دانشگاه صنعتی شریف

دانشکدهٔ مهندسی کامپیوتر

پروژه ۷ آزمایشگاه اینترنت اشیاء

نویسندگان:

احسان نادری-۴۰۰۱۰۹۷۰۲

سید علی طیب-۴۰۰۱۰۱۵۲۶

امیرحسن جعفرآبادی-۴۰۰۱۰۴۸۷۸

بهار ۱۴۰۴

۱ توضیح کلی پروژه

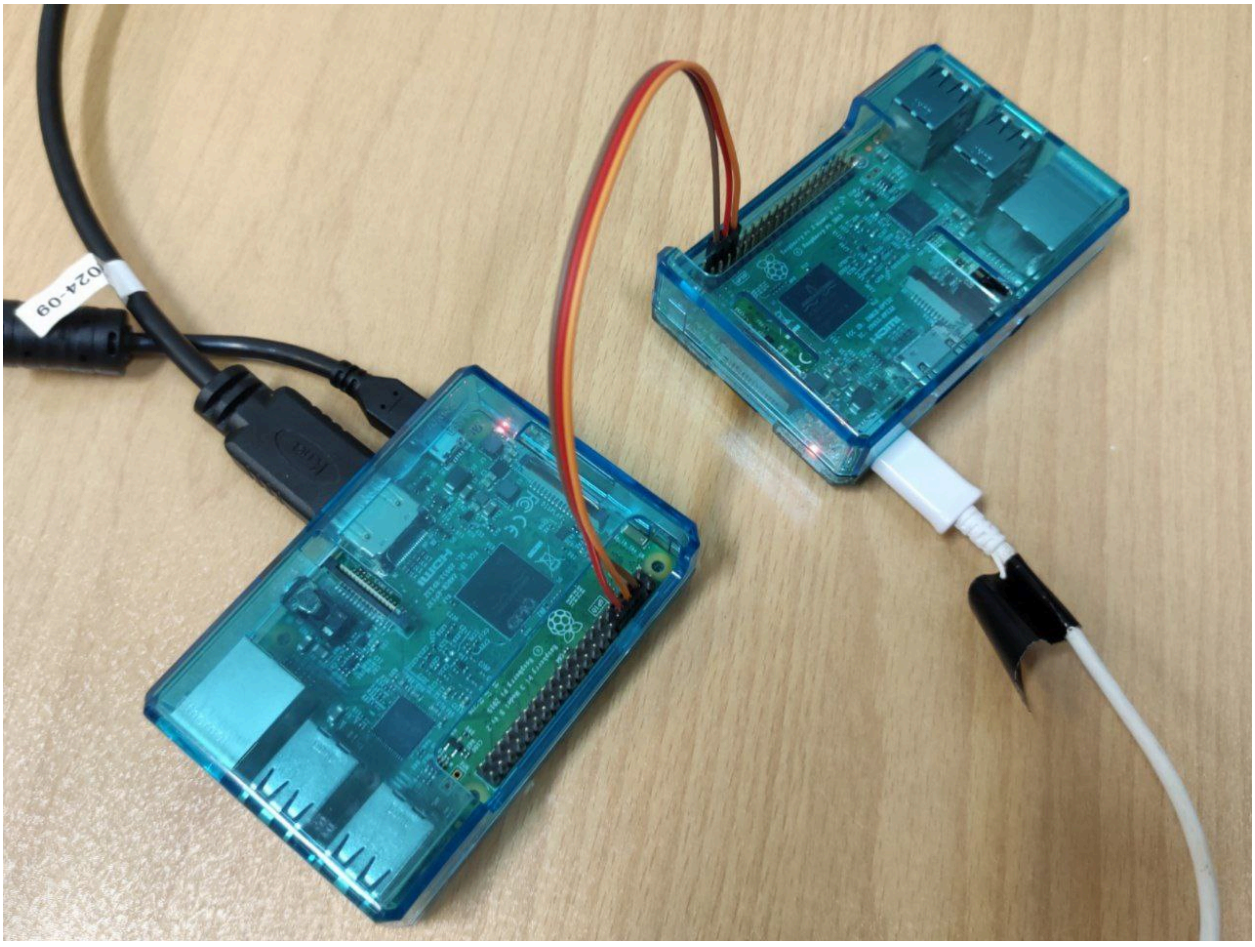
در این پروژه از دو عدد Raspberry Pi 3 استفاده شد که هر دو روی سیستم عامل Yocto را اجرا می‌کنند. هدف اصلی پروژه، ایجاد یک سامانه نظارت بر وضعیت سخت‌افزاری و نرم‌افزاری دستگاه اصلی بود. در ادامه ساختار و جزئیات این پروژه تشریح می‌شود.

معماری کلی سامانه

سامانه شامل دو برد Raspberry Pi است:

۱. برد اول (سیستم اصلی): اجرای وظایف اصلی و تولید لاگ‌های کرنل (dmesg) و ارسال آن‌ها به برد دوم.

۲. برد دوم (سیستم نظارت): دریافت لاگ‌های ارسالی از برد اول از طریق UART، تحلیل آن‌ها و گزارش وضعیت سیستم.



شکل ۱: دو رزبری‌پای استفاده‌شده در پروژه و اتصال آن‌ها

جزئیات عملکرد برد اول

برد اول به‌عنوان سامانه اصلی، دارای یک ماژول کرنل به نام **Heartbeat** است. وظیفه این ماژول ارسال سیگنال‌های ضربان قلب به صورت دوره‌ای به برد دوم است. این سیگنال‌ها نشان‌دهنده زنده بودن و صحت عملکرد سیستم اصلی می‌باشند. همچنین تمامی

پیام‌های خروجی کرنل (dmesg) از طریق درگاه UART به برد دوم منتقل می‌شوند تا امکان پایش وضعیت در زمان واقعی فراهم شود.

جزئیات عملکرد برد دوم

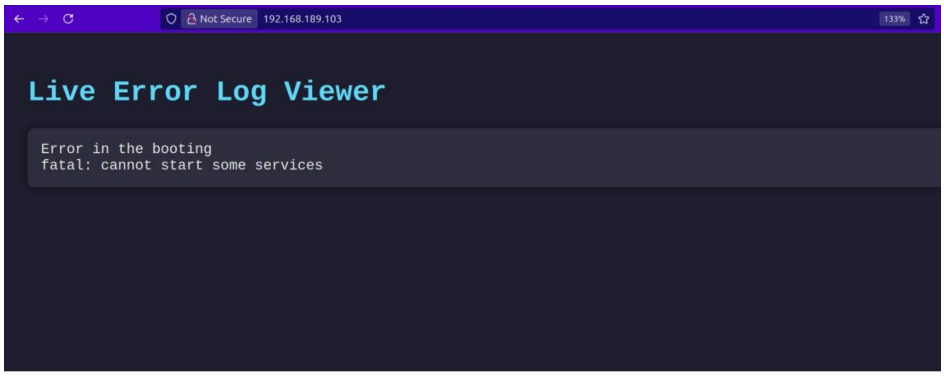
برد دوم نقش سامانه پایش و نظارت را ایفا می‌کند. این برد تمامی پیام‌های UART دریافتی از برد اول را توسط یک برنامه نوشته شده به زبان Go تحلیل می‌کند. وظایف این بخش عبارتند از:

۱. تحلیل لاگ‌ها: بررسی پیام‌های dmesg جهت شناسایی خطاها، شکست‌ها و یا ریبوت‌های ناگهانی سیستم.

۲. پایش ضربان قلب: بررسی سیگنال‌های heartbeat به‌منظور اطمینان از کارکرد درست سیستم اصلی.

۳. گزارش‌دهی: نمایش وضعیت سیستم اصلی از طریق یک وب‌سرور داخلی. این وب‌سرور امکان مشاهده لاگ‌ها و وضعیت فعلی را در اختیار کاربر قرار می‌دهد.

۴. ثبت رویدادها: نوشتن تمامی لاگ‌ها و خطاهای شناسایی‌شده در فایل‌های محلی جهت نگهداری و تحلیل‌های آتی.



شکل ۲: وب سرور برد دوم

۲ مراحل انجام پروژه

۱.۲ ساخت ایمیج **Yocto**

روند انجام:

```
mkdir yocto
cd yocto
mkdir sources
git clone git://git.yoctoproject.org/poky -b scarthgap
git clone git://git.yoctoproject.org/meta -raspberrypi -b scarthgap
git clone https:// git.openembedded.org/meta -openembedded -b scarthgap
source sources/poky/oe -init -build -env
```

conf/bblayers.conf:

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/data/uni/8/IOT/yocto/sources/poky/meta \
/data/uni/8/IOT/yocto/sources/poky/meta-poky \
/data/uni/8/IOT/yocto/sources/poky/meta-yocto-bsp \
${TOPDIR}/../sources/meta-raspberrypi \
${TOPDIR}/../sources/meta-openembedded/meta-oe \
${TOPDIR}/../sources/meta-openembedded/meta-multimedia \
${TOPDIR}/../sources/meta-openembedded/meta-networking \
${TOPDIR}/../sources/meta-openembedded/meta-python \
/data/uni/8/IOT/yocto/sources/meta-custom \
"
```

conf/local.conf:

```
MACHINE="raspberrypi3b -64"
LICENSE_FLAGS_ACCEPTED="synaptics-killswitch"
ENABLE_UART="1"
```

در نهایت با دستور bitbake core-image-base ایمیج سیستم‌عامل ساخته می‌شود و آن را روی کارت حافظه فرار داده و رزبری‌پای را روشن می‌کنیم.

```
[   14.732901] smsc95xx 1-1.1:1.0 eth0: Link is Down
udhcpc: started, v1.36.1
udhcpc: broadcasting discover
udhcpc: broadcasting discover
udhcpc: broadcasting discover
udhcpc: no lease, forking to background
ip: SIOCGIFFLAGS: No such device
Starting system message bus: dbus.
Starting rpcbind daemon...[   24.105084] NET: Registered PF_INET6 protocol family
[   24.111577] Segment Routing with IPv6
[   24.115337] In-situ OAM (IOAM) with IPv6
done.
Starting bluetooth: bluetoothd.
Starting syslogd/klogd: done
 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon
[   24.334152] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[   24.339621] Bluetooth: BNEP filters: protocol multicast
[   24.344987] Bluetooth: BNEP socket layer initialized
[   24.354876] Bluetooth: MGMT ver 1.22
   ...done.
Starting Telepho[   24.372466] NET: Registered PF_ALG protocol family
ny daemon
Starting Linux NFC daemon
[   24.561510] nfc: nfc_init: NFC Core ver 0.1
[   24.565971] NET: Registered PF_NFC protocol family
[   24.588624] Bluetooth: RFCOMM TTY layer initialized
[   24.593695] Bluetooth: RFCOMM socket layer initialized
[   24.599045] Bluetooth: RFCOMM ver 1.11

Poky (Yocto Project Reference Distro) 5.0.8 raspberrypi3-64 /dev/ttyS0

raspberrypi3-64 login:
```

شکل ۳: لاگ‌های زمان بوت

۲.۲ راه‌اندازی وای‌فای و SSH

برای اتصال با استفاده از SSH و کنترل سامانه‌ی نگهبان باید این قابلیت را روی برد فعال می‌کردیم که این‌کار با تغییرات زیر در سیستم‌عامل انجام گرفت:

با تغییر این فایل و سپس دستور ifup wlan0 شبکه‌ی وایفای فعال می‌شود.

vi /etc/wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
update_config=1
network={
    ssid="<SSID_NAME>"
    psk="<PASSWORD>"
    proto=RSN
    key_mgmt=WPA-PSK
    pairwise=CCMP
    auth_alg=OPEN
}
```

برای فعال‌سازی SSH تنظیمات زیر را به انتهای فایل local.conf اضافه‌می‌کنیم.

```
EXTRA_IMAGE_FEATURES = "ssh-server-dropbear allow-empty-password allow-root-login empty-root-
    password"
```

با این تغییر سرور DropBear SSH بالا می‌اید و می‌توانیم از طریق شبکه متصل شویم.

۳.۲ نوشتن کدهای Go و MakeFile

در این مرحله کدهای سمت نگهبان را می‌نویسیم. این کدها از یک برنامه با زبان Go تشکیل شده که وظیفه‌ی خواندن لاگ‌ها و تشخیص مشکلات و همچنین ارسال آن‌ها روی سرور وب را برعهده دارد. همچنین برای پروژه یک Makefile نوشته شده که وظیفه‌ی پیدا کردن آدرس رزبری‌پای نگهبان، اتصال به آن و دیپلوی کدها را به عهده دارد. توضیحات این کدها در قسمت بعدی نوشته می‌شود.

۴.۲ نوشتن ماژول heartbeat

سپس برای رزبری‌پای اصلی یک Kernel module نوشته شد که درون کرنل سیستم‌عامل یوکتو قرار می‌گیرد و وظیفه‌ی ارسال یک heartbeat در فاصله‌ی زمانی ۵ ثانیه را برعهده دارد. این کد هم در قسمت بعد توضیح داده خواهد شد.

۳ چالش‌های پروژه

۱. ما نتوانستیم ایمیج‌های ساخته‌شده توسط یوکتو را با استفاده از qemu تست کنیم و مجبور به استفاده از سخت‌افزار اصلی بودیم.

۲. برد رزبری‌پای۳ فقط یک رابط UART در دسترس دارد و به همین دلیل در سامانه‌ی نگهبان مجبور به استفاده از SSH بودیم.

۴ توضیح کدهای پروژه

کد اصلی سیستم نگهبان با Go

```
package main

import (
	"bufio"
	"flag"
	"fmt"
	"github.com/tarm/serial"
	"log"
	"os"
	"strings"
	"sync"
	"time"
)

var ErrorWords = []string{"error", "panic", "fatal", "fail"}
var HEARTBEAT_STRING = "heartbeat: alive"

func containsError(s string) bool {
	for _, word := range ErrorWords {
		if strings.Contains(strings.ToLower(s), word) {
			return true
		}
	}
	return false
}

func main() {
	// Command-line flags for serial configuration
	portName := flag.String("port", "/dev/serial0", "Serial port device")
	baudRate := flag.Int("baud", 115200, "Baud rate for serial communication")
	logFilePath := flag.String("logfile", "errors.log", "Path to error log file")
	flag.Parse()

	// Open the serial port
	cfg := &serial.Config{Name: *portName, Baud: *baudRate}
	port, err := serial.OpenPort(cfg)
	if err != nil {
		log.Fatalf(">>> Failed to open serial port %s: %v", *portName, err)
	}
	defer port.Close()

```

```
	fmt.Printf(">>> Listening on %s at %d baud... (press Ctrl+C to exit)\n", *portName, *baudRate)

	// Open log file
	logFile, err := os.OpenFile(*logFilePath, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
	if err != nil {
		log.Fatalf(">>> Failed to open log file: %v", err)
	}
	defer logFile.Close()

	go httpServer()
	fmt.Println(">>> HTTP server started...")

	// Read lines from UART
	scanner := bufio.NewScanner(port)
	fmt.Println(">>> Scanner started...")

	var mu sync.Mutex
	lastHeartbeat := time.Now()

	// Heartbeat monitor goroutine
	go func() {
		for {
			time.Sleep(1 * time.Second)
			mu.Lock()
			if time.Since(lastHeartbeat) > 5*time.Second {
				msg := fmt.Sprintf(">>> Heartbeat timeout at %s\n", time.Now().Format(time.RFC3339))
				fmt.Print(msg)
				_, err := logFile.WriteString(msg)
				if err != nil {
					fmt.Printf(">>> Failed to write heartbeat timeout: %v\n", err)
				}
				// Reset to avoid spamming every second
				lastHeartbeat = time.Now()
			}
			mu.Unlock()
		}
	}()

	for scanner.Scan() {
		line := scanner.Text()
		fmt.Println(line)
		if containsError(line) {
```

```
                        fmt.Println(">>> Error detected. try to appending it to the file...")
                        _, err := logFile.WriteString(line + "\n")
                        if err != nil {
                                fmt.Printf(">>> Failed to write to log file: %v\n", err)
                        }
                }
                if strings.Contains(line, "Booting Linux on physical CPU 0x0000000000") {
                        fmt.Println(">>> Reboot detected. try to appending it to the file...")
                        _, err := logFile.WriteString(line + "\n")
                        if err != nil {
                                fmt.Printf(">>> Failed to write to log file: %v\n", err)
                        }
                }
                if strings.Contains(line, HEARTBEAT_STRING) {
                        mu.Lock()
                        lastHeartbeat = time.Now()
                        mu.Unlock()
                }
        }

        // Check for scanning errors
        if err := scanner.Err(); err != nil {
                log.Fatalf("Error reading from serial port: %v", err)
        }
}
```

این کد به زبان Go پیاده‌سازی شده و وظیفه‌ی آن خواندن لاگ‌های سیستم از طریق پورت سریال UART و تحلیل آن‌هاست. ابتدا پورت سریال با پارامترهایی نظیر نام پورت و نرخ انتقال داده (baud rate) پیکربندی و باز می‌شود. سپس یک فایل لاگ برای ذخیره‌ی خطاها و رویدادهای مهم باز می‌گردد. برنامه در یک گوروتین جداگانه یک وب‌سرور محلی اجرا می‌کند تا امکان مشاهده و گزارش‌دهی وضعیت به صورت برخط وجود داشته باشد. بخش اصلی برنامه با استفاده از Scanner خطوط دریافتی از پورت سریال را می‌خواند و پیام‌ها را به‌صورت زنده در ترمینال چاپ می‌کند.

منطق اصلی تحلیل لاگ شامل شناسایی خطاها، ریبوت سیستم و ضربان قلب (heartbeat) است. برای خطاها، برنامه کلمات کلیدی نظیر error، panic، fatal و fail را جستجو کرده و در صورت یافتن، آن‌ها را در فایل لاگ ذخیره می‌کند. در صورتی که پیام ریبوت شناسایی شود، آن نیز به همان فایل اضافه می‌شود. همچنین یک گوروتین جداگانه وضعیت ضربان قلب را بررسی می‌کند؛ اگر بیش از پنج ثانیه سیگنال heartbeat دریافت نشود، پیغام «عدم دریافت ضربان قلب» تولید و در فایل ثبت می‌گردد. این ساختار باعث می‌شود سیستم به‌طور بلادرنگ مشکلات احتمالی را شناسایی و مستند کند.

کد مربوط به وب‌سرور:

```
package main

import (
        "bufio"
        "fmt"
        "net/http"
```

```
	"os"
	"path/filepath"
	"time"
)

func httpServer() {
	http.HandleFunc("/events", sseHandler)

	fs := http.FileServer(http.Dir("static"))
	http.Handle("/", fs)
	http.ListenAndServe(":80", nil)
}

func sseHandler(w http.ResponseWriter, r *http.Request) {
	logFile := filepath.Join(os.Getenv("HOME"), "errors.log")
	file, err := os.Open(logFile)
	if err != nil {
		http.Error(w, "Could not open log file", 500)
		return
	}
	defer file.Close()

	w.Header().Set("Content-Type", "text/event-stream")
	w.Header().Set("Cache-Control", "no-cache")
	w.Header().Set("Connection", "keep-alive")

	flusher, ok := w.(http.Flusher)
	if !ok {
		http.Error(w, "Streaming unsupported", http.StatusInternalServerError)
		return
	}

	// Seek to the end of file
	file.Seek(0, 2)
	reader := bufio.NewReader(file)

	for {
		line, err := reader.ReadString('\n')
		if err == nil {
			fmt.Fprintf(w, "data: %s\n\n", line)
			flusher.Flush()
		} else {
			time.Sleep(1 * time.Second)
		}
	}
```

```
}
```

این بخش از کد وظیفه‌ی راه‌اندازی یک وب‌سرور ساده را بر عهده دارد که روی پورت ۸۰ اجرا می‌شود. در این وب‌سرور، مسیر اصلی / فایل‌های ایستا را سرویس‌دهی می‌کند و مسیر events/ برای ارسال رویدادها به روش Server-Sent Events (SSE) پیاده‌سازی شده است. به این ترتیب کاربران می‌توانند با مرورگر خود به وب‌سرور متصل شوند و در لحظه تغییرات ثبت‌شده در فایل لاگ را مشاهده کنند. مکانیزم SSE یک روش یک‌طرفه برای ارسال داده‌های پیوسته از سرور به مرورگر است که بسیار برای نمایش لحظه‌ای رویدادها مناسب می‌باشد.

تابع sseHandler فایل errors.log را باز می‌کند و آن را در حالت event-stream برای مرورگر ارسال می‌کند. این تابع ابتدا فایل را به انتها (seek) می‌برد تا فقط رویدادهای جدید خوانده شوند. سپس با استفاده از یک حلقه بی‌نهایت، خطوط تازه به دست آمده از فایل را خوانده و به مرورگر ارسال می‌کند. در صورتی که خط جدیدی وجود نداشته باشد، برنامه یک ثانیه صبر کرده و دوباره تلاش می‌کند. در نتیجه این مکانیزم امکان نمایش بلادرنگ خطاها و رویدادهای ثبت‌شده در فایل لاگ را از طریق یک رابط وب فراهم می‌سازد.

کد مربوط به کرنل‌ماژول heartbeat

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/timer.h>

static struct timer_list heartbeat_timer;
static unsigned int interval = 3;

static void heartbeat_fn(struct timer_list *t) {
    printk(KERN_INFO "heartbeat: alive\n");
    mod_timer(&heartbeat_timer, jiffies + interval * HZ);
}

static int __init heartbeat_init(void) {
    timer_setup(&heartbeat_timer, heartbeat_fn, 0);
    mod_timer(&heartbeat_timer, jiffies + interval * HZ);
    printk(KERN_INFO "heartbeat module loaded\n");
    return 0;
}

static void __exit heartbeat_exit(void) {
    del_timer_sync(&heartbeat_timer);
    printk(KERN_INFO "heartbeat module unloaded\n");
}

module_init(heartbeat_init);
module_exit(heartbeat_exit);

MODULE_LICENSE("GPL");
```

این کد یک ماژول کرنل لینوکس به نام heartbeat را پیاده‌سازی می‌کند که وظیفه‌ی آن ارسال پیام‌های دوره‌ای به منظور نشان دادن وضعیت زنده بودن سیستم است. این ماژول با استفاده از یک kernel timer در فواصل زمانی مشخص (به‌صورت پیش‌فرض هر سه ثانیه) پیغام "heartbeat: alive" را در لاگ کرنل (dmesg) ثبت می‌کند. این پیغام‌ها توسط سیستم نظارتی (برد دوم) خوانده شده و به‌عنوان نشانه‌ای از سلامت و فعالیت سیستم اصلی مورد استفاده قرار می‌گیرند.

تابع heartbeat_init هنگام بارگذاری ماژول اجرا می‌شود و تایمر را راه‌اندازی می‌کند، در حالی‌که تابع heartbeat_exit هنگام خروج ماژول، تایمر را غیرفعال و منابع آن را آزاد می‌کند. هر بار که تایمر منقضی می‌شود، تابع heartbeat_fn فراخوانی شده و علاوه بر چاپ پیام، تایمر مجدداً تنظیم می‌شود تا چرخه ادامه یابد. این طراحی ساده و کارآمد، مکانیزمی مناسب برای ارسال ضربان قلب نرم‌افزاری در سیستم‌های نهفته فراهم می‌کند.

۵ نتایج

پروژه‌ی ارائه‌شده ترکیبی از سخت‌افزار و نرم‌افزار است که با استفاده از دو برد Raspberry Pi 3 و سیستم‌عامل Yocto یک بستر قابل اعتماد برای پایش و تشخیص مشکلات سیستم ایجاد می‌کند. طراحی ماژول کرنل heartbeat، تحلیل لاگ‌های dmesg و پیاده‌سازی وب‌سرور برای گزارش‌دهی، یک معماری کامل و کارآمد را به وجود آورده است که نه‌تنها قابلیت شناسایی خطاها و ریبوت‌ها را دارد، بلکه امکان نظارت بلادرنگ بر سلامت سیستم را نیز فراهم می‌سازد.