RFID-powered vending machine

A group project at D-ITET, ETH Zurich

Fabian Brun
Johannes Suter

September 2, 2012

We enhanced an ordinary vending machine to empower ETH students to get free drinks with their RFID capable student IDs. For this reason we dived into vending machine specifications, RFID communications and Python programming. As a result we are able to control the vending machine from our own embedded computer. Thanks to the stable, extensible framework, additional features are easily possible - RFID-powered vending is the first cruical step.

Contents

1 Introduction

The student association for mechanical and electrical engineering students at ETH Zurich (AMIV) bought a beverage vending machine in 2011 to be put in a student lounge. The main idea is that every student of the participating associations would be able to get free (or cheap) drinks with their student ID, depending on each associations policy.

To achieve this, we need to get control over the vending machine. We will accomplish this by connecting an embedded computer to the vending machine. This computer would also connect to a RFID reader device, the authentication device. Authorization will be provided by different web-based interfaces, one for each of the participating student associations. An overview about the system can be seen in Figure 3.1.

In the following chapter we give some background about the technologies involved in this project. In chapter 3 we describe the hardware setup and what we have to do to get each component working. Chapter 4 then contains implementation details of our own software. In chapter 5 we describe the test setup and the results, and we give an outlook to the possible future of this vending machine in chapter 6.

2 Background

This is an overview about the relevant technologies we used to reach our goal.

2.1 Controlling a vending machine: MDB

Current vending machines use the multidrop bus (MDB) protocol for communication between the different components (e.g. machine controller, coin validator, card reader). A *multidrop bus* in general describes a computer bus system where all components listen to each other on the same wire at the same time. The MDB protocol runs on such a multidrop bus topology and uses a 9-bit serial protocol for communication: 8 data bits together with 1 mode bit. Such 9-bit serial protocols are pretty rare, they usually only use 8 data bits (1 byte, no mode bit) with the ASCII[1] encoding.

The current MDB specification [1] defines three different feature profiles. While *MDB Level 1* defined the basic command set, the levels above almost only extend that commands to send more data with it. Our vending machine is modern enough to communicate with us on *MDB Level 3*, so we will use that command set.

In the MDB model, the vending machine controller (VMC) is the master on the bus, and all attached components (up to 32) are considered slaves. This means that the master is the arbiter on the bus: the slaves usually listen, and only respond to commands of the master addressed to them. The master regularly polls all components to make sure they are still online; if not, they get reset. There are several categories of peripheral components defined in the MDB spec. One such category is the "Cashless device"[2], which perfectly fits our use case.

[1] https://secure.wikimedia.org/wikipedia/en/wiki/ASCII
[2] see section 7 of [1]

There are some challenges when connecting the MDB interface to a computer:

- The 9-bit protocol in use. It makes it more complicated because we have to read the data as a bit stream rather than a simple byte stream. This is cumbersome to handle in most higher level programming languages (they usually don't really know "bits") unless you use a converter which translates the bits into a byte stream.
- There is a check byte. It is computed as the sum over all other bytes and has to be inserted at the end of each transmission.
- The timing requirements posed by the MDB specification. This is a smaller challenge since our computer should be fast enough to compute the response in time in general. If not, the vending machine controller will reset our component, which looks like a broken transaction to the user.

2.2 Reading student IDs: RFID

The ETH introduced new student IDs back in 2008[3]. They include a RFID chip of the "Legic Prime"[4] family, which is a proprietary system using the standard 13.56 MHz radio frequency[5]. The RFID chip on the ETH student ID carrys a unique six-digit card ID and some (encrypted) memory slots. It is used especially for after-hours entry to specific ETH buldings. There is no publicly available mapping of RFID card numbers to ETH students.

A word about security: The card number is the only "public" information the chip discloses. It is also printed on the back of the student ID. Since we only use this card number, an attacker could claim to be someone else.

[3] http://www.eth-karte.ethz.ch
[4] http://www.eth-karte.ethz.ch/data/security
[5] http://www.legic.com/de/legic_prime.html

2.3 Limiting the free drinks balance

The main goal of extending the vending machine is to be able to give out free drinks to the registered students. Therefore there also has to be a mechanism to enforce limits on the free drinks balance. The participating student associations all have their own member databases, and the rules for the amount of free drinks should also be individual per association. These rules can be even more diverse inside of one such association. This problem will be addressed by introducing "Identity Providers", which expose an API to the student associations databases. Basically this is a HTTP(S) endpoint per association, which returns the amount of free drinks a specific RFID card number is still eligible for.

2.4 Requirements

The final product should be able to:

1. read out student IDs,
2. check the identity providers if it is eligible for a free drink, and
3. communicate with the vending machine, especially to release such a free drink.

3 Hardware

Some parts of the complete system as seen in Figure 3.1 deserve a little more attention.

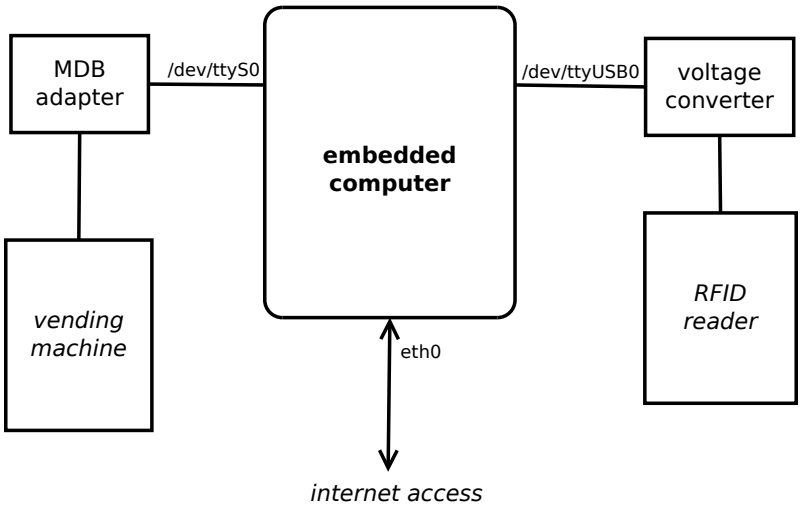

Figure 3.1: High-level overview over the system.

3.1 Vending machine

The vending machine to be extended is a second-hand, off-the-shelf beverage vending machine (see Figure 3.2). It features the aforementioned MDB capable VMC, and a state-of-the-art coin changer (connected as MDB slave). In the end we want to communicate with that VMC.

Figure 3.2: The vending machine, half-way opened.

3.2 MDB to PC converter

To avoid the hassle of connecting MDB to our serial port directly (see section 2.1), we reached out to Abrantix[1], a Zurich based company with focus on cashless payment systems. They developed a MDB-to-PC converter chip, of which they sent us a de-

[1] http://www.abrantix.com/

velopment sample (and later the real product) to experiment with. This converter does all the required conversion for us: adhere to the timing restrictions, calculate the checksum byte and hand over a simple 8-bit serial signal to our computer.

3.3 RFID reader + adapter

The RFID chip on the student IDs is proprietary system. A standard RFID reader won't be able to read out data. The only way to talk to the ETH student IDs was to get the official reader hardware from Legic itself. The reader comes with a non-disclosure agreement (NDA), so some parts of the code are left out of the (otherwise publicly available) code sources.

We also need a Serial-to-USB converter: The official reader hardware outputs its data over a serial link with voltage levels of $0V/5V$. However, the RS-232 plugs of our embedded computer expect voltage levels of $\pm 18V$. AMIV member Pascal Gohl designed a little PCB for us. It has a level shifter on it and lets us connect the RFID reader to a USB port on our computer. That is convenient, since our serial port is already blocked with the MDB connector.

4 Implementation

The whole controlling software is written with the Python scripting language, using the 2.7 version.

4.1 Basic environment

The code runs on a linux box[1] powered by an embedded AMD Geode CPU with 500 MHz, backed by 256MiB of RAM and a 8GB CompactFlash card. The RFID reader hardware will be wired up on the first USB port. The MDB-to-Serial converter chip will be wired up to the serial port of the box.

The software runs in three threads: One for the core controller (section 4.2), one for the MDB `translator` (section 4.3), and one for the `legireader` (section 4.4). This is because the latter two components must not be blocked by the core controller routine at any time. All components are implemented as Python classes. Those who are supposed to run in threads have a special `run` method to start their main loops.

4.2 System overview

The `core controller` is the main component as it controls all other components. As seen in Figure 4.1 all data flows from or to the core controller. The notable exception is the direct exchange of MDB serial data (and only that) between the `translator` and the MDB state machine.

[1] PC Engines ALIX3d3: http://pcengines.ch/alix3d3.htm

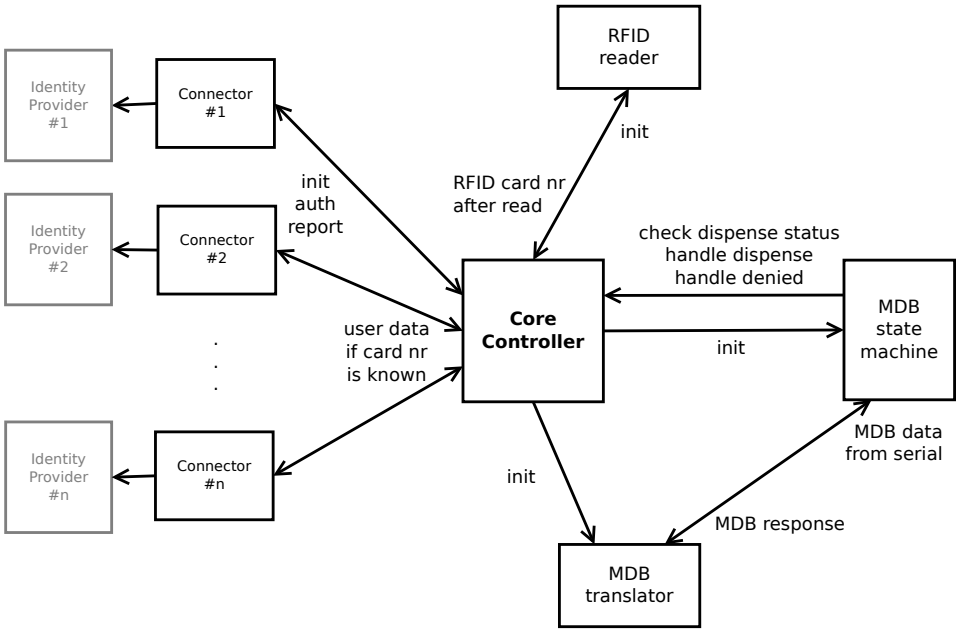

Figure 4.1: System overview

On startup, the controller initializes itself, but does not yet initialize the other components. This is done in a second step when the `start` method is explicitely called. The `start` method first makes sure that only one controller is running at the same time (in the same python process). It then initializes all `connectors`, the MDB components and the RFID reader.

After startup, the core controller waits for the RFID reader to identify a student ID. It then asks all *identity providers* for the user behind the card and his remaining free drinks balance. This is done sequentially for all connectors. It is aborted early as soon as one connector returns a user with remaining free drinks balance. The controller

then starts a timer of 8 seconds, after which the user request expires.

The MDB state machine module asks the core controller in regular intervals if there was a successful legi read/auth. With the help of the translator (see section 4.3) it then unlocks the vending machine and the user gets to chose a drink. The MDB state machine reports either "success", "denial" or "timeout" (in case the user waited too long) back to the controller. It also locks the vending machine again. If the vend was successful, the controller would instruct the responsible connector to report the vend back to the identity provider. After that, the controller will wait until the next successful RFID card readout to start over.

If there is a RFID readout while another user is still processed, the newly read out RFID takes precedence. The older session gets terminated by the MDB state machine.

4.3 MDB communication

The MDB communication part is split into two sub-components. The `translator` is connected to the serial port which is connected to the MDB-to-PC adapter. It recieves MDB commands sent by the vending machine, and responds with appropriate data. The MDB `state machine` is the other component, which takes the commands from the translator and returns the data to be sent back, depending on the state it is in (and conforming to what the MDB specification expects).

The translator recieves MDB commands byte by byte. It collects these bytes until the command (and its data) is complete, and then feeds it to the state machine. There are six different states this state machine cycles through (see Figure 4.2):

1. `DISABLED`: The Cashless Device (CD) does not do anything, until a *reset* or *enable* command is recieved.

2. `INACTIVE`: The CD is ready, but expects setup data or an *enable* command.

3. `ENABLED`: The CD is waiting on a successful RFID read (it regularly polls the core controller).

4. `SESSION IDLE`: A RFID has been successfully read out and is eligible for a free drink. The CD starts a MDB session with the vending machine.

5. `VEND`: The CD signals to the vending machine to unlock the dispense functionality

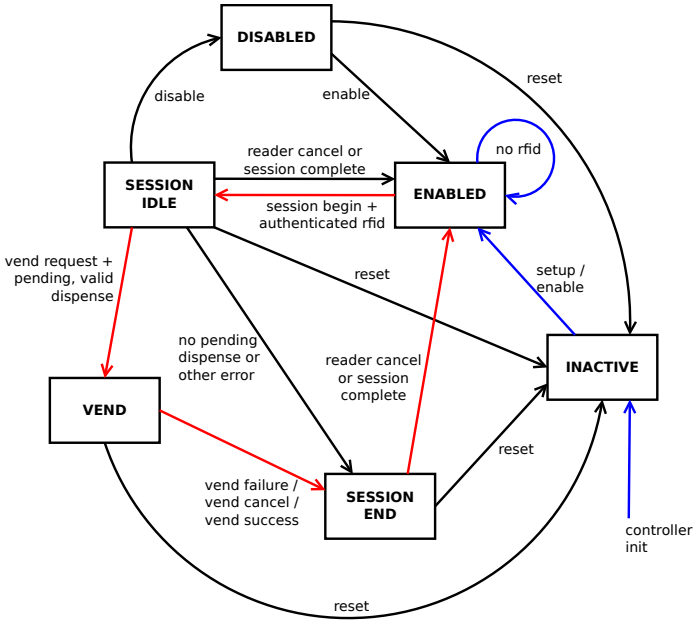

Figure 4.2: The MDB state diagram. The blue path shows the (default) state transitions after startup; The red path shows the (default) state transitions after reading and authenticating a rfid card.

to the user. It also records the choice made by the user.

6. `SESSION END`: The CD ends the open MDB session with the vending machine. In this state, the CD performs mainly cleanup tasks.

4.4 RFID reader

The `legireader` component recieves the data from the reader hardware. To make that reader work, it has to send an "enable" command to activate the reading mode. We send the `enable` command on startup of the `legireader` component, and directly after every successful read. We can't reveal the `enable` command, because this information is protected by the NDA.

Upon detection of a student ID, the RFID reader sends 14 bytes of data over the serial connection:

byte	1	2	3-10	11-13	14
value (hex)	**0D**	**80**	*(don't care)*	**(card number)**	*(data end)*

Only bytes 11 to 13 are interesting to us: they contain the card number. We don't care about the other bytes, as long as the response from the reader has 14 bytes. Our `legireader` code is basically an endless loop trying to read 14 bytes from the serial connection. Once it gets 14 bytes back, and they start with `0x0D80`, we extract and convert the bytes 11-13. They get then passed back to the controller. This is done via a callback installed on initialization of the `legireader`. The `legireader` subsequently returns to waiting for another student ID readout.

4.5 Identity providers

To make the identity provider part extensible, the interface described in Listing 4.1 has to be implemented by a connector to any such identity provider. The connectors to the participating student associations from the start have already been implemented by us.

On controller initialization time, the connectors implementing this interface would get the base configuration. From there, the connector is advised to prepare everything he will need in the `auth` and `report` methods, e.g. database connections. In general they also want to load their own configuration data here.

In the `auth` method, the connector asks his identity provider for a user with the specified RFID card number. If the identity provider knows the card, the connector should return this information together with the "free drinks balance".

When the `report` method gets called, the connector should report back to his identity provider about the (successful) vend. For this purpose it gets passed the RFID card number as well as the chosen slot.

All identity provider communication happens over an SSL encrypted connection to prevent against eavesdropper.

4.6 Audit log

The core controller maintains its own audit log. Every attempted vend (internally called "transaction") gets logged in there, even vend denials (but no timeouts). This audit log is based on *redis*[2], a fast in-memory key-value store. The transactions are recorded by day and whether they were successful or not. For every transaction, there is an additional *hash*[3] with metadata about that transaction.

[2] http://redis.io
[3] Overview over all redis datatypes: http://redis.io/topics/data-types

The redis data is saved to the disk in regular intervals (complete dumps every few minutes, and a command journal all the time).

```
class IdProvider(object):
    """ Base interface for identity providers.

    Attributes:
        orgname: The human readable name of this identity provider
            (i.e. "AMIV"). Will be used in logs etc.
    """
    orgname = None

    def __init__(self, baseconf):
        """ Initialization of the connector.

        Args:
            baseconf: The main ConfigParser configuration object.
        """
        self.baseconf = baseconf

    def auth(self, rfid):
        """ Authenticates the given legi number if possible.

        Args:
            rfid: The six digit RFID number (int as str).

        Returns:
            True if authentication was successful, False otherwise.
        """
        raise NotImplementedError(
            "Method 'auth' must be implemented by class '%s'" % (
                self.__class__.__name__
            )
        )

    def report(self, rfid, slot):
        """ Reports a vending from the given user back to the org.

        Args:
            rfid: The six digit RFID number (int as str).
            slot: The slot the user chose (int).

        Returns:
            True if reporting was successful, False otherwise.
        """
        raise NotImplementedError(
            "Method 'report' must be implemented by class '%s'" % (
                self.__class__.__name__
            )
        )
```

Listing 4.1: Interface for IdentityProviders

5 Evaluation

5.1 Testing setup

To test the proper function of our code, we set up our embedded computer with the Ubuntu 12.04 Server Edition[1]. On top of that comes a Python virtual environment with the `virtualenv` package[2]. Running our setup script with the Python interpreter of the newly created environment installs additional Python packages we depend on in our code.

We then connect everything:

1. The MDB-to-PC adapter to the vending machine's MDB plug, and on the other side to the serial port on our embedded computer.
2. The RFID reader to the Serial-to-USB converter board, and the latter to the embedded computer.
3. The power supply and an ethernet cable to our embedded computer.

We perform our tests with different student IDs. They were kindly pre-configured into the AMIV identity provider according to Table 5.1. With these student IDs, we run the following tests (expected outcome in parentheses):

1. *Banned User*: Use ID **0** to request a drink from any slot (denied, no balance).
2. *User timeout*: Use ID **1** to authenticate, don't press any button for 10 seconds, then request a drink from any slot (denied, vend request timed out).

[1] http://www.ubuntu.com
[2] http://www.virtualenv.org

student ID	configuration
0	not a member: no free drinks at all
1	normal member: 1 free drink per day
2	board member: unlimited drinks per day

Table 5.1: Configuration of test student IDs

3. *Normal vend*: Use ID **1** to request a drink from any slot (successful, correct slot reported).

4. *No-balance vend*: Use ID **1** to request a drink from any slot (denied, machine never gets unlocked).

5. *Multiple vends*: Use ID **2** to request three drinks in a row from three different slots (all successful, correct slots reported)

5.2 Results

To start the software, we run a bootstrap script with the Python interpreter from the virtual environment. The source code is prepared with a lot of debugging log messages. That gives us a good impression of where we are in the code at any given time. See Figure 5.1 for an example of the debug output.

When performing the five tests with the three pre-configured student IDs, all tests passed with the expected results. Additionally, the program ran without throwing or logging any other errors. A long term stability test was not yet possible to perform.

5.3 Security

As mentioned in section 3.3, an attacker could impersonate another student, as long as he knows the card number and can communicate with our RFID reader. This is a

feasible attack since the security of the "Legic Prime" system in use has been broken[3] in 2009. However we do not really consider that a real threat at the moment due to the following reasons:

- There is a limited amount of beverages available in the vending machine. Even if somebody tries to get a lot of free beverages with this trick, the harm is manageable from a monetary point of view.
- There is one specific person in charge for refilling. This person would recognize an increased demand and probably look closer.
- There are always a lot of people in the room. A person trying to get a lot of beverages from the vending machine would probably not go undetected.
- Students who do not get their free beverage would report a problem. A lot of such reports would be striking, and the audit log would probably help to detect the attack afterwards.

In summary, we think that such an attack would work once (or for a very limited time) only. A countermeasure would be to authenticate the student ID in any way, e.g. with an additional PIN code to enter.

Another weakness in the first system design was the reporting of the vend back to the issuing identity provider. If the machine can't report back (e.g. due to a network error because the ethernet cable has been unplugged), the identity provider would allow another free drink the next time the student tries (because it does not know about the previous ones). The countermeasure we implemented is a simple blacklist: If the reporting fails, the rfid card number gets blocked for the rest of the day.

[3] http://events.ccc.de/congress/2009/Fahrplan/events/3709.en.html

Figure 5.1: Example log output of our software.

6 Conclusion & Outlook

The initial work described in this report has set the ground for more creative work with this rather unusual piece of hardware. The delivered software stack works stable and should be easily extensible for additional features.

Such features may include:

1. The vending machine has a very powerful coin checker/changer. A possible idea to include it in our system: Only serve drinks when a valid student ID has been shown (also for paid drinks). The rationale behind this would be to restrict the vending machine to students of the participating student associations, since this is their service for their students.

2. A status screen has been wished a lot by the AMIV board. It would be built into the vending machine's body. Besides status data (display free drinks balance or history), other student related information could be displayed (e.g. upcoming events and the like).

3. Such a status screen may additionally enable students to write apps, such as minigames (to be played with a touchscreen and unlocked by the student ID) or statistics apps.

7 Bibliography

[1] National Automatic Merchandising Association (2003). *Multi-Drop Bus / Internal Communication Protocol, Version 3.0*. http://www.vending.org/technical/MDB_3.0.pdf

8 Figures and Tables

8.1 List of Figures

8.2 List of Tables

8.3 Listings