Implementing Automatic Handover Solutions for Linux-based Mobile Devices

Nickolay Amelichev
Open Source & Linux Lab (OSLL)
St.-Petersburg Electrotechnical University
St.-Petersburg, Russia
namelichev@acm.org

Abstract

In a heterogeneous network environment, transparent horizontal and vertical handover is a much desired feature. Effective handover solution would allow mobile device users to stay steadily connected, seamlessly switching between different access networks. If it also consistently connected to networks which offer best quality of service, that would dramatically improve user experience.

Expanding upon our work in [1], we present a framework for creating your own custom handover solutions which could run on the clients' Linux-based mobile devices. The framework consists of three core interacting components: *scanners*, which detect available networks; *measurers*, which measure network parameters for every detected network; and *evaluators*, which rank networks according to the values of the parameters.

To provide an example of using the framework, we reimplement parts of our previous signal strength-based handover solution to get improved flexibility, modularity and modifiability.

Index Terms: automatic handover, Linux, mobile devices, 3G, WLAN.

I. Introduction

Modern smartphone users expect to always stay connected. Therefore, seamless handover forms a crucial part of smooth network experience, especially in heterogeneous network environments.

Transparent *horizontal handover* (handover between networks of the same type) is supported in some form by nearly all smartphones. Exceptions include some of the modern Android phones that cannot perform handover between two Wi-Fi APs with the same ESSID [2], [3].

For seamless *vertical handover* (handover between different types of networks), Quallcomm offers a standard-compliant implementation [4] targeted *at service providers only*. It allows providers to switch traffic between 3G and WLAN (WiFi Mobility), or redistribute it between available networks (WiFi Mobility + IP Flow Mobility).

To the best of our knowledge, there're no solutions that run on the client smartphone and support both vertical and horizontal handover. In this paper, we extend our early handover solution prototype [1] to support multiple network types and best access network selection techniques.

II. Architecture Overview

The prototype described in [1] consisted of 3 core components (Fig. 1).

- *Switchers* to perform actual handover from one access network to another, typically implemented as shell scripts. There are different scripts for each kind of mobile device, and each pair of network types (e.g., *N950-wlan-to-3g.sh* for switching from WLAN to 3G network on Nokia N950).

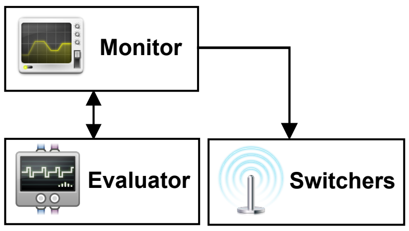

Fig. 1. Three core components of our handover solution

- *Evaluator* to detect neighboring networks, measures network parameters, evaluates each network depending on the measurements and chooses the “best” one.
- *Monitor* to supervise the process, periodically calling the *Evaluator*. If current access network does not match the “best” one chosen by *Evaluator*, *Monitor* calls an appropriate *Switcher*. The simplest implementation is a shell script running as a daemon [1].

We decided to keep this general architecture, and focus on the improvements to the Evaluator. The original Evaluator was simple and compact, but didn’t support adding custom components for network detection, parameter measurement and best network selection. It was also did not output identifier of the chosen network, only its type (e.g. ”wireless” for WLAN), because multiple WLANs were not supported.

A. Interaction between Monitor and Evaluator

Calls from Monitor to Evaluator and internal Evaluator operations are shown in Fig. 2. Monitor calls Evaluator, which proceeds to read previous measurements, scan for new and existing neighboring networks, measure parameters of these networks, and evaluate the networks.

Overall evaluation results, and results for each network type (3G, WLAN, ...) are written to the *aq.txt* file. For each best ranked network, we output its unique identifier (address, e.g. `00:50:18:64:1E:88`), name (human-readable name, e.g., ”AirCel” or ”SJCE_STUDENT”)), and type (3G/WLAN/etc.).

Finally, network parameter measurements are saved, and the application terminates. If Monitor detects that Evaluator’s best network is not currently being used, it launches an appropriate Switcher to correct this situation.

B. New Evaluator’s Features

At the core of the new Evaluator is a new framework that we created, called *FINE* (Framework for detectIon and Evaluation of Networks). The framework allows to easily add new components for detecting networks, measuring network parameters and choosing the best network. The latter can be re-used across all target platforms without any changes. The framework also offers some standard functionality.

- *Network List*. Current list of all available neighboring networks, as well as all the parameters measured, is maintained automatically. You need only to write detection code for specific network types.
- *Measurement Series*. The framework keeps track of the last measured value, as well as the mean, variance and standard deviation. Variance and mean are calculated using highly accurate Welford’s method [6].

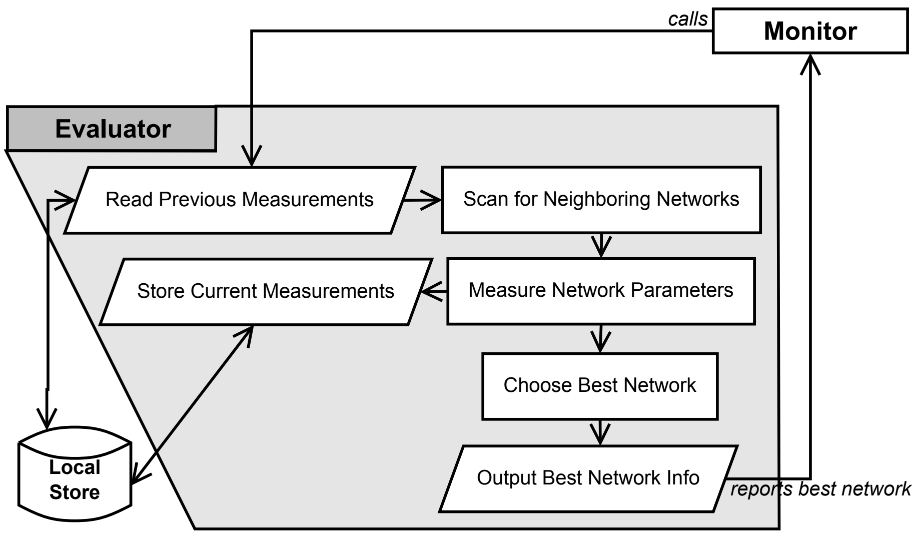
Monitor
calls
Evaluator
Read Previous Measurements
Scan for Neighboring Networks
Store Current Measurements
Measure Network Parameters
Choose Best Network
Output Best Network Info
reports best network
Local Store

Fig. 2. Interaction between Monitor and Evaluator

- *Unit Conversion*. Measurements for different networks can have different measurement units. These are transparently converted to a convenient base unit for the parameter (e.g. signal power in dBm to power in %).
- *Persistence*. Information about neighboring networks (names, identifiers, parameter measurements) is persisted across calls to Evaluator.

III. The FINE Framework

The proposed FINE framework is used to create custom network detection, measurement and ranking components which can be plugged into the new *Evaluator*. Fig. 3 shows core classes in the framework.

Note that *scanner*, *measurer* and *evaluator* interfaces correspond to the main operations of the Evaluator: scanning for neighboring networks, measuring network parameters and evaluating networks. Concrete implementation for these interfaces must be provided by the users of the framework.

A. Network Identification

FINE stores network identification information in the immutable objects of the *network* class. This includes:

- Network type (*type*). E.g. WLAN, 3G, etc.
- Unique network identifier (*id*). MAC address for WLANs, WiMAX, Bluetooth; MCC (Mobile Country Code)+MNC (Mobile Network Code) pair for 3G networks, etc. The identifier must be unique at least for all networks of the specified type.
- Human-readable network name (*name*). This is the name which is suitable for displaying in the UI. Not necessarily unique even for the networks of the same type.

How exactly this information is obtained depends on the scanning components, which are described in section III-C.

B. Parameters and Measurement Units

Each measurable network parameter is represented by an immutable object of *parameter* class. Parameter is defined by its *name* (which must be unique), and *base measurement unit*.

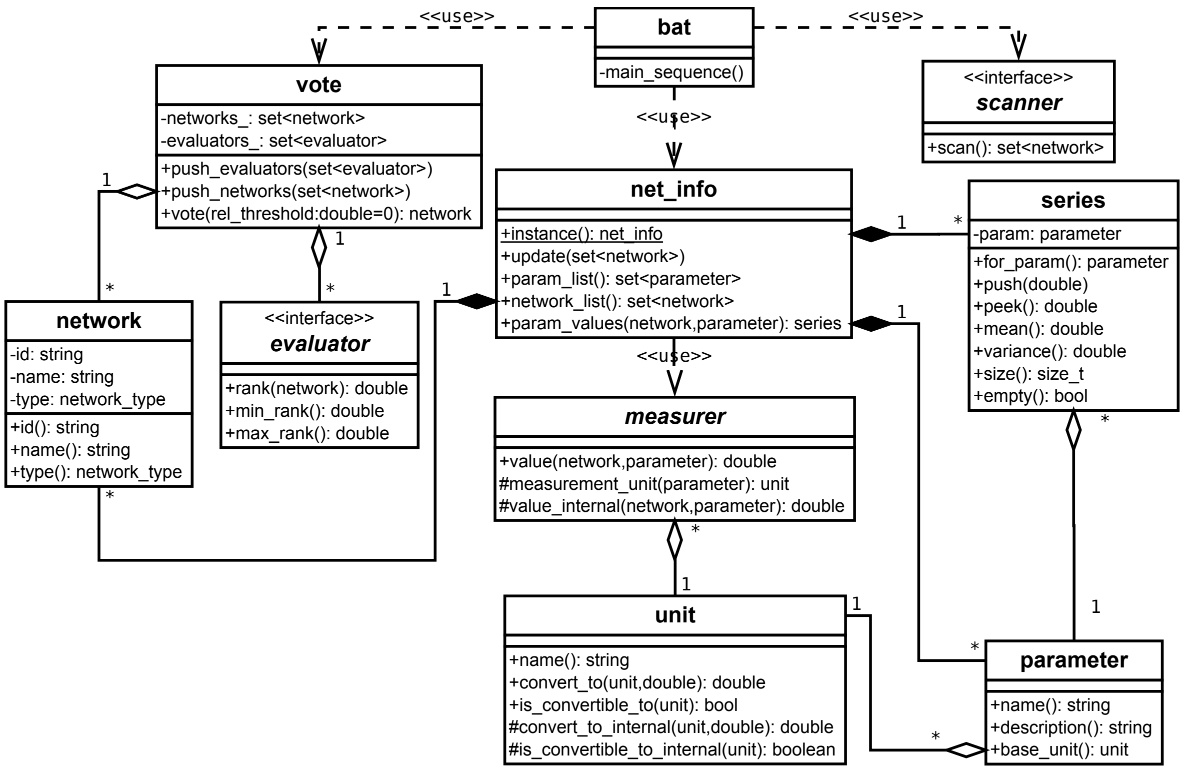

Fig. 3. Core Classes of the FINE Framework

The list of all parameters to be measured on each call to *signalstren* application is kept in the *net_info* (network information) object, which is described in section III-E.

Measurement units are represented by objects of class *unit*. Measurement unit must have a unique *name* and its values can potentially be converted to values of another compatible unit. *unit* class can be used for units not convertible to any other units; otherwise it is needed to subclass *unit* and redefine implementations of *is_convertible_to_internal* and *convert_to_internal* to check for compatibility between units and perform the requested conversion.

C. Scanning and Measurement

Each network type has an single *scanner* which detects networks of this type, and a single *measurer*, which measures parameters for the specified network.

The main class for signalstren application, called *bat*, initiates network scanning, calling the scanners for each network type. It then consolidates scanning results into a single list and requests update of network list and parameter measurements from the *net_info* object (see III-E. This is shown in detail in Fig. 4.

1) Scanner: A *scanner* interface implementation must provide code for method *scan*, which returns a list of unique *network identification* objects (providing unique network id and a human-readable network name, see III-A). How this information is obtained is device-dependent. The simplest approach is, of course, polling (e.g., executing `iw dev wlan0 scan` when the *iw* utility is available). If the target device has NetworkManager support, it can utilise the D-BUS API [7] to get information about detected networks without forcing a new scan, at least for WLAN.

2) Measurer: A concrete measurer must subclass the *measurer* abstract class and override two virtual protected methods:

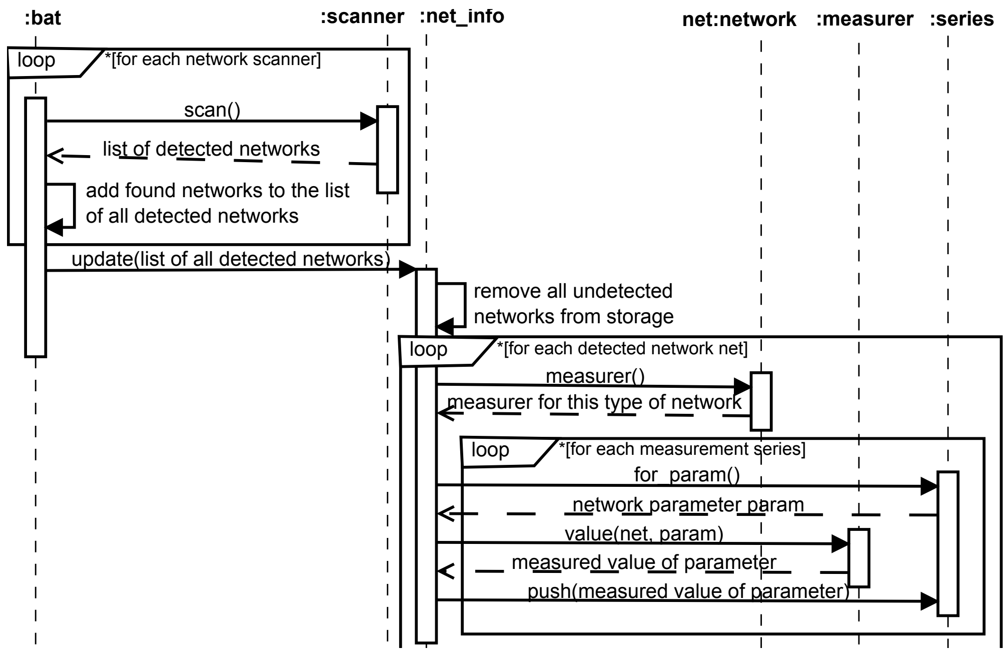
:bat
:scanner
:net_info
net:network
:measurer
:series
loop
*[for each network scanner]
scan()
list of detected networks
add found networks to the list of all detected networks
update(list of all detected networks)
remove all undetected networks from storage
loop
*[for each detected network net]
measurer()
measurer for this type of network
loop
*[for each measurement series]
for_param()
network parameter param
value(net, param)
measured value of parameter
push(measured value of parameter)

Fig. 4. Sequence Diagram for Scanning and Measurement Stages

- *measurement_unit* to return measurement unit for the specified network parameter;
- *value_internal* to return value for the specified network parameter measured in *measurement_unit.*

The *value* method ensures that if parameter is measured in some other unit than the base one (e.g., received signal power in mW vs. in dBm), its values are converted to base measurement unit. This way, all other components can treat measurements of the same parameter for different network types as always having the same scale.

3) Series: Series of parameter values (objects of class *series*) represent measurements of a specified network parameter for the given network over a period of time. FINE framework automatically *push()*es new parameter values acquired from the measurers into the respective series, which can then be accessed through the network information object (III-E).

D. Evaluation and Voting

The signalstren application could not work without defining at least one method for *ranking* detected networks according to their parameters. To define an evaluation criteria, users must implement the *evaluator* interface, providing *rank()*, *max_rank()* and *min_rank()* methods. *max_rank()* and *min_rank()* specify the range $r_{\min} \leq r \leq r_{\max}$ of network ranks r for the evaluator. The *rank()* method is central to network ranking: it calculates the rating of a specified network using information provided by the network information object (III-E).

Signalstren determines the best network by majority vote (Fig. 5). Networks are ranked by each registered evaluator. For every evaluator, the winning network (network with maximum rank) is determined. Network ranks are compared within a relative threshold $\alpha \in (0..1)$: networks are considered equal in rank if their ranks r_1 and r_2 are $|r_1 - r_2| \leq \alpha(r_{\max} - r_{\min})$. The network that won the most times (for most evaluators) is chosen as the overall winner.

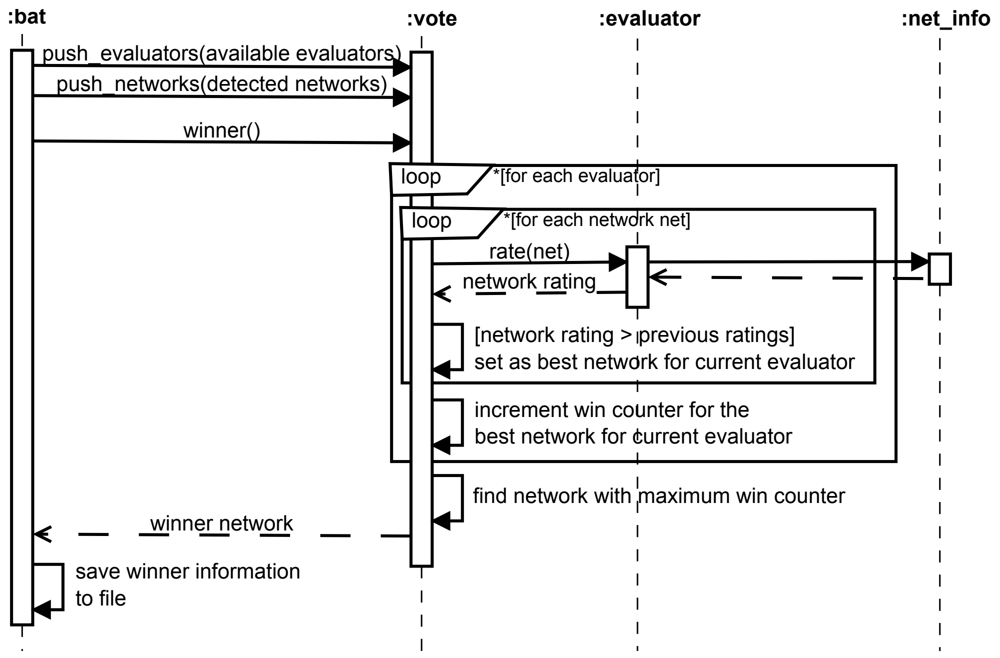
:bat
:vote
:evaluator
:net_info
push_evaluators(available evaluators)
push_networks(detected networks)
winner()
loop
*[for each evaluator]
loop
*[for each network net]
rate(net)
network rating
[network rating > previous ratings]
set as best network for current evaluator
increment win counter for the
best network for current evaluator
find network with maximum win counter
winner network
save winner information
to file

Fig. 5. Sequence Diagram for the Voting Process

E. Network Information Object

Network Information Object (*net_info::instance()*) is central to the FINE framework. This singleton object of class *net_info* holds actual information about currently available networks, network parameters, and measurements of these parameters. Its *update* method responds to ”network scanning complete” events and updates the stored information as shown in Fig. 4.

1) Features: To effectively evaluate networks, Evaluator keeps *series of measurements* for all parameters of currently active networks. If network is no longer detected, its corresponding series is deleted. The measurements for active networks can be accessed through the *param_values* method. It returns series of measurements for a parameter of a given network. The returned series has an already calculated mean, variance and standard deviation values; the last value in series can also be retrieved. This method is primarily used by the network evaluators, which can easily extract series only for parameters they need to inspect, and have no need to do averaging and variance calculations.

All the parameters which are for each network can be retrieved by calling the *param_list* method. Currently active network list can be retrieved by calling the *network_list* method.

2) Persisting Network Information across Calls to signalstren*:* Evaluator is called by Monitor, produces result and exits, so measurements must persist across Evaluator runs, device shutdowns and reboots (due to crashes, rapid discharge of battery etc.) As memory and battery of the mobile device is at premium, the series are saved in a compact way. Only the *previous measured value*, *mean*, *variance*, and *4 auxiliary variables* used for iterative calculation of mean and variance [6] are stored. Standard deviation for the series is calculated as the square root of variance; it is not stored in any way.

We used Google Protocol Buffers [5] to serialize measurement series. The following message format was used to optimize for working with the lightweight version of *libprotobuf-lite.so* protobuf shared library (Fig. 6).

```
option optimize_for = LITE_RUNTIME;

message m_network {                    // Network identification information
  required string id = 1;              // Network id
  required string name = 2;            // Human-readable Network Name
  enum network_type {
    FIRST = 0;
    WLAN = 0;
    UMTS = 1;
    LAST = 2;
  }
  required network_type type = 3;      // Network type
}

message m_series {                     // Series of values of some parameter.
  required m_network network = 1;      // Network this series corresponds to
  required string parameter_name = 2; // Name of the measured parameter
  required int32 size = 3;             // Count of processed measures
  required double last_value = 4;      // Last processed measure
  required double mean = 5;            // Mean value of the parameter
  required double variance = 6;        // Variance of the parameter
  // Next 4 variables are for iterative calculation of mean and variance:
  required double old_m = 7;
  required double old_s = 8;
  required double new_m = 9;
  required double new_s = 10;
}

message m_netinfo {                    // The whole persisted state is here
  repeated m_series all_series = 1;
}
```

Fig. 6. Message Format for Serializing Parameter Measurement Series

F. System Requirements

The Evaluator component built using FINE framework has minimal software requirements (see Table I for details). It needs Google Protocol Buffers Lite Library (*libprotobuf-lite*) to be compiled for the target device architecture, though. Other components might be required to detect wireless networks and measure network parameters, but these will depend on the device and network types supported.

TABLE I
REQUIRED SYSTEM COMPONENTS FOR *signalstren*.

Dependency	Type	Minimal Version
Linux Kernel	OS	2.6.28
sh-compatible shell	Shell	—
Qt	Libraries	4.7+
QtMobility	Libraries	1.2+
Protocol Buffers (*libprotobuf-lite.so*)	Library	2.4.1+

G. Full Handover Solution

Creating a fully functional handover solution would require using the Monitor daemon script and Switcher scripts from [1] alongside with the Evaluator developed using FINE; and modifying the target Linux-based system's `/etc/init.d` to launch Monitor daemon at startup. Monitor and Switchers would need minor changes to accomodate new format of the Evaluator's output file *aq.txt*.

One might also have to implement their own scanners and measurers, as well as custom Switcher scripts. Current version of FINE comes only with a simple implementation of WLAN scanner and measurer, which use `iwlist` utility from the `wireless-tools` package [9]. But it is possible to use something entirely different, if the target system will support it; and add support for more network types, e.g., WiMAX and Bluetooth.

Scanning wireless networks, *getting network parameters*, *changing routing tables and other settings* during handover are privileged operations, so all the solution's components (Monitor, Evaluator and Switchers) would need to run as `root`. This means that this handover solution is restricted to rooted Android devices; as well as Maemo/MeeGo-based Nokia N900, N950 and N9 (which don't need rooting). Other devices might also be supported, as long as they satisfy all the requirements outlined in Table I, allow custom applications to run as `root` and have network APIs to create scanners and measurers for all the network types needed.

IV. Preliminary Testing

To prove usefulness of the FINE framework, we reimplemented much of the original signal strength-based handover solution [1] in terms of *scanners*, *measurers* and *evaluators*. New *signalstren*, along with the FINE framework, is freely available from the *new-arch* branch of our GitHub repository[1].

There were 2 scanners and measurers: 1 for 3G, and 1 for WLAN; and 1 evaluator, which assigned ranks to networks according to their average received signal power.

To test automatic conversion between units in the framework, we used 2 measurement units for signal power: % (for 3G networks) and dBm (for WLANs). Conversion between dBm and % was possible, because typical signal power for 802.11x chips lies between -120 and -20 dBm, and the relationship between power in dBm and power in mW is *almost linear* on this interval [8, p. 4]. Therefore, 100% (maximum power) roughly corresponds to -20 dBm, and 0% (minimum power) corresponds to -120 dBm.

We introduced two significant differences from the original signal strength-based solution:

- networks were ranked according to their *average*, instead of instantaneous, signal power;
- ranks were compared with a relative threshold of 8%, instead of 0. Threshold was chosen arbitrarily, just to test that rank comparison takes threshold value into account.

TABLE II
Sample sequence of *signalstren* calls.

Call #	Detected Network	Network Type	Signal Strength, %
1	**SJCE_STUDENT**	WLAN	87
2	**SJCE_STUDENT**	WLAN	75
	__ROUTER__	WLAN	53
3	SJCE_STUDENT	WLAN	32
	__ROUTER__	WLAN	67
4	SJCE_STUDENT	WLAN	73
	shivu	WLAN	37
	MegaFon	3G	80.5

Preliminary testing of the new Evaluator was done on a regular desktop computer running Ubuntu 10.04 LTS. Testing consisted of regularly calling *signalstren*, supplying simulated network scan information and parameter measurements to it. The information sent is summarized in the Table II (expected winning network is shown in **bold**).

[1]https://github.com/OSLL/sw3g/tree/new-arch

```
=============== before update ================
00:50:18:64:1E:88 __ROUTER__ wlan
  Signal Strength[Power (%)]: size=2,mean=60,variance=98,stdev=9.89949,last_value=67
00:18:E7:8C:B6:D2 SJCE_STUDENT wlan
  Signal Strength[Power (%)]: size=3,mean=64.6667,variance=836.333,stdev=28.9194,last_value=32
=============== after update ================
25520 MegaFon 3g
  Signal Strength[Power (%)]: size=1,mean=80.5,variance=0,stdev=0,last_value=80.5
00:1E:58:B8:AB:A3 shivu wlan
  Signal Strength[Power (%)]: size=1,mean=37,variance=0,stdev=0,last_value=37
00:18:E7:8C:B6:D2 SJCE_STUDENT wlan
  Signal Strength[Power (%)]): size=4,mean=66.75,variance=574.917,stdev=23.9774,last_value=73
================= ranking ===================
[signal_strength_evaluator] MegaFon rank=80.5
[signal_strength_evaluator] shivu rank=37
[signal_strength_evaluator] SJCE_STUDENT rank=66.75
[signal_strength_evaluator] found best net, it is: MegaFon
================== voting ===================
determining overall...
found best count - 1 - for MegaFon
found overall winner, it is: MegaFon

The winner is: 25002 - MegaFon (type 3g)
```

Fig. 7. *signalstren* log for the sample

```
best=3g 25520 MegaFon
best[3g]=25520 MegaFon
best[wlan]=00:18:E7:8C:B6:D2 SJCE_STUDENT
```

Fig. 8. *signalstren* output for the sample

New *signalstren* application performed well, choosing the expected winner in each of the test cases. Signalstren log and output on the last step are shown in Fig. 7 and Fig. 8, respectively.

V. CONCLUSION

We created the FINE framework for network detection, parameter measurement and ranking. We have successfully redone the monolithic Evaluator application using FINE, easily adding averaging and comparison of network ranks using a certain relative threshold. According to the preliminary test, results obtained with our new *signalstren* application based off the FINE framework are consistent with the results we got from the old application. The new application is much easier to maintain and improve, though, largely due to the separation of concerns, e.g., the evaluator need not know how the parameters are measured to process their values.

Extracting existing monolithic Evaluator functionality into three kinds of interacting components: scanners, measurers and evaluators, lays the groundwork for implementation of more sophisticated network ranking schemes. These include *using multiple parameters for ranking networks* (e.g., SNR, collision rate, average throughput, packet loss and network latency, jitter); *ranking depending on the environment* (remaining battery charge, device velocity, etc.) and so on. Exploring these possibilities for intelligent and automatic selection of best access network will be the topic of our future works.

References

[1] N. Amelichev, K. Krinkin, S.P. Shiva Prakash, TN Nagabhushan, ”Signal Strength-Based Approach for 3G/WLAN Handover on Nokia N900 Devices”, *Proceedings of 10th Conference of Open Innovations Association FRUCT*, Tampere, Finland, 7-11 November 2011, pp. 10–15.

[2] ”WiFi AP Handover/Switching”. - http://androidforums.com/captivate-all-things-root/270139-wifi-ap-handover-switching.html, 2011.

[3] ”WiFi Handover Between Access Points not Happening Automatically”. - http://androidforums.com/android-lounge/220090-wifi-handover-between-access-points-not-happening-automatically.html, 2010.

[4] ”3G/WiFi Seamless Offload”. - http://www.qualcomm.com/media/documents/files/3g-wifi-seamless-offload.pdf, 2010.

[5] ”Developer Guide - Protocol Buffers”. - http://code.google.com/intl/ru-RU/apis/protocolbuffers/docs/overview.html, 2011.

[6] B.P. Welford, ”Note on a method for calculating corrected sums of squares and products”. Technometrics 4(3), pp. 419–420, 1962.

[7] ”NetworkManager D-BUS Interface Specification”. - http://projects.gnome.org/NetworkManager/developers/spec.html, 2008.

[8] ”Converting Signal Strength Percentage to dBm Values”. - http://www.wildpackets.com/elements/whitepapers/Converting_Signal_Strength.pdf, 2002.

[9] ”Wireless Tools for Linux”. - http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html, 2008.