CS 211 project report: Remote access to OpenWRT

Zhehao Wang
Department of Computer Science, UCLA
zhehao@cs.ucla.edu

Haitao Zhang
Department of Computer Science, UCLA
haitao@cs.ucla.edu

Jeffrey Chen
Department of Computer Science, UCLA
jc4556@g.ucla.edu

ABSTRACT

content...

1. GUIDELINE

abstract; introduction; background; design; implementation; evaluation/demo; discussions; related work; conclusion and future work.

2. INTRODUCTION

OpenWRT is a free, open-source, Linux-kernel based operating system (OS) for network-routing embedded systems. This OS is notable for being able to run on various types of devices and for simplifying cross-platform building of OpenWRT software packages, including fixes for devices no longer supported by the devices' manufacturers. OpenWRT receives regular updates, and allows basic router configuration, and installation of features through a package repository.

OpenWRT can be configured using a SSH command-line interface or a pre-packaged LuCI web interface. The SSH command-line interface is more suitable to professional and developer users, while a web interface is more friendly to common users, providing access to basic OpenWRT functions. However, SSH is tedious on a mobile device, because the user must perform all configurations via text modifications, and data and results cannot be interpreted in visual graphics. On the other hand, OpenWRT's existing remote access web interface is made for desktop web browsers, and not smart devices. On mobile smart devices, the in-browser interface is not scaled to the dimensions of the device's screen nor is it touch-friendly, hampering user comprehension and control of the interface. Additionally, depending on the device, loss of connection or suspension of the browser to another application can force renewal of the session or prevent retrieval of network information, causing issues for presenting real-time data and visualizations.

A native smart device application is more distribution-friendly for smart devices and can be designed on smart devices to be more user-friendly and streamlined to OpenWRT uses. Therefore, this project seeks to create a lightweight, easy-to-use generic Android application available for products running OpenWRT.

3. BACKGROUND

3.1 What is OpenWRT

OpenWRT [?, ?] is a Busybox/Linux based embedded platform which is developed following GPL license. It minimizes its own functions so that it fits for lots of memory constrained devices. Specifically, it builds the appropriate toolchain for devices, compiles appropriate kernel with patched and options, and provides software as IPKG packages.

3.2 OpenWRT System Structure

OpenWRT System Structure covers four aspects: directory structure, packages and external repositories, toolchain, and software architecture.

There are four key directories in the base: tools, toolchain, package and target. Tools and toolchain refer to common tools which will be used to build the firmware image, the compiler, and the C library.

In OpenWRT, almost all the packages are .ipk files. Users can choose what packages to install and what packages to uninstall based on their specific needs. Packages are either part of the main trunk or maintained out of the main trunk. For the second case, packages can be maintained by the package feeds system.

To compile the program for a particular architecture, the OpenWRT system will automatically create the toolchain during the cross-compilation process, simplifying the development tasks. However, if the toolchain needs to be created manually, OpenWRT also provides an easy way to configure the arguments.

Figure 1 shows the software stack of OpenWRT. We can see that the common embedded Linux tools such as uClibc, busybox, shell interpreter are used by OpenWRT.

3.3 How to Develop With OpenWRT

It is easy to port software to OpenWRT. Various fetching methods such as GIT, Subversion, CVS, HTTP, local source can be used to download package source. In a typical package directory, there should always be a

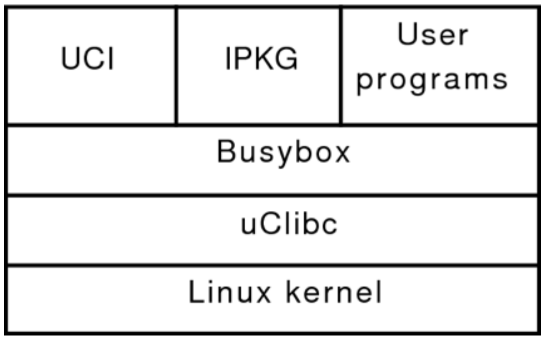

Figure 1: the software stack of OpenWRT

`package/<name>/Makefile`. After running "make menuconfig", the new package will show up in the menu; and after running "make", the new package will be built.

3.4 Web-based Access to OpenWRT

In the OpenWRT system, some important features are provided: a built-in web server with CGI support, an SSH server and a package management tool. We can make use of these existing tools to build our Android application.

Apart from the basic tools, OpenWRT also supports LuCI [?], which is a browser-based tool to remotely configure the OpenWRT system. Specifically, it provides status visualization functions, system administration functions and network configurations. We would provide similar functions in our Android application.

Another useful reference is the Netgear [?] mobile application, which designs nice UIs that we can borrow ideas from. We need also to design the web-based access application UIs based on their work.

4. DESIGN AND IMPLEMENTATION ROADMAP

Based on the functional modules of LuCI, we will design the functional components of the Android application, which are organized into three major categories:

- **Network configuration** covers common functionalities in configuration tools that often come with commercial APs. These configurations include, but may not be limited to, managing network interfaces, DHCP and DNS settings, static routes, and firewall.
- **System configuration** provides an interface to customize the OpenWRT box. Common adminstration functions include: system and user configuration (setting device adminstrator password, creating system backup image and restoring system from backup image, generating user SSH keys, etc), software management (installing and configuring software packages) and task management (managing scheduled task and startup task). If the time allows, an in-application command line tool can be implemented for advanced users to execute console commands from the application to further customize the OpenWRT box.
- **Status/Statistics visualization** offers a mobile-phone friendly view of the system status (firmware and kernel version, uptime, current time; CPU and memory usage, currently running processes, system and kernel log) and network-related status (interface, route, firewall status, etc). The visualization component can provide real-time graphs of system load and traffic statistics, such as historical system memory usage, network traffic per interface and traffic per transport layer connection.

The design and implementation effort will be organized by the three function categories, with approximately one and a half weeks dedicated to each.

4.1 Graphical User Interface

The goal of the graphical user interface design was to simplify the user interface on a smartphone. To that end, the limitations of the LuCI web interface were studied to provide design guidelines. The first issue analyzed was that LuCI had an issue in its navigation on smartphones. To navigate through the application categories, the user needed to select a category in the navigation menu, then select a subcategory from the dropdown menu. Additionally, changing subcategories within the same category still required selecting the overarching category again. Therefore, a design goal of the application would be to maintain the current category and simply swap subcategories.

Another LuCI WebView issue was that unsaved changes would be tracked in a session until committed. Tracking unsaved changes on a web browser can result in session complications, depending on browser settings for caching. Furthermore, the WebView relied on in-browser scripting to provide functional elements, which is a dependency that can be optimized. Therefore another aspect of our design was to make all actions atomic and contained to the screen they are accessed on, to avoid carrying changes. To make all actions atomic, all functional elements in the original WebView would be rebuilt natively in Android.

Based on the LuCI framework, the designed Android application's user interface screens consisted of a login screen, then three major categories: status, network, and system. Each category then presented a subnavigation menu that persisted until another major category was selected, allowing users to move more freely within same category.

Each separate screen in the subcategories of the major categories was designed to maintain discrete actions and information. Rather than having multiple configuration forms and submission buttons in the same screen, the screen would be limited to at most one form each, with other forms being accessible on a new screen that is linked to by a list on the current screen.

4.2 Application specific traffic statistics

One of our goals in this project is to identify and provide statistics for traffic belonging to specific user applications, for example, identifying how much incoming and outgoing traffic's related with the Youtube application, and knowing which host behind this OpenWRT box is generating these traffic. To achieve this goal, we considered the following design alternatives:

- Install a service on each user device, which monitors application network usage information from the operating system and reports the monitored statistics to a service on the OpenWRT box. This approach can likely generate more accurate results and provide each specific application's statistics, however, it requires that each user device voluntarily run this service, which is not ideal for our goal: this service should not require additional actions from user devices, and the OpenWRT box should be able to collect statistics with or without end user cooperation.

- Capture the traffic on the OpenWRT box, and run per-packet analysis. The following information from packet header or payload could be helpful:

  - Source and destination IP addresses.

    Popular service providers, such as Google and Facebook, own large chunks of IP addresses, and certain destination IP address ranges may correspond to servers for a specific application. By collecting such information and building a mapping from destination IP address to an application's backend services, we can infer the user application that's generating this traffic. The problem with this approach is that this mapping takes time to build, and it's hard for this mapping to change dynamically, especially considering the fact that many popular services are using CDNs.

  - Source and destination port number.

    Certain services can be identified by IANA's port number allocation [?], and this information is easily retrievable from the Internet. The problem with this approach, is that the amount of applications that can be identified purely by port number is very limited, for example, the Youtube app and Facebook app on Android both go through port 443 (TLS).

  - Application layer payload.

    Decoding application payload and trying to find characteristic plain text is another way to identify which application the packet belongs to. This approach is promising, if such characteristic texts can be found and the payload's not encrypted, which is not the case for most popular applications, such as Youtube or Facebook. Packet capture of these apps on Android shows that their traffic goes on top of TLS, and packet capture of watching Youtube videos on a desktop indicate that the traffic goes on top of QUIC, which has encryption over UDP.

  - Source and destination host names.

    DNS host names often give information about the service provider, even when the service provider's using CDNs. By doing a DNS reverse lookup on the IP addresses, we can find out the service's domain name, thus infer what application's generating the traffic. The assumption of this approach is that most servers or CDN boxes do have a DNS name that corresponds to the application backend that they run. Our initial experiments suggest that this assumption is indeed true for popular applications like Youtube and Facebook, though mapping a DNS domain to a certain service may not be straightforward. For example, Facebook app on Android talks to both **.facebook.com* domain, and **.fbcdn.net* (owned by the CDN provider, Akamai Technologies) domain, and Youtube app talks to both **.google.com*, and **.1e100.net*, another Google owned domain.

    Given this caveat, the authors chose this approach since it's still more applicable than other options listed above. For a proof-of-concept implementation, the mapping from domain names to service names is statically configured. It's also worth mentioning that this approach will limit our statistics to per service provider, rather than per exact application (for example, we can't differentiate Google Hangout application traffic from Youtube application traffic), but the authors believe this is enough for the purpose of providing statistics on the router end that could give the adminstrator rough ideas of which application on which device is generating how much traffic.

5. IMPLEMENTATION

5.1 Graphical User Interface

The development of the user interface considered native Android solutions to the navigation. Typically, Android activities are single interface screens. Inside the Android activities, Android fragments can be used use to provide different tab screens. Therefore, tab navigation was implemented with Android activities for the categories, and Android fragments for the subcategories and forms.

However, the implementation of the user interface reached complications. The major problem leading to complications was that in using the LuCI backend, content extraction using HTML retrieved whole WebView pages. JSON would extract smaller elements, but required specific queries, and would make code reuse difficult. Using both required parsing, and the extent of parsing needed for each category and subcategory complicated the implementation, taking more development time than available. The issue of parsing elements also complicated the implementation of buttons and other interactive objects, which needed to be re-made natively. Therefore, the graphical user interface was not completed in time.

5.2 Application specific traffic statistics implementation

Based on the design described in section 4.2, the proof of concept implementation pipes the output of *tcpdump* to a custom backend, which reads the source and destination IP addresses, tries to use *nslookup* to find the domain names by DNS reverse lookup, and match the domain names with statically configured characteristic strings; if a match is found, the number of bytes is added for corresponding service provider name and corresponding host, otherwise, the traffic is considered identified. The backend writes statistics in JSON format to a file, and the file is served by a simple web server. In the statistics, incoming and outgoing traffic, as well as different IP addresses within the local subnet are differentiated. An example of the JSON object is given below, in which the numbers are number of bytes per IP address and application. We use Python for the simple backend and web server, for easy prototyping purposes.

```
{"outgoing":
  {"google": {"10.0.5.15": 812},
   "facebook": {"10.0.5.15": 622}},
 "incoming":
  {"google": {"10.0.5.15": 148865},
  "facebook": {"10.0.5.15": 76947}}
}
```

Basic optimizations for the backend include introducing the following

- A cache for IP and domain mapping, so that instead of calling *nslookup* each time a packet arrives, *nslookup* is tried only once for each IP address within a given time.
- A minimum interval of one minute between writing statistics back, so that the number of file writes is limited.

6. EVALUATION

6.1 Application specific traffic statistics backend evaluation

This section describes the disk, CPU and memory usage of the Python backend for application specific traffic analysis.

Hard drive volume's usually quite limited for an OpenWRT box. The backend analyzer has dependency on opkg packages *tcpdump*, *python-light* and *python-codecs*; and the associated http server has dependency on *python-logging* and *python-ssl* (optional, the module would not give out warnings if installed). On our test TP-link WDR4300 router flashed with OpenWRT 15.05, 500KB disk space still remains for mount point "/", after all the necessary dependencies are installed. The mount point "/tmp" has over 60MB temporary space, and we store our tcpdump file, and statistics output file under mount point "/tmp".

CPU and memory usage on the OpenWRT boxes's measured with users behind the AP watching a Youtube video. When only one user's behind the AP watching a Youtube video at 360p, our Python backend uses 8% 22% CPU, tcpdump uses 1% 10% depending on whether the video is pre-buffered long enough. Both tcpdump and Python backend use 5% of the virtual memory. With two users behind the AP, the Python backend CPU usage grows by 7% on average, and we did not see tcpdump CPU usage ramp up. Since the communication between tcpdump and the Python backend is through file system interface, IO is more likely the bottleneck here than CPU.

7. DISCUSSIONS

8. FUTURE WORK

9. CONCLUSION

10. TIMELINE

A rough timeline for the project is given in table 1.

Table 1: Project timeline

Week No.	Task
5, 6 (first half)	Implement status/statistics visualization module
6 (second half), 7	Implement network configuration module
8, 9 (first half)	Implement system administration module
9 (second half), 10	Prepare final report and presentation