System Description

Al Hinai, Abdullah Andersson, Conny Forouzani, Sepehr
Halteh, Faris Ivanou, Aliaksandr Karkanis, Iosif
Klingsbo, Lukas Lan, Fangming Lång, Magnus
McCain, Daniel Sean Noorani Subramanian, Varun
Omer, Enghin Santos Rivera, Juan De Dios

January 16, 2015

Abstract

This document contains an overview of the different services that compose the entirety of the project. The router part describes how the Ericsson Erlang NetInf Router (EENR) works and how the modules within it interact. There is also a Name Resolution Service (NRS) section of the document that covers the centralized database and its structure. Another part describes WallE, the cleaning robot with the purpose of removing faulty streams from the application. The Android service section resides further down in the document, and contains information about the android-service library, which runs in the background of the Android OS. Lastly, the Android application part describes the structure of the application that does streaming and playback.

Contents

Chapter 1

Router

1.1 Overview

The Ericsson Erlang NetInf Router (EENR) is an implementation of a NetInf router originally implemented by the Project CS 2012 group and extended by Marcus Ihlar in his master thesis. This implementation has been updated with several features and fixes. A few examples of this include a stand-alone NRS featuring document-database-like searches in metadata, a statistics collection platform, and a new static routing model storing data using Mnesia, which is a distributed, soft real-time database management system written in the Erlang programming language.

1.2 Module overview

1.2.1 eenr_database_mnesia

The NetInf routers features an implementation of the named data object (NDO) database (NRS) using Mnesia, which supports all functionality defined in *eenr_storage.erl*. The routers have their own persistent database to make sure that ICN functionality is maintained. The router database contains two tables: db and search. The db record contains a key/value pair where the key is the NetInf URI and the value is the record *ndo_info*, and the search record contains another key/value pair tuple where the key is the keyword used to find the URI and value is the set of NetInf URIs matching that keyword. It is currently not able to perform the metadata search functionality that the central NRS has.

1.2.2 eenr_nrs_client

This module contains a client library for the centralized NRS API. It implements API functions such as *get_locator*, which specifies where the object might be found, *exists* which checks whether an NDO with a URI is known by the NRS, two *search* functions to search using keywords and metadata, and *publish* to publish. It also contains some internal functions that are mostly for parsing the JSON using a library named *jiffy*.

1.2.3 eenr_signature_validation.erl

This module validates signatures attached to notification messages.

1.2.4 eenr_stat_collector.erl

This module handles the top level functionalities of the statistics collection platform. It contains several functions such as *update_stats* which adds a request to the statistics, *poll_stats* whose purpose is to collect and reset the statistics, and *avg_latencies* to compute the average latencies.

1.2.5 eenr_stat_handler.erl

The module features a cowboy handler module and serves statistics as JSON by long-polling *eenr_stat_server*.

1.2.6 eenr_stat_server.erl

This module contains functions that collect statistics from several resources. The statistics will be distributed among the consumers periodically.

1.2.7 eenr_stat_writer.erl

This module contains functions which are responsible for maintaining CSV files for statistics and writes samples pushed from *eenr_stat_server* to those files. It is responsible for starting and initializing the statistics collection server and of writing the statistics to file.

1.2.8 eenr_storage_sup.erl

This module contains the code of Supervisor for *eenr_storage* workers. Supervisors are responsible of organizing which process must stay alive for an application to function as intended.

1.2.9 eenr_udp_utils.erl

This module contains essentials for testing "get request" and "get response" when the application is using UDP.

1.3 Testing

The router has two common_test suites. One is called integration_SUITE and is a fairly complete test of the HTTP convergence layer. The other is called udp_SUITE and only tests GET requests over UDP.

The tests are executed with the `tests` make target. The makefile automatically instruments for coverage analysis.

Chapter 2

NRS

2.1 Overview

The NRS is a centralized NetInf database which the EENR router uses. It also allows queries in the metadata of NDOs, similar to a document-oriented database. However, it is currently implemented using Mnesia rather than a document database.

2.2 Module overview

The NRS is organized into the following modules

- nrs
- nrs_api
- nrs_api_client
- nrs_app
- nrs_db
- nrs_sup

2.2.1 nrs

The nrs module is the main module of the NRS application. It provides the start up for the NRS, including all its dependencies.

2.2.2 nrs_api

A *ranch_protocol* callback that implements the network API of the NRS.

2.2.3 nrs_api_client

A simple client for the NRS network API, used by the test suite.

2.2.4 nrs_app

The application module of the NRS.

2.2.5 nrs_sup

Main supervisor of the NRS.

2.2.6 nrs_db

Implements the NRS database on top of Mnesia. The main reason for choosing Mnesia was because of its persistent storage capacity, allowing the datasystem to survive even if the nodes crash.

2.3 Testing

The nrs has two common_test suites. One is called eunit_SUITE and bootstraps a single trivial EUnit test in the *nrs_db_tests* file. The other is called proper_SUITE and uses PropER[4], the property testing tool to compare the behaviour of the *nrs_db* module against a state machine model of its behaviour, defined in the *nrs_proper_statem* module.

The tests are executed with the `tests` make target. The makefile automatically instruments for coverage analysis.

Chapter 3

WallE

3.1 Overview

WallE is the cleaning robot of the Falun 2015 Android streaming solution. Its purpose is to tag streams that have been interrupted by a network outage or application crash, so they will no longer appear as live. It also deletes recorded streams that are shorter than a set length (currently 2 seconds by default), stopping streams from showing up in the history if a user starts and quickly stops them.

3.2 Module overview

WallE is organized into the following modules

- walle
- walle_app
- walle_length_guard
- walle_mdns
- walle_monitor
- walle_nrs_client
- walle_services
- walle_subscriber
- walle_subscriber_sup
- walle_sup

3.2.1 walle

This module is the main module of the WallE application. It provides the start of WallE, including all of its dependencies.

3.2.2 walle_app

The application callback of the WallE application, which provides the interface so that OTP can start it.

3.2.3 walle_length_guard

This module is a gen_server that periodically queries the NRS for finished streams and deletes those that are too short.

3.2.4 walle_mdns

A simple mDNS client library for doing service queries.

3.2.5 walle_monitor

Periodically searches the NRS for new streams and starts walle_subscriber processes for those streams. It maintains a table of all monitored streams so that no more than one walle_subscriber is started per stream.

3.2.6 walle_nrs_client

A client library for the centralized NRS.

3.2.7 walle_services

Caches mDNS service discovery results and is the frontend for all service discovery functionality. In particular, it serves the UDP endpoint used by walle_subscriber. When WallE is configured with a static UDP endpoint, it will return that instead of performing mDNS.

3.2.8 walle_subscriber

Monitors a single stream by subscribing to it. When it stops receiving notifications, it checks if it was tagged as dead in the NRS. If it was not, it either tags it as dead or deletes it, depending on its length.

3.2.9 walle_subscriber_sup

Supervisor for the walle_subscriber processes.

3.2.10 walle_sup

Supervisor for the walle app.

Chapter 4

Android service

4.1 Overview

The library android-service is a bound Android service that runs in the background of the Android OS. This section explains each package and the class files in them, as well as their responsibilities in the system.

4.2 Package overview

The android-service is organized in 18 packages:

- edu.projectcs.netinf
- edu.projectcs.netinf.messages
- edu.projectcs.netinf.ndo
- edu.projectcs.netinf.node
- edu.projectcs.netinf.node.api
- edu.projectcs.netinf.node.get
- edu.projectcs.netinf.node.notify
- edu.projectcs.netinf.node.publish
- edu.projectcs.netinf.node.search
- edu.projectcs.netinf.node.services.database
- edu.projectcs.netinf.node.services.http
- edu.projectcs.netinf.node.services.udp

- edu.projectcs.netinf.node.sub
- edu.projectcs.netinf.service
- edu.projectcs.netinf.service.queue
- edu.projectcs.netinf.service.queue.impl
- edu.projectcs.netinf.service.sender
- edu.projectcs.netinf.service.sender.impl

4.2.1 edu.projectcs.netinf

This package contains files which provide common functionality. *NetInfException.java* is an Exception class used to signal generic NetInf related errors. *NetInfUtils.java* contains utility methods for working with NetInf. *SignatureUtils.java* contains utility methods for signing NetInf messages.

4.2.2 edu.projectcs.netinf.messages

This package describes the five messages of the NetInf protocol and their corresponding responses. These messages are: notify, get, publish, search and subscribe. Each one of the messages has a corresponding class representing its response, e.g. *GetResponse.java*, which is the result of a *Get*, from *Get.java*. These responses are extended from an abstract class named *Response*, which is extended from *Message.java*. *Response.java* is a small class that returns the status and type of the response. *Message.java* is also another abstract class that returns the message ID, type and NDO (if it has one). Both the request and response classes are immutable and are constructed with the builder pattern. Requests are executed by calling Request.submit(). However, the local NetInf node must have been started before doing so.

4.2.3 edu.projectcs.netinf.ndo

NetInf is an architecture that relies on the forwarding of requests for NDOs. This package is related to these NDOs, which are defined in the *Ndo* class, along with their functionalities. This package also has a *Metadata* class, an immutable wrapper around the Android class *JSONObject*, representing the information that belongs to an NDO, and a *Locator* class, which specifies where the object might be found.

4.2.4 edu.projectcs.netinf.node

This package, including its sub packages, define the logic implementing a NetInf node. This package contains two classes. *Node.java* is a singleton class that represents a NetInf node and is used to stop and start it. *Settings.java* provides utility methods for accessing the settings for the node.

4.2.5 edu.projectcs.netinf.node.api

There is only one file, *Api.java*, residing inside this package. It is the base class for convergence layers, whose responsibility is to accept requests, transform them into the internal representation, pass them to the node, and finally send back the response.

4.2.6 edu.projectcs.netinf.node.get

This package contains files that define how to interact with GET and GET-RESP messages. The file *GetController.java* controls the execution of GET messages. *GetService.java* is an interface to be implemented by convergence layers that represents the output of NetInf GET messages from the *Node*. Its responsibility is to send GET messages over some NetInf convergence layer, receive the response, transform it into the internal *GetResponse* representation, and return it to the Node. *InProgressTracker.java* keeps track of requests that are in progress and their associated asynchronous results. The file *RequestAggregator.java* keeps track of aggregated GET messages.

4.2.7 edu.projectcs.netinf.node.notify

This package has a similar functionality as edu.projectcs.netinf.node.get, except this one works with NOTIFY messages. *NotifyService.java* is an interface to be implemented by convergence layers to send NOTIFY messages, transforms them into *NotifyResponse* and return them to a node. *NotifyController.java* implements the common logic to handle the notify messages, both received by a convergence layer and locally submitted. *NotifyCallback.java* is a callback that is invoked after a *NotifyResponse* is received.

4.2.8 edu.projectcs.netinf.node.publish

This package contains files that define how to interact with PUBLISH and PUBLISH-RESP messages. The interface provided by *PublishService.java* is an interface to be implemented by convergence layers that represents an output of NetInf PUBLISH messages from the node. Its responsibility is to send PUBLISH messages over some NetInf convergence layer, receive the response, transform it into the internal *PublishResponse* representation, and return it to the Node. The class file *PublishController.java* controls the execution of PUBLISH messages.

4.2.9 edu.projectcs.netinf.node.search

This package has a similar functionality as edu.projectcs.netinf.node.publish, except this one works with SEARCH and SEARCH-RESP messages. *SearchController.java* is responsible for controlling the execution of the SEARCH messages and *Search-Service.java* is an interface to be implemented by convergence layers that sends SEARCH messages over a NetInf convergence layer, receives the responses, transforms them into internal *SearchResponse* objects and return them to the node.

4.2.10 edu.projectcs.netinf.node.services

This package contains the services and convergence layers provided by the NetInf node. It also contains *ServiceDiscovery.java*, which implements discovery of other NetInf nodes via Multicast DNS.

4.2.11 edu.projectcs.netinf.node.services.database

This package only contains the class *Database.java*, which defines the database. The database contains all NDOs known to the node. The contents of the NDOs are stored in the filesystem and not in the Database, but the *Database* class abstracts this separation.

4.2.12 edu.projectcs.netinf.node.services.http

The HTTP package defines the HTTP convergence layer. *GetHandler.java* handles incoming GET messages. *HttpApi.java* is the class that handles starting and stopping the HTTP convergence layer. The class *HttpCommon.java* defines the utility methods of the package. *HttpGetResponseHandler.java* is used to parse GET-RESP messages. *HttpGetService.java* handles GET messages. *HttpPublishService.java* handles PUBLISH messages. *HttpSearchService.java* is responsible for handling SEARCH messages. *HttpServer.java* accepts incoming connection attempts, but it is currently only supporting GET messages.

4.2.13 edu.projectcs.netinf.node.services.udp

The UDP package handles all the functionalities related to the UDP protocol in the NetInf service. *UdpApi.java* handles the top level functionalities of the protocol, from which the protocol can be started and/or finished. *UdpCommon.java* defines the utility methods. The GET, NOTIFY and SUB NetInf messages are handled by *UdpGet.java*, *UdpNotify.java* and *UdpSub.java*, respectively. *UdpInProgressTracker.java* keeps track of the UDP convergence layer requests. *UdpServer.java* accepts incoming UDP messages which are parsed by *UdpParser.java*.

4.2.14 edu.projectcs.netinf.node.sub

The sub package defines the interface used to provide output of SUB and input of SUB-RESP messages, as well as the common control logic. The file *SubController.java* controls the execution of SUBs. The *Subscriber.java* class represents a subscriber. *SubService.java* represents an output of NetInf SUB message from the Node. The responsibility of a SubService is to send SUB messages over some NetInf convergence layer, receive the response, transform it into the internal *SubResponse* representation, and return it to the Node.

4.2.15 edu.projectcs.netinf.service

The service package collectively implements the glue required to use the NetInf library as an Android Bounded Service[1]. Files that are connected to the service can be found in this package. *NetinfService.java* is the implementation of android.app.Service; it handles starting and stopping the service as well as incoming requests over IPC. *ClientApplication.java* starts up the service and a node. *MessageCodes.java* is an enum of the IPC message types sent and received by the service. *MessagingNotifyCallback.java* implements node.notify.NotifyCallback by relaying notifications over a android.os.Messenger. *NdoProcessor.java* maintains the threads that execute requests sent to the service. *ServiceUtils.java* contains help-methods for serialization/deserialization and default properties.

To use the service in another application, the following has to be added to that application's manifest file

```
 <service android:process="edu.projectcs.netinf.service"
android:name="edu.projectcs.netinf.service.NetinfService" />
```

There is no client-side API for the service. For efficiency, the user of the service will implement the client side of the IPC. A good starting point and reference for that is the code of the edu.projectcs.falun.api package, which implements that part of the Falun 2015 Android app.

4.2.16 edu.projectcs.netinf.service.queue and edu.projectcs.netinf.service.queue.impl

These packages are responsible for queueing the requests that will be executed by the appropriate sender. The most important class of the package is an abstract class named *NdoQueue.java*. Five classes are extended from *NdoQueue*, these are: *GetQueue.java*, *NotifyQueue.java*, *PublishQueue.java*, *SearchQueue.java* and *SubscribeQueue.java*.

4.2.17 edu.projectcs.netinf.service.sender and edu.projectcs.netinf.service.sender.impl

The class *NdoSender.java* from the sender package represents the consumer in the Producer-Consumer pattern. It executes requests and calls appropriate callback functions. Like *NdoQueue*, *NdoSender* is also an abstract class from which five other classes, located in the sender.impl package, are extended. These are: *GetSender.java*, *NotifySender.java*, *PublishSender.java*, *SearchSender.java* and *SubscribeSender.java*.

Chapter 5

Android application

5.1 Overview

The application section describes the Android application used to let end users watch currently or previously streamed content or stream videos of their own.

This section also describes how the Android application is structured and what each package and class therein are responsible for. This is to simplify for other developers to update and/or extend the code in the future. There is also a section containing information on how the application's user interface was designed.

5.2 User Interface (UI) design process

In order to have a design that would be user friendly enough to not need major changes, the Android application went through a very thorough design process. The design was made by a group of seven people, in the time span of two weeks. Six members of this group formed part of a "design team", and the seventh member acted as an auxiliary member.

Initially, six separate designs were created. Each model would show the designer's interpretation of how the application would look, as well as the functionalities that the user would be able to use. The six designs were then compared and discussed within the group, and merged into three different designs. Each model would be intentionally different from the others, allowing to test different features.

Paper prototypes were created for each of the three designs, allowing the team to quickly test how users would interact with the models. All of the interviews followed the same structure: explain the task to the users, give them a number of tasks to perform on the paper prototype (such as start a recording, view a stream, etc.), and evaluate how each user performed. For each prototype, four users were interviewed except for the last prototype where six people were interviewed, giving the team enough results to improve the design and build a final prototype. This final prototype included the best features of each previous idea, as well as the improvements that emerged from the previous interviews. The same interview process was followed for this final prototype, resulting in a final application design. No programming for the user interface was performed until the team had enough qualitative data about the prototypes to produce a final design.

5.3 Package overview

The android-netinf project is organized in 8 packages:

- edu.projectcs.falun
- edu.projectcs.falun.activities
- edu.projectcs.falun.api
- edu.projectcs.falun.player
- edu.projectcs.falun.send
- edu.projectcs.falun.streamer
- edu.projectcs.falun.watch
- edu.projectcs.falun.watch.drawer

5.3.1 edu.projectcs.falun

The top level package of the application. Contains the subclass of android.app.Application, *ClientApplication.java.*

5.3.2 edu.projectcs.falun.activities

This package contains all the Android activity classes.

AboutActivity.java presents information about the team that developed the application.

MainActivity.java is the main page of the application. When it is visited for the first time, a tutorial will be presented. From this activity the user can navigate to all of the activities of the application. The preferences are also loaded here.

SettingsActivity.java is the settings page. It contains some developer options to tweak settings such as the UDP port number, the type of routing, among other options. Upon opening the activity, it will load the settings using an Android feature called PreferenceFragment from a file named *preferences.xml*, which resides in the *src/xml* folder of the android-service library. Note that the activity *SettingsActivity.java* is launched after touching the screen four times using three fingers. It is meant to be used by developers, and is thus hidden from the users.

SendActivity.java is used for testing and debugging several features of the platform. It allows the user to send NDOs, publish NDOs, publish messages, get messages, search for messages and perform subscriptions.

StreamingActivity.java is where the user streams the video. This class shows the preview of the camera and handles the recording and switching between the front and back camera. When the activity opens, it will ask the user to add a name and a description to the video (this step can be avoided if the user decides to check the “do not show again” checkbox).

VideoActivity.java is responsible for playing the video streams using a player called ExoPlayer. The activity has a custom UI that displays the title and description of the video, and a label that will show the text *“Live”* if the video currently playing is a live stream.

WatchActivity.java is responsible for showing the user the videos (live or not live). The activity has three main components: a navigation drawer, a list view, and a map view. The navigation drawer allows the user to customize the content of the views, the list view shows all the videos using a list, and the map view shows the location of the videos as pins on a map.

5.3.3 edu.projectcs.falun.api

The api package provides an abstraction of the backend and the serialization format of streams and chunks. It handles the publish and subscribe process, the creation of chunks and the metadata of the video.

Api.java represents a connection to the backend and allows the access to information about, and contents of, streams and creates the necessary objects that perform the tasks discussed above. It must be manually closed when it is no longer needed.

ApiNotifyCallback.java is responsible for receiving notification messages and calling NextChunkCallback.

Callback.java is the callback that is called when an operation completes.

ChunkInfo.java describes a chunk of the stream. Related to *ChunkInfo* are the classes *ChunkQueryBuilder.java*, which constructs queries for chunks belonging to a stream, and *NextChunkCallback.java*, which is called when a new chunk is available and when a subscription succeeds or fails. *QueryBuilder.java* constructs queries for the streams. It contains information such as the liveness status of the stream and the date when it was started. *Result.java* represents a result, which is either a result of type $< T >$ or an error. Lastly, there is the *StreamInfo.java* class which describes a current or past stream by using its title, description, starting and end time, location, region etc. *ApiNotifyCallback.java* is used for abstracting the notification interprocess communication with the service and calling the *NextChunkCallback*.

5.3.4 edu.projectcs.falun.player

This package is responsible for the playback of videos. ExoPlayer, a project that provides a pre-built player that can be extended, is used for this task. It supports features such as persistent caching and dynamic adaptive streaming over HTTP (MPEG-DASH)[2]. Of the seven classes that are contained in the package, three of them (*NetinfDataSource.java*, *NetinfStreamChunkSource.java* and *NetinfStreamRendererBuilder.java*) are related to the managing of the videos chunks. *LivenessTracker.java* decides if a playback is considered to be a live stream. *VideoPlayer.java* is a wrapper around ExoPlayer that provides a higher level interface, and *VideoUtil.java* provides utility methods for the video player.

5.3.5 edu.projectcs.falun.send

This package contain classes related to *SendActivity*. *ResponseHandler.java* handles the responses from the NetInf Service.

5.3.6 edu.projectcs.falun.streamer

This package deals with the camera functionalities and the encoding of video streams. *CameraPreview.java* handles the preview of the camera. This allows for, among other actions, setting the maximum resolution and desired frames per second of the video stream, depending on the bandwidth capabilities of the network.

ChunkPublisher.java is a process running on a separate thread that is responsible for publishing the collected chunks.

DetailsDialogue.java contains the code for the small popup dialog box where the user enters the title and description of the stream.

Encoder.java handles the encoding of the video stream. This class sets the interval for the i-frames, bit rate and MIME type. This class is also in charge of sending the video chunks through the api, which is connected to the backend.

EncodingPreviewCallback.java is responsible for allocating callback buffers and attaching them to the camera.

ImageTransformation.java contains methods to transform images to different color formats.

MovingAverage.java is a class that computes a moving average over a fixed amount of samples.

OutputStreamChannel.java implements a WritableByteChannel around an OutputStream.

TimeUpdater.java handles the timer present on the screen while streaming.

Utils.java contains some useful utilities for the encoder, like computation of the size of video frames in YV12 and YUV420 formats.

5.3.7 edu.projectcs.falun.watch

This package contains classes which are related to the list view, map view and the slider menu. Available videos will be displayed as a list of clickable items in the list view tab and as pins on the map view tab. Displayed videos can be filtered by using the slider menu. They can be filtered by choosing whether they are live or not, choosing a specific date and time, and choosing a specific region. In addition to filtering, the displayed videos in the list view can be sorted by title, date and region. The available videos will be displayed as pins on the map based on the location where they were recorded.

ListViewFragment.java holds the list of streams and is responsible for updating it. *MapViewFragment.java* defines the map view and everything therein. The *StreamInfoAdapter.java* class holds the information regarding each stream, such as the title and region. *TimeManager.java*, is responsible for managing the time.

5.3.8 edu.projectcs.falun.watch.drawer

The drawer package contains everything related to the side menu of the watch page. *Drawer.java* defines the side menu, while *RegionSpinner.java* and *SortSpinner.java* define the two spinners for selecting the region or selecting how to sort the list of streams.

5.3.9 The *res* folder

The res folder contains images and XML files used for the UI of the Android application. These XML files are divided into several folders:

- layout: Contains the layout of every activity of the application. These files use the following naming convention: *activity_activityname.xml*. This folder also contains *list_fragment.xml* and *map_fragment.xml*, both of which are fragments (portion of user interface) implemented in the watch activity. *drawer_item.xml* is the navigation drawer from *WatchActivity.java*. *spinner_item.xml* is used as a template for the view that populates the navigation drawer, and *list_view.xml* populates the list of stream from the watch activity.
- drawable: Contains all the images and icons that are used in the application.
- values: Contains strings, colors, dimensions, themes and styles that are referenced from the code.
- menu: Contains the options the user will have when touching the overflow button from the action bar.

5.4 Testing

The testing of the Android application is performed using a UI/Application exerciser called *Monkey*[3], which can be used to stress-test applications by generating pseudo-random streams of clicks, gestures, and touches. Throughout the duration of the project, this test proved to be effective in finding race conditions and other flaws in the application.

Bibliography

[1] Android bounded services. `http://developer.android.com/guide/components/bound-services.html`. Accessed: 2014-12-12.

[2] Exoplayer. `https://developer.android.com/guide/topics/media/exoplayer.html`. Accessed: 2015-01-16.

[3] Ui/application exerciser monkey. `https://developer.android.com/tools/help/monkey.html`. Accessed: 2014-12-16.

[4] Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.