Małgorzata Wielgus*, Przemysław Dąbek**, Roman Janusz***, Tomasz Kowal****, Wojciech Turek*****

PLATFORMA DO AKTUALIZACJI OPROGRAMOWANIA NA URZĄDZENIACH MOBILNYCH OPARTA NA TECHNOLOGII ERLANG

To pisze się na samym końcu

Słowa kluczowe: *Erlang, aktualizacja oprogramowania, urządzenia przenośne*

ERLANG-BASED SOFTWARE UPDATE PLATFORM FOR MOBILE DEVICES

To be written at the very end

Keywords: *Erlang, software updates, mobile devices*

1. Introduction

Fast development of mobile devices creates new domain of applications for large systems composed of many devices communicating over the internet. Those devices can be far from each other, what makes direct access difficult. Our system was designed to simplify software updates on groups of such devices.

There are many areas of applications including:

- Monitoring - with our system we can perform software updates on cameras in the building. Especially if they are equiped with software for face recognition.
- Management of geographically spread devices.
- Security - updates on sensors used in cars or alarms in homes.
- Robotics - autonomic robots used in factories and in industry.

*AGH University of Science and Technology, Kraków, Poland, malgorza@student.agh.edu.pl
**AGH University of Science and Technology, Kraków, Poland, przemyslaw.dabek@gmail.com
***AGH University of Science and Technology, Kraków, Poland, roman@student.agh.edu.pl
****AGH University of Science and Technology, Kraków, Poland, tomekowal@gmail.com
*****AGH University of Science and Technology, Kraków, Poland, wojciech.turek@agh.edu.pl

The need: management and software updates, use cases General usecase of our system is to perform software updates on multiple devices over unreliable network. There is a need of such updates because of growing amount of remote devices such as sensors, mobile phones or network infrastructure devices.

Existing solutions include Mobile Devices Management systems created by Nokia, Motorola, Oracle and by Open Mobile Alliance.

"The tool described in this paper is a response for all the needs world has... "

2. Requirements and assumptions

- Network infrastructure assumptions
  - Possibility of NAT
  - Slow and unreliable connections
- Need for scalability (up to about 10000 devices)
- Simple, intuitive interface
- Managing groups of devices, batch updates

3. Erlang Technology Overview

3.1. Usability of Erlang Technology in distributed systems

Erlang is a technology and a programming language that mixes functional programming with an approach to easily build heavily parallel and distributed, highly available systems. It achieves these goals using a set of unique features, including:

- Virtual machine implementing message-passing concurrency model with lightweight Erlang processes
- Built-in, language-integrated engine for communication in a distributed environment
- Hot code swapping with a fine control over the software upgrade process. The aim of these is to allow an upgrade to be performed automatically without stopping any services.
- Fault-tolerance features like supervisors.
- Takeover and failover mechanisms for cluster systems.

3.2. General description of Erlang OTP

Erlang OTP (Open Telecom Platform) is an Erlang distribution released by Ericsson in 1998 when the language became open source. OTP is a set of standard erlang libraries and corresponding, well-defined design principles for Erlang developers. OTP defines patterns for basic elements that make up the software as well as the general layout of completed, deployed environment.

OTP principles include:

- Supervisors and supervision trees
  Erlang software can be thought of as a set of lightweight erlang processes communicating with each other. In supervision tree principle, these processes form a tree where the leaves are called workers and are doing the actual job, while other nodes are called supervisors. Each supervisor is responsible for monitoring its children and reacting accordingly when any of them crashes. Supervisors allow to design well-structured and fault-tolerant software.
- Behaviours
  Behaviours are a set of basic design patterns used to build common types of software pieces. Fundamental behaviours are:
  - `gen_server` for implementing simple servers and client-server relation between erlang processes
  - `gen_fsm` for implementing generic finite state machines
  - `gen_event` for implementing event handling subsystems
- Applications and releases
  These patterns define general layout of a self-contained, deployable piece of Erlang software. Applications and releases will be described in the next section as the Software Update Platform deals heavily with them.

3.3. Applications and releases in Erlang

An example of deployable package of software written in Erlang is so-called embedded node. An embedded node is a self-contained, configured Erlang environment along with actual software written in Erlang that can be deployed and run using a simple command. An embedded node contains:

- Erlang Runtime System
- A set of Erlang applications, the actual code
- Configuration for ERTS and applications
- An Erlang release

3.3.1. Erlang applications

An erlang application is an independent piece of software that serves some particular functionality. An application is defined by its name, version, code (set of modules), dependencies (other applications) and other more finegrained settings and attributes. These are all configured in an `.app` file. Every application defines a way of starting it, stopping it and possibly upgrading or downgrading it to another version (optional `appup` file). It also has its own piece of configuration. A running application is often made up of a single supervision tree.

3.3.2. Erlang release

An erlang release is a configuration of what an embedded node contains and how it is started, stopped and upgraded. Thus, an erlang release, defined by an `.rel` file, states which version of ERTS should be used in the node, lists a set of applications in particular versions that should be part of the release, and defines one or more way the Erlang node is started and stopped. Separate, optional `relup` file defines how the release is upgraded or downgraded (without stopping the node). Erlang release has its own version number.

3.4. Upgrading Erlang software

As a part of focus for high available systems, Erlang supports hot code swapping and very finegrained control over the upgrade process without stopping running node. Every application may define how its version should be changed to higher or lower in the `appup` file. Based on a set of `appup` files, a `relup` file may be generated which merges all operations listed in `appup` files into one big script that upgrades or downgrades the whole release. This script may be executed using standard erlang API for release handling (the `release_handler` module).

Because each application is responsible for defining how its version should be changed, the upgrade process is very straightforward and requires only a few calls to the release handling functions. Thus, it can be easily performed without human interaction, by automatic tools.

Still, there are some problems with that method. Mainly, it is low-level as it requires calling the erlang API on the target node. What is needed and what our platform aims to provide is a way of easy installation and deinstallation of the erlang node on the target system using simple tools like package managers and also a way of easy management of a large number of devices.

4. Packaging of erlang software

4.1. What it is about and motivation for it

By ‘packaging of erlang software‘, we mean a way of putting the contents of erlang node into packages like `.deb`. These would be easy to install, upgrade and remove with standard package manager commands. We also want to maintain ability to perform upgrades without stopping the node (hot code swapping).

There are a few reasons why we decided to implement this:

- Easy installation and deinstallation of erlang node on the target system using a single command.
- Reduction of amount of data downloaded during the upgrade (only the actually changed packages are fetched by the package manager).
- Overall better integration with the target system.

4.2. How it was implemented

We decided to create tools to build debian packages `.deb` that would span the contents of the erlang node and create a set of packages ready to be pushed to a repository and easily installed on the target device. This also required general implementation of debian maintainer scripts included into these packages. These scripts are responsible for management of the node during installation. Most importantly, they contain the code that performs the upgrade process.

4.2.1. Decomposition of the erlang node into debian package set

The erlang node is decomposed into a set of debian packages in the following way:

- Base package contains ERTS and its version is equal to the ERTS version. This package is architecture-dependent. It has no dependencies.
- Each erlang application gets its own package. Its version is equal to the application version. Package dependencies reflect the list of dependent applications in the `.app` file.
- Erlang release files are packaged into the final, main package. This package is dependent on the base package and packages for all applications contained in the release. These dependencies are strict in terms of version of dependent packages - the release requires a concrete version of every application as well as the ERTS.

The only package that should be explicitly maintained by system administrator is the main package containing the release files. This package, thanks to its well-defined dependencies, represents the whole erlang node and its installation will cause the whole node to be deployed on the device.

The point of splitting up the node into several packages related through dependencies was to reduce the amount of data downloaded during the upgrade process. It is a very common situation when we want to upgrade e.g. only one erlang application. Standard erlang features do not allow us to independently upgrade each application - we must always create a new version of the whole release and upgrade the release itself. Without proper decomposition of the contents of the release, this would force us to download a big package containing every application, even though most of them did not change.

Usage of package manager solves this problem. When the node is decomposed into several packages, upgrade of the main package forces the package manager to download only these application packages whose version changed. This is all thanks to automatic dependency resolution provided by the package manager.

4.2.2. Upgrade of the release - maintainer scripts' job

There are a few maintainer scripts in all the packages spanning the erlang node. They are responsible for starting the node when it gets installed, stopping it when it gets removed and, most importantly, performing the release upgrade process when the main package is being upgraded by the package manager.

The script performing the release upgrade checks whether the node is running, using tools available in the node `bin` directory. When the node is running, the script remotely calls appropriate erlang code on the running node, causing it to switch to the higher version of the release (standard erlang release upgrade using the `release_handler`, without stopping the node). Hot upgrade may fail or the node may have been down from the beginning. If so, a manual replacement of the old version of the release with the new version is performed (manual replacement of some files) and the node is started.

4.3. Results and effects

4.4. Problems and limitations

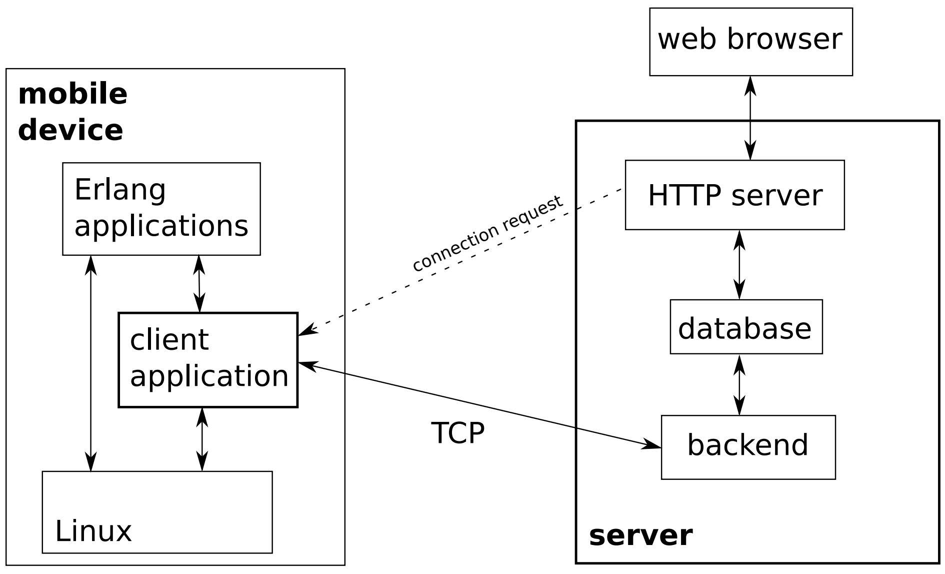
5. Software Update Platform

Software Update Platform is responsible for performing updates on multiple devices and monitoring installed applications. Developers prepare *.deb* packages with applications and add them to repository. When connection with device is established and information about update is written in database, platform will perform software update on that device.

5.1. Architecture

Software Update Platform consists of two main parts:

- client application installed on mobile device,
- server.



Client application

We assume that client application is running on Erlang VM installed on some Linux distribution and *apt* is available. Client application connects with server and perform given operations.

Server

Server consists of 3 components:

- HTTP server
- database
- backend

HTTP server

Mochiweb – our web server of choice is responsible for following tasks. It manages user interaction through web interface (serves content, performs user requests) and it is repository for *.deb* packages.

Database

Mnesia database stores information about:

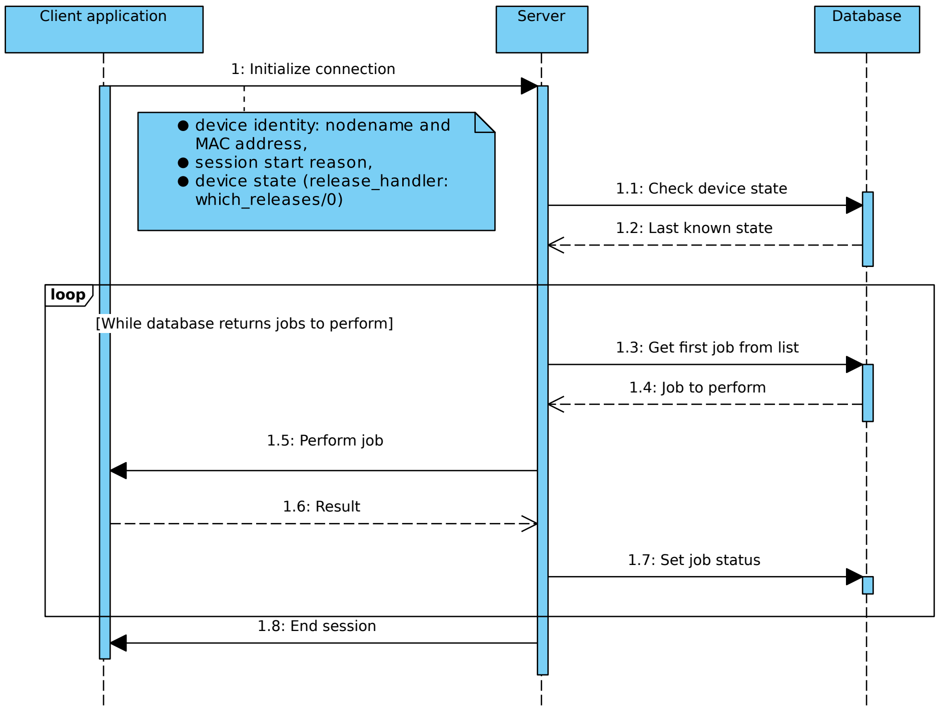
- device and installed applications on it,
- jobs (e.g. update) for device.

Database is connector beetwen web interface and backend. It stores user requests – jobs to perform on device.

Backend

Backend is the core of platform. It is Erlang application responsible for whole automatic device management. Every new device in platform connected with backend is stored in database. From now on backend can monitor state of that device and performs operations on it.

5.1.1. Device-server session



5.1.2. On-device upgrade logic

5.2. Functionality

- General description of the application: Web interface, technology
- Development model for application, debian packaging utilities
- Direct usage of package manager on remote devices

gui: 1 or 2 images

6. Conclusions and Further Work

- Support for other package managers
- Communication security
- Development towards more general management platform
  - monitoring
  - configuration
  - maintenance
  - diagnostics

7. Acknowledgements

To be added at the very end

Literatura

[1] Palermo D. S., Jenkins J. J.: *Word Association Nouns: Grade School trough College*. 1st ed., Univeristy of Minnesota Press, 1964