JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Androgios

Android Interface for Nagios

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Netzwerke und Sicherheit

Eingereicht von:
Markus Koppensteiner, BSc

Angefertigt am:
Institut für Informationsverarbeitung und Mikroprozessortechnik

Beurteilung:
Assoz.-Prof. Mag. Dipl.Ing. Dr. Michael Sonntag

Linz, Juli 2012

Contents

List of Figures

List of Tables

Listings

Abstract

Nagios is a widely used open source network monitoring system. It enables administrators to continuously monitor hosts and services on the network. Through a web interface administrators are able to watch the state of monitored entities. However, an administrator can not only watch problems, but also handle them in an easy way, for example he acknowledges a problem or adds a comment. In that way it can be avoided that unintentionally multiple persons work on the same problem the same time.

The growing distribution of the operating system Android for mobile devices (smartphones, tablets) opens up the possibility to bring the user interface of the network monitoring system still closer to the administrator: It is no longer necessary to constantly monitor the web interface at a computer. When problems arise, mobile devices can alert automatically (e.g. by sound, or vibration).

The developed software "Androgios" displays the main features of the Nagios web interface on Android devices. Through the use of Androgios users can inspect both hosts and services. Overview lists of all monitored entities can be displayed. These lists can be filtered to e.g. only display problems. In addition to overviews there is a detailed view for each monitored entity available. Androgios also supports to send external commands such as acknowledging problems or adding comments. Changes in the configuration of the Nagios server are not necessary.

Kurzfassung

Nagios ist ein weit verbreitetes Open Source Netzwerkmonitoring System. Es ermöglicht Administratoren Hosts und Dienste eines Netzwerkes kontinuierlich zu überwachen. Über eine Weboberfläche können Administratoren den Zustand überwachter Einheiten einsehen. Ein Administrator kann Probleme jedoch nicht nur einsehen, sondern auch in einfacher Weise behandeln, beispielsweise indem er ein Problem bestätigt oder einen Kommentar hinzufügt. Auf diese Weise wird verhindert, dass sich ungewollt mehrere Personen mit dem gleichen Problem gleichzeitig beschäftigen.

Die zunehmende Verbreitung des Betriebssystems Android für mobile Geräte (Smartphones, Tablets) eröffnet die Möglichkeit eine Benutzerschnittstelle zum Netzwerkmonitoring System noch näher an den Administrator zu bringen: Es ist nicht mehr notwendig ständig an einem Rechner die Weboberfläche zu überwachen. Bei auftretenden Problemen kann ein mobiles Gerät selbstständig alarmieren (z.B. durch Ton, Vibration).

Die entwickelte Software "Androgios" stellt die wichtigsten Funktionen der Nagios Weboberfläche auf Android Geräten zur Verfügung. Durch die Verwendung von Androgios können Benutzer sowohl Hosts als auch Dienste inspizieren. Es können Listen von allen überwachten Geräten angezeigt werden. Diese Listen können gefiltert werden um beispielsweise nur Probleme anzuzeigen. Neben den Übersichten steht eine detailierte Ansicht für jedes überwachte Gerät zur Verfügung. Androgios unterstützt außerdem die Übermittlung externer Kommandos wie beispielsweise das Bestätigen eines Problems oder das Hinzufügen eines Kommentars. Änderungen in der Konfiguration am Nagios-Server sind dabei nicht notwendig.

1 Introduction

This thesis deals with the network monitoring system Nagios and the operating system Android. An important benefit of Nagios is that it enables administrators to detect network problems early. To ensure that an administrator handles network problems as soon as possible, even when he does not sit at his computer and watch the Nagios software, Nagios is capable to send notifications by E-Mail or SMS. This approach has the disadvantage of forcing the administrator to his computer immediately. Android devices as small computers are always near at hand. It is therefore natural to want to have an interface to Nagios on the Android device. This would give administrators the chance to get an first overview of what happened in the network and to initiate first steps of remediation remotely.

Nagios has a web user interface and there is a web browser available on Android. So it is already possible to handle Nagios from Android devices. However, the screens on Android devices are quite small, so that the usage of the web browser becomes very hard. Figure 1 illustrates the problem.

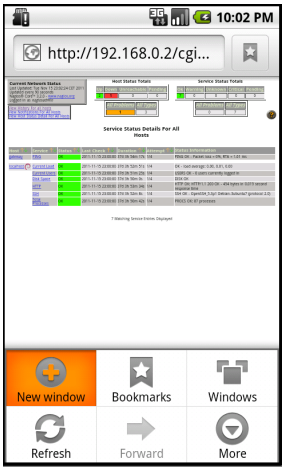


Figure 1: Problem of the Web UI on Android

Sought is therefore a system that combines Nagios and Android with a better usability. The basic ideas for Androgios, the software to develop, are the following.

- A system to show and manage Nagios from Android devices.
- It should support most of the functionality of the web interface, but with an easier use on touchscreens. No web interface, but a native application.
- The main Nagios system must not be changed.
- The system must support encrypted communication (HTTPS).
- Management must include: re-scheduling checks, acknowledging problems, disabling notifications/checks, scheduling downtime, flap detection, comments.
- The system should run (and ideally look good too) on mobile phones and on tablets.

This thesis is organized as follows:

Section 2 is the theoretical part regarding network monitoring with Nagios. After the introduction of Nagios' basic principle of operation, the meaning of its monitoring features and the capabilities of the web user interface are explained. This should help to understand the idea and benefit of Androgios' functionality better.

Section 3 describes the fundamentals of Android, Androids architecture and how to get started with developing on the Android platform. In addition core concepts of Android development are introduced as necessary to understand the implementation of Androgios.

Section 4 is the main part that describes the practical aspects and the implementation of this work. Competiting software and problems with this software are discussed here. Then Androgios' base architecture is explained, followed by a detailed description of the implementation of Androgios core components. How Androgios was tested finishes this section.

Section 5 finally recapitulates the thesis and discusses ideas for future work.

2 Network Monitoring

IT departments cannot monitor big networks manually all the time. However, in complex networks it is crucial to notify administrators about problems as early as possible. Network monitoring systems can check network infrastructure automatically. Goal of network monitoring systems is to recognize problems, ideally before they become critical. In that way administrators can handle upcoming problems before they impact business processes.

A widely used network monitoring system is the open source software Nagios. Administrators can use the the web interface of Nagios to get an idea what's up in the network. Nagios uses traffic light colors to denote states: green: ok, yellow: warning, red: problem. What is considered a warning and what a problem can be configured by the administrator. The following sections explain Nagios in more detail.

2.1 Nagios

The Nagios homepage [Ent12b] introduces Nagios as follows:

"Nagios is a powerful monitoring system that enables organizations to identify and resolve IT infrastructure problems before they affect critical business processes."

and furthermore:

"Nagios monitors your entire IT infrastructure to ensure systems, applications, services, and business processes are functioning properly. In the event of a failure, Nagios can alert technical staff of the problem, allowing them to begin remediation

processes before outages affect business processes, end-users, or customers. With Nagios you'll never be left having to explain why a unseen infrastructure outage hurt your organization's bottom line."

Nagios distinguishes between hosts and services when it checks for problems. A host check tests if a host is reachable, usually via ICMP echo (ping). Hosts can be computers or other network devices such as routers, switches and so on. A service check on the other hand tests network services such as HTTP or SSH. Service checks can also include CPU-load or disk space. Basically everything measurable can be monitored. Services always run on a particular host. This implies: if a service is ok then the corresponding host must be up and Nagios does not perform a host check in that case. The host check is only performed when none of the service checks were successful.

Nagios has a modular structure. The Nagios core does not contain any checking functionality. To perform checks, Nagios uses external programs called plugins. Nagios already ships with a number of plugins. Plugins may also be implemented by the user (plugins are written in PERL) or may be downloaded (NagiosExchange [Ent12a] is an excellent source for plugins). Plugins return one of the states: ok, warning, critical or unknown. The state unknown denotes an error in the usage, such as an illegal option. Additionally if the check is performed the first time and no data has been collected yet, Nagios will display the state pending.

Nagios always tries to support administrators to find problem causes efficiently. Unsuccessful service checks are denoted as unimportant when the corresponding host is down. A service may not only depend on a host, but also on another service. E.g. a service may need a database in order to work correct. Such service dependencies can be configured. Also configurable is that hosts may have parent

hosts on that they depend. The network topology can be determined in that way and Nagios uses this dependencies to give hints where to find the actual problem cause.

Nagios comes with a notification system to avoid that administrators have to watch the Nagios web interface all the time. It is configurable which contacts should be informed, how the contacts should be informed and about what state changes. Common ways to notify contacts are SMS and e-mail. For escalation management it is possible to notify a supervisor in case of a problem that is not fixed within a certain time period.

Once an administrator is aware of a problem, he acknowledges the problem using the web interface. This instructs Nagios to not send further notifications. The acknowledgement includes a comment set by the administrator that should contain meaningful hints such as e.g. what is intended to be done or not, an external support team has been informed ... Other contacts can be notified about the acknowledgement.

It's possible that hosts or services are down for maintenance reason. In this time periods no notifications should be sent. For such cases it is possible to set a downtime. Downtimes include a comment to indicate the reason. The time period can be treated in a fixed and in a flexible way. Fixed time periods start and end at an exact defined time. Flexible time periods start when a host (or service) changes its state to down and end after a certain time elapsed.

Acknowledging problems and setting downtimes are examples for sending external commands. Nagios supports many more external commands. Therefore Nagios is not only a pure viewing system. External commands also allow e.g. to enable or disable the monitoring features described in section 2.1.1

Nagios allows to define host and service groups. This enables administrators to see states of a specific group quickly. Host and service groups can also be used to send an external command to all members of the group at once to avoid forcing an administrator to enter the same command over and over again.

2.1.1 Monitoring Features

Nagios offers a set of different monitoring features that can be enabled or disabled on individual hosts/services or globally for all.

Active Checks are initiated by the Nagios server. They are scheduled by the Nagios server on basis of the configured *check_interval*. The check intervals can be configured for each host and service separately. In case of negative check results a *retry_interval* can also be defined. The retry interval is used as long as the host or service is in soft state. The parameter *max_check_attempts* tells how often retries are performed. After that it turns into hard state and the normal check interval is used. In contrast to soft states, hard states are considered as "real" problem. This distinction is used to prevent false alarms.

Passive Checks While active checks are initiated and performed by the Nagios server, passive checks are done by external applications that transmit the results to Nagios. Passive checks are useful e.g. to reduce traffic (active checks generate a lot of network traffic, only to determine everything is ok) or if active checks are not possible perhaps due to a firewall that blocks the check. Typical passive checks are SNMP traps.

Obsessing A command that is executed after each check may be configured. The obsessing feature allows to enable or disable the execution of the defined command. By default obsessing is disabled globally.

Notifications are sent when a hard state changes or if a hard state other than ok remains longer than in the parameter *notification_interval* specified. Notifications will not be sent if one of the following points is true:

- the host or service is in a scheduled downtime
- the host or service is flapping (see below)
- the host or service is not configured for notifications
- the notification has to be sent in a non-valid time period for sending notifications (the notification is rescheduled in that case)

Event Handlers are commands that are executed when a host or service is in soft state or changes to a hard state for the first time. Event handlers may trigger self healing mechanisms, such as restarting faulty services. They can be of global type or host/service specific.

Flap Detection A host or service is flapping when it frequently changes its state. Flapping is detected by calculating a percentage of state changes. To calculate the percentage of state changes, the last 21 check results are considered. By comparing the percentage of state changes against a high flapping threshold it is detected that a host or service has started flapping. To determine flapping has stopped, a low flapping threshold is used.

2.2 Nagios Webinterface - CGI

The Nagios web interface allows more than only to view information. Commands can be sent and Nagios can be actively controlled, e.g. adding a comment or restart Nagios. All web interface functionality is provided by CGI (based on C-programs). In the following the CGI-programs are explained to give an overview of their capabilities.

status.cgi This CGI-program is responsible for displaying status information in several forms. There are three parameter groups that control what is displayed in particular. The first group controls whether hosts, hostgroups or servicegroups are displayed.

```
...cgi-bin/status.cgi?host=hostname
...cgi-bin/status.cgi?hostgroup=hostgroupname
...cgi-bin/status.cgi?servicegroup=servicegroupname
```

To display all hosts, hostgroups or servicegroups the keyword `all` can be used. The second parameter group controls the style. A different style means different granularity.

```
...cgi-bin/status.cgi?host=all&style=detail
```

Five styles are supported:

- `overview` displays hosts within a table and services are summarized by states.
- `summary` hosts are summarized by states too.

- `grid` displays hosts and services within a table, service states are indicated by color only.
- `detail` displays detailed information for each service in a separate row.
- `hostdetail` displays detailed information for each host in a separate row.

The third parameter group allows to filter by status types (e.g. UP, DOWN, ...) and by properties (e.g. Flap Detection enabled, notifications disabled, ...).

```
...cgi-bin/status.cgi?host=all&hoststatustypes=2&hostprops=8
```

Detailed information about the application of filters can be read in section 4.3, tables 4, 5 and 6.

extinfo.cgi This CGI-program displays the most detailed information about a host, service, hostgroup, servicegroup, comments, downtimes, the Nagios process, performance data or scheduled checks. Additionally, it provides external commands. The detailed state information on the left side shows e.g. the state itself (and since when), the last check time, the next scheduled check, is the entity flapping or is it in scheduled downtime. Furthermore it showns if monitoring features are enabled or disabled. At the bottom a list of comments is displayed. Commands for adding a new comment and deleting all comments are offered there too. All other commands are displayed at the right side. Commands refer to the cmd.cgi that is explained in its own paragraph.

What in particular to display is controlled by the *type* parameter. Possible values are:

- `extinfo.cgi?type=0`
  displays information about the Nagios process such as total time running, process id or what monitoring features are enabled.
- `extinfo.cgi?type=1&host=hostname`
  displays information about a host
- `extinfo.cgi?type=2&service=servicename`
  displays information about a service
- `extinfo.cgi?type=3`
  displays all host and service comments
- `extinfo.cgi?type=4`
  displays information about the performance of Nagios, divided by active host checks, passive host checks, active service checks, passive service checks
- `extinfo.cgi?type=5&hostgroup=hostgroupname`
  displays information about a hostgroup
- `extinfo.cgi?type=6`
  displays all scheduled host and service downtimes
- `extinfo.cgi?type=7`
  displays all scheduled checks
- `extinfo.cgi?type=8&servicegroup=servicegroupname`
  displays information about a servicegroup

cmd.cgi is the CGI-program that accepts external commands. The parameter *cmd_typ* controls what command should be executed. For more detailed information about what commands may be sent and what additional parameters are

required see tables 3 and 2 in section 4.3. Nagios' web interface provides for each command a separate input form. A command description is also available, shown in figure 2.

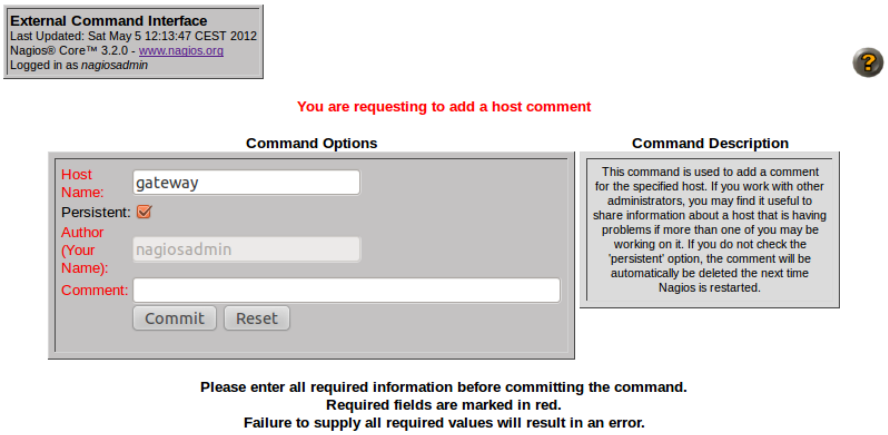


Figure 2: Adding a comment

tac.cgi The CGI-program tactical overview displays summarized information in a single page. As shown in figure 3 the tactical overview focuses on displaying problems. Network outages are shown first, followed by the states of hosts and services. At the bottom is shown if certain monitoring features are globally enabled or disabled. In case of disabled monitoring features on single hosts or services, the number of them is displayed.

If everything is ok, only the number of unproblematic hosts and services is shown. In case of problems additional information is displayed such as number of unhandled problems (these are problems without an acknowledgement). Nagios distinguishes between important (e.g. a service in state critical without an acknowledgement) and unimportant problems (e.g. a service in state critical that is acknowledged). Important problems are highlighted in red, unimportant problems

in pink.

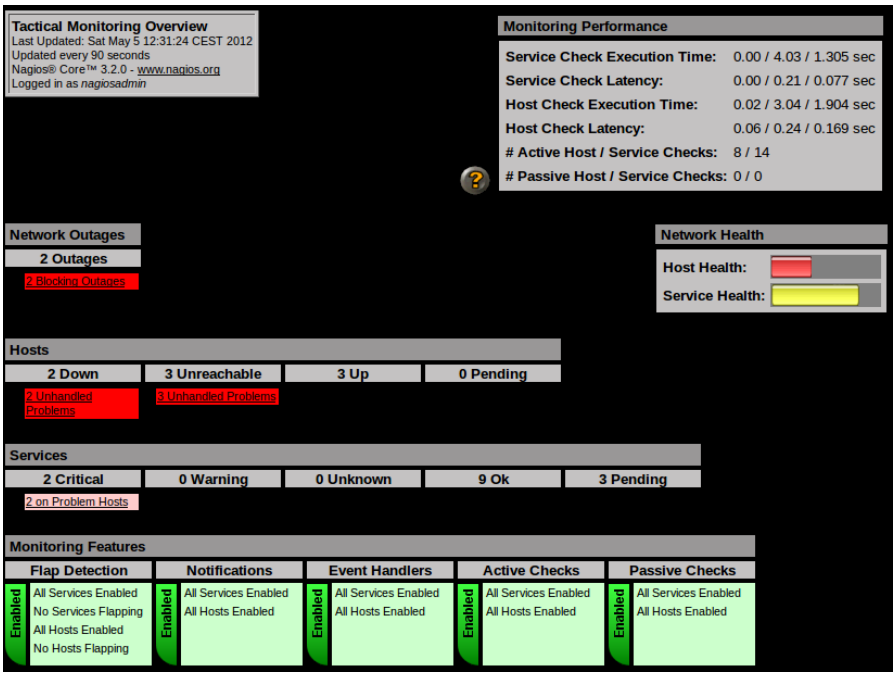


Figure 3: Tactical Overview

outages.cgi This CGI-program displays a list of all hosts that cause an outage. This list includes the number of hosts and services that are affected. Based on the number of affected hosts/services Nagios calculates a severity level that is also shown in the list. Possible actions are navigating to: detailed host information, status map, trends, log entries or notifications.

others There are many other CGI-programs available. Since they are not used by Androgios, they are only introduced briefly.

- *statusmap.cgi*

  displays the monitored hosts topologically

- *statuswrl.cgi*

  displays the monitored hosts topologically in 3D (a VRML supporting browser is required)

- *statuswml.cgi*

  simple status page for WAP devices

- *config.cgi*

  displays the Nagios configuration

- *avail.cgi*

  displays availability reports

- *histogram.cgi*

  displays a histogram of events happened

- *history.cgi*

  displays all events ever happened

- *notifications.cgi*

  displays all notifications sent

- *showlog.cgi*

  displays all logfile entries

- *summary.cgi*

  displays reports of events, arranged by hosts, services, errors and timeperiod

- *trends.cgi*

  displays a timeline including states occurred

3 Android

According to [Inc12] "Android is a software stack for mobile devices that includes an operating system, middleware and key applications."

Android Applications (Apps) are written in the Java programming language. Although there is an Android specific library used, the Java Standard Edition API [Cor12] is available. The Android library supplements the Java SE API by an Android specific API. This section shows the basic concepts of Android as necessary to understand the implementation of Androgios.

3.1 Architecture

Android is based on a Linux kernel. The kernel provides the drivers and is responsible for security, memory/process management and the network stack. Figure 4 shows Androids main components.

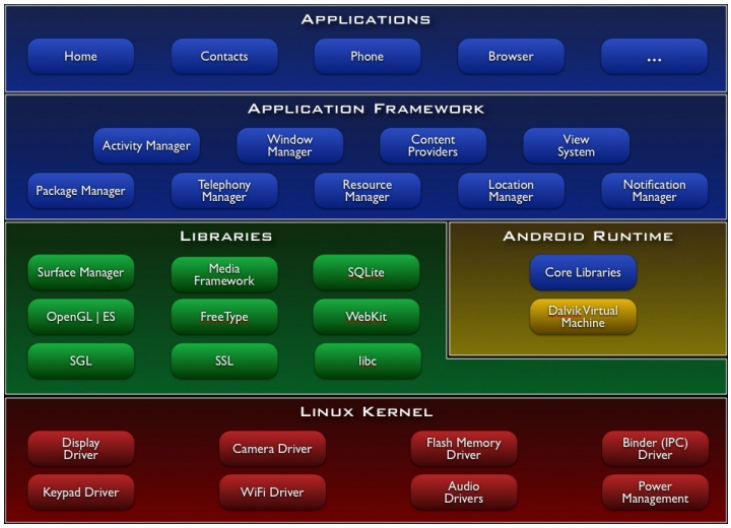


Figure 4: Android Architecture (from [Inc12])

Core part of the Android runtime is the Dalvik virtual machine. For each Android application a separate Dalvik virtual machine is started, which runs in its own process. The Dalvik virtual machine is therefore optimized for running multiple virtual machines concurrently. Although Android applications are written in java, the Dalvik virtual machine does not use java bytecode, but its own format, the dex bytecode. This format is required because the Dalvik virtual machine is a register machine, in contrast to the java virtual machine that is a stack machine. Dex bytecode is created from java bytecode by the dx-tool which is part of the Android SDK. A overview of the Dalvik virtual machine can be found in [Nic09] and [BP10].

Android includes several C/C++ Libraries. These libraries enable Android to provide capabilities needed by applications such as graphics, database or network connections. Applications cannot access the libraries directly, but only through the application framework.

The application framework provides system classes, e.g. for accessing hardware. Many of these classes are called Manager. All classes of the application framework are written in java and developers can use them. Some important components are:

- Views like buttons, lists, ...
- Content Providers to manage the access to data.
- Resource Manager to provide and access e.g. different layouts for different screen sizes, or language resources.
- Notification Manager to manage user notifications such as displaying icons, vibrating, or flashing LEDs.
- Activity Manager to interact with different screens of a running application.

The components are described in more detail in section 3.3.

On the applications level reside the actual Apps. This may be both standard applications shipped with Android and applications developed by third parties. All applications use the application framework.

3.2 SDK/AVD

Before development for Android can start the Android software development kit needs to be downloaded and installed. To run Android applications on the developing machine an Android virtual device is needed. [Inc12] recommends to use Eclipse in conjunction with the Android Development Tools (ADT) plugin. The ADT plugin provides e.g. custom XML editors for manipulating the manifest file or user interface definitions. Guided project setup and support for creating signed .apk files make developing easier. Appendix C shows how to install the SDK, the ADT plugin and how to get started with Android programming in detail.

3.3 Android Components

Android components that are used by an application must be declared in a so called manifest file. An Android application needs a manifest file with the exact name ”manifest.xml”. The manifest file contains information about the application. The Android system needs this information in order to be able to start and run the application. Content of the manifest file is e.g.:

- Activities
- Services

- Broadcast Receivers
- IntentFilters
- Permissions
- Minimum API level

Activities, services, and broadcast receivers are explained in the following sections. They are started by special objects called intents. Intents hold certain information for the Android system and also for the receiving component. This information is: component name, action, data, category, extras and flags. The component name represents the name of the component that should handle the intent. The action is a string value representing the name of the action to perform. Data represents the URI of the data to be acted on. Category denotes the kind of the component that should handle the intent. Extras are used to transfer key value pairs from sender to receiver. Flags (static defined in the intent class itself) can be set.

Intent filters describe what intents a component wants to receive. For example if an activity is able to initialize an application it may declare the intent filter shown in listing 1.

```
1 <intent-filter>
2     <action android:name="code␣android.intent.action.MAIN" />
3     <category android:name="code␣android.intent.category.LAUNCHER" />
4 </intent-filter>
```

Listing 1: Intent Filter

3.3.1 Activities

Activities represent screens that enable users to do different tasks. Android applications typically consist of several activities. One activity must be declared in the manifest file as the main activity. Further activities can be started by calling the *startActivity(Intent)* method.

The lifecyle of activities is shown in figure 5. An activity can be in one of the states running, paused, or stopped. A state transition always cause the invocation of a certain callback method. When an activity is newly created the *onCreate()* callback method is called. If the activity is the main activity, this will be the entry point for programming. *onCreate()* is followed by *onStart()*, which is called before the activity becomes visible. After that, two things may happen: First the activity becomes indeed visible, in this case *onResume()* is called and the activity is then in the state running. Second the activity becomes hidden, in this case *onPause()* is called and the activity is in the state paused. If an activity is in the state running, the only following callback method is *onPause()*. *onPause()* is called when another activity should come in the foreground.

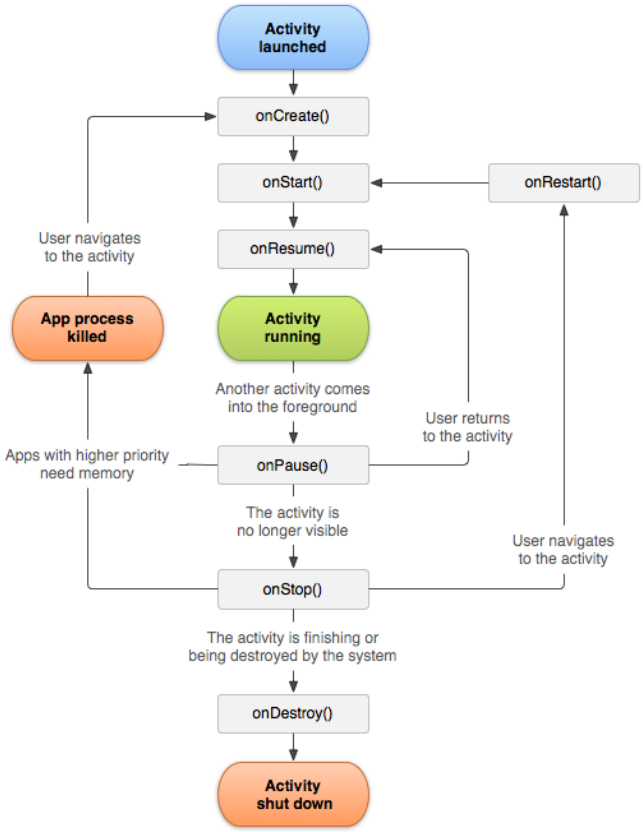


Figure 5: Activity Lifecycle (from [Inc12])

After *onPause()* the activity may directly come in the foreground again. In this case *onResume()* is called. The small cycle between *onResume* and *onPause()* is also known as foreground lifetime. If the activity becomes invisible (because another activity covers it entirely or its about to be destroyed), *onStop()* is called. When the user navigates back so that the activity becomes visible again, *onRestart()* is called, followed by *onStart()*. The cycle between *onStart()* and *onStop()* is also known as visible lifetime. If the activity is about to be destroyed, *onDestroy()* is called. The cycle between *onCreate()* and *onDestroy()* is also known as entire lifecycle.

In the activity lifecycle unexpected events may happen. Device configuration changes such as screen orientation cause the android system to call *onDestroy()*, immediately followed by *onCreate()*. To handle such events the methods *onSaveInstanceState()* and *onRestoreInstanceState()* can be used as shown in figure 6.

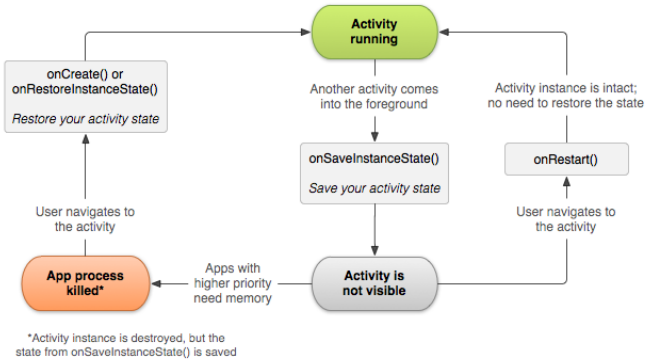


Figure 6: Activity Instancestate (from [Inc12])

When an activity A starts another activity B, then activity B is pushed on the back stack (A must already exist on the back stack, since it has already been created) as shown in figure 7. *onStop()* has been called on activity A and activity B is in state running. The topmost activity on the back stack has the focus. When the back button is pressed, activity B is popped from the back stack and it is about to be destroyed. Activity A is then brought back to the foreground and its state changes to running.

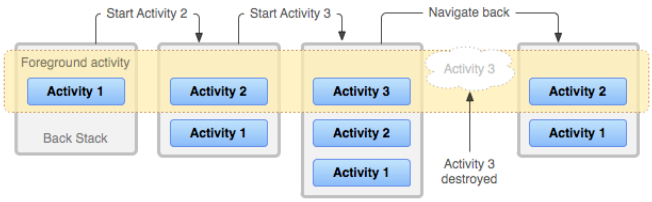


Figure 7: Activity Backstack (from [Inc12])

The basic user interface layout of an activity is declared in an xml file. To define the user interface of the activity, *setContentView(int)* must be called within the *onCreate()* method as shown in listing 2. The parameter "R.layout.some_layout" is an integer value identifying the xml file to load from. This integer value is created automatically by the SDK when the project is built (more precisely the class *R* is generated, that holds identifiers for all resources).

```
1 public void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(R.layout.some_layout);
4 }
```

Listing 2: Activity Layout

3.3.2 Services

In contrast to activities, services do not provide a user interface. Services are used for background tasks. They have to be declared in the manifest file. There are two forms of services: a started form and a bound form.

- Started: created by calling *startService()*. This form may run even if the component that created the service was destroyed.
- Bound: created by calling *bindService()*. Components can bind to this service in order to be able to interact with it. Multiple components may bind to a service. If the last component unbinds, the service will be destroyed.

Services do not create their own thread. However it may be necessary to create a separate thread within a service for long running tasks such as networking. This

avoids blocking of the main thread and therefore reduces the risk of an "Application not responding" error.

Figure 8 shows the lifecycle for both started and bound services. A started service is created when a component calls *startService()*. When this service is in state running there are two ways to destroy it: First the service destroys itself, when the task is done, by calling *stopSelf()*. Second a client component calls *stopService()*.

A bound service is created when a component calls *bindService()*. Clients can communicate with the service via the *IBinder* interface. The service stops if there are no more clients bound to it.

The Android system destroys services when they are stopped. Its worth mentioning that services can be started and bound at the same time. Although there is no obvious reason for hybrid services its technically possible and developers should bear in mind that such a hybrid service will only stop when the stopping conditions for both service forms are met.

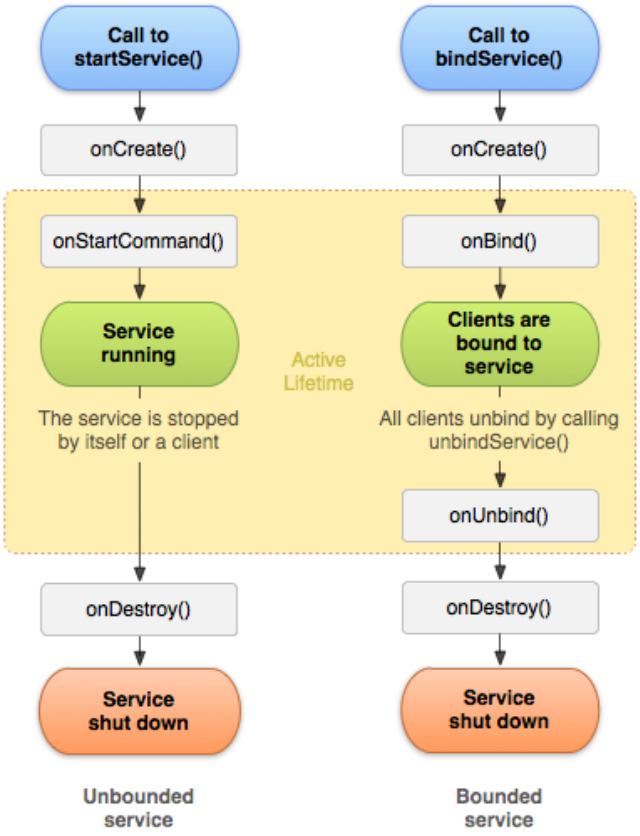


Figure 8: Service Lifecycle (from [Inc12])

3.3.3 Broadcast Receivers

As their name imply broadcast receivers are components that can react on broadcasts. Broadcasts may be sent by the Android system or by another component that called *Context.sendBroadcast()*. They are delivered as *Intent* Objects. There are two ways to use a broadcast receiver. First way (static) is to declare it in the manifest file. In this case the broadcast receiver is registered when the application is installed. An application receives broadcasts it has registered for, e.g. it may declare a receiver for ACTION_BOOT_COMPLETED in order to start itself

after system reboots. Second way (dynamic) is to implement a subclass of the abstract baseclass *BroadcastReceiver* and use it within an activity or service. Such a receiver exits only for the lifetime of its parent component.

3.3.4 Content Providers

Many applications allow to store or load data. A content provider manages that data. Using credentials content providers can provide their data to any application. Content providers are loosely coupled to applications via an interface. Since the developed software Androgios does not use content providers they are not explained in more detail here.

4 Androgios

Androgios is an Android application, which enables users not only to be informed about the current network status but also to react on newly appeared problems. Figure 9 shows the main setup.

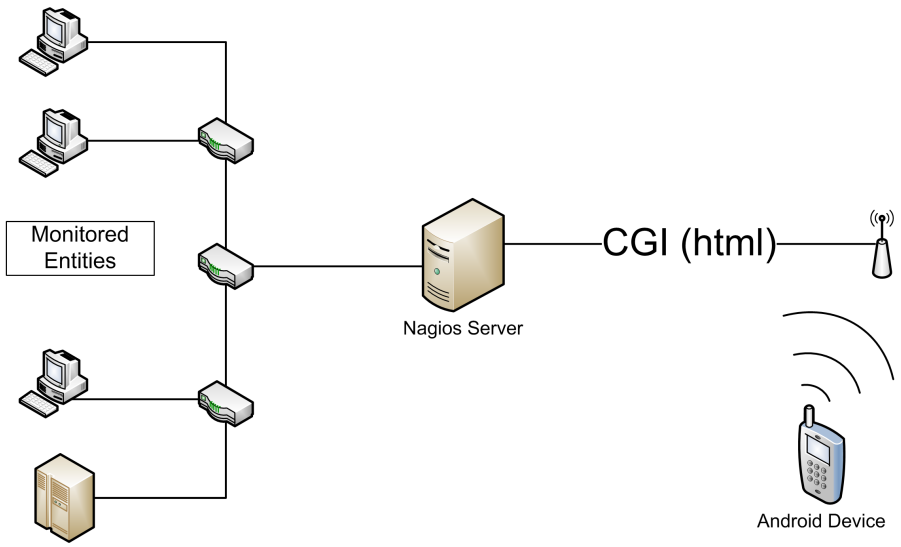


Figure 9: System Overview

The Android device is capable to connect to a Nagios server. Since there is no web service available on Nagios servers by default, Androgios uses the cgi programs and parses information from HTML results. The gathered information is then displayed in a way optimized for small screen sizes. Androgios provides most of the functionality of the standard web user interface. Furthermore, Androgios allows to filter information, in order to find the host (or service) of interest quickly. Androgios does not require changes to the Nagios server or to any monitored entity.

This section is structured as follows: First an overview is given about what com-

petitor software is available in the Android marketplace. Then an architectural overview of Androgios gives an idea how Androgios is designed and how it fulfills its tasks. A detailed description of the concrete implementation builds the main part. The testing strategy and ideas for future work complete this section.

4.1 Competitor Software

According to c't article [Rie10] from November 2010, Nagroid and NagMonDroid were the first Apps available.

Nagroid is a pure problem viewer. Its capabilities are very limited. Hosts and services can only be shown if they are not OK. In case of problems there is no way to acknowledge them or to send any external command. However, Nagroid can start a service that checks for problems and alerts the user if necessary.

NagMonDroid is also very limited. In contrast to Nagroid, NagMonDroid is capable to display hosts and services in state OK. Problems are not identified reliably. This is reported in [Rie10] and also in experience reports from the Android marketplace. A manual refresh is not possible and the service, once started cannot be stopped (there is a button for stopping the service, but polls and alerts go on). Additionally there is no support for HTTPS.

Both, Nagroid and NagMonDroid are not further developed, but the problem was realized by the Android developer community. C't article [Döl12] reports in march 2012 about the better developed Apps aNag and uNagi.

ANag is capable to handle multiple Nagios servers. When the App is started a list of configured Nagios servers is displayed. The list entries show how many hosts and services are in which state. Details can only be displayed for problem hosts

and services. The same is true for external commands: only commands regarding problems, such as to acknowledge a problem, are available.

UNagi is capable to handle multiple Nagios servers too. When it starts it displays all hosts including their state. Services are display after a click on a host. Details and external commands are available for hosts and services in all states. UNagi is probably the most developed Nagios App so far. Drawbacks are that uNagi needs at least Android 2.2 and only Nagios version 3 is supported.

Table 1 shows the main aspects of the different applications in comparison.

	Nagroid	NagMonDroid	ANag	UNagi
Minimum Android	1.1	1.5	1.5	2.2
Multiple Nagios Servers	no	no	yes	yes
Display hosts/services with problems	yes	yes	yes	yes
Display hosts/services with no problem	no	yes	yes	yes
Auto reload	yes	yes	yes	yes
Manual Reload	yes	no	yes	yes
Commands regarding problems	no	no	yes	yes
Other commands	no	no	no	yes
Widget	no	yes	yes	yes
HTTPS	yes	no	yes	yes
Compatible Nagios versions	3.x	3.x	2.x, 3.x	3.x

Table 1: Competitor Software Comparision

4.2 Architecture

In order to satisfy the requirements, a basic architecture has been developed. Figure 10 views the main aspects of the entire architecture in a simplified way (i.e. only base classes are shown).

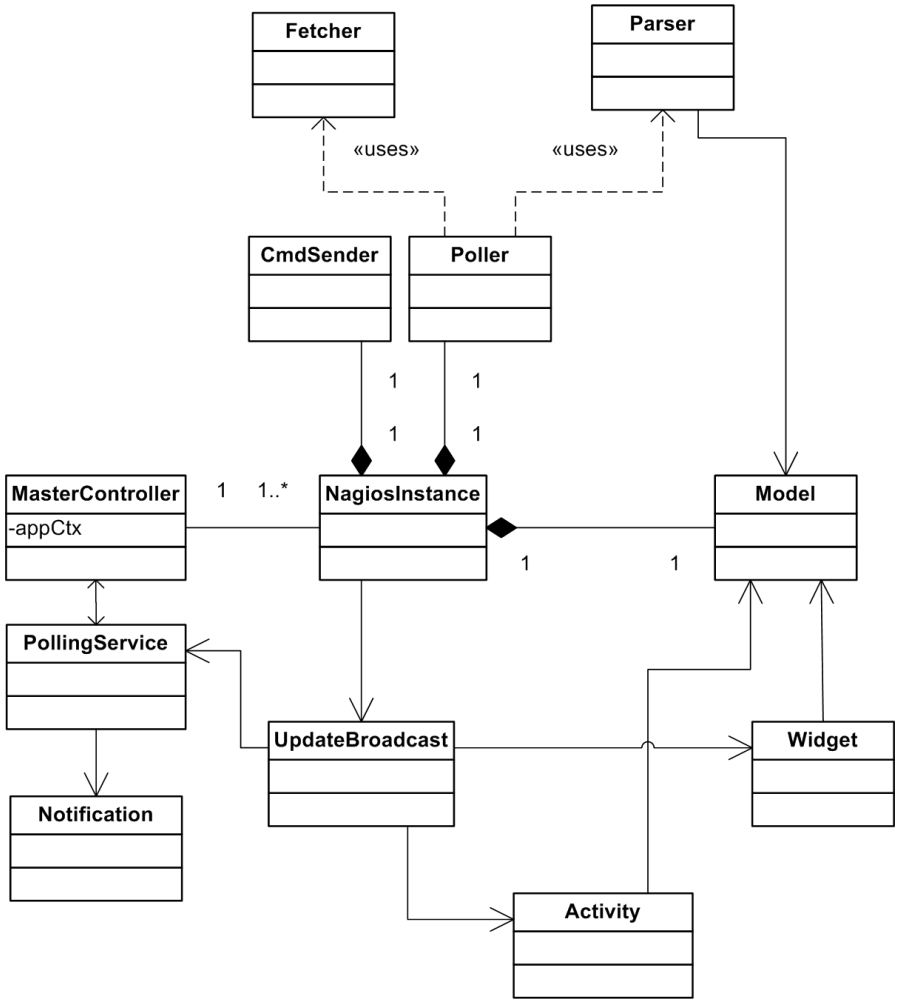


Figure 10: Base Architecture

The class *NagiosInstance* represents a Nagios server. It holds basic information about this Nagios Server such as the URL and credentials. Furthermore it stores user preferences related to this Nagios server instance. Whenever new information from the Nagios server should be fetched, the *NagiosInstance* class uses its *Poller* class. Goal of this action is to crate an updated model of the current network state and to inform view components about the change. In the first step the *Poller* gathers the desired information in HTML form using a *Fetcher* class. Once the HTML

representation is available, it needs to be transformed and written in a model class, since the HTML representation is not a proper way to share information within Java code. A *Parser* class is used to fulfill this transformation task. When the transformation is done, view components have to be informed about the update. For this reason an update broadcast is sent. Registered broadcast receivers (any open activity screen, the polling service, or the widget) can then read the updated model and refresh the information they view. Due to the fact that there are not only display requirements but also management requirements the *NagiosInstance* class can use a *CmdSender* class to send external commands to the Nagios server.

The *MasterController* class holds information which is not only of interest for a particular *NagiosInstance*, but Application wide. This is e.g. the application *Context* class, or user preferences such as the poll free interval. The *MasterController* is notably able to store multiple *NagiosInstance* classes. However, the current implementation of Androgios does only support one *NagiosInstance*. The ability to store multiple *NagiosInstance* classes is intended to be used in conjunction with future improvements, in order to avoid major architectural changes. Another important issue of the *MasterController* is to enable and disable the polling service.

The *PollingService* class gets a reference to the *NagiosInstance* by using the *MasterController*. On the basis of the user preference it periodically initiates a poll as described before. That way the *PollingService* does not need to inform view components about updates, this is done automatically by *NagiosInstance*. The only issue left is to alert the user in case of network problems. This is done by the *PollingService* itself.

4.3 Implementation

This subsection describes the concrete implementation of the architecture in a much more detailed fashion. It also discusses which challenges arose and how they were solved.

4.3.1 Android Manifest

A manifest file is necessary for each Android application. This file must be located in the projects root directory. It declares the minimum API Level, user permissions and each component used. Listing 3 shows Androgios' manifest file.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3        package="at.jku.koppensteiner.androgios"
4        android:versionCode="1" android:versionName="1.0">
5      <uses-sdk android:minSdkVersion="3" />
6      <uses-permission android:name="android.permission.INTERNET" />
7      <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
8      <uses-permission android:name="android.permission.VIBRATE" />
9      <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
10
11     <application android:icon="@drawable/icon_green" android:label="@string/app_name">
12
13         <activity android:name=".AndrogiosActivity" android:label="@string/app_name">
14     <intent-filter>
15         <action android:name="android.intent.action.MAIN" />
16         <category android:name="android.intent.category.LAUNCHER" />
17     </intent-filter>
18         </activity>
19
20         <service android:name=".service.PollingService">
21     <intent-filter>
22       <action android:name=".service.PollingService"></action>
23       </intent-filter>
24         </service>
```

```
25
26          <receiver android:name=".widget.AndrogiosWidget">
27            <intent-filter>
28              <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
29          </intent-filter>
30          <meta-data android:name="android.appwidget.provider"
31                     android:resource="@xml/androgios_widget_providerinfo" />
32          </receiver>
33
34          <receiver android:name=".service.BootReceiver" android:exported="true">
35      <intent-filter>
36        <action android:name="android.intent.action.BOOT_COMPLETED" />
37        <category android:name="android.intent.category.DEFAULT" />
38      </intent-filter>
39    </receiver>
40
41          <activity android:name="ConfigActivity"></activity>
42          <activity android:name="NavigationActivity"></activity>
43          <activity android:name="MonitoringFeaturesActivity"></activity>
44          <activity android:name="CommentDetailActivity"></activity>
45          <activity android:name="DowntimeDetailActivity"></activity>
46          <activity android:name="DowntimesActivity"></activity>
47          <activity android:name="HostCommentsActivity"></activity>
48          <activity android:name="HostDetailActivity"></activity>
49          <activity android:name="HostGroupsActivity"></activity>
50          <activity android:name="HostListActivity"></activity>
51          <activity android:name="HostListFilterActivity"></activity>
52          <activity android:name="HostProblemsActivity"></activity>
53          <activity android:name="OutageDetailActivity"></activity>
54          <activity android:name="OutagesActivity"></activity>
55          <activity android:name="ProcessDetailActivity"></activity>
56          <activity android:name="ServiceCommentsActivity"></activity>
57          <activity android:name="ServiceDetailActivity"></activity>
58          <activity android:name="ServiceGroupsActivity"></activity>
59          <activity android:name="ServiceListActivity"></activity>
60          <activity android:name="ServiceListFilterActivity"></activity>
61          <activity android:name="ServiceProblemsActivity"></activity>
62      </application>
63  </manifest>
```

Listing 3: Manifest

Androgios' manifest file declares API Level 3 as minimum required. An API Level smaller than 3 is not applicable, because it is the minimum requirement for the widget (Widgets were introduced with API Level 3).

Then the four permissions INTERNET (to do the polling), RECEIVE_BOOT_COMPLETED (to autostart Androgios), VIBRATE (to notify the user) and ACCESS_NETWORK_STATE (to detemine available network connections), are declared. Android applications should not be overprivileged. The tool Stowaway discussed in [FCH+11] (available: http://www.android-permissions.org/)was used to check if the principle of least privilege is satisfied. Figure 11 shows the output of Stowaway.

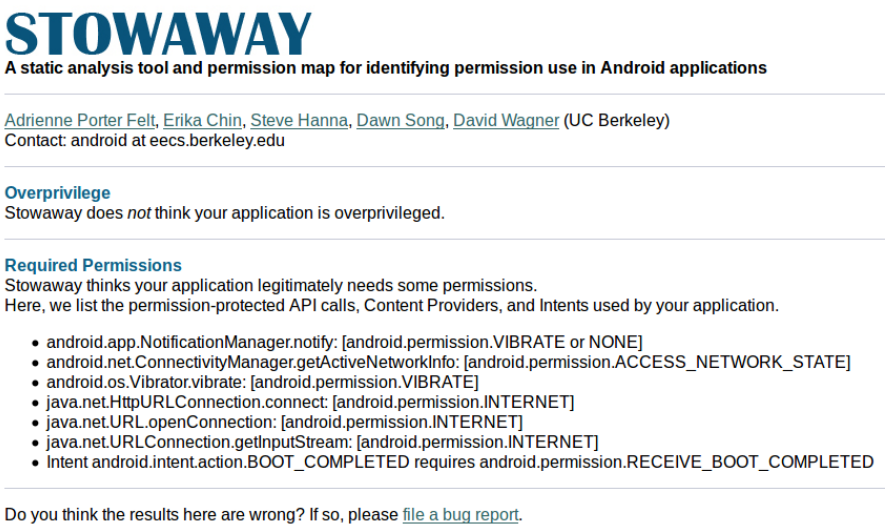


Figure 11: Stowaway output

Additionally all components used by Androgios are declared: The main activity, the polling service, the widget, the boot receiver and all other activities as discussed in section 4.3.8.

In order to make an activity the main activity, a special intent filter has to be declared (Listing 3 line 13 to 18). The Widget is declared as an update receiver (Listing 3 line 26 to 32) in order to enable the widget to get information about changes in the model.

4.3.2 Model

Network state information which is represented in HTML format needs to be parsed into a model. This model must be suitable to share the network state information between different components. Androgios distinguishes two different models: The *TacticalOverview* and the *MonitoringModel*. The *TacticalOverview* only consists of information fetched from the tactical overview URL (i.e. tac.cgi). This information is sufficient to display the start screen, display the widget and also to determine if important problems arose. Figure 12 shows the composition of the *TacticalOverview* model. The HTML file delivered from tac.cgi is also quite small (typically less than 20kB). Therefore the *TacticalOverview* is the model used when the polling service processes its continuous task. Due to the fact that there are separate models used for overview and detailed data, the polling service and user initiated polls do not interfere.

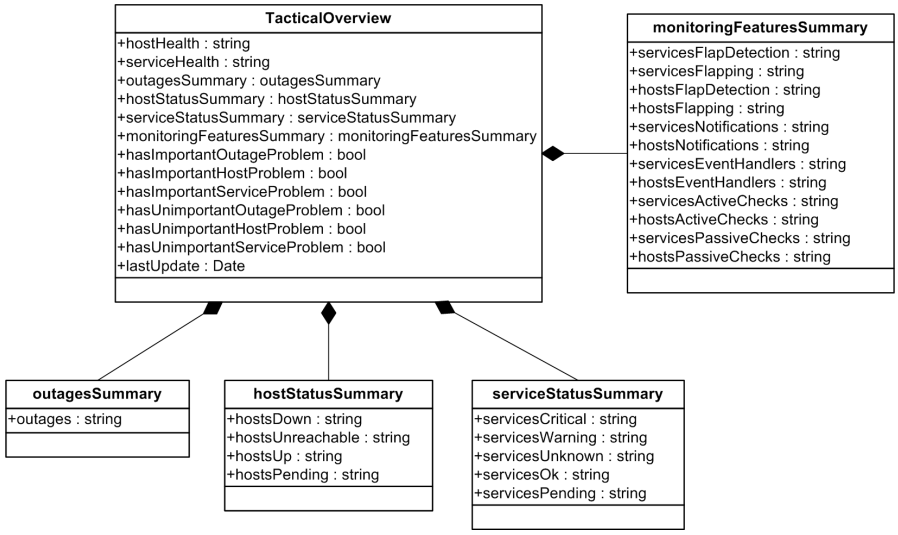


Figure 12: Tactical Overview Model

The more detailed *MonitoringModel* showed in figure 13 is used by displaying components that need more detailed information. The *MonitoringModel* essentially consists of several lists of: hosts, hostgroups, servicegroups, downtimes, comments and outages as well as a process object which represents the Nagios server process. Whenever a specific activity needs certain data in the model, it should not be necessary to fetch HTML from more than one URL. Therefore lists containing comments or downtimes are held by the *MonitoringModel* itself. The fields host name respectively service name are provided for both cases by Nagios. So there is no need to build up a big data structure that holds all hosts/services. This would cause overhead in memory and time complexity. This argument is not true for the list of services. Nagios does not provide a field for services that would allow to identify the host it belongs to. However the service list provided by Nagios does also contain the according host names. For that reason the list of services is not held by the *MonitoringModel*, but by the *Host* class. That way it is implicitly clear to which host a service belongs and no mapping information has to be maintained.

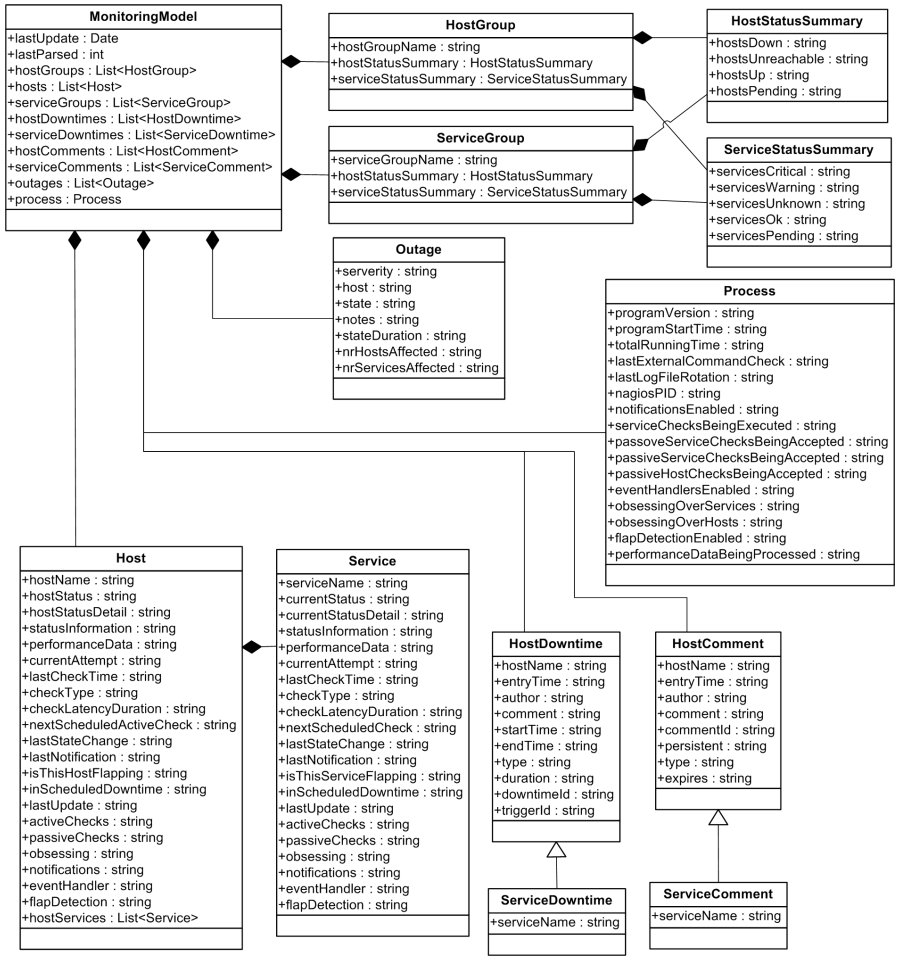


Figure 13: Detailed Monitoring Model

In contrast to the *TacticalOverview* the *MonitoringModel* is never complete. It does only hold information currently needed by the active view. That way the *MonitoringModel* can be kept as small as possible and consistency can be ensured easily. Most parsers create the *MonitoringModel* from scratch. An exception is when details of a particular host is parsed. This shall not destroy an already existing list of hosts. The same is true for services (see also section 4.3.5).

Building a model may take a while. A model is therefore always built within

a separate thread to avoid blocking the GUI thread. Android would display an ”application not responding” error message if the GUI thread blocked for more than 5 seconds. Parallel model building arises the problem that changes may occur while a displaying component wants to read the model. To handle this issue, data held by a model is never changed after is has been built. For the case that parts of the current model needs to be preserved while updating, the whole model can be deep copied by calling the *clone()* method. This immutability avoids the necessity of locking while reading. When a model change finishes a broadcast is sent to trigger an update of UI elements.

4.3.3 Polling

The purpose of polling is to update model and view based on newly fetched data. The *Poller* is implemented as an inner class of *NagiosInstance*. There are two poll methods implemented: One for polling the *TacticalOverview* and the second for polling (a part of) the *MonitoringModel*. The *poll()* methods are responsible for several tasks:

- Check if a network connection is currently available.
- Determine the correct URL.
- Fetch the HTML from that URL.
- Initiate the according parser.
- Send broadcasts to the Widget and other UI elements.

Polling of the TacticalOverview: At this point, first the connectivity has to be checked. The actual connectivity check is implemented in the *NagiosInstance* class. In case of no connectivity a flag in the model is set to indicate an unsuccessful update attempt. Update broadcasts are sent immediately in this case.

A user has to provide a base URL which points to the cgi-bin folder of the Nagios server. To determine the correct URL for fetching the *TacticalOverview* is easy. It's only baseURL + tac.cgi. In order to fetch the HTML one of two fetcher types may be used, HTTP or HTTPS. Both types implement the interface *Fetcher* (see section 4.3.4). Once the HTML representation is known, the *TACParser* is used to build the new *TacticalOverview* (see section 4.3.5). As last step two update broadcasts are sent. One to inform all activities displaying *TacticalOverview* information and another to inform the Androgios widget. All these tasks are performed within their own thread where fetching, parsing and broadcast sending are synchronized on the *TacticalOverview* object.

Polling of the MonitoringModel: This is similar to the polling process of the *TacticalOverview*. The main difference is that this method needs a parameter of the URL to process. On the basis of this parameter the corresponding parser can be determined (see also section 2.2 for details on cgi URLs). However the URL does not always exactly tell what parser should be used or how it should be initialized. E.g. the URLs for the details of two different hosts are of course not equal, since the URLs consist of different host names. Nevertheless the same parser is needed. This can be resolved by only comparing the starting sequence("extinfo.cgi?type=1&host=") and ignoring the host specific part. A similar problem is that the URL for the full list of hosts starts with the same sequence as a filtered list of hosts. Additionally the list of host problems is a special case of the

filtered list. To overcome this, the order of the case distinction is important: First is has to be checked if the URL equals the host problems URL, second is has to be checked if the URL equals the URL for the full host list and third it has to be checked if the URL starts with the URL for the full host list. To not respect this order would cause a wrong initialization of the *HostListParser* i.e. the constructor flags *problemsOnly* and *filtered* could not be set correctly. The same problems are true for services and they can be treated the same way.

4.3.4 Fetching

Goal of the *Fetcher* is to provide a HTML string delivered by the Nagios server. Due to the fact that Androgios must be able to handle secure and insecure connections, the interface *Fetcher* is used. This interface only demands to implement a *fetchHTML()* method which delivers the HTML string.

HttpFetcher: This implementation needs the URL to fetch and additionally username / password for authentication purpose. Authentication can be done by calling the *setRequestProperty()* method of a *HttpURLConnection* object. The "username : password" combination has to be Base64 encoded in basic authentication mode. Since there is no Base64 encoder available in the Java library the open source implementation [Heu10] is used.

Tests have shown that HTML could often not be fetched at the first attempt, especially when the mobile network connection is in use. The reason for this behavior is not entirely clear but only an observation. Therefore by default HTML fetching is tried three times. This procedure improves the rate of successful attempts significantly.

HttpsFetcher: This implementation needs an additional parameter which signals if self signed certificates should be accepted. By default the Java *HttpsURLConnection* class does not accept self signed certificates. However, self signed certificates are widely used, especially for "internal" websites. To accept those certificates two things need to be done: First a *TrustManager* has to be installed. This is done when the corresponding user preference is set (or read on startup). Second a *HostnameVerifier* has to be set that accepts the host name. Since the permanent import of new root certificates is currently not supported by Android (at least not on "non-rooted" devices) an "all accepting" host name verifier is implemented in a *SSLHelper* class. The *setHostNameVerifier()* method of the *HttpsURLConnection* object is called only if the according flag is set. In case of a certificate related error a SSLException is thrown.

4.3.5 Parsing

A parser has to transform HTML represented information in a model. Every parsing process involves only one HTML string. The following HTML representations have to be parsed:

- TacticalOverview
- Host list
- Host groups
- Host details
- Service list
- Service groups

- Service details
- Outages
- Comments
- Downtimes
- Nagios process

For maintainability reasons a separate parser for each HTML representation is implemented. However, representation differences due to different Nagios versions have to be handled by one parser. The parsers extend an abstract base class called *Parser*. Subclasses of *Parser* must implement the method *updateModel()*.

No parser may alter an existing model. Each parser needs to create a new model and set the according field in the *NagiosInstance* when the whole new model is created. That way a model update appears as an atomic operation and displaying components do not need to get a lock on the model. For displaying components it is sufficient to store their own reference to the model (see also section 4.3.8). Most parsers do not need to preserve old model information. This is however not true for host detail respectively service detail parsers. Both extend an already created list of hosts/services. To ensure that, once created, a model is immutable in those cases, it is necessary that parsers are able to create deep copies of the model. All classes belonging to the model therefore implement the *clone()* method.

For the actual parsing the open source library jsoup is used. Jsoup is distributed under the MIT license. It allows to create a DOM easily and extract data, using DOM traversal, CSS selectors or jquery like methods, according to [Hed12]. The complete API reference of jsoup can be found at the jsoup homepage.

4.3.6 Sending External Commands

A prerequisite for sending external commands is to have the line ”check_external_commands=1” in the Nagios server configuration. Otherwise the Nagios server would reject all commands sent. External commands are sent to the Nagios server via HTTP POST method. In order to create the correct POST data string, the utility class *PostDataBuilder* has been implemented. It is capable to provide POST data strings for all the external commands demanded by the management requirements in section A.

The correct format of the commands is not well documented. The tables 2 and 3 are available in [Bar09] and show possible values for the parameter ”cmd_typ” which indicates what command should be executed. (Androgios does not implement all commands shown.)

Global Parameter	Start/Enable	Stop/Disable
NOTIFICATIONS	11	12
SVC_CHECKS	35	36
ACCEPTING_PASSIVE_SVC_CHECKS	37	38
HOST_CHECKS	88	89
ACCEPTING_PASSIVE_HOST_CHECKS	90	91
EVENT_HANDLER	41	42
FLAP_DETECTION	61	62
PERFORMANCE_DATA	82	83

Table 2: Command Types for global parameters(from [Bar09])

Command	Host	Service	HostGroup	ServiceGroup
ADD_HOST_COMMENT	1	-	-	-
DEL_HOST_COMMENT	2	-	-	-
DEL_ALL_HOST_COMMENT	20	-	-	-
ADD_SVC_COMMENT	-	3	-	-
DEL_SVC_COMMENT	-	4	-	-
DEL_ALL_SVC_COMMENT	-	21	-	-
ENABLE_ACTIVE_SVC_CHECK	15	5	67	113
DISABLE_ACTIVE_SVC_CHECK	16	6	68	114
SCHEDULE_SVC_CHECK	17	7	-	-
ENABLE_ACTIVE_HOST_CHECK	47	-	103	115
DISABLE_ACTIVE_HOST_CHECK	48	-	104	116
SCHEDULE_HOST_CHECK	96	-	-	-
ENABLE_HOST_NOTIFICATIONS	24	-	65	111
DISABLE_HOST_NOTIFICATIONS	25	-	66	112
DELAY_HOST_NOTIFICATIONS	10	-	-	-
ENABLE_SVC_NOTIFICATIONS	28	22	63	109
DISABLE_SVC_NOTIFICATIONS	29	23	64	110
DELAY_SVC_NOTIFICATIONS	19	9	-	-
ACKNOWLEDGE_PROBLEM	33	34	-	-
REMOVE_ACKNOWLEDGE	51	52	-	-
ENABLE_PASSIVE_HOST_CHECKS	92	-	107	119
DISABLE_PASSIVE_HOST_CHECKS	93	-	108	120
ENABLE_PASSIVE_SVC_CHECKS	-	39	105	117
DISABLE_PASSIVE_SVC_CHECKS	-	40	106	118
SCHEDULE_HOST_DOWNTIME	55	-	84	121
DEL_HOST_DOWNTIME	78	-	-	-
SCHEDULE_SVC_DOWNTIME	-	56	85	122
DEL_SVC_DOWNTIME	-	79	-	-
ENABLE_EVENT_HANDLER	43	45	-	-
DISABLE_EVENT_HANDLER	44	46	-	-
ENABLE_FLAP_DETECTION	57	59	-	-
DISABLE_FLAP_DETECTION	58	60	-	-

Table 3: External Command Types (from [Bar09])

The parameter ”cmd_typ” is not the only required one. ”cmd_mod” is always set to the value 2 which means the command is immediately committed, without further confirmation. Additional parameters depend on the command itself.

Host commands always require the parameter ”host” to indicate the host name. The following host commands require additional parameters:

ADD_HOST_COMMENT:
”com_data” is an arbitrary string value representing the comment. ”com_author” indicates who has written the comment. Androgios uses the login name from its configuration for this parameter. ”persistent” indicates whether the comment should remain once the acknowledgement is removed. Its value is either ”on” or the parameter is omitted.

DEL_HOST_COMMENT:
”com_id” identifies a particular comment. Its an integer value delivered from the Nagios server.

SCHEDULE_HOST_CHECK:
”start_time” is a date string indicating when the check should be executed. ”force_check” indicates if Nagios will force a check of the host regardless of both, what time the scheduled check occurs and whether checks for the host are enabled or not. Its value is either ”on” or the parameter is omitted.

ACKNOWLEDGE_HOST_PROBLEM:
”com_data”, ”com_author” and ”persistent” are treated the same way as when adding a host comment. ”sticky_ack” indicates to disable notifications until the host recovers. Its value is either ”on” or the parameter is omitted. ”send_notification” indicates whether an acknowledgement notification is sent out by the Nagios server.

Its value is either ”on” or the parameter is omitted.

DISABLE/ENABLE_HOST_NOTIFICATIONS:
”ptc” indicates whether the command should be propagated to child hosts. Its value is either ”on” or the parameter is omitted.

SCHEDULE_HOST_DOWNTIME:
”com_author” and ”com_data” are treated the same way as when adding a host comment. ”start_time” and ”end_time” are date strings indicating the downtime period. ”fixed” indicates whether the downtime starts at a given time (fixed) or at the time when the host becomes unavailable (flexible). Its value is either ”0” (flexible downtime) or ”1” (fixed downtime). ”hours” and ”minutes” indicate the duration of a flexible downtime. Both are omitted in case of a fixed downtime. ”childoptions” indicates how to handle child hosts. Its value is either ”0” (do nothing), ”1”(schedule triggered downtime for all child hosts) or ”2”(schedule non-triggered downtime for all child hosts).

DEL_HOST_DOWNTIME:
”down_id” identifies a particular downtime. Its an integer value delivered from the Nagios server in the downtime list.

DISABLE/ENABLE_ACTIVE_SVC_CHECK
DISABLE/ENABLE_SVC_NOTIFICATIONS:
These commands affect all services running on the host. The parameter ”ahas” indicates whether the command should affect the host too. Its value is either ”on” or the parameter is omitted.

SCHEDULE_SVC_CHECK:
Does also affect all services running on the host. Both parameters, ”start_time”

and ”force_check”, are treated the same way as when scheduling a host check.

Service commands always need the parameters ”host” and ”service” to indicate host name and service name. The commands SCHEDULE_SVC_CHECK, ACKNOWLEDGE_SVC_PROBLEM and SCHEDULE_SVC_DOWNTIME require the same additional parameters as the corresponding host commands do.

4.3.7 Date Formats

Nagios supports four different date formats. It is important to recognize that the Nagios documentation does not correspond with the actual implementation.

The following formats are documented:

- us (MM/DD/YYYY HH:MM:SS e.g. 06/30/2002 03:15:00)
- euro (DD/MM/YYYY HH:MM:SS e.g. 30/06/2002 03:15:00)
- iso8601 (YYYY-MM-DD HH:MM:SS e.g. 2002-06-30 03:15:00)
- strict-iso8601 (YYYY-MM-DDTHH:MM:SS e.g. 2002-06-30T03:15:00)

Nagios does implement the following formats:

- us (MM-DD-YYYY HH:MM:SS e.g. 06-30-2002 03:15:00)
- euro (DD-MM-YYYY HH:MM:SS e.g. 30-06-2002 03:15:00)
- iso8601 (YYYY-MM-DD HH:MM:SS e.g. 2002-06-30 03:15:00)
- strict-iso8601 (YYYY-MM-DDTHH:MM:SS e.g. 2002-06-30T03:15:00)

Androgios must format date parameters according to the Nagios configuration, in order to ensure that commands work properly. This is done by the *DateHelper* class. This issue additionally requires the user to configure the correct date format in the Androgios configuration.

4.3.8 Activities

Most activities in Androgios are used to present model information to the user. All these activities have the same user interface design and similar needs regarding the handling of model updates. For this reason these activities are subclasses of the abstract base class *AbstractModelActivity*.

AbstractModelActivity implements a model update receiver. This receiver is registered when the activity is created and unregistered when the activity is destroyed. A reference to the *NagiosInstance* is also stored in the base class. Every activity needs a standard menu, that is also implemented here. The standard menu includes buttons for refreshing, starting the configuration activity and for enabling/disabling the polling service. Two methods are responsible for handling updates: *update()* that does only update the activities viewing components without altering the model and *poll(url)* that initiates the polling process. As shown in figure 14 after an initiated poll has finished, a broadcast is received and the receiver calls the *update()* method. When the activity is created a decision has to be made which of the both methods should be called. The *poll(url)* method may also be called when the user presses the refresh button from the standard menu when the activity is in running state.

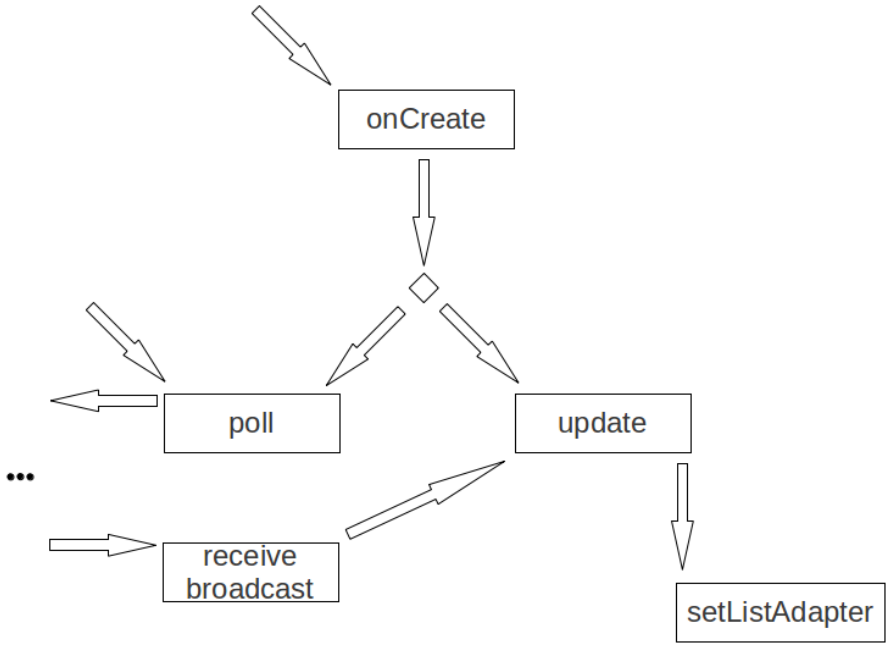


Figure 14: Typical Life Cycle

AbstractModelActivity implements for both *update()* and *poll(url)* small parts that are always treated equally such as setting header texts. Subclasses have to extend both methods according to their specific needs. The decision of which of the two methods have to be called after instantiation, has to be implemented by the subclasses too.

Main task of the subclasses that extend *AbstractModelActivity* is to display the model information. Each subclass has therefore its own *BaseAdapter* implementation (as inner class) to fill a *ListView*. An extension of *BaseAdapter* needs to implement the following methods: *getCount()*, *getItem(position)*, *getItemId(position)* and *getView(position,...)*. The most interesting method here is *getView(position,...)*. It specifies what to display at the given position. In this method the model is accessed in order to create a proper *View*. The maximum position is determined by

the return value of *getCount()*. Android does not allow to alter data that is used to compute list entries from within another thread (if Android detects such situations an IllegalStateException is thrown). However, the time intensive procedures for fetching and parsing are done from within separate threads. To avoid unexpected changes in data each adapter has its own reference to the model. This reference is never changed. Whenever the *update()* method is called, a new adapter is instantiated. The new adapter is then set on the *ListView*. Although the *NagiosInstance* class provides constants for the URLs it may be necessary to append parameters such as a host name. This must be implemented in the *poll(url)* method.

Some activities allow to send external commands. These activities have a slightly different layout of the user interface: They implement a *Spinner* (the Android version of a drop down list) to choose a command and a button at the bottom to finally send the command. To fill the *Spinner* with proper commands another implementation of *BaseAdapter* is used. The listener for the button that sends the command needs to read what command is chosen. As pointed out in section 4.3.6 there are several commands that need additional parameters. Therefore the button listener displays a custom dialog to ask the user which parameter values should be used. When all parameter values are known the utility class *PostDataBuilder* is used to create the correct parameter string and finally the command is sent using the *sendCmd(Postdata)* method of the *NagiosInstance*. After the message is sent the result message from the Nagios server is presented to the user. This information helps the user to decide if he should retransmit the command (in case of an error) or if he should do a refresh after a few moments. Nagios uses a queue to process commands, therefore it may take some time until the command is processed. Therefore it is not possible to refresh automatically in a reliable way.

Host and service lists can be filtered by Nagios itself. The GET parameters `host-`

`statustypes`, `servicestatustypes`, `hostprops` and/or `serviceprops` have to be appended to the URL in order to tell Nagios to do so. The parameter values are integer types. The tables 4, 5, and 6 show the meaning of the integer values.

Value	Meaning
1	PENDING
2	UP
4	DOWN
8	UNREACHABLE

Table 4: Host Status Types

Value	Meaning
1	PENDING
2	OK
4	WARNING
8	UNKNOWN
16	CRITICAL

Table 5: Service Status Types

Value	Meaning
1	in scheduled downtime
2	not in scheduled downtime
4	acknowledged
8	not acknowledged
16	checks disabled
32	checks enabled
64	eventhandler disabled
128	eventhandler enabled
256	host flap detection disabled / service flap detection enabled
512	host flap detection enabled / service flap detection disabled
1024	flapping
2048	not flapping
4096	notifications disabled
8192	notifications enabled
16384	passive checks disabled
32768	passive checks enabled
65536	passive check result
131072	active check result

Table 6: Host/Service Properties for filtering

Androgios uses separate activities to ask the user about the filter criteria. These activities simply present checkboxes for choosing the criteria. The checkboxes can

be transformed into the correct integer values, as shown in listing 4. Since all the values are a power of two, they can be added to combine them. Nagios treats combinations in status types and properties differently: While multiple selection in status types are treated as logical OR, multiple selection in properties are treated as logical AND. An example: if hoststatustypes is set to 6 (2..UP, 4..DOWN) the resulting list will contain hosts that are either in state UP or in state DOWN, the result increases the more selections are made. On the other hand if hostprops is set to 3072 (1024..FLAPPING, 2048..NOT FLAPPING) the resulting list is empty, because no host can have both properties at the same time.

```
1  public static int getFilterParam(CheckBox[] checkBoxes){
2    int param = 0;
3    for(int i=0;i<checkBoxes.length;i++){
4      if(checkBoxes[i].isChecked()){
5        param +=   0x1<<i;
6      }
7    }
8    return param;
9  }
```

Listing 4: Creating Filter Parameters

4.3.9 Resources

Resources such as different layouts or images sould always be stored separately to ensure that they can be maintained without changes in the source code.

The Androgios project places different resource types in specific subdirectories of the projects ”res” folder. The structure looks as follows:

```
androgios/
  src/
   ...
  res/
   drawable/
    icon_green.png
    widget_background.png
    ...
   layout/
    abstract_model_cmd_layout.xml
    abstract_model_layout.xml
    main_activity_layout.xml
    androgios_widget_layout.xml
    ...
   raw/
    msg.wav
   values/
    downtime_types.xml
    preference_values.xml
    strings.xml
   xml/
    androgios_widget_providerinfo.xml
    config.xml
```

The drawable folder contains graphics such as icons in different colors and the background image of the widget.

The layout folder contains definitions for different activity layouts. Layouts are defined in XML and describe how all the UI elements are nested. Activities that offer to send external commands share the same layout (abstract_model_cmd_layout.xml) and activities that only show model information also share a common layout (abstract_model_layout.xml). All other activities have their own layout (e.g. main_activity_layout.xml). A layout must also be defined for the widget (androgios_widget_layout.xml).

The raw folder only contains the audio file to play when the user has to be alerted.

The XML files in the values folder provide strings that have to be displayed anywhere. Downtime_types.xml provides two string arrays that are used to fill the two spinner elements of the dialog that is shown when the user schedules a new downtime for a host or a service. Preference_values.xml provides string arrays that are used when the user configures poll interval or the Nagios date format. Strings.xml provides all other strings displayed in Androgios, except the strings displayed within the preferences activity (these strings are defined in xml/config.xml). Androgios does not display any strings that are written directly in the source code. Putting string values in resource files brings the advantage that internationalization can be done in a simple way. Section 5.1 describes how to add further languages. In order to use the string provided as resource, subclasses of *Activity* have a *getString(resID)* method. The resource ID comes from the generated class R, described in section 3.3.1. For more convenience the *setText()* method of *TextView* objects is overloaded so that it accepts resource IDs.

The XML folder contains the androgios_widget_providerinfo.xml that essentially defines what layout resource has to be used for the widget. The config.xml file provides the information needed to create the layout of a *PreferenceActivity* auto-

matically. This may also include sub-activities.

4.3.10 Preferences

The following preferences are available to the user:

- Nagios URL to the cgi-bin folder (Text)
- Nagios User (Text)
- Nagios Password (Text/Pwd)
- Accept self signed certificates (Boolean)
- Polling interval (ListEntry)
- Nagios date format (ListEntry)
- Play sound (Boolean)
- Vibration (Boolean)
- Alarm on update failed (Boolean)
- Use mobile network (Boolean)
- Poll free interval (DateTime 2x)
- Alarm free interval (DateTime 2x)
- Auto start after reboot (Boolean)

Android provides the base class *PreferenceActivity*. This class can be used to create configuration related activities easily. Androgios' *ConfigActivity* extends

this base class. The layout can be created by calling the *addPreferencesFromResource(R.xml.config)* method. The only functionality that is automatically provided by this activity is that changes are stored on the sd-card of the Android device. Therefore the *ConfigActivity* adds for each preference an *OnPreferenceChangeListener*. This is necessary to be able to react on preference changes immediately, without the need of an Androgios restart.

The resource file `config.xml` defines all preferences displayed. Androgios uses different kinds of preferences:

- EditTextPreference - stores a freely selectable string
  (e.g. Nagios URL)
- ListPreference - stores a from a list chosen string value
  (e.g. Nagios date format)
- CheckBoxPreference - stores a boolean value
  (e.g. Accept self signed certificates)
- TimePreference - stores a string that represents a time value
  (e.g. Poll free interval start/end)

The elements *PreferenceCategory* and *PreferenceScreen* may be used to organize a *PreferenceActivity*. A *PreferenceCategory* adds a header, a *PreferenceScreen* adds a sub-activity. Listing 5 shows an excerpt of the file `config.xml` and how to integrate the preference elements. An example for every kind of preference used in Androgios is present in the excerpt. Self defined preferences, such as the *TimePreference* must be specified with their fully qualified name.

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Nagios">
        <EditTextPreference android:key="configuration_nagios_url"
    android:title="Nagios Url" android:dialogTitle="Url (cgi-bin)"
    android:dialogMessage="Specify the url pointing to your nagios cgi-bin directory"
    android:defaultValue="http://localhost/cgi-bin/nagios3" android:singleLine="true"/>
...
  <CheckBoxPreference android:title="Accept SelfSigned Cert"
    android:key="configuration_acc_self_cert"
    android:summaryOff="do not accept"
    android:summaryOn="accept"></CheckBoxPreference>
...
  <ListPreference android:key="configuration_poll_interval"
    android:entries="@array/pollIntervalsEntries"
    android:title="Polling Interval"
    android:entryValues="@array/pollIntervalsValues"
    android:defaultValue="300"></ListPreference><ListPreference
    android:entries="@array/dateFormatEntries"
    android:entryValues="@array/dateFormatValues"
    android:title="Nagios Date Format"
    android:key="configuration_date_format"
    android:defaultValue="iso8601"></ListPreference>
...
  <PreferenceScreen android:title="Alarm Free Intervals"
    android:summary="manage alarm free intervals">
    <PreferenceCategory android:title="Poll Free Interval">
...
      <at.jku.koppensteiner.androgios.util.TimePreference
        android:title="Start Time"
        android:key="configuration_poll_free_start"
        android:defaultValue="00:00"></at.jku.koppensteiner.androgios.util.TimePreference>
...
        </PreferenceScreen>
...
```

Listing 5: Excerpt of config.xml

The *TimePreference* extends the base class *DialogPreference*. It presents a *TimePicker* to the user and persists the chosen value as a string value when the dialog is closed. That way time preferences can be set more convenient and syntactically incorrect

entries can be avoided.

On Android startup it is necessary that the stored values are read. This is done by the *MasterController* and discussed in section 4.3.11.

As already mentioned *OnPreferenceChangeListeners* are responsible for handling changes. Two types of preferences may be distinguished. The first type applies to Androgios, all preferences that are not child of the *PreferenceCategory* "Nagios". The second type applies to a specific *NagiosInstance*. These are all preferences that are child of the *PreferenceCategory* "Nagios". The *MasterController* provides public variables for setting first type preferences. The *NagiosInstance* provides public variables for setting second type preferences.

4.3.11 MasterController

The *MasterController* handles application wide issues. It is implemented as singleton. This ensures that only one instance is created and that the MasterController is available from each point of code.

As a central component the *MasterController* manages all *NagiosInstances*. Although there is only one *NagiosInstance* supported currently, the *MasterController* holds a list of *NagiosInstances*. The *getActiveNagiosInstance()* method returns a reference to the first entry. The list is intended for future use. Supporting multiple *NagiosInstances* is discussed in more detail in section 5.1.

The *ApplicationContext* can be obtained via the method *getAppCtx()*. The Android API provides many methods, that need a *Context* object as an argument, e.g. when displaying a toast message (the Android version of a pop up window). Dealing with *Context* objects is hard when they are needed outside of Activities

(extensions of the *Activity* class have their own method to get a context object). Providing the *ApplicationContext* at a central location simplifies this issue. The component that starts Androgios must set the *ApplicationContext* at the *MasterController*. This can be either the main *Activity* or the *BootCompleteReceiver* that starts the polling service after a reboot.

The *MasterController* also provides convenience methods for starting, stopping and restarting the polling service. The *ApplicationContext* is used to start or stop a service.

Alarm free and poll free intervals are handled by the *MasterController*. Methods for setting start- and end times and also to enable or disable these intervals are implemented. Whether the current time is within one of these intervals can be determined by the usage of the method *withinAlarmFreeInterval()* or *withinPollFreeInterval()*. Both methods check whether the current time is within the corresponding interval and in addition it checks whether it is enabled. Only if both is the case, these methods will return true.

The *MasterController* must read the Androgios configuration on startup and set all variables accordingly. This cannot be done when the singleton is instantiated, because the *ApplicationContext* is not known, but required at this point of time. Since the Androgios starting component sets the *ApplicationContext*, this call can be used as trigger to read the configuration. A call of *PreferenceManager.getDefaultSharedPreferences(appCtx)* reads the configuration from file. This returns a *SharedPreferences* object that allows to get the configuration entries via methods such as *getString(String key, String defValue)* or *getBoolean(String key, String defValue)*. Once the configuration entries are known, the corresponding variables in the *MasterController* or *NagiosInstance* are set.

4.3.12 Polling Service

The polling service initiates regular model updates based on the users configuration. In addition it is responsible for alerting the user via a notification icon, sound, or vibration.

The user should be alerted as soon as a problem is detected. This requires to wake up Android from stand-by. A service cannot do this by itself. Androids *AlarmManager* class can be used to schedule future tasks even if Androgios is not running and Android is in stand-by. The static method *AlarmManager.set(int type, long triggerAtTime, PendingIntent operation)* allows to schedule a future task. If the task is already scheduled, the first will be canceled. The type *AlarmManager.ELAPSED_REALTIME_WAKEUP* is used to ensure Android will not remain in stand-by. To calculate the trigger time *SystemClock.elapsedRealtime()* is used, it includes time elapsed in stand-by. *System.currentTimeMillis()* is not ideal, because this time can be altered by calling *setCurrentTimeMillis(long)* so this time may jump unpredictably. The *PendingIntent* to perform when the trigger time is reached is simply the polling service itself.

When the polling service is started first time, it only sets the *AlarmManager* accordingly. When it is started next time, a polling thread is started. This thread first checks if the current time is within a poll free interval. If this is the case it will not make sense to schedule the next poll within this interval, due to energy efficiency reason. Therefore the next poll is scheduled at the end of the poll free interval. If the current time is not within a poll free interval, the polling service initiates a poll and sets the *AlarmManager* for the next check time. In addition a *BroadcastReceiver* is registered that handles the result as soon as the model update is available.

The preference ”alarmOnUpdateFailed” and the *TacticalOverview* are of interest when an update broadcast is received. The alerting procedure is executed in two cases: First if the polling was unsuccessful and the user set the preference to be alerted in this case. Second if the polling was successful and an important problem occurred. In case of unimportant problems, such as acknowledged problems, the user will not be altered. If there is no important problem an eventually displayed notification icon will be removed at this time.

The alerting procedure is implemented in the method *doAlarm()*. A notification icon is always displayed, because it will not disturb a user. Vibration and sound depend on the user preferences. Androids *NotificationManager* class helps to display icons on the status bar. To send an icon to the status bar a *Notification* object and an identification number is needed. If a notification icon with the same id is sent a second time, the old icon will be replaced. Therefore it cannot happen that the status bar is flooded with many icons with the same meaning. The notification id is also used to remove a certain icon. A *Notification* object allows to set a *PendingIntent*. Androgios uses this feature to start the main activity when the user clicks at the icon. While *Notification* objects can be instantiated, the *NotificationManager* can be obtained by calling *getSystemService(Context.NOTIFICATION_SERVICE)*, this method is inherited from *Service*. A *Vibrator* object can be obtained similarly by calling *getSystemService(Context.VIBRATOR_SERVICE)*. For playing sounds *RingtoneManager.getRingtone(Context context, Uri ringtoneUri)* returns a *Ringtone* object.

The user may want to start the polling service automatically with Android. This feature is implemented in the class *BootReceiver*. It is registered as broadcast receiver in the Androgios manifest for the action ”BOOT_COMPLETED”. If it receives this action and the corresponding preference is set to true, it will set the

ApplicationContext at the *MasterController* and enable the polling service.

4.3.13 Widget

The Androgios widget provides the time of the last model update and a quick overview of the host/service health and of outages. When the user clicks at the widget Androgios is started.

Widgets must extend the base class *AppWidgetProvider*. The Androgios widget implements the callback method *onReceive(Context c, Intent i)*. This method must ensure, that all displayed widgets are updated, because there may be more than one widget on the Android home screen. Multiple widgets will display the same information in the current implementation. In future multiple widgets could display different Nagios instances, see also section 5.1. To find out how many widgets are displayed, the *AppWidgetManager.getAppWidgetIds()* method can be used. It returns an array of widget IDs that are bound to the *AppWidgetProvider*.

To start Androgios when the user touches a widget click handlers need to be set. This is done by calling *setOnClickPendingIntent()* on a *RemoteViews* instance. The *PendingIntent* (which is actually the main activity) is then executed as soon as a click event occurs. Because the number of displayed widgets may vary between update broadcasts (e.g. the user added an additional widget meanwhile), this procedure is done on each update.

4.3.14 NagiosInstance

NagiosInstance is the class that supports all actions related to a Nagios Server, such as polling, sending commands or handling the network connection. It serves

as a wrapper around specialized classes like *Fetcher*, *Parser*, *CmdSender* and so on.

The *NagiosInstance* holds information about a Nagios server: the URL to the cgi-bin folder and credentials to log in (username and password). Besides this basic information, it also stores the polling interval, the date format configured at the Nagios server and if to accept self signed certificates in case of SSL connections. The *NagiosInstance* finally provides the model that is built by a poll operation.

Poll operations are done in a separate thread to not block the GUI. In addition to the base URL to the cgi-bin folder, an extension of the URL is required by the *poll(extURL)* method. *NagiosInstance* provides the URL extension as publicly accessible, static constants. The concatenation of base- and extended URL is the URL from where to fetch the HTML string. The extended URL is additionally used to determine what parser has to be instantiated. When the HTML is fetched and the correct parser is known, the actual parsing is done. Two errors may occur: A SSLException while fetching or a ParserException while parsing. In both cases two fields of the model are set accordingly: updateSuccessful and errorCause. This information is used by GUI elements to determine if a model is accurate. The final step of the poll operation is to send a broadcast that informs about the model update.

The *NagiosInstance* allows to send external commands. This is also done in a separate thread. Callers of the *sendCmd()* method must provide a proper command string to send. The utility class *PostDataBuilder* helps to create the command string. The correctness of the command string is not checked at this point. The command string is simply passed to the Nagios server. The Nagios server returns a success or error message in HTML form. Since it is unpredictable when the

resulting HTML is returned from the Nagios server, the *sendCmd()* returns only a message that the command has been sent. When the response of the Nagios server is available, it is displayed as toast message.

Both operations, polling and sending commands, require a network connection. Without checking the connectivity before doing one of the operations, waiting for timeouts and retries could cause long waiting times for the user and a waste of energy resources. In addition a user may prefer to not use mobile connections but only WIFI due to cost reasons. These checks are done in the *checkConnectivity()* method. Android provides a *ConnectivityManager* and a *NetworkInfo* class to do this task as shown in listing 6.

```
1   private boolean checkConnectivity(){
2     ConnectivityManager cm = (ConnectivityManager) MasterController.getInstance()
3         .getAppCtx().getSystemService(Context.CONNECTIVITY_SERVICE);
4     NetworkInfo info = cm.getActiveNetworkInfo();
5
6     if(MasterController.getInstance().useMobileNetwork){
7       return info!=null && info.isConnected();
8     }else{
9       return info!=null && info.isConnected() &&
10        info.getType() != ConnectivityManager.TYPE_MOBILE;
11    }
12
13  }
```

Listing 6: Connectivity Check

4.4 Testing

The testing of a newly developed application is crucial to increase the confidence that the application works as expected. The most error prone components of

Androgios are the parsers and the graphical user interface. These are tested in an automated way. Androgios is also tested manually, e.g. in the case of sending external commands.

JUnit is suitable to test the parsers. The unit tests were not executed against a running Nagios server. This would require to fetch HTML before the actual test could be done and would therefore be very slow. In addition it would be hard to predict what exactly is delivered by the Nagios server. To test the different parsers delivered HTML files were collected and stored locally on the file system. These locally stored files can be read fast and it is clear how the model should look like after parsing a file. Each unit test follows the same structure: First read a specific file from the file system, second use the proper parser to build the model and third check if the model finally holds the corresponding information. Changes of a parser can be checked for side effects with very little effort. When a new bug becomes known it is easy to append a further test case to avoid this bug in future.

To test the graphical user interface the tool "Monkey" was used. These tests aim to find errors provoked due to unexpected user interaction. Monkey creates random user events and is therefore well suited to find such bugs. It is a command line tool delivered with the Android SDK. The syntax is documented at [Inc12] in the "Tools" section and looks basically as follows:

```
$ adb shell monkey [options] <event-count>
```

The command `adb shell` starts a remote shell in the running emulator, so `monkey` is executed there. Important options are `-p <package.name>` that constrains monkey to only visit activities defined in this package and `-s <seed>` that allows to generate the same random events more than once for debugging reasons. If an unhandled exception is thrown during the test, monkey will stop running at this

point. A stack trace is logged in such cases.

In addition to the automated tests, Androgios is also tested manually in form of field tests. One problem in testing Androgios is that Nagios servers may be configured very differently. However, common Nagios configurations can be covered by doing field tests. With field tests errors such as a wrong date format implementation (due to an incorrect Nagios documentation) or a GUI bug that only arises when no ServiceGroup is configured were discovered. The feedback of the users also gave some usability related hints. That way user expectations, such as what happens if a certain element is clicked, could be considered and implemented. Based on this knowledge e.g. shortcuts from the main screen to the host list (or problem list in case of problems) were implemented.

4.5 License

Androgios is distributed under the open source license GPL, GNU General Public License, V2 or later (http://www.gnu.org/licenses/gpl.html).

Third party libraries are JSOUP [Hed12] and Base64 Coder [Heu10]. JSOUP, licensed under MIT is GPL compatible, Base64 Coder is multi licensed among others under GPL v2 or later.

5 Conclusions

To make Nagios usable on Android devices is an interesting topic. It is a way to make daily practice of monitoring responsible IT stuff easier. Previously available software have shortcomings, especially in the management part. Androgios, that meets all the requirements demanded in section A, results in an improved benefit for the users.

Although Androgios can keep up with competitor software, there are possibilities to further improve Androgios, as mentioned in section 5.1. Androgios takes this into account by its architecture that allows the implementation of further features without major changes. Future work therefore will only require limited effort.

Androgios is already useful in practice. It is easy to learn how to use Androgios, because its user interface design is inspired by the well known Nagios web user interface. More importantly it covers the major aspects that are of interest regarding the work with Nagios and therefore Androgios fulfills its purpose to aid in the daily work. Table 7 summarizes the features of Androgios.

	Androgios
Minimum Android	1.5
Multiple Nagios Servers	no
Display hosts/services with problems	yes
Display hosts/services with no problem	yes
Auto reload	yes
Manual Reload	yes
Commands regarding problems	yes
Other commands	yes
Widget	yes
HTTPS	yes
Compatible Nagios versions	2.x, 3.x

Table 7: Androgios Features

5.1 Future Work

There exist several ideas how to improve or extend Androgios. The following paragraphs describe basic ideas and steps necessary to implement the improvements.

External Commands Androgios does not implement all possible external commands. It is obvious that providing missing external commands would improve Androgios' benefit. Missing commands are mainly commands regarding host- and servicegroups, such as scheduling downtimes for all hosts in a hostgroup, or enable/disable active checks of all services in a servicegroup.

Additional commands can be implemented as follows: If an activity does not display the *Spinner* element to choose a command, it will be necessary to change the layout resource. The command name must be added to the adapter that fills the *Spinner*. The click listener for the "send" button must be edited. In particular the case that the new command is chosen must be added. It may be necessary to show a dialog to ask the user for additional parameters. The *PostDataBuilder* must be extended by a method in order to be able to create a proper command string. When the command string is built, the *NagiosInstance* can be used to send the command.

Multiple Nagios Instances Currently there is only one NagiosInstance supported. In large networks maybe more than one Nagios server is running. To support multiple *NagiosInstances*, there are at least changes in the GUI, the user preferences, the polling service and the *MasterController* necessary.

The GUI must provide the possibility to choose the *NagiosInstance* that should be handled currently. This *NagiosInstance* has to be marked in the *MasterController*

so that it is able to return the proper object when the method *getActiveNagiosInstance()* is called. An additional activity that displays an overview of all configured *NagiosInstances* may be desirable. The polling service must initiate polls on all *NagiosInstances*, because the user wants to be alerted in case of problems, regardless on what server they were detected. Finally it is necessary to extend the preference activity so that as many *NagiosInstances* as wanted can be configured.

Multiple Alarm- and Poll free intervals To provide multiple alarm- and poll free intervals would be another improvement. It is expected that the single intervals implemented yet will be used mainly during night hours. However there are additional time spans where it is desirable that Androgios is quiet, e.g. during meetings or on weekends.

Implementing this feature would require to adapt the the preference activity so that additional intervals can be configured. The *MasterController* methods that check a given time for these intervals would have to be adapted too.

Reporting The web user interface of Nagios provides reporting functionality that is not yet taken into account in Androgios. Possible are e.g. reports on trends, availability, alerts or notifications. Figure 15 shows a sample report on trends.

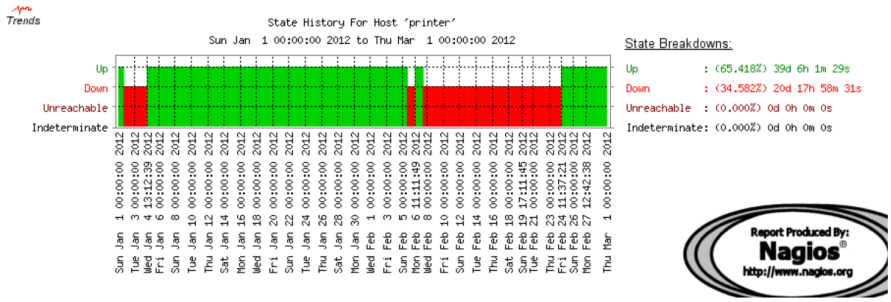


Figure 15: Nagios Report on Trends

The report uses HTML <map> tags on the image. Information within these tags may be parsed, or converted to an image in order to present the report on Android.

Localization The user interface language of Androgios is english by default. However users may prefer to use Androgios in their natural language.

Androgios has strings that are displayed externalized in the resource files `res/-values/strings.xml` and `res/xml/config.xml`. These are the default resources. In order to provide localized resources, the adopted XML files must be put in sub folders that are named according to the following scheme: `res/values-<qualifier>/strings.xml` and `res/xml-<qualifier>/config.xml`. For the german language e.g. this would result in the files `res/values-de/strings.xml` and `res/xml-de/config.xml`. These additions are easy to do, but it must be considered that data provided by the Nagios server is displayed without translation. Therefore localization would also require to change the language of the Nagios server. Changing the Nagios server language will not require changes in the parsers.

Appendices

A List of Requirements

Androgios should be able to show and manage Nagios from android devices. This means that the requirements of the application can be divided into display requirements and management requirements. Additionally there are separate requirements for the polling service, which polls continuously for current network problems, and also for a widget, that should give a quick overview of the network status.

A.1 Display Requirements

Required information on the Start Screen

- The host- and service health in percent.
- The number of outages.
- The number of hosts which are in status UP, DOWN, UNREACHABLE or PENDING.
- The number of services which are in status OK, CRITICAL, WARNING, UNKNOWN or PENDING.
- The background color should indicate if there is no problem (green), there are unimportant problems (yellow) or there are important problems (red).

Required information about Hosts

- A list of all monitored hosts and their status.
- A hostgroup overview with host/service status summary for each group.
- Detailed information about a particular host.
- A list of all problem hosts and their status.
- A list of hosts and their status after application of a custom filter. Filter criteria can be host status types and host properties.

Required information about Services

- A list of all monitored services and their status.
- A servicegroup overview with host/service status summary for each group.
- Detailed information about a particular service.
- A list of all problem services and their status.
- A list of services and their status after application of a custom filter. Filter criteria can be service status types and service properties.

Required information about Downtimes

- A list of all scheduled downtimes including host-/service name, start time and end time.

- Detailed information about a particular downtime including host-/service name, entry time, author, comment, start time, end time, type, duration and downtime ID.

Required information about Comments

- A list of all host comments including host name and comment.
- A separate list of all service comments including service name and comment.
- Detailed information about a particular comment including host-/service name, entry time, author, comment, comment ID, persistent, type and expires.

Required information about Outages

- A list of all outage causing hosts including the number of affected hosts and services.
- Detailed information about a particular outage including severity, host, state, notes, state duration, number of affected hosts and services.

Required information about Monitoring Features

- Overview with flap detection, notifications, event handlers, active/passive checks.

Required information about the Nagios Process

- Detailed information about the Nagios core process.

A.2 Management Requirements

Required commands regarding Hosts

- Enable/disable active checks
- Re-schedule next host check
- Start/stop accepting passive checks
- Acknowledge host problem / remove acknowledgement
- Start/stop obsessing over this host
- Enable/disable notifications
- Schedule host downtime
- Enable/disable notifications for all services
- Schedule check of all services
- Enable/disable checks of all services
- Enable/disable event handlers
- Enable/disable flap detection
- Delete all host comments

Required commands regarding Services

- Enable/disable active checks
- Re-schedule next service check
- Start/stop accepting passive checks
- Acknowledge service problem / remove acknowledgement
- Start/stop obsessing over this service
- Enable/disable notifications
- Schedule service downtime
- Enable/disable event handlers
- Enable/disable flap detection
- Delete all service comments

Required commands regarding Comments

- Delete particular host comment
- Delete particular service comment

Required commands regarding Downtimes

- Delete particular host downtime
- Delete particular service downtime

Required commands regarding the Nagios Process

- Shutdown the Nagios process
- Restart the Nagios process
- Enable/disable notifications
- Start/stop executing service checks
- Start/stop accepting passive service checks
- Start/stop executing host checks
- Start/stop accepting passive host checks
- Enable/disable event handlers
- Enable/disable flap detection
- Start/stop obsessing over services/hosts
- Enable/disable processing performance data

A.3 Polling Service Requirements

- Polls continuously the network state. The polling interval should be configurable. The usage of (probably costly) mobile network connection should be avoidable per configuration.
- Ensures to alert the user on problems (via sound, vibration and notification icon).
- A poll free interval should be definable.
- An alarm free interval should be definable.

A.4 Widget Requirements

- Should display if last poll was successful.
- Should display host problems.
- Should display service problems.
- Should display outages.

A.5 Non-functional Requirements

- Androgios must run without changes to the Nagios server.
- Nagios version 2.x and 3.x should be supported.
- Smallest Android Version as possible.
- HTTPS shall be supported.
- Androgios shall run on mobile phones and tablets.

B Androgios Installation

Androgios requires Android 1.5 or higher. Androgios is not yet available at the Google Marketplace. Therefore the installation of Non-Market-Applications must be allowed. An arbitrary file manager is needed to access the installation file "androgios.apk"

The installation steps are:

- Store "androgios.apk" somewhere on the SD card.
- Using the file manager open "androgios.apk".
- Read the permissions required and click install.

After the first launch at least the URL to the Nagios cgi-bin folder and the credentials must be configured. The configuration activity can be opened by pressing the (hardware) menu button and then choosing "Configuration". In order to ensure that external commands are sent correctly, the Nagios date format should also be configured properly.

In many cases the acceptance of self signed certificates (SSL) and the usage of mobile networks will have to be enabled. Both are disabled by default.

A sample configuration can be seen in appendix D, picture (i). This configuration can be used for testing Androgios. It refers to a publicly available Nagios test instance (the password is like the username "nagiosadmin").

C Android development with Eclipse

Androgios was developed on Ubuntu 11.04 (64 bit) in conjunction with the Eclipse IDE. This Appendix describes how to get started with development and how to export a signed application package (apk file).

Under the assumption that Eclipse is already installed, the first step to get started is to install the "Android Development Tools" (menu: Help - Install New Software...) from `https://dl-ssl.google.com/android/eclipse` After the installation of the Android Development Tools, a new project type "Android Project" is available. On 64 bit Ubuntu the "ia32-libs" (ia32 shared libraries for use on amd64 and ia64 systems) are necessary. They are available in Ubuntu's default package sources.

In the menu "Window" the entry "Android SDK and AVD Manager" is now available. This window is used to download and install SDK platform packages for certain API levels (see figure 16).

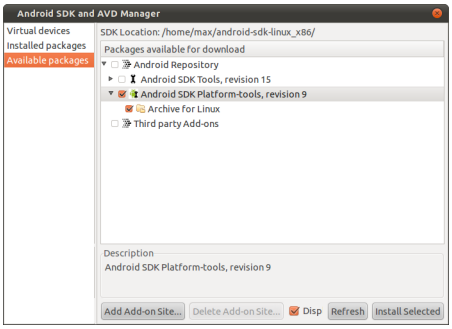


Figure 16: Android Virtual Device Manager

When the desired SDK is installed, an Android Virtual Device can be created (see

figures 17 and 18). Android projects can be run on available virtual devices.

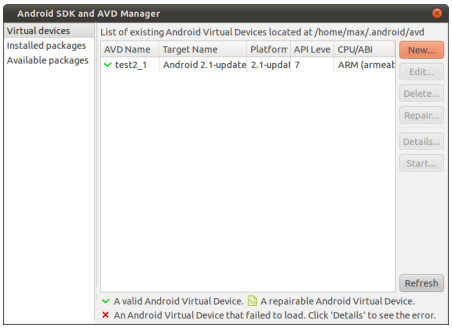


Figure 17: Android Virtual Device Manager

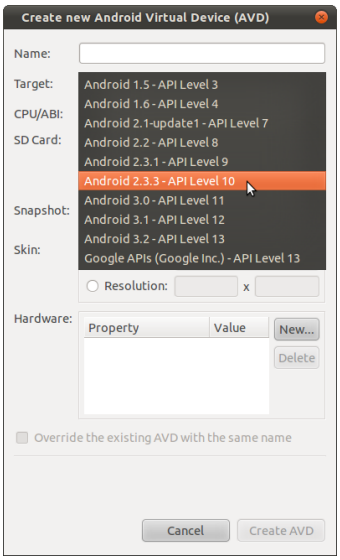


Figure 18: Android Virtual Device Manager

The Android Development Tools provide a wizard that supports the export of an Android project into an installable application package. Figure 19 shows where the wizard is located in the projects context menu. To sign the application package the location of a keystore and the name of a key is required. The wizard supports the creation of both when needed. Keys need not to be signed by a CA, the only issue

is that the Android Market currently requires certificates to be valid until 2033. According to [Inc12], "A validity period of 25 years or more is recommended."

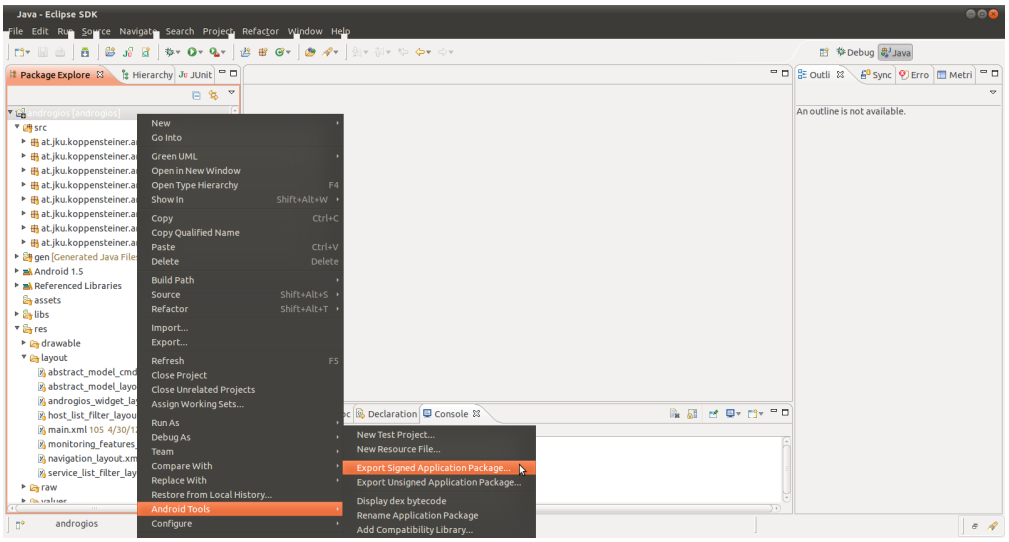
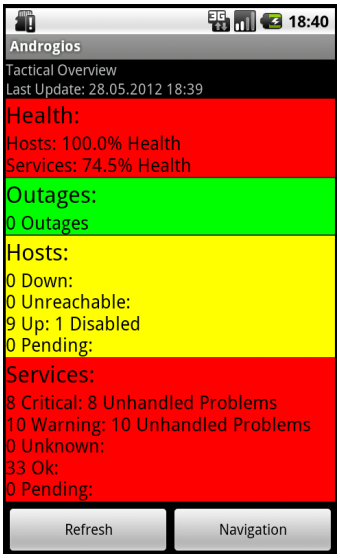
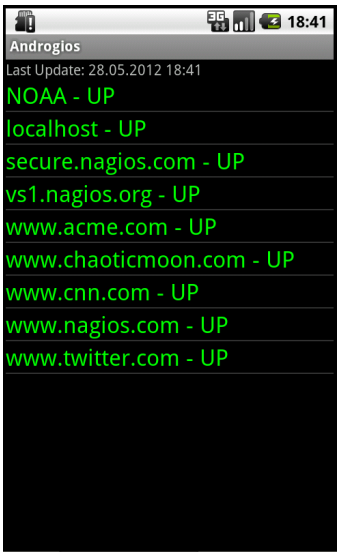


Figure 19: Export an Android Project

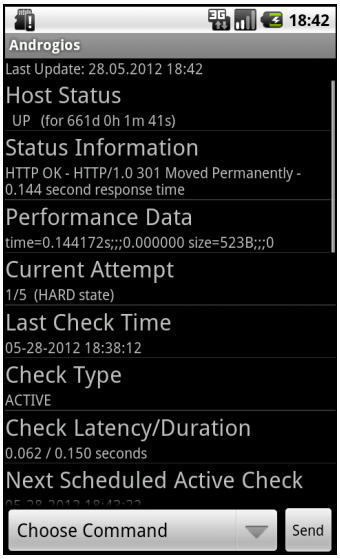
D Androgios Screenshots



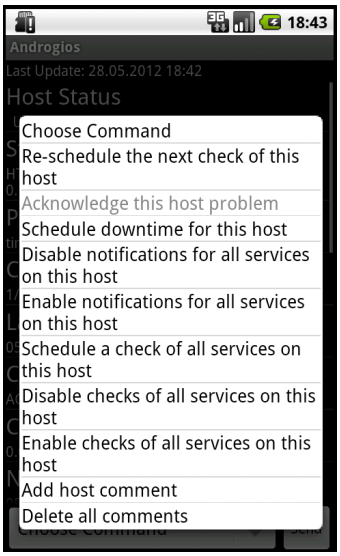
(a) Main Activity



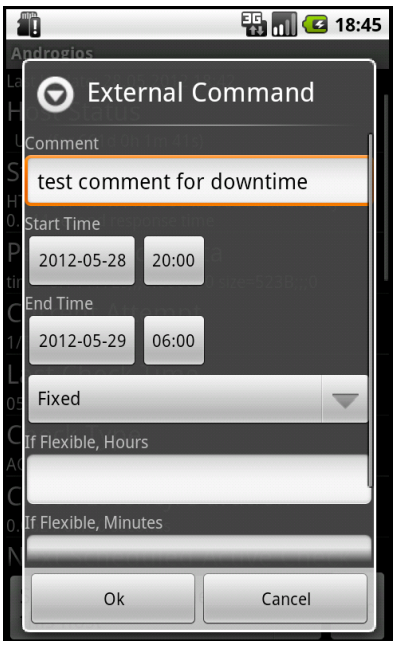
(b) Host List



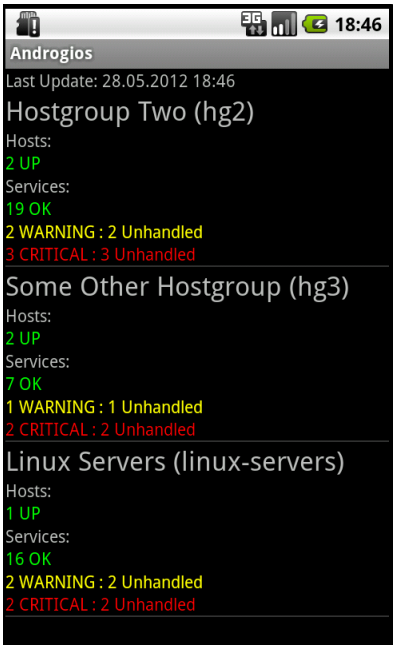
(c) Host Details



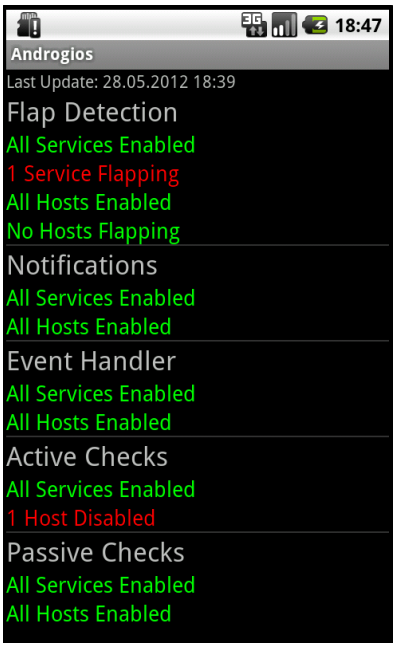
(d) Host Commands



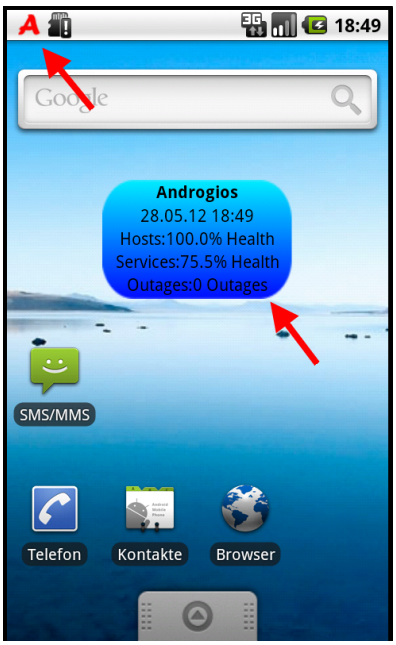
(e) Scheduling a Host Downtime



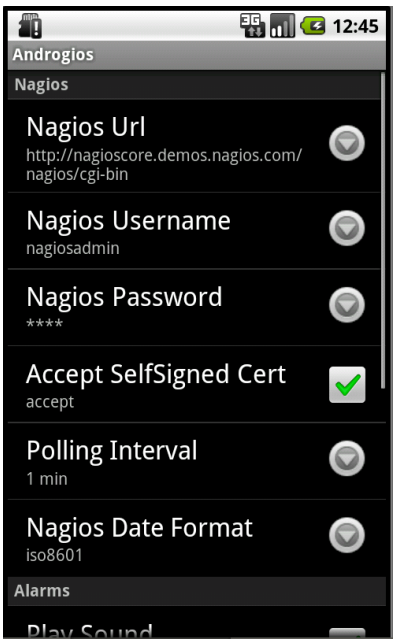
(f) Host Groups



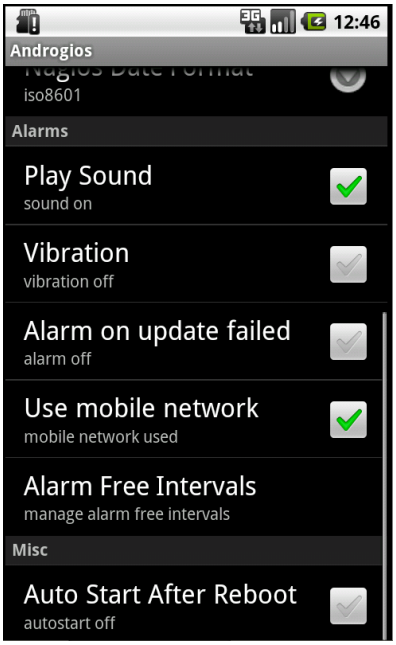
(g) Monitoring Features



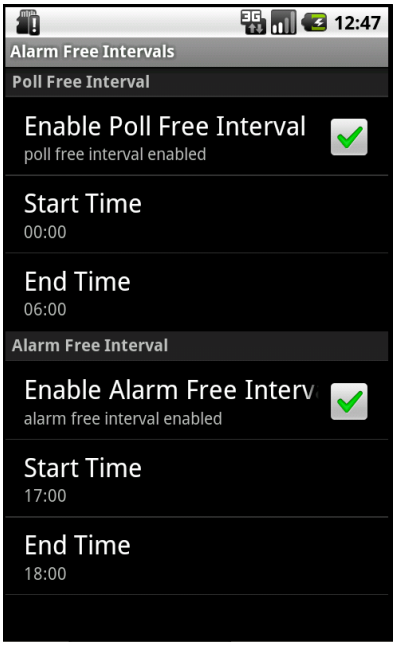
(h) Widget and Alert Icon



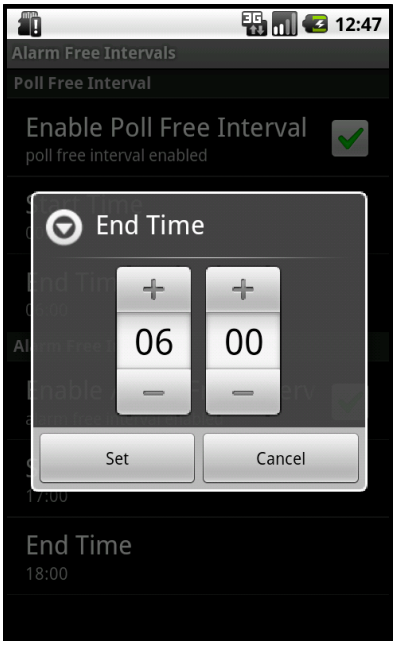
(i) Nagios Instance Config



(j) Androgios Config



(k) Free Intervals



(l) Choosing a Time

References

[Bar09] Wolfgang Barth. *Nagios - System und Netzwerkmonitoring*. Open Source Press, Munich, 2009.

[BP10] Arno Becker and Marcus Pant. *Android 2 - Grundlagen und Programmierung*. dpunkt.verlag, Heidelberg, 2010.

[Cor12] Oracle Corporation. Java se 6 api specification. http://docs.oracle.com/javase/6/docs/api/, 2012.

[Döl12] Mirko Dölle. Fernüberwacht. *c't Magazine*, 5/2012:176–178, March 2012.

[Ent12a] Nagios Enterprises. Nagios exchange website. http://exchange.nagios.org/, 2012.

[Ent12b] Nagios Enterprises. Nagios website. http://www.nagios.org/, 2012.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[Hed12] Jonathan Hedley. Jsoup parser. http://jsoup.org/, 2012.

[Heu10] Christian Heureuse. Base64coder. http://www.source-code.biz/base64coder/java/Base64Coder.java.txt, 2010.

[Inc12] Google Inc. Android developers guide. http://developer.android.com/guide/index.html, April 2012.

[Nic09] Carlo U. Nicola. Einblick in die dalvik virtual machine. http://www.fhnw.ch/technik/imvs/publikationen/fokus-report/2009, 2009. Fachhochschule Nordwestschweiz, Hochschule für Technik.

[Rie10] Götz Rieger. Nagios to go. *c't Magazine*, 23/2010:150–155, November 2010.

Acknowledgements

I would like to thank all contributors to the success of this thesis and all professors who did gave support during the time of my study.

Special thank goes to Michael Sonntag, the referee of this theses, who gave me an interesting topic and the opportunity to solve it independently.

In addition, I want to thank my colleague Phillip Lengauer for his valuable information in several discussions.

CURRICULUM VITAE

Markus Koppensteiner, BSc

Zur Person

Name	Markus Koppensteiner
Anschrift	Freytagstraße 6, A-4020 Linz
Telefon	+43 676 6800239
E-Mail	markus.koppensteiner@aon.at
Geburtsdatum	23.01.1977
Familienstand	ledig
Staasbürgerschaft	Österreich
Führerscheinklassen	A,B,C

Ausbildung

seit 03/2010	Studium zum Dipl.Ing. in Netzwerke und Sincherheit in Linz
03/2007 - 02/2010	Studium zum BSc in Informatik in Linz
09/2004 - 07/2006	Berufsreifeprüfung in Linz
02/1998 - 02/2001	Krankenpflegediplomausbildung in Innsbruck
08/1992 - 04/1994	Lehre Großhandelskaufmann in Linz
09/1991 - 07/1992	HAK in Linz
09/1987 - 07/1991	Hauptschule in Linz
09/1983 - 07/1987	Volksschule in Linz

Berufliche Aktivitäten

seit 03/2009	Studentischer Mitarbeiter im Lehrbetrieb, JKU
02/2002 - 03/2009	DGKP im KH der Barmherzigen Schwestern Linz
10/1995 - 02/2002	Sanitätsunteroffizier beim österreichischen Bundesheer

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 10. Juli 2012

Markus Koppensteiner