Android Iodine Entwicklerdokumentation

Inhaltsverzeichnis

Zusammenfassung

Die Dokumentation ist zweigeteilt. Dieser Teil enthält eine technische Beschreibung. Die Bedienung und Funktionsweise ist in der *Anwenderdokumentation* beschrieben.

1 Aufbau

Die Anwendung besteht im groben aus 4 Komponenten

- Activity `.IodineMain`
- Activity Verbindungseinstellungen `.IodinePref`
- Tunnel Service `VpnService` und den JNI Bindings `IodineClient`
- Konfigurationsverwaltung `.config.ConfigDatabase` und `.config.IodineConfiguration`

Abbildung 1 zeigt Architektur der Anwendung:

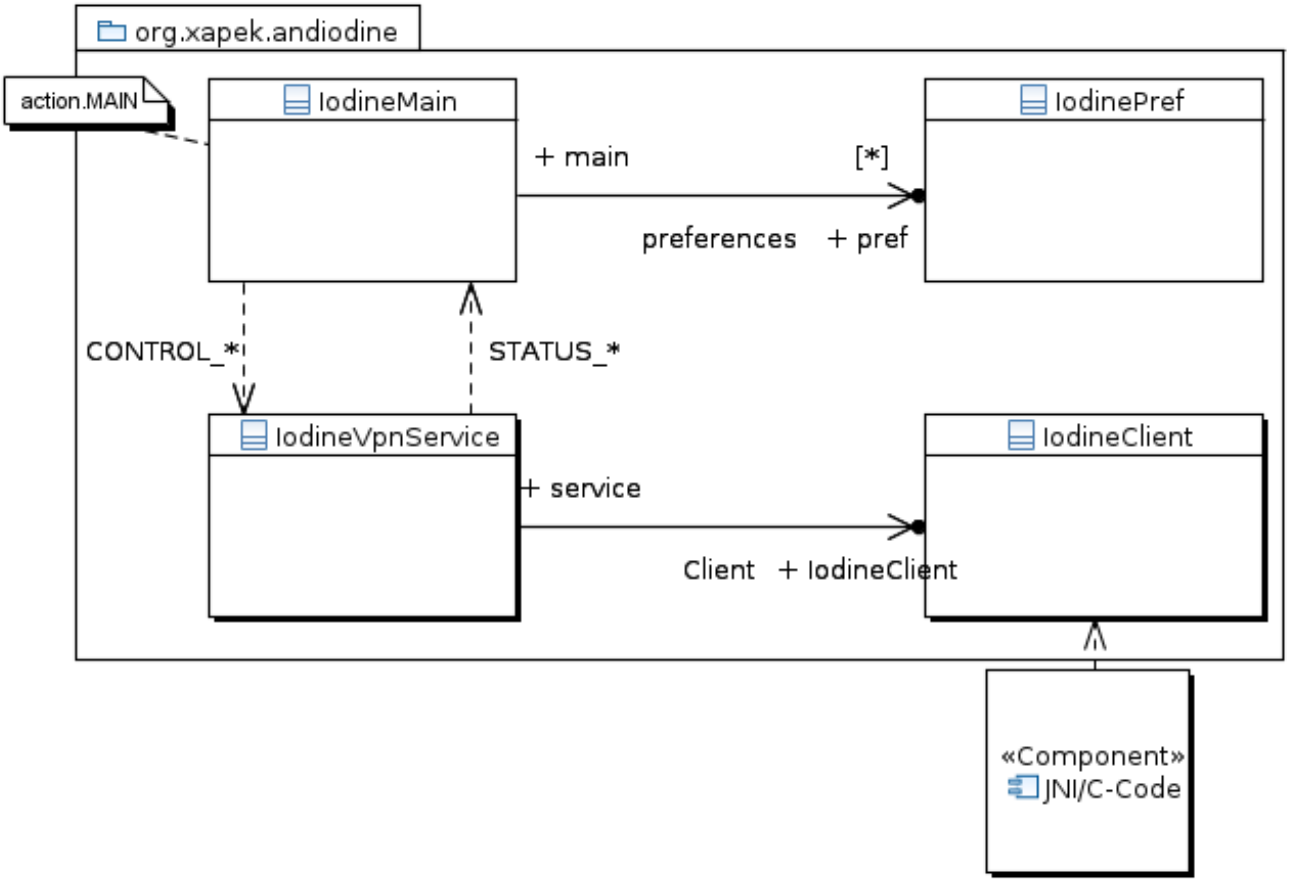

Abbildung 1: Architektur der Anwendung

1.1 Benutzeroberfläche

Die Haupt Activity `.IodineMain` startet den "VpnService" und steuert ihn über Broadcast Intents. In dieser Activity steuert der Benutzer den Auf- und Abbau der Tunnel. Über ein Button in der ActionToolbar kann eine neue Tunnelkonfiguration angelegt werden.

Die Interaktion zwischen des Benutzers in der Anwendung ist in Abbildung 2 visuell dargestellt:

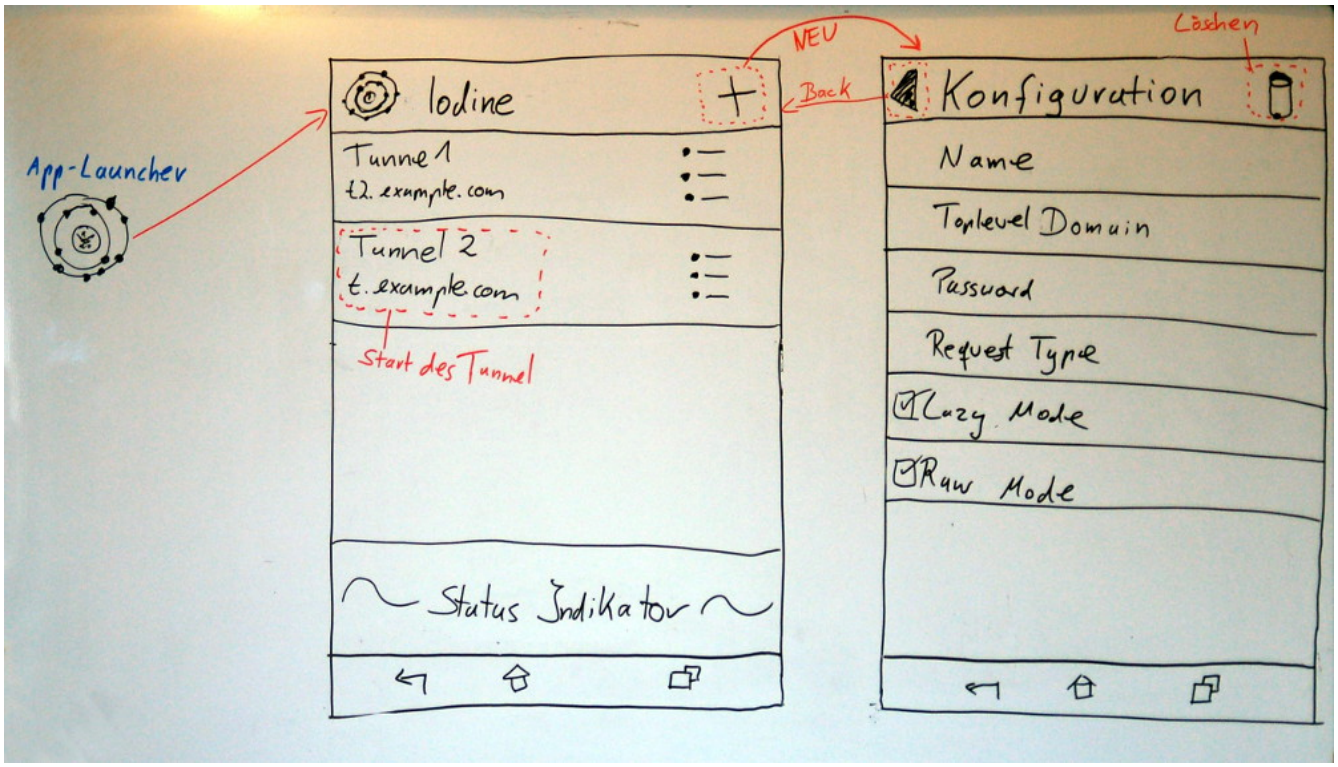

Abbildung 2: Graphischer Aufbau der GUI

1.2 Konfiguration

Die Tunnelkonfigurationen werden in einer SQLite Datenbank abgelegt. Es existiert mit `.config.IodineConfiguration` ein leichtgewichtiger Proxy um die Android `ContentValues` Klasse. Die `.config.ConfigDatabase` Klasse ist ein `SQLiteOpenHelper` und kann mehrfach instanziert werden.

1.3 VPN-Service

Der VPN Service hat 5 Zustände die er über Broadcast Intents mitteilt. Eine solche Mitteilung wird verschickt wenn sich der Zustand ändert oder dies über ACTION_CONTROL_UPDATE angefordert wurde.

Die Kommunikation der Oberfläche mit dem VPN Service erfolgt mit Broadcasts Intents.

Abbildung 3 zeigt die Zustände des Iodine VPN-Service. Rot nummeriert sind die Intents die der Service verschickt um über Statusänderungen zu informieren. Blau nummeriert sind Intents mit denen der Service gesteuert werden kann.

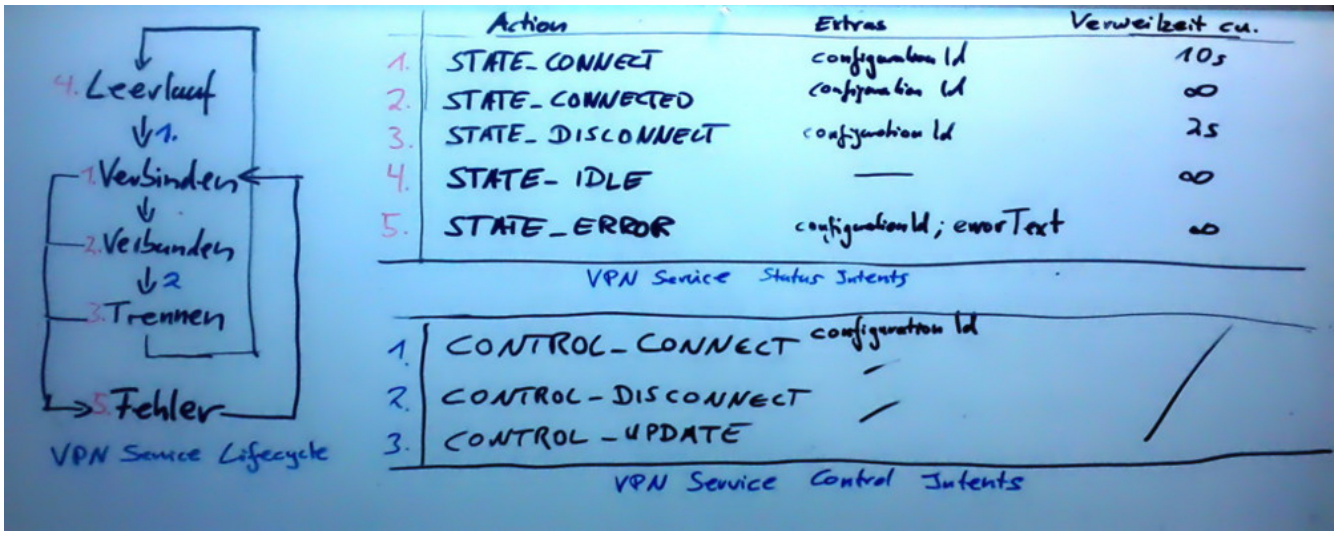

Abbildung 3: Status Informations und Steuerungs Intents des VPN Service

1.4 JNI

Die JNI Methoden für iodine befinden sich in der Klasse `.IodineClient` bzw. `/jni/iodine-client.c`. `IodineClient#connect` ersetzt dabei prinzipiell die `main()` des ursprünglichen iodine Client.

Weitere Methoden dienen dem Austausch der vom Server übermittelten Konfiguration und des im System eingestellten DNS Server.

2 Android VPN-Framework

Seit API Level 14/Android 4 ist es möglich VPN Verbindungen mit Android Anwendungen aufzubauen und zu verwalten.

Die Application benötigt dazu die Permission `android.permission.BIND_VPN_SERVICE`.

Bevor eine Anwendung das erste mal eine VPN Verbindung aufbauen darf wird Android sicherheitshalber den Benutzer explizit um Erlaubnis fragen.

Dazu wird `IodineVpnService.prepare(this)` [vpnapi] aufgerufen. Wird null zurückgegeben hat der Benutzer VPN Verbindungen dieser App bereits früher zugestimmt. Andernfalls wird ein Intent zurückgegeben mit dem die Benutzernachfrage initiiert werden kann.

```
public void tunnel() {
  Intent intent = IodineVpnService.prepare(this);
  if (intent != null) {
    // Ask for permission
    intent.putExtra(IodineVpnService.EXTRA_CONFIGURATION_ID, configuration.getId());
    startActivityForResult(intent, INTENT_REQUEST_CODE_PREPARE);
  } else {
    // Permission already granted
    startVPNService();
  }
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (requestCode == INTENT_REQUEST_CODE_PREPARE) {
    if (resultCode == RESULT_OK) {
      startVPNService();
```

```
    } else {
      // User denied permission
    }
  }
}

private void startVPNService() {
  // Start VPN with VPNService.Builder
}
```

Der weitere Weg mit dem `VPNService.Builder` ist geradelinig. Im Fall von iodine wird zunächst der Tunnel über DNS aufgebaut bevor das tun-Interface geöffnet wird.

Nachdem vom Server die IP-Konfiguration mitgeteilt wurde, wird diese im `Builder` gesetzt und der Tunnel geöffnet:

```
// .... IodineVpnService.java :: runTunnel()
b.addAddress(hostAddress, netbits);
b.addRoute("0.0.0.0", 0); // Default Route
b.setMtu(mtu);

// Opens tun device
ParcelFileDescriptor parcelFD = b.establish();

// prevent dns traffic to get through its own tunnel
protect(IodineClient.getDnsFd());

// get the filedescriptor
int tun_fd = parcelFD.detachFd();

// pass the filedescriptor to iodine
IodineClient.tunnel(tun_fd);
```

3 iodine

3.1 Verbindungsaufbau (Handshake)

Der folgende Text zeigt ein Beispiel für den Ablauf eines Handshake. Der genaue Ablauf kann variieren jenachdem wie die Verbindungsparameter gewählt werden.

Hier sind gewählt -m 768 fragment size und ein 9 Zeichen Passwort. Die Gegenstelle ist `t.yves.tw`. Eine Raw (direkte UDP) Verbindung wurde verhindert indem der Rechner zum Testzeitpunkt keine default Route hatte.

RX/TX aus der Sicht des Servers. Die "*" in den Hostnamen markieren Zeichen die sich aus Random Daten ergeben.

```
== Der Client testet die Qualitaet der Uebertragung
<-- client.c:handshake_qtype_autodetect()
          -> handshake_qtypetest()
          -> send_downenctest()
  hostname[0] = "y"
  hostanme[1] = downenc = 'r'
  hostname[2] = variant = 1 = 'b' (b32)
  hostname[3..5] = rand_seed++
  RX: yrb***.t.yves.tw

--> 48 bytes aus encoding.h:DOWNCODECHECK
  TX: yrb***.t.yves.tw, 48 bytes data

== Austausch der Versionsinformationen
<-- client.c:send_version()
  VERSION=0x00 00 05 02
```

```
    hostname[0] = cmd = 'v'
    hostname[1..6] = b32(0,0,5,2,random<<8,random)
    hostname = "vAAAAKAR__"
    RX: vaaaaka****.t.yves.tw

  --> iodined.c:send_version_response()
    der Server bestaetigt mit
    data[0..8] = "VACK" b32(seed>>24, seed>>16, seed>>8, seed, userid)
    TX: vaaaaka***.t.yves.tw, 9 bytes data


  == Senden von Passwort und IP-Konfiguration (Subnetz)
  <-- client.c:send_login()
    cmd = 'l'
    hostname[1..16] = login/password mit seed xored und md5
    hostname[17..18] = seed
    RX: lad24srn4ezmg21qjsfy13msagd0srfq.t.yves.tw

  --> iodined.c:handle_null_request()
    Sendet bei Erfolg die IP Einstellungen wie
    "172.16.0.1-172.16.0.2-1130-16"
    server="172.16.0.1"
    client="172.16.0.2"
    mtu=1130
    netbits=16
    TX: lad24srn4ezmg21qjsfy13msagd0srfq.t.yves.tw
      = 3137322e31362e302e312d3137322e31362e302e322d313133302d3136 (_16)
      = 172.16.0.1-172.16.0.2-1130-16

  == Senden der IP Adresse des Clients
  <-- Request for IP address
    RX: iamin.t.yves.tw

  --> iodined.c:handle_null_request()
      addr = externe IP Adresse des Server (-n Switch)
      reply[0] = 'I';
      reply[1] = (addr >> 24) & 0xFF;
      reply[2] = (addr >> 16) & 0xFF;
      reply[3] = (addr >>  8) & 0xFF;
      reply[4] = (addr >>  0) & 0xFF;
    TX: iamin.t.yves.tw
      = 494e2f737d (_16)

  == Testen auf EDNS Erweiterung
  <-- client.c:handshake_edns0_check()
            -> send_downenctest()
    downenc = 'r' fuer T_NULL 't'
    variant = 1 = 'b' (b32)
    data[0..5]  = "y" downenc variant rand_seed[0..2]
    RX: yrb***.t.yves.tw

  --> iodined.c:handle_null_login() : 937
             -> write_dns( type='R')
    Der Server antwortet mit 48 bytes aus encoding.h:DOWNCODECHECK
    TX: yrb***.t.yves.tw, 48 bytes data


  == Testen der Kodierungen mit verschiedenen Patterns
  <-- client.c:handshake_upenc_autodetect()
    In den folgenden Tests testet der Client ob mit Base128
    kodierte Nachrichten vom DNS Relay korrekt verarbeitet werden.
```

```
--> Der Server schickt die Patterns einfach wieder zurueck.

== Client legt Kodierung fest, Server bestaetigt
<-- client.c:handshake_switch_codec()
  hostname[0] = command 's'
  hostname[1] = b32(userid)
  hostname[2] = 'h' (7)
  hostname[3..5] = rand_seed++
  rand_seed++;
  RX: sahmiut.yves.tw

--> iodined.c:840
  Schreibt den Namen des ausgewaehlten Codecs:
  data="Base128" (kein encoding!)
  TX: sahmiut.yves.tw, 7 bytes of data

== Anschalten lazy mode (an: Server beantwortet Anfragen nicht sofort)
<-- client.c:send_lazy_switch()
  hostname[0] = 'o'
  hostname[1] = b32(userid) = 'a'
  hostname[2] = 'l' fuer lazy mode oder 'i'
  hostname[3..5] = rand_seed++
  RX: oalmiv.t.yves.tw

--> iodined.c:919
  data="Lazy" (kein encoding!)
  TX: oalmiv.t.yves.tw, 4 bytes of data

==
<-- client.c:send_set_downstream_fragsize()
  data[0] = userid;
  data[1] = (fragsize & 0xff00) >> 8;
  data[2] = (fragsize & 0x00ff);
  data[3] = (rand_seed >> 8) & 0xff;
  data[4] = (rand_seed >> 0) & 0xff;
  hostname = 'n' + b32(data)
  RX: naabqbmiw.t.yves.tw

--> iodined.c:1042
  bestaetigt empfangene Framesize durch Wiederholung

== Regelmaesige pings fragen den Server nach anstehenden Daten ab
<-- client.c:send_ping()
  data[0] = userid;
  data[1] = ((inpkt.seqno & 7) << 4) | (inpkt.fragment & 15);
  data[2] = (rand_seed >> 8) & 0xff;
  data[3] = (rand_seed >> 0) & 0xff;
  hostname = 'p' + b32(data)
  RX: paaalcfy.t.yves.tw

--> iodined.c:1067
  Der Server nutzt die regelmaessigen Pings um Daten an den Client zu liefern.
```

3.1.1 Der lazy Modus

Wie in der Anwenderdokumentation beschrieben erhöht der Lazy Modus den Durchsatz und senkt die Latenzzeit, wird aber nicht von allen DNS-Relays unterstützt.

Lazy bezieht sich auf das Verhalten des Servers. Der Server wird im Lazy-mode alle Antworten auf Anfragen solange zurückhalten bis er neue Daten für den Client erhalten hat. Im Idealfall also bis das Antwortpaket der getunnelten IP Verbindung angekommen ist.

Diese Verzögerung kann mit manchen DNS-Relays Probleme machen. Der Server kann dies jedoch anhand der Duplikate in den Anfragen erkennen und damit den lazy-mode ausschalten.

Ohne diesen Mechanismus müsste der Client jedoch viel häufiger nach neuen Daten pollen (vgl. HTTP Long polling in Comet oder BOSH).

3.2 iodine base(32) Kodierung

Dieses Programm bietet die Base32 Kodierung von iodine für die Kommandozeile zum Debuggen an.

```
#include <stdio.h>
#include <string.h>

#include "src/base32.h"
#include "src/encoding.h"

int main(int argc, char *argv[]) {
    struct encoder *b32 = get_base32_encoder();
    char buf[512];
    size_t len = 512;

    if (argc != 3) return 0;
    if (*argv[1] == 'd') {
        int r = b32->decode(buf, &len, argv[2], strlen(argv[2]));
        int i;
        printf("Decoded %d bytes:\n", r);
        for (i = 0; i< r; i++) {
            printf("0x%02hhx (%c) ", buf[i], (buf[i] >= '0' && buf[i] <= 'z') ? buf[i] : '  ↩
                ');
        }
        printf("\n");
    } else if (*argv[1] == 'e') {
        int r = b32->encode(buf, &len, argv[2], strlen(argv[2]));
        printf("Encoded %d bytes in %ld output bytes: >%s<\n", len, r, buf);
    }
    return 0;
}
```

```
# gcc test.c src/base32.c -o test
# ./test e abcdefg
Encoded 7 12 bytes: >mfrggzdfmztq<
```

3.3 Änderungen an iodine

Der Code basiert auf der letzten Iodine Version 0.6.0-rc1. Die Änderungen wurden absichtlich möglichst gering gehalten und betragen im wesentlichsten nur ca. 80 Zeilen.

Ein Hauptteil der Änderungen verhindern, dass Android als Linux erkannt wird. Im Gegensatz zu vielen Linux Installationen verwenden Android nicht die GNU libc sondern *Bionic libc*. Dies ist eine besonders kleine, auf die BSD libc zurückgehende standard C Library. Es fehlen einige Features der glibc wie wide-character support, volle POSIX Thread Unterstützung oder locale Unterstützung. Das Ziel von Bionic ist nicht eine vollständige C Standardbibliothek sondern lediglich eine schlanke Implementierung aller für ein Android nötigen Funktionen.

Im einfachsten Fall scheitert die Ausführung von iodine unter Android an einem `system()` Aufruf mit dem iodine die IP-Konfiguration anwendet.

3.3.1 Android.mk

Das ursprüngliche iodine Makefile wird nicht verwendet. Es wird das Android NDK Buildsystem verwendet, die Anweisungen dazu liegen in `jni/Android.mk`. Aus dem Projektverzeichnis kann die Übersetzung der C-Quellen angestossen werden.

```
org.xapek.andiodine % ~/$NDK_ROOT/ndk-build clean
Clean: iodine-client [armeabi]
Clean: stdc++ [armeabi]
org.xapek.andiodine % ~/$NDK_ROOT/ndk-build
Compile thumb  : iodine-client <= iodine-client.c
Compile thumb  : iodine-client <= tun.c
Compile thumb  : iodine-client <= dns.c
Compile thumb  : iodine-client <= read.c
Compile thumb  : iodine-client <= encoding.c
Compile thumb  : iodine-client <= login.c
Compile thumb  : iodine-client <= base32.c
Compile thumb  : iodine-client <= base64.c
Compile thumb  : iodine-client <= base64u.c
Compile thumb  : iodine-client <= base128.c
Compile thumb  : iodine-client <= md5.c
Compile thumb  : iodine-client <= common.c
Compile thumb  : iodine-client <= client.c
Compile thumb  : iodine-client <= util.c
SharedLibrary  : libiodine-client.so
Install        : libiodine-client.so => libs/armeabi/libiodine-client.so
```

Die Library wird vom Android SDK automatisch in die APK-Datei eingefügt.

3.3.2 common.c daemon()

Die `daemon()` Funktion in src/common.c ist gedacht um iodine als Hintergrundprozess laufen zu lassen. Sie ist nur für Linux und BSD vorgesehen.

Das `#ifdef` erkennt Android als Linux, Bionic unterstützt `daemon()` jedoch nicht, da die Funktionalität der `daemon()` Funktion für eine Android App nicht benötigt wird.

Auch in diesem Fall brauchen wir die `daemon()` Funktion nicht, da iodine in einem von Java gesteuerten Thread laufen wird.

3.3.3 common.c warn()

Die `warn()` Funktion existiert nicht in der Bionic libc. Die bereitgestellte Implementierung verwendet `fprintf` auf stderr. Die Meldungen werden in das Android Logging System umgeleitet und sind auch über Logcat nutzbar.

3.3.4 tun.c write_tun() / read_tun()

Wie bei FreeBSD und Windows muss beim schreiben auf das Tun device (`write_tun`) kein 4 byte großer Header mit der Adress Family angefügt werden. Entsprechend wird dieser in `read_tun()` im Fall von Android, FreeBSD und Windows nicht entfernt.

3.3.5 tun.c tun_setip()

Je nach Plattform werden wird die IP-Adresse unterschiedlich gesetzt. Im Fall von Linux geschieht dies mit einem fragwürdigen `system("/sbin/ifconfig")` Aufruf.

Dies ist unter Android so nicht möglich. Es wurde daher eine globale Datenstruktur `tun_config_android` (tun.h) angelegt in welcher die zu setzende IP-Adresse, Gegenstelle IP-Adresse und Netzmaske abgelegt wird. Die Inhalte dieser Datenstruktur können von Java über JNI Funktionen abgefragt werden.

Das setzen der IP-Adressen und Routen geschieht über Methoden des Android VPN-Framework in Java.

3.3.6 DNS Headerfiles

Iodine benötigt Konstanten aus arpa/nameser_compat.h und arpa/nameser.h das nicht Teil der Android Libc ist. Die Header wurden als src/dns_android.h hinzugefügt.

4 Projekt öffnen und bauen

4.1 C Quellcodes übersetzen

Um das Projekt zu bauen ist neben dem Android SDK auch das Android NDK erforderlich. Mit dem daraus bereitgestellten Kommando `ndk-build` werden die C-Quellcodes unterhalb des Verzeichnisses `jni/` übersetzt.

```
andiodine$ $NDK_ROOT/ndk-build clean
Clean: iodine-client [armeabi]
Clean: stdc++ [armeabi]
Clean: iodine-client [x86]
Clean: stdc++ [x86]

andiodine$ $NDK_ROOT/ndk-build
Compile thumb  : iodine-client <= iodine-client.c
.....
SharedLibrary  : libiodine-client.so
Install        : libiodine-client.so => libs/armeabi/libiodine-client.so
Compile x86    : iodine-client <= iodine-client.c
.....
SharedLibrary  : libiodine-client.so
Install        : libiodine-client.so => libs/x86/libiodine-client.so
```

4.2 Entwickeln mit Eclipse

Das Projekt kann über den Importassistenten eingebunden werden:

Import → Android → Existing Android Code Into Workspace

4.3 Entwickeln mit Android Studio

- Choose Import Project, choose project Folder.
- Select "Create project from existing sources".

4.4 Übersetzen mit ant

Using ant

```
android project --path .
ant debug
```

Die APK liegt unterhalb von `bin` und kann mit dem ant target `install` über adb installiert werden.

5 Anhang

5.1 Literatur

[1] [vpnapi] http://developer.android.com/reference/android/net/VpnService.html Dokumentation zu den Android VPN Service API