Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Design and implementation of an RFID access control system

Bachelor's Thesis

2016

Kamila Součková

Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Design and implementation of an RFID access control system

Bachelor's Thesis

Študijný program:	Informatics
Študijný odbor:	2508 Informatics
Školiace pracovisko:	Department of Computer Science
Školiteľ:	RNDr. Richard Ostertág, PhD.

Bratislava, 2016

Kamila Součková

53637720

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Kamila Součková
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Design and implementation of an RFID access control system
Návrh a implementácia RFID prístupového systému

Cieľ: Cieľom práce bude navrhnúť a implementovať server pre prístupový systém Deadlock. Práca by mala obsahovať:
• návrh servera, ktorý bude udržiavať databázu identifikátorov RFID prístupových kariet a to najmä návrh:
o architektúry servera,
o dátových štruktúr pre efektívne uloženie a spracovanie prístupových práv,
o komunikačného protokolu so zariadeniami riadiacimi prístup do chránených priestorov,
o riešenia aktualizácie firmvéru k serveru pripojených zariadení,
o zaznamenávania prístupu do chránených priestorov,
• implementáciu navrhnutého riešenia v jazyku Python 3.

Vedúci:	RNDr. Richard Ostertág, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 28.10.2015

Dátum schválenia: 28.10.2015

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent vedúci práce

Acknowledgements: I would like to thank RNDr. Jaroslav Janáček, PhD., RNDr. Richard Ostertág, PhD., and Mgr. Tomáš Vinař, PhD. for their valuable input on the design and requirements of the system. Without their insightful discussions, Deadlock would lack the strengths that make it unique, such as reliability and simplicity.

I would also like to thank Adam Dej, who designed and is at the time of writing implementing the embedded devices. His input about what to expect from the controllers has been invaluable not only, but especially for the implementation phase.

Finally, I would like to thank Michal Hanula and RNDr. Richard Královič, PhD. for listening and asking questions. They helped me to see things from a different viewpoint, which led to considerable improvements.

Abstract

Project Deadlock is a system that controls access to a number of points of access (e.g. doors, appliances) using RFID cards. Deadlock is designed for security and reliability, assuming untrusted and unreliable network. It is fully open-source and open-hardware, and designed to be flexible, maintainable, and cost-effective. It is easy to integrate with existing systems and customize to the needs of the user.

This thesis first lists the requirements and introduces the high-level design choices we made in order to fulfill them. It then focuses on the design and implementation choices of some parts of the system. We start with the design and implementation of the server/controller communication protocol, focusing on the conflicting requirements of reliability, extensibility and simplicity. Next, we describe the access rules format and evaluation, especially the compromise of generality vs. user-friendliness. Third, we provide an overview of the server design and implementation, telling the tale of how good design choices and modularity led to a clean and simple implementation. We conclude with the future plans for Project Deadlock.

Keywords: access control, networked system design, network protocol design, reliable system design

Abstrakt

Deadlock je systém na kontrolu prístupu do miestností alebo k zariadeniam na základe identifikáce RFID kartou. Je dizajnovaný s dôrazom na spoľahlivosť a bezpečnosť, predpokladajúc nezabezpečený a nespoľahlivý komunikačný kanál.

Táto práca sumarizuje dizajn systému a podrobne popisuje dizajn a implementáciu komunikačných protokolov, prístupových pravidiel a centrálneho servera.

Kľúčové slová: kontrola prístupu, dizajn distribuovaného systému, dizajn sieťového protokolu, dizajn spoľahlivého systému

Table of Contents

Introduction

Project Deadlock is a system that controls access to a number of points of access (e.g. doors, appliances) using RFID cards. Deadlock is designed for security and reliability, assuming untrusted and unreliable network. Unlike existing commercial solutions, Deadlock is fully open-source and open-hardware, and designed to be flexible, maintainable, scalable, and cost-effective. We provide tools and expose all interfaces and components, making Deadlock easy to integrate with existing systems and customize to the needs of the user.

Deadlock is a project of the Student Development Team[1] at the Faculty of Mathematics, Physics and Informatics of Comenius University. It is implemented by students and supervised by faculty members.

This thesis first lists the requirements (chapter 1) and introduces the high-level design choices we made to fulfill them (chapter 2). These were developed jointly by the Student Development Team. We then focus on the author's contribution in the rest of the thesis. Chapter 3 describes the design and implementation of the server/controller communication protocol, focusing on the conflicting requirements of reliability, extensibility and simplicity. Chapter 4 describes the access rules format and evaluation, especially the compromise of generality vs. user-friendliness. Chapters 5 and 6 provide an overview of the server design and implementation, telling the tale of how good design choices and modularity led to a clean and simple implementation. We conclude with the future plans for Project Deadlock (chapter 7).

[1]http://svt.fmph.uniba.sk

Chapter 1

Requirements

Project Deadlock aims to create a complete system to allow cards compatible with the ISO/IEC 14443a standard (commonly known as *RFID cards*), such as International Student/Teacher Identification Cards, to be used to unlock doors and access other electronic equipment (hereafter *points of access* or *PoA*).

For this system to be useful at our university, Deadlock must meet the requirements outlined below.

1.1 Trustworthiness

As Deadlock may be used to protect valuable resources, such as computer rooms or labs, it must allow access when and only when it should.[1] We must provide the user with reasons to trust this promise.

1.1.1 Reliability. Points of access should be accessible even when things go wrong; specifically partial power or network outages must not prevent the system from allowing access, nor cause loss of access logs. Temporary server failure must also cause no problems. Furthermore, the design and implementation should allow for a simple failover mechanism.

[1]See section 1.3.1 for the discussion of behavior during power outages.

1.1.2 Security. Deadlock must not allow illegitimate access. Also, to protect privacy, logs or card IDs must not leak. We cannot assume a private communication channel. Therefore all communication in both directions must be authenticated and kept confidential.

1.2 Practicality

Deadlock must be an effective solution for our use case. This must hold even if the use case changes in the future.

1.2.1 Extensibility. In order to be prepared for the future, and also to make incremental development possible, all software and all hardware must be modular, with well-defined interfaces, and extensible.

Functions not implemented in the first iteration, but expected to be added in the future, are

- arbitrary communication with the card,[2]
- controlling arbitrary appliances, not just door locks,
- WiFi module (for cases when power is available but Ethernet is not).

1.2.2 Ease of development. In the future Deadlock will likely be developed and maintained by students, not full-time developers. Therefore the codebase must be simple, easy to understand and change, the tools and libraries must be easy to use, and the overhead of introducing a new developer to the project must be minimal. When possible, general, well-known solutions should be used instead of solutions developed in-house.

1.2.3 Ease of use. Setting up access rules should be simple and convenient. This should not come at the expense of generality. Synchronization with the university's

[2]RFID cards are capable of complex actions, such as cryptographic verification of identity, or local data storage. Currently we only support reading the card's ID, but the communication stack is ready for extension.

electronic information system is required, so that card information and groups like "CS teachers" or "PhD students" can be imported automatically.

The system should notify the operator if human intervention is required, but simple tasks and predictable issues should be handled automatically.

1.2.4 Ease of deployment and maintenance. Deployment should be simple and with minimal overhead. On the hardware side, it should be possible to leverage existing infrastructure in order to not need extra cables for communication or power. On the software side, importing data from existing sources (such as our university's Academic Information System) should be possible. Replacing any failed components should be quick and should not require substantial training. The system should check its state and automatically fix whatever can be fixed automatically, e.g. reboot a device if it gets locked up.

1.2.5 Availability. Hardware should be cheap to manufacture and components should either be available in the foreseeable future or painlessly replaceable by newer alternatives.

In order to make Deadlock as available as possible, we release both the hardware schematics and the software to the public under the MIT license.

1.3 Further considerations

1.3.1 Power outage behavior. In case of a power outage at entrance/exit PoAs, some doors should stay locked (to avoid the risk of breaching security), and some doors should open (e.g. emergency exits). Both can be supported by using different lock hardware and changing configuration.

1.3.2 Emergency open. The hardware locks on entrance/exit PoAs must support manual opening and locking by authorized personnel. This is useful in emergencies.

Chapter 2

System overview

2.1 Existing systems: overview and comparison

The solution currently employed at the university, as well as most commercially available alternatives, consist of a number of simple card readers and a centralized decision-making server and access management interface. They usually require custom wiring; introduce vendor lock-in because of proprietary communication protocols; and cannot operate when the server is unavailable. In contrast, Deadlock will rely on standard, hopefully already existing infrastructure; provide the communication protocol specification and libraries to extend our system; and make sure Deadlock continues to operate when the server cannot be reached.

Existing commercial solutions are also expensive (usually several hundred dollars per unit) and because of vendor lock-in parts cannot be replaced by alternatives. Deadlock aims to be almost an order of magnitude cheaper, and fully open.

2.2 Key design principles

2.2.1 Modularity. Separating the functionality into independent modules with well-defined, simple, minimal interfaces simplifies development, makes the design much easier to grasp and minimizes the learning curve for a new developer. Things are better if one knows where to look for certain functionality (and where not to).

2.2.2 Principle of Least Astonishment. "People are part of the system. The design should match the user's experience, expectations, and mental models." [8] If the design, implementation or behavior of a part of the system is obscure enough to surprise you, it should be redesigned.

We should prioritize predictability of the parts of the system which are hard to observe (such as the embedded devices), so that they do not unpleasantly surprise the user.

2.2.3 State is ugly. State adds complexity to a system: if something has state, it has code managing it, and the developer must keep track of how the internal state influences the system. Also, if the system is stateful, it complicates failure recovery, as the state must be replicated or otherwise re-creatable. Therefore when at all possible, the components of Deadlock should have little internal state and depend only on explicit, well-defined, easily replicated data.

2.3 Main components

The system consists of a server and a number of controllers. Each controller serves a single point of access, holds a copy of the access rules and evaluates them locally. The server provides controllers with rules updates and collects access logs.

2.3.1 Server. The server holds the authoritative version of the access rules, collects logs and provides software updates and time synchronization for the other devices. It monitors system state (and reports it to the management UI).

Except for the contents of the database, it is entirely stateless. This simplifies the code and makes replication and failover trivial.

2.3.2 Controller. The controller controls its associated point of access (e.g. unlocks its door). It takes actions (such as opening the door or logging) based on events observed (such as a card being presented or the door being opened). It periodically contacts the server, reporting its status and checking for updates.

The controller is "almost stateless" – logs are sent to the server, and rules and firmware updates can be retrieved from the server. Therefore a device can be swapped simply by writing the correct device ID and encryption key to either the device or the database.

2.3.3 Reader. Up to two card readers may be attached to one controller. A library to interface with our readers is provided, so they can be used independently of our controller.

2.3.4 Hardware. The server is hardware-agnostic – it runs on anything with networking and a Python environment. Deployments will usually use generic server hardware.

ŠVT has designed and built custom hardware for the controllers and readers. They focused on making it available and future-proof, extensible, and cheap. The schematics and other documents are available in the Deadlock source repository [9].

In order to simplify installation, we have attempted to leverage existing infrastructure wherever possible: we use Ethernet for server/controller communication, adding optional Power over Ethernet, so we don't require any extra cables. Optionally, we can add a WiFi module to the controller for cases where electricity is available but connectivity is not. We even designed our reader boxes and connection cables to be easy to customize, so that they can be made compatible with existing holes in walls.[1]

2.4 Access rules

The decision whether to grant access is a function of user identity, point of access, date, time, and day of week. The access rules are designed to be simple without sacrificing generality. For details see chapter 4.

[1]For example, instead of using an expensive mold for the reader boxes, we make them from several layers of Plexiglas that can be cut individually. The source files for the cut pattern are available and easy to modify.

2.5 Technical challenges

2.5.1 Reliability. As required by section 1.1.1, controllers must work during network failures. Continued operation is ensured by storing and evaluating the rules locally on the controller, and only needing the server for updating the local copy. Loss of access logs is avoided by saving them locally and never forgetting a log entry before it has been stored to disk by the server.

Multiple servers may be deployed for higher availability, as long as the backing database is synchronized by generic database replication mechanisms (eventual consistency is sufficient).

See chapter 3 for details.

2.5.2 Security. See section 3.6.

2.5.3 Easy deployment and maintenance. Due to requirements in sections 1.2.4 and 1.2.5, communication is built on standard Ethernet, and we support the Power over Ethernet standard. Adding and replacing devices must be quick and easy, and therefore we made sure it was possible to pre-configure devices and then just plug in as needed.

Deadlock must be usable decades from now, therefore it must depend only on components, libraries and tools which are likely to stay. This requirement needed to be taken into account when designing the hardware and software.

TODO fix page breaks in tables and after headings

Chapter 3

Server/controller communication protocol

3.1 Design requirements

The server/controller communication protocol must facilitate reliability, security, extensibility and ease of development and deployment, as defined in sections 1.1.1, 1.1.2, 1.2.1, 1.2.2 and 1.2.4. This led to the following decisions:

3.1.1 Statelessness and idempotence. In order to keep the protocol simple, yet reliable, communication should be stateless and all messages should be idempotent. This allows to simply retry anything that failed for any reason, at any time.

3.1.2 Simplicity. The protocol should be simple. This includes not only simplicity in protocol implementation, but also simplicity in message parsing on any device and in any programming language. Specifically, parsing must be efficient on embedded devices with low CPU frequency and memory.

3.1.3 Extensibility. The protocol must be easily extensible. Additionally, a mechanism for seamlessly transitioning to a newer version in a live system should be provided. These must not interfere with the simplicity requirement.

3.1.4 Security. We do not want to rely on security mechanisms provided by lower layers (as they may or may not be adequate, or present). Therefore the application layer of the protocol must provide sufficient authentication and secrecy.

3.1.5 Built on standard, well-known technologies. Code is a liability. Our code has our bugs, requires specific knowledge, and is our problem. Therefore the protocol should require as little own code as possible.

Note that fulfilling these requirements is not at all trivial, as some, especially extensibility vs. simplicity, or security vs. simplicity, may end up contradicting each other.

Also, statelessness and idempotence are not strictly required, as reliability may be achieved in different ways, too. However, as shown in the following sections, statelessness is a very useful approach, as it allows to keep things simple even in the face of requirements that would otherwise need significant complexity.

3.2 Protocol design

3.2.1 Alternatives considered. The original suggestion was a connection-oriented protocol, where either the server or the controller could initiate communication. This would abstract away details like maximum packet size, handle lost packet retransmissions, and allow for pushing rule or firmware updates from the server side. However, this approach presents multiple problems, such as

- high server failover cost: in a stateful protocol, connection state would either have to be synchronized among multiple servers, which is impractical, or all communication would have to be restarted from the beginning on failure;
- added reliability only at the cost of added complexity: again because of problems with server state;

- more complexity on the server: the server would need to keep track of active controllers, and who has said and heard what;
- extra CPU cycles, flash and memory usage on embedded devices: while an efficient implementation of these abstractions might be unnecessary or could be written, nonexistent code is easier to write, more efficient, and has fewer bugs.

3.2.2 Chosen design: overview, rationale. The chosen communication protocol is connectionless and stateless, and all information exchange is of the form "controller request $\rightarrow$ server response". All requests (including retries) can be served independently of any other past, present or future requests from this or another controller.

This implies that there is in fact no strict requirement to serve all requests, or any particular subset of requests, by a single server. Multiple server instances can be used as long as the data they need can be synchronized (and even for this synchronization, eventual consistency is sufficient). Therefore, the system may be configured to use several servers, and controllers are expected to send requests, including retries, in a more or less round-robin fashion.[1]

The round-robin scheduling is particularly useful for retries: if there is a problem with a specific server or a specific part of the network, the controller will simply resend the request to a different server until a good response is received. Therefore the probability that no server can be reached can be significantly reduced by deploying multiple servers in different parts of the network.

As will be shown in section 3.4, under normal circumstances the server/controller communication is not latency-sensitive, so the round-robin retries approach does not pose a latency problem. Therefore the controllers use round-robin with generous timeouts and exponential back-off to avoid network congestion.

A "bad" response (such as one that cannot be parsed, or an error) is treated the same as if no response were received (except for possibly different timeouts, logging and such), i.e. a retry is sent to the next server. This allows for uniformly handling all

[1]"More or less" means that the controller is allowed to cache "server dead" information in order to skip dysfunctional servers.

kinds of transient and permanent problems with the server, network or other resources.

3.2.3 Live system upgrades. A welcome consequence of the message independence and round-robin retries for all errors is that even if a server cannot parse a controller's request, or a controller cannot parse a server's response, the controller will simply retry with a different server. Therefore in order to transition to an incompatible protocol version, all that is needed is deploying servers with both the "old" and the "new" protocol, and the controllers will simply retry until they find a compatible server. Together with the fact that the server can automatically deliver firmware updates, and that controllers report their firmware version to the server, this makes any and all online system upgrades trivial and fully automatic.

3.3 Network stack

The standard network stack is used: Ethernet (IEEE 802.3) as the physical and data link layers, IP as the network layer[2] and UDP as the transport layer. The server and controller IP addresses are configured statically on both sides.

3.3.1 UDP vs TCP. As a standard TCP implementation is available for all devices we will use (it is even bundled with the real-time OS used for the embedded devices), it seems like using TCP would provide benefits at no additional cost. However, as our protocol is stateless and packet-oriented, and manages retransmissions on the application layer, the only benefit of TCP would in fact be unlimited "packet" length (as opposed to 64kB for UDP [6]), and other than that we would end up emulating a UDP-like service on top of TCP if we chose to use it.

While the unlimited message size looks useful, it is in fact not that helpful – the only messages that do not fit into a single UDP packet are rule database and firmware blobs, and for these it is more efficient to deliver them in explicit chunks, so that the transfer of these large files does not need to start over in case something goes wrong.

[2]Both IPv4 and IPv6 are supported, and standard ARP or NDP, respectively, is supported for network to link address resolution. IP addresses may be configured statically or obtained via DHCP.

Therefore, as the benefits of TCP are in our case not worth the TCP overhead and flash space on embedded devices is limited[3], we have chosen to use UDP.

3.4 Message types, controller behavior

Controllers are expected to download a local copy of the rules database and query that instead of contacting the server whenever access is requested. They send access logs, report their status, and request updates of the rules database and firmware.

As all communication must be initiated by the controller, it must periodically contact the server in order to find out if an updated rules database or firmware is available.

All responses have a response status tag, the value of which is one of `RESPONSE_OK`, `RESPONSE_ERR` (permanent error), `RESPONSE_TRY_AGAIN` (transient error). Any non-`OK` response must be treated as if the response did not arrive (i.e. usually a retry as in section 3.2.2 is necessary), except for possibly different timeouts, logging or scheduling. In the following, only `OK` responses are shown.

Note: The following details the "semantic" data types. The details of the encoding are specified in section 3.5.1. The type "byte string" represents a binary-safe string, or an array of bytes of arbitrary length.

Currently, the following message types are recognized:

3.4.1 `PING`: keepalive, DB and FW version info.

Contacts the server to report current status and request info about updates. Also used to adjust controller time.

Request:

Field	Type	Description
time	integer	what time the controller thinks it is
db_version	integer	version of the rules database currently in use

[3]The current controller model will have slightly less than 256kB of programmable flash memory, of which less than 128kB is usable, because we need to store two versions of the firmware when doing online firmware upgrades. Therefore the several kB saved by not compiling in the TCP stack might come in handy.

Field	Type	Description
fw_version	integer	currently running firmware version

Table 3.1: `PING` request

OK response:

Field	Type	Description
time	integer	server time
db_version	integer	newest available version of the rules database
fw_version	integer	newest available firmware version

Table 3.2: `PING OK` response

The controller is expected to adjust its clock to match the server time.

3.4.2 `ALOG`: transfer access logs.

Sends access logs to the server.

Controllers attempt to send access logs as soon as possible, but in order to not lose them, they are saved to the SD card until the server confirms they have been written to disk.[4]

Request:

Field	Type
records	array[5] of `log_record`s (defined in table 3.4)

Table 3.3: `ALOG` request

[4]We recommend at least 4GB SD cards in order to have enough space for flash wear leveling. As each log record is about 20-30 bytes (depending on the encoding), at a rate of 1 access per second (which is somewhat overstated) it would take about 5 years to run out of space.

[5]The array may end with a termination symbol instead of having an explicitly specified length. See section 3.5.1 for details of the encoding.

Field	Type	Description
time	integer	timestamp
card_id	byte string	card that requested access
allowed	boolean	was access granted?

Table 3.4: `log_record` structure

OK response: All sent records have been written to disk. (Response body empty.)

3.4.3 `XFER`: transfer a file chunk.

Firmware and rule database updates are treated as opaque binary blobs by the `XFER` command. They are identified by type and version. In order to trivially support incremental downloading and also arbitrary chunk sizes, the controller explicitly requests the offset and length of the chunk. The same version must always refer to an exactly identical blob (if it exists), even if requested from a completely independent server.[6] The server may return a smaller chunk (e.g. because end of file was reached), but never longer. A chunk of length 0 indicates end of file.

Request:

Field	Type	Description
filetype	enum	`DB` and `FW` currently supported
fileversion	integer	same version => same contents
offset	integer	offset from the beginning of the blob
length	integer	

Table 3.5: `XFER` request

OK response:

[6] This is implemented by using the file's 64-bit hash as version, but that is a detail irrelevant for the protocol, as versions should be treated as opaque integers.

Field	Type	Description
length	integer	may be less than requested
chunk	byte string	the file chunk contents

Table 3.6: `XFER OK` response

The server will return a `TRY_AGAIN` error if the file was not found. This would usually happen because one server already received and processed an update and another one is behind. The controller will simply retry until it finds a ready server.

Note: These are the only responses longer than a few bytes. The server will send whatever size it is asked for (up to the generous packet size limit). It is each controller's responsibility to not ask for chunks that may result in replies that are too long for it to process. This is to allow maximum efficiency with controllers with different capabilities.

3.4.4 `CRITICAL`: report a critical problem.

Used to report a critical problem, upon which the server should take immediate action.

Request:

Field	Type	Description
code	enum	error code
message	optional text string	details of the error, if any

Table 3.7: `CRITICAL` request

Currently the only recognized code is `LOCK_FORCED_OPEN` (a physical lock was opened without permission), but we assume that more uses will emerge when preparing for real-world deployments.

OK response: Acknowledged, action taken. (Response body empty.)

3.4.5 `ASK`: ask if access should be granted now. Because of the potentially high latency of roundtrips, local evaluation rather than querying the server should be used in production. However, we include this for special cases and as a fallback.

Request:

Field	Type	Description
card_id	byte string	card that requested access

Table 3.8: `ASK` request

Whether access should be granted is a function of identity, time and the PoA's type. In this case, this card's identity, the current (server) time and the type of the PoA associated with this controller are used.

OK response:

Field	Type	Description
allow	boolean	should we grant access?

Table 3.9: `ASK OK` response

3.4.6 `ECHOTEST`: echo for testing purposes. Echoes the response body. This is helpful in integration testing. Live deployments are recommended to run a process that will act as a controller sending `ECHOTEST` (and possibly other) requests and report any problems. (Such a process is run by default – see section 5.2.4.)

3.5 Packet format

3.5.1 Record encoding. All requests and responses, as well as the outer packet envelope, are "records", i.e. small key-value mappings with fixed key names and types. Therefore we originally wanted to simply transmit "C structs" (i.e. binary blobs with fixed offsets for fields) and hard-code field offsets in the server and controller firmware. However, this approach has multiple disadvantages:

- Any extension would be an incompatible change, and therefore would require the full upgrade procedure as described in section 3.2.3. While this procedure is simple, when it is running, the system requires more servers to achieve the same level of redundancy; and it may make administrators nervous.
- When parsing fails, we don't know anything more specific than "parsing failed".
- Sometimes (e.g. when length was not fixed or did not change) we may parse a packet incorrectly without noticing.
- The blob is not self-describing, and therefore it cannot be parsed correctly without the context of the outer envelope specifying the version and the description of fields for this version.

Especially the concerns around parsing errors are significant enough to justify a self-describing encoding. Therefore we need an encoding with the following properties:

- self-describing: key names and types must be present in the the encoded data
- expressive: it must be possible to include all the necessary types and arbitrarily nest them as arrays or sub-records; fields must be optional
- binary-safe: able to transmit arbitrary binary data (e.g. card IDs or file chunks) without the need for extra encoding
- not incompatible by default: when a backwards-compatible change is introduced (such as adding a new optional field, or removing a field that was optional), old and new code must be able to communicate without change
- suitable for embedded devices: encoding and decoding must be fast, using small code size and producing small messages
- standard, with existing libraries: our code is our problem – the less code we write, the less code we will need to maintain in the future

These requirements are perfectly fulfilled by the Concise Binary Object Representation (CBOR, see [4]) – a data format designed for communicating with constrained nodes. We use arrays of CBOR semantically tagged items to represent records.[7] (These are equivalent to arrays of $(tag, data)$ pairs, where $data$ is strongly typed.) Unknown tags are ignored and from the parsing viewpoint all fields are optional. In this way the only thing that a server and a controller must have in common to communicate are the tag interpretations (which makes sense, if they want to use the values for something useful).

3.5.2 Requests, responses. For all requests and responses, the record as specified in 3.4 is tagged by a semantic tag for the corresponding message type, and in case of response records this is in turn tagged by the response status.

3.5.3 "Envelope" – version, addressing, encryption. The outer layer of the messages (common to requests and responses) provides addressing and encryption. It is a record with fields as specified in table 3.10. The encoded record is prepended with a 4-byte "magic number" containing the bytes $[68, 69, 65, 68]$ ('DEAD' in ASCII) identifying this as a Deadlock message.

Field	Type	Description
Version identifier	integer	unknown version must be ignored
Controller ID	integer	addressing
Nonce	24 bytes	random bytes
Payload	byte string	encrypted request/response

Table 3.10: Message record.

Version identifier Packet must be considered invalid if this does not match a known version. This is to support live system upgrades, as detailed in section 3.2.3.

Controller ID Unique identifier of the sender or intended recipient. Serves as address-

[7]If a duplicate tag is encountered, it is considered an error. In addition to serving as a sanity check, this might also prevent some overflow-related attacks.

ing. Including a form of addressing on the application layer decouples "logical" addressing from "physical" (i.e. network) addressing, thereby allowing Deadlock to function over NAT, with broadcast/multicast/anycast IP addresses, and such.

Nonce Randomly generated bytes. Matches a response to a request: when a request nonce is x, the associated response's nonce must be $x \oplus 1$. Used as detailed in 3.6.

Payload Request/response, encoded according to section 3.5.1. Encrypted with the key for the given controller using the nonce, as detailed in 3.6.

Note: Maximum message size (when encoded and encrypted) is 63kB (in order to comfortably fit into a UDP packet).

3.6 Security

Security is complicated. While libraries implementing cryptographic primitives exist, they usually do not make securing an application particularly easy: the developer must be aware of what needs what kind of security; which primitives (such as cipher, cipher mode, checksums, signatures) are suitable for which use case, what they promise, what their weaknesses are and whether they are a problem in the given use case; she must consider potential side channel attacks, replay attacks, and such; and she must ensure other developers are aware of all these considerations. As the numerous vulnerability reports published each month signify, this is no easy task.

Short of locking one's computer in a closet without electricity, the best way to secure a system is to leave it to an expert. Luckily, in 2013 the NaCl library interface specification [2] and several implementations were published, with the aim of providing developers with a simple, "sane defaults" crypto toolkit. See [3] for a discussion of the impact of such a library.

In Deadlock, we assume operation over untrusted networks, and we must resist both passive and active attacks. Therefore we encrypt and authenticate all messages from/to a given controller with a device-specific symmetric key, using NaCl's `secret_box(nonce, key, payload)` function, which promises secrecy and integrity provided the nonce is not used more than once [2]. We construct the nonce by generating 24 random bytes,[8] which ensures negligible collision probability (quick birthday paradox approximation says the probability reaches 50% after more than 10^{28} packets, which is a lot). Symmetric cryptography was chosen for performance, but once the actual controller hardware and firmware exists, we are planning to run benchmarks and switch to asymmetric cryptography if possible, in order to avoid the need to copy the secret to more than one place.

The default NaCl primitives in NaCl are the Salsa20 stream cipher for symmetric encryption and the Poly1305 MAC for message authentication. As detailed in [1], these are performant and secure without depending on any form of hardware acceleration, which goes well with our requirements.

3.6.1 Security guarantees. Provided a nonce is not used more than once, `secret_box(nonce, key, payload)` guarantees

- **secrecy**: it is infeasible to decrypt a message without knowledge of the key;
- **integrity**: if a message is decrypted successfully, no accidental or purposeful third party modification of the nonce or the encrypted payload can have occurred;
- **resistance to timing attacks**: the implementations try to always perform the same amount of work.

Furthermore, the protocol's idempotence and use of nonces **prevents replay attacks**: if an attacker attempts to replay a request to a server, nothing bad will happen as all requests are idempotent; if she replays a response to a controller, its nonce will not match any of the requests the controller is currently expecting and therefore it will ignore the fake response.

[8]"Random" in this case does not mean cryptographically secure randomness – nonces may be predictable (they are sent in cleartext along with the payload anyway), the only requirement is a uniform distribution to ensure low collision probability. The fact that NaCl does not require a source of good randomness is in embedded environments very welcome.

Chapter 4

Access rules

4.1 Generality vs. convenience

The key observation when designing the access rules is:

Generality comes at the cost of complexity. For any given application, most rules will look the same, and therefore if the rules are general, then they will be unnecessarily complex in the typical use case. Most of the time, the user will be annoyed by inputting similar rules every time, instead of making use of the generality.

Because of this problem, we have decided to create two distinct layers of access rules: a low-level layer, which is general and simple, and a domain-specific high-level layer, which is optimized for the typical usage in the given domain. The high-level layer builds on the primitives provided by the low-level layer, and different high-level rules should be developed for different use cases (such as campuses vs. hotels). This allows for flexibility and convenience at the same time.

In order to support both the typical use case and the unique snowflakes in a single installation, high-level rules implementations are expected to not assume anything about the rules installed in the system – they are not allowed to carelessly delete existing rules, or assume only rules they know about exist. They should display the low-level representation for rules they cannot interpret in their high-level model.

To facilitate this cooperation between high-level and low-level rules, and also to ensure consistency, we have come up with the notion of a *ruleset*: every rule in the system is tagged as belonging to exactly one ruleset, and the high-level layer can create, update or delete only whole rulesets, not individual rules. Operations on rulesets are atomic. An application implementing the high-level rules should operate only on rulesets created by that application. A mechanism for enforcing this restriction exists.[1]

The low-level, internal rules must be generic enough to support any use case, yet easy to compile by both computers and humans.

4.2 Rules format

In order to cover all possible use cases, the straightforward approach is to allow access rules to be specified as any Boolean formula over identities, access points and time specifications. However, this brings the following problems:

- it is hard for humans to quickly reason about the result of any given query
- complete evaluation on every query is necessary, which might be costly in memory or time; it is impractical to pre-compute anything
- for offline functionality, the evaluation logic and all data required for evaluation need to be embedded in the controllers, which violates the "keep embedded devices simple" design principle;
- a small change in input data or formulas can have arbitrarily large effects, which hinders attempts at both understanding why something happens and pre-computation.

In order to avoid these problems, we have instead chosen the following model:

Every access point is of exactly one *type*; for each type, rules that match a *time specification* and an *identity expression* to an *Allow* or *Deny* response may be added. Rules are strictly ordered by priority.

The evaluation flow, as depicted in in figure 4.1, is as follows:

1. Find this AP's type, select its rules.

[1]This is implemented in our DBMS of choice, PostgreSQL, by the row-level security mechanism [11].

2. Select rules with matching time specification.
3. Select rules with this ID in the rule's expression.
4. Select the (single) rule with the highest priority.

This selects a single rule, which unambiguously allows or denies access.

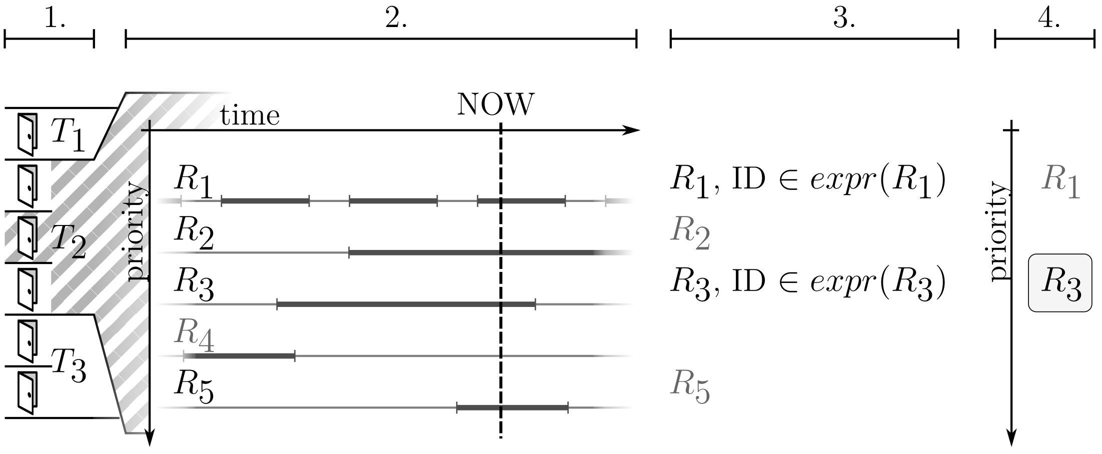

Figure 4.1: Rule evaluation flow

4.2.1 Identity expressions. An identity expression is a (restricted) Boolean formula over identities only, and it implements a generalization of access control by groups.

Implementing general Boolean formulas (e.g. using AND, OR and NOT operators) would be possible, but to support NOT we would have to either store the complement, which may require a lot of memory for a small input, or make the computation less straightforward, which clashes with keeping controllers simple. Therefore identity expressions use the INCLUDE and EXCLUDE operators, which are equivalent to set union and set difference. These are equivalent (even in expression complexity) to general Boolean formulas as long as the set of "interesting" identities is given (which it is, as "any ID whatsoever except for this one" is not a useful rule).

4.2.2 Rationale for the separation. Splitting rule evaluation into identity expressions and time+place expressions means that rules are easier to evaluate: a human (or a computer) can evaluate the two independently, and "why does this happen" questions are easier to answer. Moreover, in this way classes of equivalence on inputs are easier to find, as in this model a single time+place rule matches a single identity expression instead of arbitrary combinations. This makes it practical to pre-compute some rules, and implement re-computing this incrementally on change.

4.3 Answering access queries

Typically, rules will be queried often (especially when creating local rules databases for controllers) and changed infrequently. Therefore we can save work and time by pre-computing some information. Currently, we assume that in a typical deployment there will be few rules and multi-level identity expressions. Therefore we pre-compute an "in identity expression" relation – for every identity (i.e. for all leaves of the expressions) we climb the expression DAG and save the $(identity, expression)$ tuple when the identity is included by an expression (taking into account the INCLUDE/EXCLUDE operations). As the expressions are acyclic, whenever we need to INCLUDE/EXCLUDE a sub-expression, we can just wait until we've computed it, thereby computing everything once.

An access query selects a single rule (and a single expression) via the mechanism described in 4.2, and then we just check whether an $(expression, identity)$ tuple exists to answer the query.

When an identity expression changes, it is easy to incrementally re-compute only the affected parts: we simply search the DAG, re-computing nodes as we visit them.

See section **TODO** ?? for notes on the implementation of re-computation.

4.4 Local evaluation on embedded devices

The local database copy on the controllers builds on this two-level approach of separating identity expressions and time specifications. Note that any embedded device serves a single point of access, and therefore the "where" part of the rules is already taken care of – every accesspoint selects only rules belonging to its type.

The server provides a mechanism to re-create the local databases on rules change. The specific format of the local databases is out of scope of this thesis.

4.5 Integration with existing systems

As required by section **??**, data may be imported from other systems, and transparently "patched" into access rules. This is done via an application that generates flat identity expressions of the form $X :=$ [INCLUDE $student_1$, INCLUDE $student_2$, ...] for every group X that needs importing.

Chapter 5

Server Design

Like all other Deadlock components, the server design and implementation follows from the key design principles listed in section 2.2.

5.1 Data organization

Server replication and failover should be easy, and therefore any data/state that the server needs must be easy to replicate. Therefore the server is allowed to depend only on the contents of the relational database (which has built-in replication mechanisms) – it may cache or pre-compute some values, but otherwise all output must be a pure function of the database contents.

The following sections give an overview of the data the server works with.

5.1.1 Access rule specification and identities management. TODO this is fun

TODO in_expr

Normally, the access rules will be queried often and changed only occasionally. Therefore, it is beneficial to partly pre-compute the queries on rules change, in order to answer more efficiently. To do this,

5.1.2 Point of access management and access logs. For each access point, we store its type, optional description, and which controller is attached to it.

Access logs (with data as specified in the message, according to section 3.4.2) are written to disk on insert. In order to fulfill the protocol idempotence guarantee, only logs with a unique combination of attributes are stored.

5.2 Main components

The server functionality has been split into 3 separate components with minimalistic interfaces. These run as separate processes and it is not assumed that they run on the same machine.

5.2.1 The "common files" interface. In order to avoid tight coupling between these modules, generally the only common interface among these (apart from the shared database) is the filesystem: when configuring the deployment, a filesystem directory with read and write access is given, and the components communicate by creating and accessing files in that location. This is usually sufficient, as the components are designed to run independently and only collect whatever happens to have been created by the other components. In particular, the purpose of several components is to create files meant for transfer to controllers via the `XFER` message (see section 3.4), and for these we have created a simple common library for opening files meant for a controller (optionally with a fallback to files common to all controllers).

5.2.2 `deadserver`: communicates with controllers. Listens for controller requests on a UDP socket, and sends responses according to the protocol specified in chapter 3. For `PING` and `XFER` requests (see section 3.4), looks for files via the mechanism mentioned in section 5.2.1.

5.2.3 `deadapi`: the API for the outside world. Provides the HTTP API used by the web management and monitoring interface, and the provided commandline interface. Thereby bridges the outside world and the database via a simple CRUD REST API.

Supports pushing events via a streamed long-running HTTP response, in accordance with the Server-sent events/Eventsource specification [5]. Events are triggered via the LISTEN/NOTIFY pub/sub mechanism in Postgres, and the database in turn contains triggers that send a NOTIFY on certain table row changes. Therefore data changes can bubble all the way to clients, which can use the standard Eventsource API to subscribe to these.

Provides a quick way to stage firmware updates: a firmware image together with a list of controller IDs can be uploaded, and `deadapi` simply drops (or links) the file into subfolders dedicated to the given controllers (see section 5.2.1).

5.2.4 `deadaux`: auxiliary jobs supporting the other components. `deadaux` is a collection of auxiliary modules that support the functionality of `deadserver` and `deadapi`, plus a very simple dispatcher that runs the modules in separate threads. By default, the following modules are part of `deadaux`:

- **`offlinedb`:** The main responsibility of this module is to build the copy of the rules database that the controllers use for local offline evaluation. Its main thread uses the pub/sub mechanism in Postgres, LISTEN/NOTIFY, to be notified on rule changes. On change, it generates new versions of the files, and drops them where controllers can find them via the mechanism mentioned in section 5.2.1.
- **`echotest`:** Uses the controller client library to periodically send `ECHOTEST` messages to `deadserver` and check the response. Sends an email if any problem occurs.

Chapter 6

Server Implementation

6.1 Chosen programming language

Several alternatives were considered for the server code, notably C++ and Haskell, but in the end we chose Python 3, for the following reasons:

- **Low entry cost for new developers**: Most potential future developers are already familiar with Python: the Applied Informatics branch at our faculty teaches it in the first year, and the Informatics students are expected to know Python in various courses in the curriculum. Also, Python is quite simple, and therefore becoming proficient at it is a lot easier than with most other languages.
- **Great simplicity/awesomeness ratio**.
- **Principle of Least Astonishment**: Unlike e.g. C++, Python is simple and consistent.
- **Effective constructs that encourage good design and correct code**: Language constructs such as decorators encourage modularity and composition, and e.g. context managers help ensure resources, transactions and such are managed correctly.
- **Good libraries available**: Libraries for common tasks such as interfacing with the DB, serving UDP or HTTP requests, and much more, are readily available, well known and well tested.

- **Fast prototyping**: For the above reasons, getting something up and running is quick with Python.

The obvious, and considerable, disadvantage of Python, is the lack of static typing – without static typing, many errors which could be discovered by a compiler will only appear at runtime. In fact, the type system is the main reason for the author's ongoing desire to switch to Haskell. However, not many people know Haskell, and we want it to be easy to contribute to Deadlock, so it is a much better idea to pick a well known language.

The Python 3 language and standard libraries are documented in [7].

6.2 Targeted environment, dependencies

- The Deadlock server is meant to be run on **a Unix-like server**. While it may work on multiple platforms, it is tested only on Debian-like Linux distributions and FreeBSD 10.
- We are targeting **Python 3.4 or newer**. This is because at the time of writing Python 3.4 is available in all relevant OSs and distributions (in particular Debian Stable), and contains useful features not present in previous versions.
- For the DBMS, **PostgreSQL >= 9.3** is required. We use non-standard Postgres-specific features, such as the PL/pgSQL in-database procedural language [10] for rules pre-computation, or the NOTIFY/LISTEN pub/sub system for notifying `deadaux` of access rules changes.
- Several of the used libraries (at least `psycopg` and `pynacl`) use native bindings, and therefore only work with the CPython implementation of Python 3.

6.3 Database structure

As explained in section 5.1, the database structure is the complete information (except for caching/pre-computation) about the data Deadlock is concerned with.

Figure 6.1 shows the entity-relationship diagram for the basic database schema.

In addition to this, the `in_expr` table as described by section **TODO i**s computed from the data. This computation is implemented in-database using PL/pgSQL [10].

6.4 Interesting problems encountered

TODO put it here, or have own sections about "stuff stuff problem solution stuff stuff"?

- avoid running around with secret keys by passing just a crypto black box (TODO but do it :D) to avoid e.g. accidentally logging it
- guarantees:
  - blob version –> contents: This is implemented by using a blob's hash as the version, but this is an implementation detail not relevant for the protocol

---

References: [12]

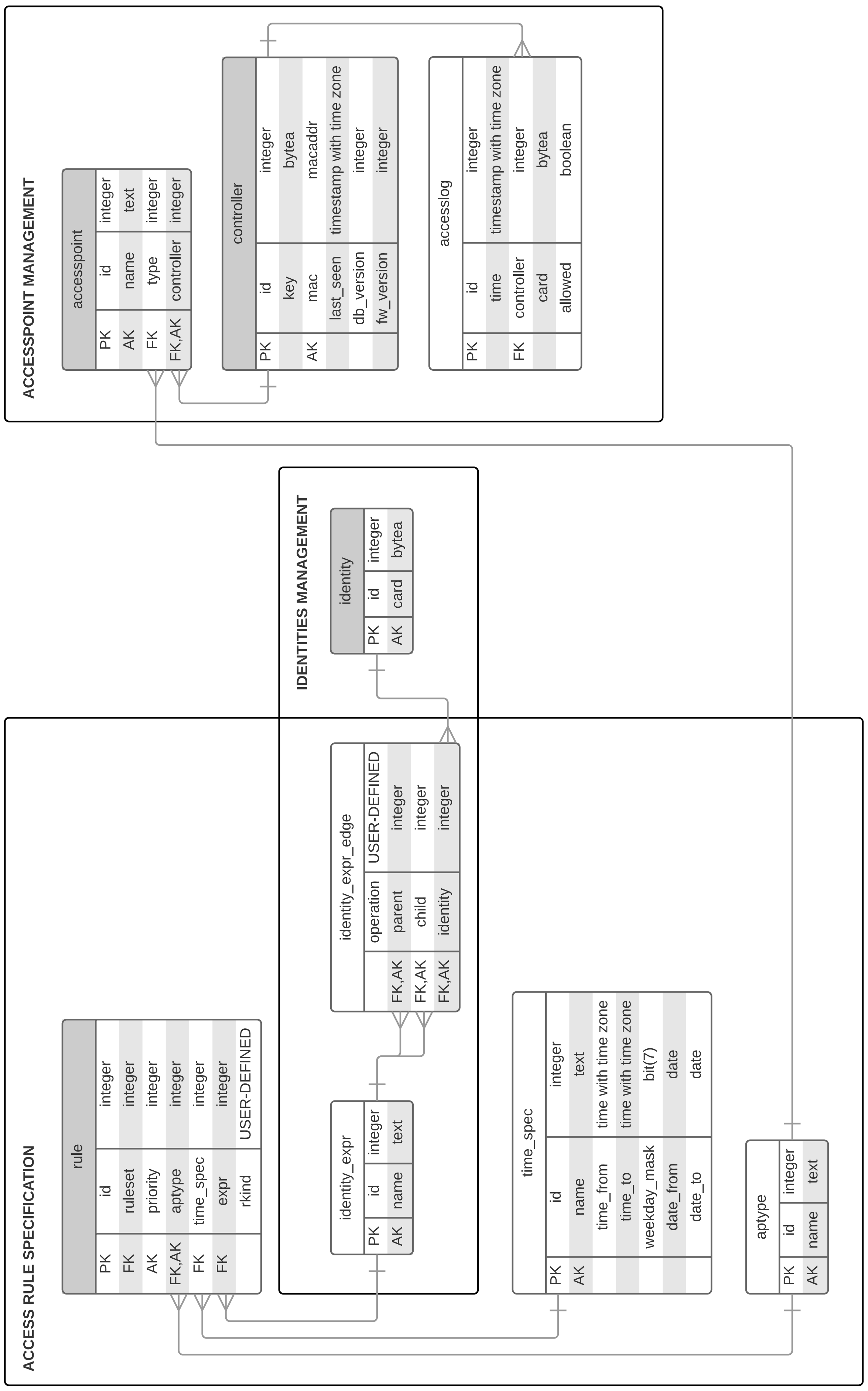

Figure 6.1: Entity-relationship diagram for the database scheme.

Chapter 7

Future plans

7.1 Real-world deployment

As we prepare Deadlock for deployment at our faculty, more issues will certainly surface. We intend to make Deadlock a reliable long-term solution for our faculty. Later we are planning to expand to larger deployments.

7.2 Testing

Due to time constraints, currently Deadlock does not have unit tests, although a simple integration test, plus the continuously running `ECHOTEST` sanity check included in `deadaux` (as described in section 5.2.4), exist. Unit tests and more comprehensive integration tests would ease development. We are planning to reach 100% unit test coverage and setup continuous integration as soon as time allows.

7.3 System status monitoring

If one wants a system to work, one needs to monitor it. In particular, metrics assessing the system health and performance should be exported; when a problem occurs, actions that can be taken automatically should be automatically taken; and actions that require human intervention should alert a human. A way to monitor the system and take appropriate actions (ideally based on an existing general solution) should be found.

Some basic watchdog functionality is present in Deadlock itself: controllers have a hardware watchdog that restarts them on lockup, and the integration test included in `deadaux` (see section 5.2.4) can alert a human if things obviously don't work. However, we intend to explore more comprehensive solutions.

7.4 High-level rules and UI optimized for usage at universities

As part of deploying at our university, a domain-specific rules model will be developed, and the corresponding rules management interface will be created.

7.5 Tools and libraries

Tools and libraries that further ease deployment and integration should be provided. In particular, a tool for importing data from often used systems, such as directory databases using the LDAP protocol or SQL databases, will be made available prior to the faculty-wide deployment estimated for autumn 2016.

7.6 A server implementation in Haskell

The current server implementation (in Python) is production-ready. However, the lack of compile-time type checking is a considerable weakness. The type system in Haskell is very strong, and therefore it can find bugs which would normally not be found at compile time. We believe that a Haskell implementation would be far more trustworthy, and intend to write one.

Conclusion

TODO Deadlock is awesome.

Glossary

`deadapi` the HTTP API for Deadlock management, rules configuration and status monitoring

`deadaux` auxiliary jobs supporting tasks such as offline database creation

`deadserver` the Deadlock server that communicates with controllers

PoA, point of access a door lock, a printer or any other device, access to which is controlled by a Deadlock controller

ruleset A tagged set of access rules. Every access rule in the system belongs to exactly one ruleset. Creating, updating or deleting a ruleset is an atomic operation.

Appendix: Source code and documentation

Project Deadlock is and will for some time remain under development. The newest source code, hardware schematics and documentation for all components is available at https://github.com/fmfi-svt-deadlock/.

Due to time constraints, the attached source code does not yet implement everything as described in the thesis. The notable differences are:

TODO sections

- identity expressions precomputation is not incremental (**TODO section**)
- the `CRITICAL` message handler is not implemented yet
- command-line interface directly queries and modifies the database instead of contacting the HTTP API

Attached: Server source code, as of May 16, 2016.[1]

The newest version can be found at https://github.com/fmfi-svt-deadlock/server.

[1]It is strongly recommended to look at and use the newer online version rather than the attached version of the code.

References

[1]Bernstein, D.J. 2009. Cryptography in NaCl. *https://cr.yp.to/highspeed/naclcrypto-20090310.pdf*. (2009).

[2]Bernstein, D.J., Lange, T. and Schwabe, P. 2013. NaCl reference.

[3]Bernstein, D.J., Lange, T. and Schwabe, P. 2012. The security impact of a new cryptographic library. *Progress in cryptology–LATINCRYPT 2012*. Springer. 159–176.

[4]Bormann, C. and Hoffman, P. 2013. Concise binary object representation (cBOR). (2013).

[5]Hickson, I. 2009. Server-sent events. *W3C Working Draft WD-eventsource-20091222, latest version available at< http://www. w3. org/TR/eventsource*. (2009).

[6]Postel, J. 1980. RFC 768: User datagram protocol. *https://tools.ietf.org/html/rfc768*. (1980).

[7]Python Software Foundation 2015. Python 3 documentation.

[8]Saltzer, J.H. and Kaashoek, M.F. 2009. *Principles of computer system design: an introduction*. Morgan Kaufmann.

[9]ŠVT, F. 2016. Deadlock repositories.

[10]The PostgreSQL Global Development Group 2016. PL/pgSQL reference.

[11]The PostgreSQL Global Development Group 2016. PostgreSQL row security policies – documentation.

[12]Ullman, J.D. 1984. *Principles of database systems.* Galgotia publications.