certain type of network traffic. The `monitor-flush` subcomponent periodically transmits the buckets to the back-end.

3.2 Hooks

Each hook extracts some specific metadata from the network stack and associates it with the packet currently being transmitted. Just before a packet is either copied into the hardware buffer for transmission or dropped because of an error, the accumulated metadata is sent to the `monitor-core` subcomponent. The metadata associated with each transmitted network packet can be divided into five categories:

– **Type**: The packet type such as TCP, UDP, ICMP Echo Reply, ...

– **Size**: The size of the packet as a whole and of the different headers.
– **Addresses**: Source and destination address at link and network layer. This allows differentiation between local and forwarded traffic.
– **Protocol-specific properties**: Additional properties, such as the source and destination ports for TCP and UDP packets.
– **Error information**: The cause if a packet is dropped by the network stack.

This metadata is collected without any modifications to the applications. However, for some use cases an application developer might find it useful to collect additional metadata. Therefore, FAMoS allows applications to attach additional metadata to transmitted packets.

3.3 Monitor Component

The `monitor` component is subdivided into the following three subcomponents:

Monitor-core The `monitor-core` subcomponent maintains the *buckets*. `Monitor-core` maintains two kinds of buckets: *simple buckets* and *BSD-buckets* (figure 2).

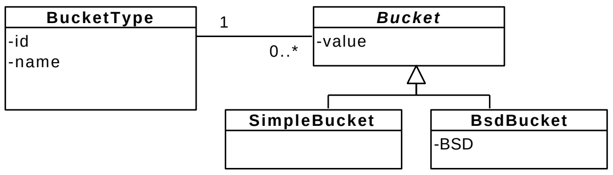

Fig. 2. Structure of simple and BSD buckets

Simple buckets are only associated with a certain *bucket type*, which determines the type of network traffic counted by that bucket, e.g. ICMP traffic and routing traffic. Each bucket type has a unique identifier to be able to distinguish buckets.

BSD-buckets are also associated with a bucket type but additionally contain a *bucket specific discriminator* value (BSD). This is a value indicating additional information about the network traffic counted by the BSD bucket. The monitor profile can fill out the BSD value as it sees fit. For instance, if a user wants to monitor traffic on each separate UDP port, he can define a single bucket type for UDP traffic and let the monitor profile allocate BSD buckets, with their BSD values equal to the UDP port for which they count the traffic.

Besides maintaining buckets, the `monitor-core` subcomponent has two other tasks: (1) receiving incoming notifications from the network stack and delegating them to the `monitor-profile` subcomponent, and (2) notifying the `monitor-flush` subcomponent to initiate a flush whenever a bucket reaches the threshold value.

Monitor-profile A monitor profile is a *compile-time replaceable* component that determines what network traffic information is collected and transmitted.

Users can choose a monitor profile to compile into the FAMoS framework for each node. Each monitor profile consists of two parts: (1) a number of configuration parameters and (2) the code that determines the behaviour of the profile. The configuration parameters consist of an ID and the number of simple and BSD buckets. Whenever `monitor-core` receives a notification from the network stack, it calls the monitor profile. Based on the collected metadata, the profile determines the delta for each bucket and returns this information back to `monitor-core`. This component then adds the delta to his buckets.

Monitor profiles allow the collected information and the amount of network traffic and processing power consumed by the monitoring service to be customized to the needs of the specific use case.

Monitor-flush After a predetermined period of time, all non-empty buckets are transmitted to the back-end. The `monitor-flush` subcomponent first creates a *flush packet*, which contains the value and identification number of each non-zero bucket. Next, `monitor-flush` clears all buckets, to allow traffic information for the next time period to be collected. Finally, the network packet is transmitted to the back-end. These steps must be performed atomically; if any step fails, the previous steps must be undone to prevent corruption or loss of traffic data.

To prevent a bucket from getting full, a non-periodic flush can be triggered by the `monitor-core` whenever a bucket reaches a certain threshold.

3.4 Back-end

Overview The back-end is responsible for the storage and visualisation of the traffic data collected by the sensor nodes. After successful reception of a flush packet, the back-end stores the associated traffic information into a database. The back-end user interface allows users to perform queries on the collected data, enabling them to get an overview of the traffic data collected from different nodes or to view the evolution in time of the traffic transmitted by a specific node.

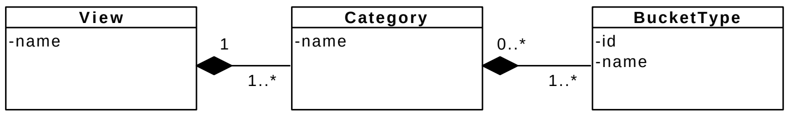

Fig. 3. Structure of views and categories

Data structuring In order to arrange the acquired data in a meaningful, structured way, FAMoS uses the concept of *views*. A view consists of a number of *categories* and each category aggregates a number of bucket types (3). Categories from the same view should be comparable to each other. For instance, a view named *summary* could consist of categories such as *ICMP traffic*, *routing traffic*, *forwarded traffic*, etc. For a valid comparison between the categories of a view, each bucket type included in the view should be based on the same packet size information, either including or excluding headers and data from higher or lower layers. Different views allow the acquired data to be structured in different ways, customized to the kind of information the user is trying to obtain. Furthermore, by aggregating matching bucket types from different monitor profiles into a single category, readings taken with different profiles can still be compared.

Whether or not a certain view can be used for structuring data depends on the monitor profile used to collect that data. Three cases can be identified regarding the compatibility between the categories of a view and the bucket types of a monitor profile: **(1) incompatible:** the category does not reference any bucket type of a certain profile, **(2) one-to-one compatible:** the category references exactly one bucket type of a certain profile, and **(3) one-to-many compatible:** the category references multiple bucket types from a single profile: the values from those buckets are aggregated. If all the categories in a view are compatible with one or a group of bucket types of a profile, the profile and the view are compatible.

4 Implementation

We implemented FAMoS for Contiki rc1 [**?**] running on AVR Raven sensor nodes using the uIPv6 network stack. Contiki is a lightweight, event based, open source operating system, written in C. This platform was selected because of its popularity and IPv6 support. The Contiki and back-end implementations are available online at *`http://code.google.com/p/famos/`*.

4.1 Hooks

Throughout the uIPv6 stack, 38 hooks are placed, ranging from the transport layer down to the MAC layer. At all locations where a packet exits the stack, either because it is transmitted or dropped, a hook sends the acquired metadata to the `monitor-core`. These transfers are implemented as function calls.

4.2 Buckets

Data types The buckets are implemented as 16-bit unsigned integers. This allows a maximum of 65 535 bytes to be collected in a single bucket.

BSD values are implemented as 16-bit values, a compromise between a large range and a small memory footprint. The BSD field is able to store a single TCP or UDP port number. Even though TCP and UDP packets contain a source and a destination port, generally only one port number offers useful information about the kind of traffic. For UDP packets, this is the destination port. For TCP packets, this depends on whether the TCP connection was initiated locally. For a locally initiated TCP connection, it is the destination port, while for a remotely initiated TCP connection, it is the source port.

Bucket allocation Simple buckets are implemented as a statically allocated array of unsigned integers. The size of this array is the number of simple buckets, determined by the monitoring profile. A simple bucket's identifier serves as the index, removing the need to store this separately.

For BSD buckets, memory is allocated statically and filled up as necessary. This method doesn't not suffer from memory fragmentation and enables very fast allocation and deallocation. Because each BSD bucket is given a position dynamically, the bucket identifier must be stored explicitly for each BSD bucket.

4.3 Flushing

The buckets are transmitted to the back-end using a custom protocol, which implements reliable connection on top of UDP by using acknowledgements. During development, a TCP-based protocol was considered, but tests have shown the TCP overhead to be significant.

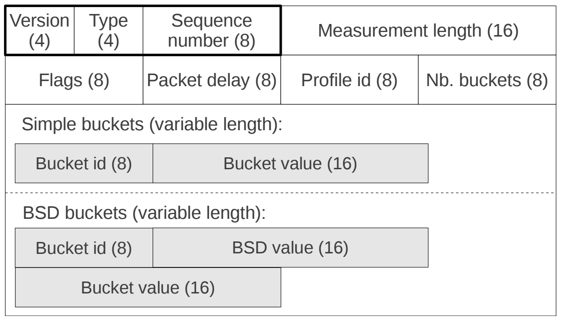

Fig. 4. The flush packet format. The numbers in parentheses indicate the number of bits required by each packet field.

Flush packets Figure 4 shows the flush packet format. The first three fields are for the basic transport protocol. The *measurement length* field indicates the collection time in seconds. The *flags* field indicates (1) whether there was a problem allocating a BSD bucket from the statically allocated array, (2) whether the previous attempt to transmit the buckets failed, and (3) whether this flush is periodical or triggered. The *packet delay* indicates the delay between packet creation and transmission, in case of retransmission. The *profile id* field is the monitor profile identifier. The *nb buckets* field indicates the number of buckets that follow. Next, the flush packet contains the simple buckets and finally the BSD buckets. Only non-zero buckets are transmitted.

4.4 Monitor Profiles

Four monitoring profiles have been developed, but users can easily add their own. The profiles broadly correspond to the use cases listed in section 2.1.

- **Research profile**: The research profile corresponds to the *WSN research* use case and collects detailed traffic data from all layers of the network stack. It defines 25 simple bucket types which count all ICMP, ND and RPL packets and 7 BSD bucket types which count TCP and UDP packets.
- **Application profile**: The application profile corresponds to the *shared sensor networks* use case and collects information about the traffic generated by each application. It defines three bucket types: TCP (BSD), UDP (BSD), and other traffic (simple).
- **Summary profile**: The summary profile corresponds to the *network administration and security* use case. The traffic is divided into nine traffic categories: application, ICMP, ND, RPL, FAMoS, forwarded non-FAMoS, FAMoS to back-end, FAMoS from back-end and errors.
- **Load balancing profile**: The load balancing profile corresponds to the load balancing part of the *network administration and security* use case and uses only one simple bucket type: total traffic.

4.5 Back-end

The back-end is implemented as a J2EE web application. J2EE is chosen for its ease of use to make a prototype and the absence of resource constraints in the back-end. It can display the total traffic per category per node in a certain time period or the evolution of traffic per category for a single node in a certain time period. Several views have been included by default, but users can add their own. Some example views are:

View	**Description**
Summary	Traffic divided into eight general categories
Application	Traffic divided per TCP/UDP port
RPL	Traffic for different types of RPL packets
Total	The total amount of transmitted traffic
Errors	Traffic not transmitted because of different kinds of errors

5 Evaluation

The above implementation was evaluated in terms of runtime overhead, memory footprint and communication overhead. Tests were performed on the AVR Raven platform. No energy consumption tests were performed. However, most energy is spent by sending and receiving packets, hence the communication overhead is a fairly accurate and platform independent predictor of energy overhead.

5.1 Runtime Overhead

We performed two tests for measuring the runtime overhead for sending a packet. The first test measures the overhead in *processing time* without transmission. The second test measures the total time for sending a message including transmission. Both tests are based on measuring the time for sending 1000 UDP packets with a payload of 16 bytes to 20 different destination ports. Table 1 shows the results of these measurements. All measurements were performed 5 times and had a relative standard deviation of less than 0.5%. These results indicate a maximum processing time overhead of 11% and a maximum delay of 3%, both when using the research profile.

	Processing (ms)	Delay (ms)
Without FAMoS	1360 (100%)	4848 (100%)
Research profile	1504 (+11%)	4982 (+3%)
Application profile	1464 (+7%)	4944 (+2%)
Summary profile	1424 (+5%)	4896 (+1%)
Load balancing profile	1416 (+4%)	4898 (+1%)

Table 1. Runtime overhead in milliseconds of sending 1000 packets

5.2 Memory Footprint

Table 2 shows the program and data memory footprint of the monitoring service. These numbers indicate the amount of additional memory usage for the monitoring service, using the four implemented monitor profiles. The baseline case is a minimal Contiki build without running any applications. The maximum program memory overhead is 9% and the maximum data memory overhead is 6%, both when using the research profile.

	Program memory (B)	Data memory (B)
Without FAMoS	47 028 (100%)	9898 (100%)
Research profile	51 280 (+9%)	10 504 (+6%)
Application profile	50 776 (+8%)	10 269 (+4%)
Summary profile	50 494 (+7%)	10 154 (+3%)
Load balancing profile	50 318 (+7%)	10 114 (+2%)

Table 2. Memory footprint of FAMoS in bytes of memory

5.3 Communication Overhead

To measure the communication overhead of the monitoring service, a test setup with 6 AVR Raven sensor nodes was used. These nodes were connected to the back-end using an AVR Jackdaw USB stick. The test used RPL as routing protocol and used 6LoWPAN stateful IPv6 header compression (HC06). Figure 5 shows the network set up. Four nodes were programmed to send a temperature reading every 10 seconds, two using UDP and the other two using TCP. The remaining two nodes were set up as routers which only forward traffic.

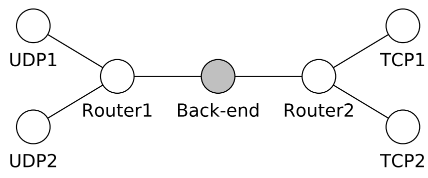

Fig. 5. The setup of the evaluation environment. The edges indicate the routing paths created by the RPL protocol.

	Communication overhead (B/s)
Research profile	2.8 (+9%)
Application profile	2.5 (+8%)
Summary profile	2.2 (+7%)
Load balancing profile	1.6 (+5%)

Table 3. Communication overhead of FAMoS in B/s with a 60 second flush interval

Comparison of monitor profiles Table 3 shows the communication overhead of FAMoS using each of the four monitor profiles. Each of these tests was performed over a time period of 20 minutes, during which the buckets were flushed every 60 seconds. This means a flush is performed every 6 temperature transmissions, which can be considered a worst-case scenario. The overhead is calculated as the additional number of bytes transmitted by all nodes in the setup, divided by the length of the measurement period in seconds. The largest communication overhead occurred with the research profile and was equal to 2.8 B/s. This is an overhead of 9% in comparison to the total traffic without the monitoring service.

Influence of flush interval When using a flush interval of 60 seconds, the largest transmitted bucket contained only 6229 bytes, which is less than 10% of a bucket's maximum value.

Figure 6 shows the effect on the communication overhead of increasing the time interval between flushes. For this test, the research profile was used. The figure indicates that at small flush intervals by doubling the flush interval, the overhead is halved. At large flush intervals, this effect stops since some buckets will get full, requiring a non-periodic flush. In the test setup, the nodes were programmed to trigger a flush whenever a bucket reached 80% of its maximum value. At interval lengths up to and including 8 minutes, this situation did not occur. Beyond 8 minutes, non-periodic flushes were triggered by routing node 2. When the flush interval increases further, more and more nodes started performing non-periodic flushes, eventually reaching the point where all nodes only perform non-periodic flushes. This point was not reached during the tests, even when using a flush interval of 128 minutes.

From these results we can conclude that the communication overhead drops quickly when increasing the flush interval. At an interval of 8 minutes, the overhead is 0.305 B/s, which is less than 1%. If the flush interval can be large, the

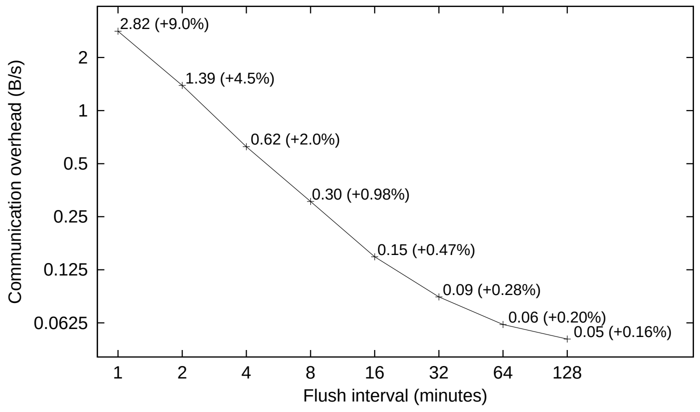

Fig. 6. Communication overhead as a function of the flush interval, when using the research monitor profile. The percentages in parentheses indicate the communication overhead relative to a setup without monitoring averaged over all nodes in the test.

communication overhead can even be brought down to less than 0.05 B/s. Some situations require a high resolution of the evolution of network traffic or require a low response time to changes in network activity. These situations are not suited for large flush intervals. However, in situations where the WSN must be monitored for an extended period of time or where quick response times are not important, large flush intervals are acceptable.

6 Related Work

Table 4 shows a comparison between FAMoS and related work, based on the requirements listed in 2.2. The first three rows indicate the types of traffic information collected by each system. The last four rows show how the systems adhere to non-functional requirements.

	FAMoS	SNMP	Sympathy	Passive
Application data	++	-	+	++
In-depth protocol data	+	-	+	++
General overview	++	-	+	++
Flexible monitoring	++	+	-	-
Low node resource consumption	+	-	+	+++
Transparent for applications	++	+	-	++
Scalable	+	-	+	-

Table 4. Comparison between FAMoS, SNMP, Sympathy and passive monitoring

SNMP The Simple Network Management Protocol [**?**] is an application-layer protocol that allows a host to request information from networked devices.

SNMP only provides a protocol and does not define what information a device must offer. Instead, each device has a management information base (MIB), determining the information it offers. As a consequence, SNMP does not directly provide end-to-end support for monitoring any kind of network traffic. MIBs, like monitor profiles, provide flexibility in the information to be collected and are a piece of shared state between the nodes and the back-end. However, SNMP uses hierarchical, globally unique identifiers to request variables from devices. These identifiers are much longer than those used in FAMoS, requiring more bytes to send and store, increasing resource consumption and reducing scalability.

Sympathy Sympathy [**?**] is a debugging tool for sensor networks. In Sympathy, nodes periodically send metrics to a back-end, which aggregates these metrics and combines them with passively collected metrics, in order to detect and debug failures. Metrics include nodes' next hops, neighbours and uptime, and the number of packets transmitted and received for different kinds of network packets. However Sympathy requires applications to be instrumented with custom code. As for resource consumption, Sympathy consumes little RAM and ROM but does have a relatively large overhead in bandwidth usage, reducing scalability.

Passive monitoring Passive monitoring uses a secondary network of sniffer nodes to capture node transmissions and send them to a gateway computer. Three systems each propose a different way to organize the secondary network: Pimoto[**?**] uses Bluetooth to connect each sniffer node to the gateway, SNIF [**?**] uses Bluetooth to create a secondary WSN and SNDS[**?**] connects each sniffer node using wired ethernet. This approach offers a very fine grained view on traffic from all layers of the network stack and does not pose any additional burden on the WSN under observation. The downside however is that a lot of additional hardware is necessary, limiting scalability and comprising node mobility. Also, passive monitoring does not allow monitoring encrypted traffic nor detection of errors in the network stack.

7 Future Work

The current evaluation was performed in a limited test setup. Future work will include validating FAMoS in an industrial, large scale setup and in networks with high rates of loss and churn.

FAMoS does not implement any security features. Malicious users can intercept and modify flush packets. This vulnerability is especially critical if the traffic data is used, for instance, to check and enforce a network traffic policy. Therefore, future work will focus on improving the security of the framework.

Parameters such as the flush interval and the non-periodic flush bucket threshold are fixed at compile-time. The framework will be extended to support run-time change of these. Additionally, support for using a run-time adjustable monitor profile will be added. These features would allow the back-end to dynamically reconfigure the behaviour of the `monitor` component.

8 Conclusion

We have developed FAMoS, a flexible active monitoring service for wireless sensor networks, which provides an overview of the volume and distribution of network traffic over different nodes and applications of a wireless sensor network. FAMoS is flexible in the sense that it can be customized to many different use cases by the means of monitor profiles. In most configurations, runtime, memory and communication overheads are under 10%.

These results indicate that using an active approach for network monitoring in wireless sensor networks is not only feasible but also efficient, and can be applied in many practical circumstances.

Acknowledgements

Research funded by a Ph.D. grant of the Agency for Innovation by Science and Technology (IWT). Pieter Agten holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven and was conducted in the context of the IWT-SBO-STADIUM project No. 80037.