Monitoring Subsystem for Wireless Systems Based on Miniature Spectrum Analyzers

Ivan Bogachuk,
Department of Information and Cyber Security
State University of Telecommunications
Kyiv, Ukraine
vanya.bogachuk@gmail.com

Volodymyr Sokolov, Volodymyr Buriachok
Department.of Information and Cyber Security
Borys Grinchenko Kyiv University
Kyiv, Ukraine
vladimir.y.sokolov@gmail.com,
v.buriachok@kubg.edu.ua

***Abstract*— The paper presents a substantiation of the effectiveness of the implementation of IEEE 802.11 wireless network analysis subsystem using miniature spectrum analyzers. A practical implementation scheme, approaches to the software solution and hardware are shown. Design and implementation is presented.**

Keywords*— *dynamic channel allocation; access point; integrity; availability; spectrum analyzer

I. INTRODUCTION

To ensure the safety of roaming wireless infrastructure, namely, the integrity and availability at the same time. The development of backup and monitoring subsystems allows solving problems related to the availability and integrity of the transmitted data, thus increasing the security of the entire telecommunications system as a whole.

The problem of accessibility and integrity is solved in different ways: by modifying protocols [1], antenna solutions [2] or by changing the architecture [3]. On the basis of the scheme and algorithm given in [4], the data analysis sub-system for the wireless system of the IEEE 802.11 standard is implemented in this paper (fig. 1).

Wireless systems can be in three states:

- •Regular work mode.
- •Critical mode.
- •Denial of service.

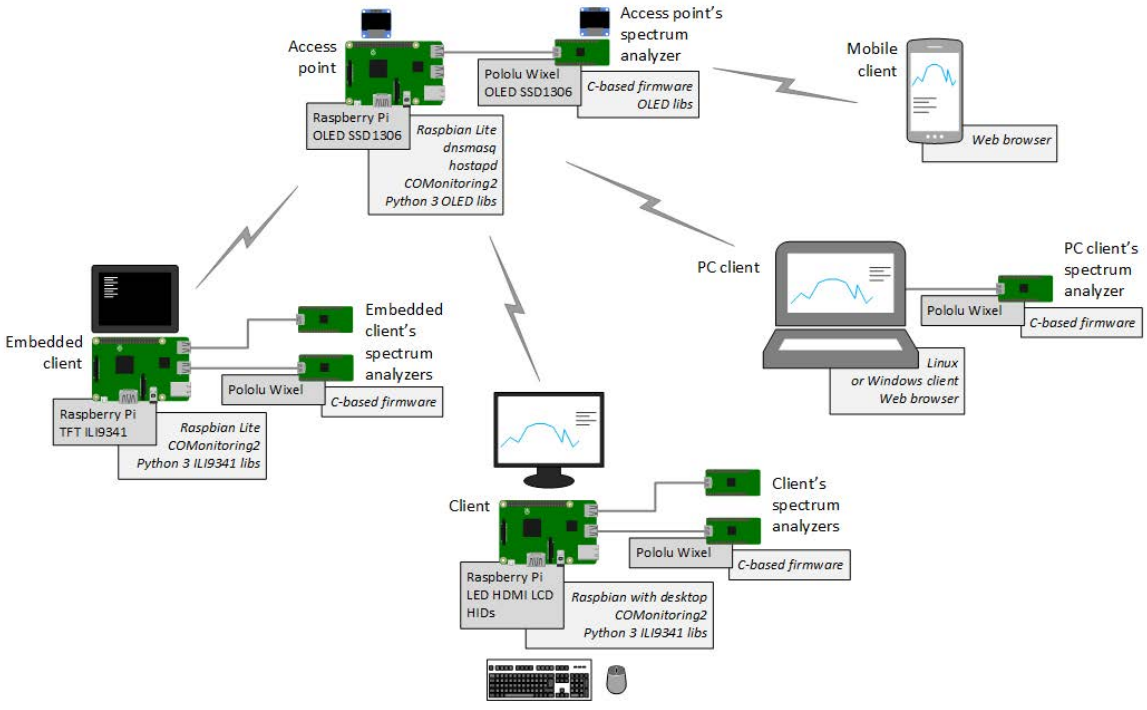

Fig. 1. Spectrum analysis subsystem

Introduction to the wireless system of the subsystem with spectrum analyzers leads to the fact that the onset of a critical mode of operation is less likely, since subscribers on overloaded access points are redirected to adjacent access points not only at the maximum signal level, but load distribution algorithms can be much more complicated [5].

The reminder of the paper is organized as follows. In section II "Rationale for the Efficiency of a Wireless Network with Spectrum Analyzers" provides a rationale for the effectiveness of using a subsystem with spectrum analyzers. Section III "Development Toolkit" gives an overview of software and related hardware. Our approaches to implementation and evolution of software

978-1-5386-6611-1/18/$31.00 ©2018 IEEE

are given in section IV "Monitoring System Development Approaches". The current state of subsystem development is given in section V "Current Development Stage". Design and hardware implementation of our system is given in section VI "Implementation". The paper end with section VII "Conclusion and Future Work and References".

II. Rationale for the Efficiency of a Wireless Network with Spectrum Analyzers

Determine the efficiency factor for the wireless system:

$$K = {}^{E}/_{C}, \quad (1)$$

where E is the efficiency, and C is the cost.

Determine the cost of the usual wireless system:

$$C = P_i + P_s, \quad (2)$$

where P_i is the cost of infrastructure; P_s is the cost of service.

And the cost of a system with miniature spectrum analyzers, taking into account (1):

$$C_{sa} = P_i + P_s + P_{sa} = C + P_{sa}, \quad (3)$$

where P_{sa} is the cost of the subsystem with spectrum analyzers. The cost of service is formed in both cases with the constant salary of the attendants, and therefore remains unchanged.

Wireless subscribers of an enterprise or organization can be divided into three categories:

- Mobile (cell phones, tablets, etc.).
- Conditionally moving (laptops).
- Fixed (fixed PCs).

Determine the number of movable as Nm, conditionally moving is N_{cm} and fixed is N_f. The number of movable roughly equals the total number of conditionally moving and stationary, since virtually every worker has a personal mobile terminal (if the rules of the internal order do not provide for restrictions on the use of the corporate network):

$$N_m \approx N_{cm} + N_f, \quad (4)$$

The cost of the subsystem with spectrum analyzers is:

$$P_{sa} = \left(N_{cm} + N_f\right)P^*_{sa} + P_c, \quad (5)$$

where P^*_{sa} is the cost of one spectrum analyzer, P_c is the cost of the controller responsible for collecting, analyzing data from all spectrum analyzers, and generating tips for the main wireless controller. The number of mobile subscribers is not taken into account, since it is difficult to install additional equipment on them.

The cost of infrastructure depends on the number of subscribers:

$$P_i \sim N_m + N_{cm} + N_f + N_g, \quad (6)$$

where N_g is number of guest subscribers.

We introduce the openness parameter σ of the wireless system:

$$N_g = \sigma N_m, \quad (7)$$

Then from (4), (6) and (7) we have a full number of subscribers:

$$N \approx (2 + \sigma)(N_{cm} + N_f), \quad (8)$$

The efficiency of an ordinary wireless network is directly proportional to the subscriber's minimum access time to access point resources:

$$E \sim T_a^{min} = \frac{\Delta T}{N^{max}}, \quad (9)$$

where ΔT is the size of the time window, $N_{\max}$ is the maximum number of subscribers per access point (depending on the manufacturer is 30 to 50).

Wireless network performance with spectrum analyzers:

$$E_{sa} \sim \frac{N_{AP}}{N}\Delta T, \quad (10)$$

where N_{AP} is the number of access points.

Estimate the quality of work can be based on the ratio of efficiency coefficients:

$$\frac{K}{K_{sa}} = \frac{E \cdot C_{sa}}{E_{sa} \cdot C} = \frac{N}{N_{AP}} \cdot \frac{1}{N^{max}} \cdot \left(1 + \frac{P_{sa}}{P_i + P_s}\right), \quad (11)$$

III. Development Toolkit

The common monitoring system schema that was described above requires that the application embodies such main features:

- Decentralization.
- Cross-platform.
- Ability to launch by the single-board computer (e.g. Raspberry Pi v3 or higher).

The Python is very suitable tool to satisfy our task's requirements. Beside of that Python has powerful standard library, it also has big repository with useful packages and modules for different development directions. For example, you can use *peserial* package to works with Pololu Wixel modules [6]. This is great and simple tool for managing COM ports. As our system have to be cross-platform, we choose such group of used operating systems: *Raspbian*, *Ubuntu*, *Windows*. The first two are Linux distributions that supplied with the Python interpreter installed by default. Other good points are:

- Great friendly community.
- Good documentation for modules and packages.
- Development speed.

Although many Linux distributions have Python 3.5 interpreter by default, it was decided to use Python of 3.6.6 stable version (Pyhton 3.7 stable is available now) [7],

because it has important improvements for our app, in direction of:

- Security.
- CPython implementation.

IV. Monitoring System Development Approaches

The current development stage has three different app versions. The last app version is named cmtg (the short abbreviation of the first version name COMonitoring). The monitoring system development has significant path that associated with different solutions, issues, improvements and optimization methods.

The first version has client-server architecture with client and server Python script separated (fig. 2).

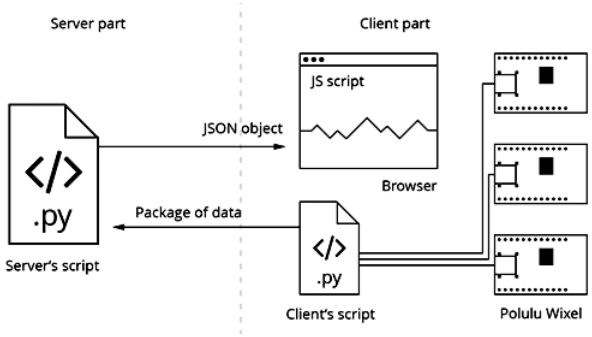

Fig. 2. COMonitoring working scheme

To run the system they have to be on different machines that connected to the same access point. Firstly, you need to launch the server's script and then the client's script. The client's script have to find server and starts to handle Wixel modules data to send them to the server. Client and server use WebSocket protocol for communication. After that, you can monitor the signal level values via browser.

In general, this version shows that idea of the monitoring system on the Python works and browser interface is very convenient solution for data analyzation. However, there are some important disadvantages:

- This isn't decentralized monitoring system, so it has low level of fault tolerance.
- Complicated launch scheme.
- Low server reliable.
- Bad thoughtful of app logic.

These disadvantages were the reason to make conclusions and create the system from scratch. The second app version has new working scheme and several important improvements. The key point is that we have single application that can do both of client's and server's jobs. We used the threading Python module to run the server's job on the separate thread. To provide this functional we came up with such algorithm:

- The app runs.
- It tries to find the working server in the current local network. The application sends the UDP package to signalize to the server, that new client was appeared in the network.
- If the worked server is in the current network and retrieved this package, it sends response to the client with self-address (URL) via TCP connection. After that, the client can communicate to the server. Otherwise if the server is absent, client runs the own server thread. In the future, other clients can connect to the current client's server, using the same steps.

In addition to, there is the method to save the all connected clients at the current network. It means that all clients know who is connected to the network and who the server is. This list is used in the case, when the server goes down and all clients have to be able to reconnect to the new server. The selection of new server depends on the list order, specific client have to run the server thread. This functional implements the system decentralization, but works very unstable and unpredictable. The reason is that we had little time for testing. However, there are some other cool features:

- Added new algorithm to store and handle signal level values. We save some part of values for each Wixel module's channel in the queue data structure (FILO) and can calculate the average values for specific range. This method allows to us to analyze the network state a much better.
- Strong changed the web-interface structure. There were added the unique color creation method for visual module identification. In addition, the interface functional provides the chart control. You can chose the range for average signal values calculation and disable chart's rendering for specific modules.
- Some security improvement. There was added the signature to check data integrity of incoming packages from client.
- Added simple console interface to output the information about the app working state.

Beside of this features, in the result it turned out that the system is unreliable. It has the same problem with app logic as the first version, but in this case, the system is very complicated and bulky. The most of entities in the app depend between themselves that creates additional problems with flexibility and scalability.

Therefore, the third (current) version is union of the best from the both first two versions and new key changes. It was developed the new working schema to solve all issues that were in previous versions (fig. 3).

We have the single application as in the second version. Each client in the system is named "node". The app structure combines two ideas: using threads (as the second version) and the microservice architecture. You can see that each app includes three main entities: controller, node manager and device manager. These entities are analogue of the service in the microservice architecture. They are fully separated and work in different threads.

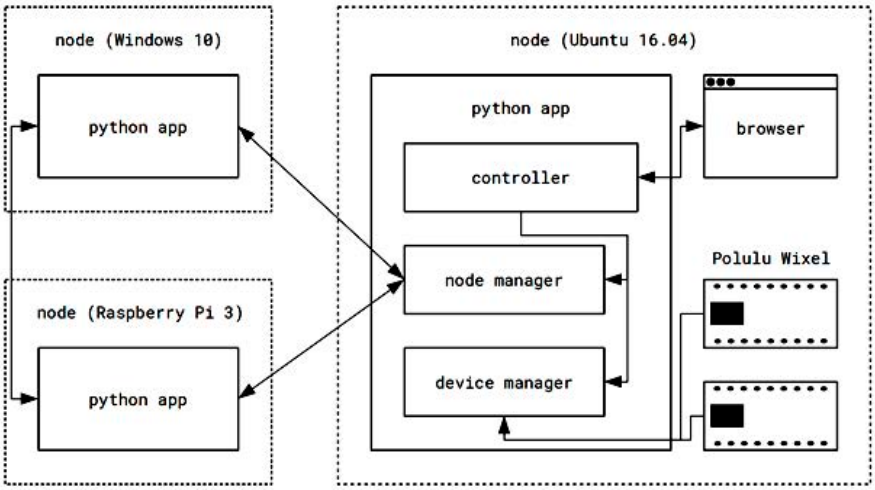

Fig. 3. Cmtg working scheme

The device manager provides functional for working with modules. First two versions works with a single thread for every module. The device manager creates new thread for each module to work with it in the current version. The main task of device manager is to retrieves data from module and store it in special data structure.

The node manager is responding for nodes communication in the system. It binds the listening UDP socket that accept broadcast packages, in the one thread. In another thread, it sends broadcast UDP packages. This package contains data about the current node, such as, machine name, operating system name, MAC address, host, port. Therefor every node is listening and sending packages, so they all know about who is connected to the current network. Every node stores all this information in the system.

The controller entity handles client requests and communicates with node and device managers. This is the app server. The node manager and the device manager doesn't know about each other, unlike the controller. For example, when the client sends request to get data about nodes that are connected now, controller takes specific data from the node manager storage and response to the client. It makes the same steps, when client want to get data about devices and signal level values, but uses the device manager.

The most interesting part is how web-interface allows us to communicate with all system nodes, because we doesn't have centralized node that handle all connection, every node has controller entity and server. To understand this scheme better, it would be good to divide the app lifecycle by steps:

1. The user run the app in the terminal (console). He can see the console app interface, that shows the system working state. The second version has the same functional.
2. The node manager and device manager start working in their own threads.
3. After that, user can see if the network has other connected nodes. The console interface shows node's URL address that allows us to communicate with this node via browser.
4. When we open browser and connect to the node, the node's server response the html page with JS script. The JS script makes three main functions:

- Store current URL.
- Start the infinity loop that always requests data about all connected nodes.
- Start the infinity loop that always requests data about devices of the current node.

Both of the loop is associated with the stored URL. Therefore, if we want to request data from another node, we just need to change this URL to the specific node's URL. This action can occurs CORS errors in browser, so we use the flask-cors package to solve this issue.

In the third version, we have achieved that all previous issues are solved. The current app scheme works better. Besides of that it is decentralized system, it also uses separated threads, so system has high fault tolerance level. This is very important for our field. The monitoring system has powerful foundation for future scaling.

V. CURRENT DEVELOPMENT STAGE

Today we work to implement new functional for such directions:

- Security.
- Logging system.
- Web-interface improvements.
- CPU load and memory usage optimization.

To make the app more safety, there is idea to use the encryption algorithm for packages that are transmitted between nodes in the monitoring system.

The user of the monitoring system needs to be able to trace the system working process in convenient form via browser. It is necessary feature to provide the monitoring system condition assessment. Based on the previous point, we need to change web-interface for logging functional supporting.

Most success we have achieved in optimization direction. It was decided to dive in through studying of used Python modules and built-in functionality. As a result, we were able to reduce CPU load by 80% and memory usage by 50%, but this isn't the final result. In addition, it is very important to provide good testing to get the output of reliable, stable and high-quality system.

VI. Implantation

As a sensor you can use separate Pololu Wixel modules or together with OLED SSD1306 128x64. Examples of such sensors are shown in fig. 4 and 5.

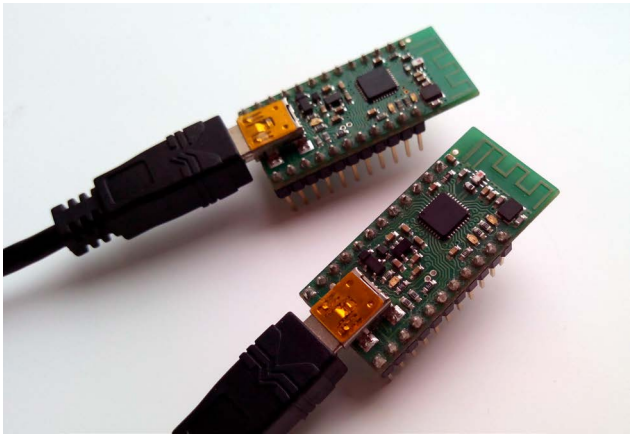

Fig. 4. Pololu Wixel sensors

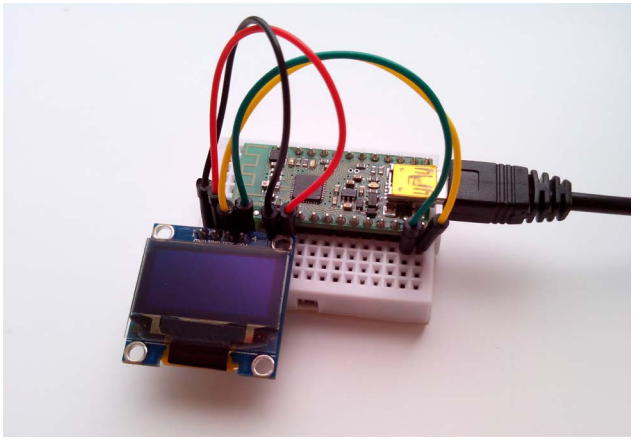

Fig. 5. Pololu Wixel sensor with OLED

As a controller, Raspberry Pi versions older than 3 are used together with the liquid crystal TFT display or others, including electronic ink technology.

VII. Conclusion and Future Work

The monitoring system can be used to provide safe and stable working of the network. Spectrum analyzers can help us to create such type of the system. For our system we choose to use Pololu Wixel modules with specific C firmware on the board.

There were consider several methods of the monitoring system building that are based on programmable modules and Python programming language. Every version of the monitoring system is different by own key features. During the development process is was created and tested many solutions, that formed the basis of the final system version. It was shown that the current realized application scheme is enough viable and can be used in commercial or industrial fields.

The development of monitoring systems will allow to increase the efficiency of frequency resource use, will enable to monitor regular narrowband interference and flexibly redistribute the load, which will increase the availability and integrity of the transmitted data.

References

[1] A. Karakaya and S. Akleylek, "A survey on security threats and authentication approaches in wireless sensor networks," *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, Antalya, 2018, pp. 1-4. doi: 10.1109/ISDFS.2018.8355381

[2] V. M. Astapenya and V. Y. Sokolov, "Experimental evaluation of the shading effect of accelerating lens in azimuth plane," *2017 XI International Conference on Antenna Theory and Techniques (ICATT)*, Kiev, 2017, pp. 388-390. doi: 10.1109/ICATT.2017.7972671.

[3] B. Chatfield and R. J. Haddad. "RSSI-based spoofing detection in smart grid IEEE 802.11 home area networks," *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, Washington, DC, 2017, pp. 1-5. DOI: 10.1109/ISGT.2017.8086064.

[4] V. Sokolov, A. Carlsson and I. Kuzminykh, "Scheme for dynamic channel allocation with interference reduction in wireless sensor network," *2017 4th International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)*, Kharkov, 2017, pp. 564-568. doi: 10.1109/INFOCOMMST.2017.8246463.

[5] V. Buryachok, G. Gulak, and V. Sokolov. "Miniaturization of Wireless Monitoring Systems 2.4–2.5 GHz Band," *Proceedings of the II International Scientific-Technical Conference on Actual Problems of Science and Technology*, Dec. 2015, Kiev, p. 41.

[6] *Pololu Wixel User's Guide*, Pololu Corporation, 2015 p.

[7] Python 3.6.6 Documentation, *Python Software Foundation*, 2018.