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

ERLANG-BASED SOFTWARE UPDATE PLATFORM FOR MOBILE DEVICES

To be written at the very end

Keywords: *Erlang, software updates, mobile devices*

PLATFORMA DO AKTUALIZACJI OPROGRAMOWANIA NA URZĄDZENIACH MOBILNYCH OPARTA NA TECHNOLOGII ERLANG

To pisze się na samym końcu

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

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. There are systems designed to simplify software updates on groups of such devices.

Areas of applications include:

- 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 including configuration management software update, position tracing, error detection. It can be applied to mobile phones and industrial robotics.
- 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

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. All of these devices need to run without breaks. That is where Erlang is useful with its updating mechanisms. We can perform software updates without interrupting running system and without the need to restart it.

Existing solutions include Mobile Devices Management systems created by Nokia [1], Motorola [2], Oracle [3] and by Open Mobile Alliance [4].

2. Requirements and assumptions

During system design we made following assumptions:

- Managing groups of devices, batch updates – user can create groups of devices so that our system can be used to manage more than one distributed system at once. User can send requests to those groups. He alse has fine-grained control over particular device.
- Use of Erlang programming language – features of Erlang programming language provide tools simplifying implementation of distributed, fault-tolerant systems.
- Possibility of NAT – usually mobile devices we want to update are not in the local network. This implies that those devices are behind a NAT. It makes ssh connection difficult. Devices should connect to central server with public IP.
- Slow and unreliable connections – while developing we thought mainly about devices that are connected via GSM. This connection can be slow and may be interrupted easily.
- Need for scalability – distributed systems has to be prepared for heavy load because they consist of large amount of devices. Our system should be able to manage up to about 10000 devices.
- Simple, intuitive interface – we wanted our system to be user-friendly.

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. The most important one is supervisor behaviour for writing special control processes responsible for monitoring other processes and reacting accordingly when they crash. This allows programmers to take an *happy case programming* approach which means that they can ignore any exceptions as long as they don't have to be handled explicitly in some special way.
- 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.

Erlang is a technology that from the very beginning incorporates patterns, conceptions and approaches that are crucial for heavily parallel and distributed systems. Since such systems are becoming ubiquitous these days, Erlang definitely has the chance to become a technology of the future for large, distributed computer systems.

In this article, we will briefly describle basic OTP principles with more attention on the ones important for the Software Update Platform. You can read more on OTP principles in the official documentation in [5].

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 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:

- ERTS (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. The idea and motivation

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. Implementation details

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 (see [6]) 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

We were able to implement all crucial features regarding integration of package manager with Software Update Platform.

From the point of view of an erlang developer maintaining the software being upgraded, the main feature implemented are a few helpful scripts. They allow easy generation of debian packages from an Erlang release generated with `rebar` (you can read more about development model in 5.2). There is also a script for easy generation of the `relup` file.

From the point of view of a target system administrator, the main advantage of package manager is mentioned earlier easy way to install, upgrade and remove the Erlang node. The only configuration needed at the target system is addition of debian repository in the `apt` configuration file.

From the point of view of an user of the Software Update Platform, main feature implemented is a built-in debian packages repository along with an UI to manage its contents.

We were also successful in obtaining desired non-functional features of the system regarding its use of package manager. This includes smaller downloads from the server to the target devices and ability to use hot-code swapping by the package manager during upgrade process.

4.4. Problems and limitations

Although the integration of package was successful, due to a significant mismatch between how package manager works and how standard Erlang upgrade tools work, some workarounds were required.

The main problem that we encountered was an issue that we had to resolve when trying to preserve ability to perform hot-upgrade without stopping the system. Erlang upgrade tools require that at the point of a release upgrade, the whole old version of the release and the whole new version of the release are present in the filesystem. When the release is split into several packages, it is impossible to find a moment where this requirement is met. It is also impossible to influence the way package manager works to force it to behave as we wanted.

Because of that, we had to introduce a workaround. Erlang application and Erlang release packages do not contain their contents directly. Instead, an intermediate `tar.gz` archive is created that contains actual contents of the package. This archive is then put into the `.deb` file directly. Like this, we have full control over when the files from the intermediate package are unpacked and removed. The intermendiate archive is unpacked from the `.deb` file by the package manager directly, while the actual files are unpacked by the maintainer scripts.

One downside of this approach is that we also had to manually remove application and release files using maintainer scripts. Another one is that when an application is removed from the release, we have to manually remove the package containing that application. This cannot be done from the maintainer scripts as it is probably a bad idea to invoke package manager from scripts invoked by the package manager.

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.

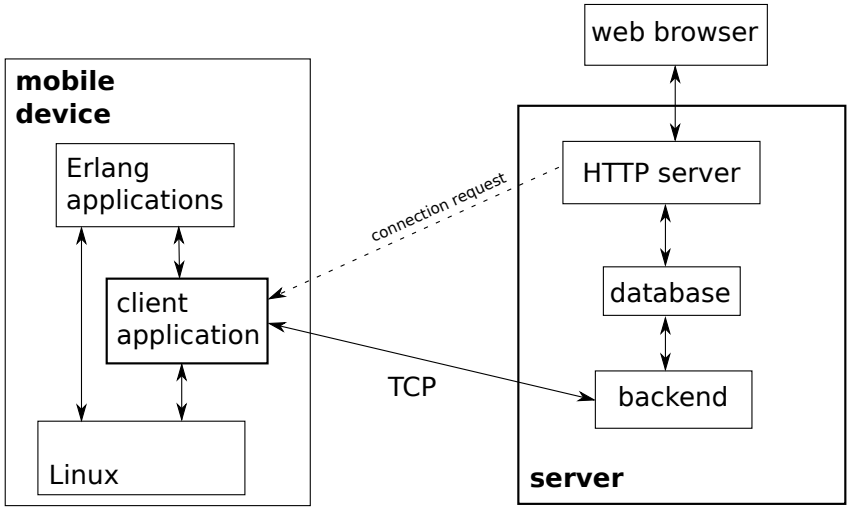

Fig. 1. Architecture overview

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

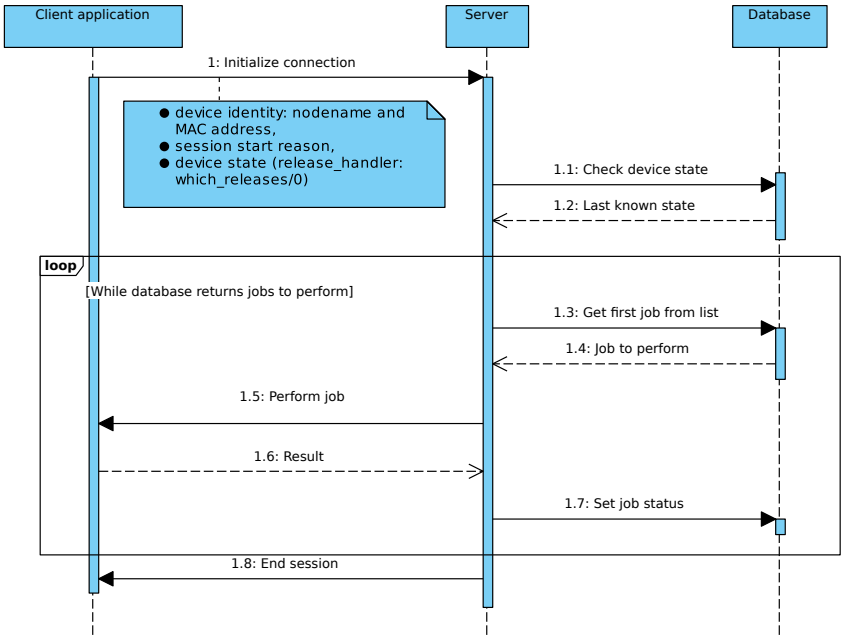

Fig. 2. Session communication diagram

Firstly device initializes connection which carries following data:

- device identity: nodename and MAC address,
- session start reason,
- device states.

These pieces of information are used to unambiguously identify the device. After connection is established, server checks device state in database. If there are any enqueued jobs, they are sent to device. The device returns result of performed job. This status is written to database.

5.1.2. On-device upgrade logic

Updating software consists of two jobs:

- upgrade
- check release

Upgrade immediately returns with status `ok`. It stops periodic connection requests, closes session and uses *apt* to perform upgrade. It waits for *apt* to finish and checks its result. Then it restores periodic connection requests.

Check release reads value returned from *apt* and notifies server about changes in next session.

5.2. Functionality

- General description of the Web interface
  Web interface provides simple way to manage groups of devices. User can:
  - check last known state of device: IP, version of release, installed application
  - assign device to one or more categories
  - schedule jobs for device

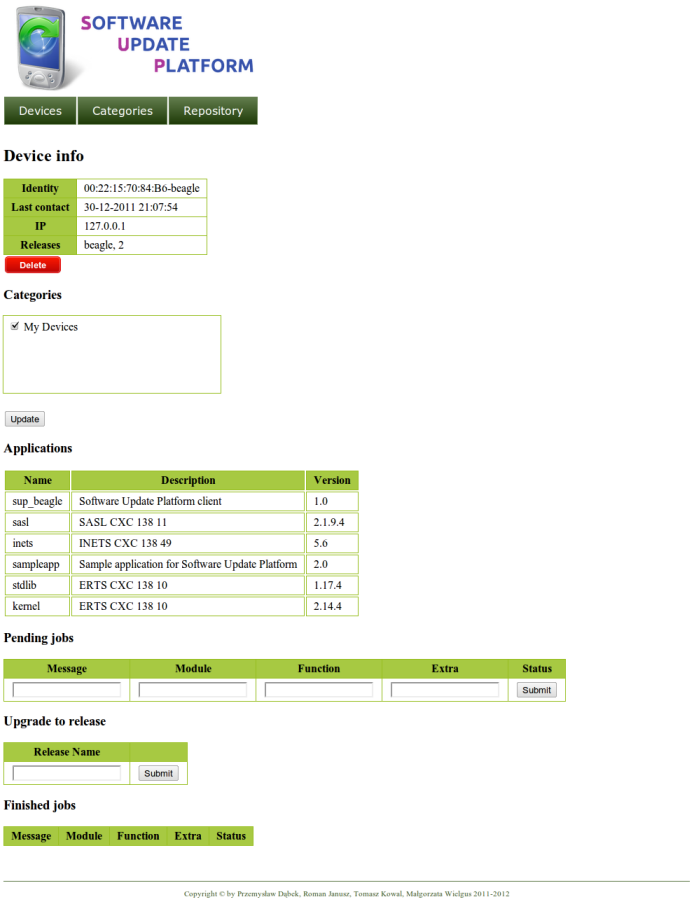

Fig. 3. Device view

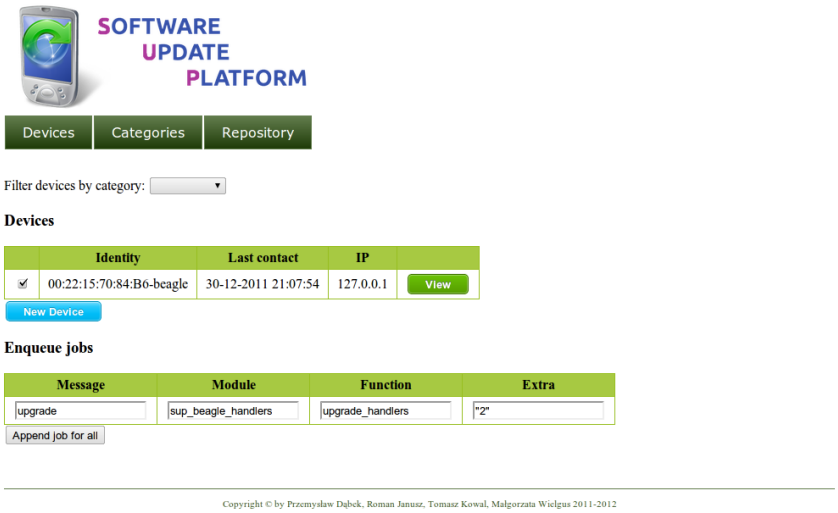

Fig. 4. Device view

There is also special page for repository where user can upload previously prepared packages.

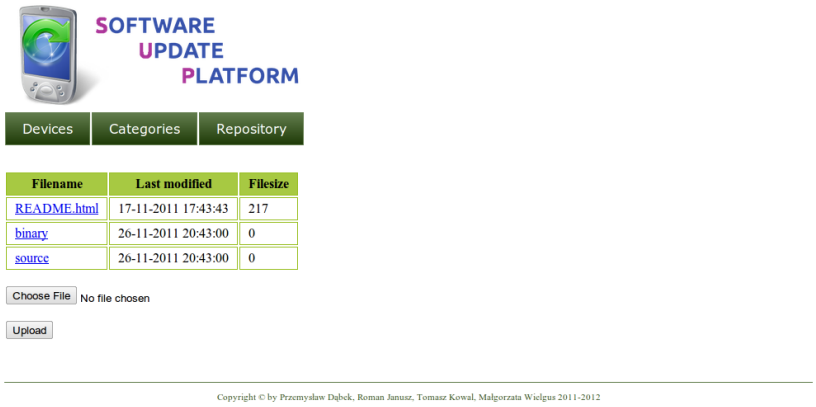

Fig. 5. Repository view

- Development model for application, debian packaging utilities
  Software Update Platform proposes a model and provides some tools for development of Erlang software installed on the target systems. The main assumption is that the developer uses a tool called `rebar` for managing Erlang software (see [7] for more information). `rebar` is a script providing functionality for Erlang

similar to what `maven` provides for Java development. This includes automatic generation of stubs for Erlang applications, build, `.appup` file generation, creation of complete, self-contained Erlang nodes (with a runtime), documentation etc. Generated Erlang nodes are used as a basis for `.deb` packages generation. This is implemented by a set of scripts provided by the Software Update Platform itself.

6. Conclusions and Further Work

We managed to solve most of occuring problems. Our system is fully functional. Even though *apt* does not provide simple solution to hot upgrades, we implemented it. This gave us some advantages over standard tools like *rebar*. We optimized amount of data to download. Using package manager to install and upgrade Erlang applications is easier for people who had never used Erlang before.

- Support for other package managers
  One obvious improvement for our platform would be support for more package managers, as debian package format `.deb` was chosen arbitrarily for experimental development. Since its integration was successful, support for `rpm`-based package managers (like `yum`), `pacman` or `port` is planned in the future.
- Communication security
  Right now, communication between devices and the server is not encrypted. Wrapping device-server sessions in TLS and adding some authentication is another important potential feature for our platform.
- Development towards more general management platform
  The protocol used in communication between devices and the server is very general and extensible. New types of jobs can be easily added. Thus, the platform can be turned into more general management platform, providing a lot more of different functionality:
  - monitoring of the devices, their status, statistics, etc.
  - configuration of software installed on the device
  - maintenance features (viewing logs, etc.)
  - peforming diagnostics, troubleshooting, tests etc.

References

[1] *Nokia Device management* http://europe.nokia.com/find-products/nokia-for-business/device-management

[2] *Motorola Mobility Services Platform (MSP)* http://www.motorola.com/Business/US-EN/Business+Product+and+Services/Software+and+Applications/Mobility+Software/Mobile+Device+Management+Software/Mobility+Services+Platform_US-EN

[3] *Oracle Database Mobile Server 11g* http://www.oracle.com/technetwork/database/database-mobile-server/overview/index.html

[4] *Open Mobile Alliance* http://www.openmobilealliance.org/

[5] *OTP Design Principles* http://www.erlang.org/doc/design_principles/users_guide.html
[6] *Debian Policy Manual - Package maintainer scripts and installation procedure* http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html
[7] *Rebar: Erlang Build Tool* https://github.com/basho/rebar/wiki