BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

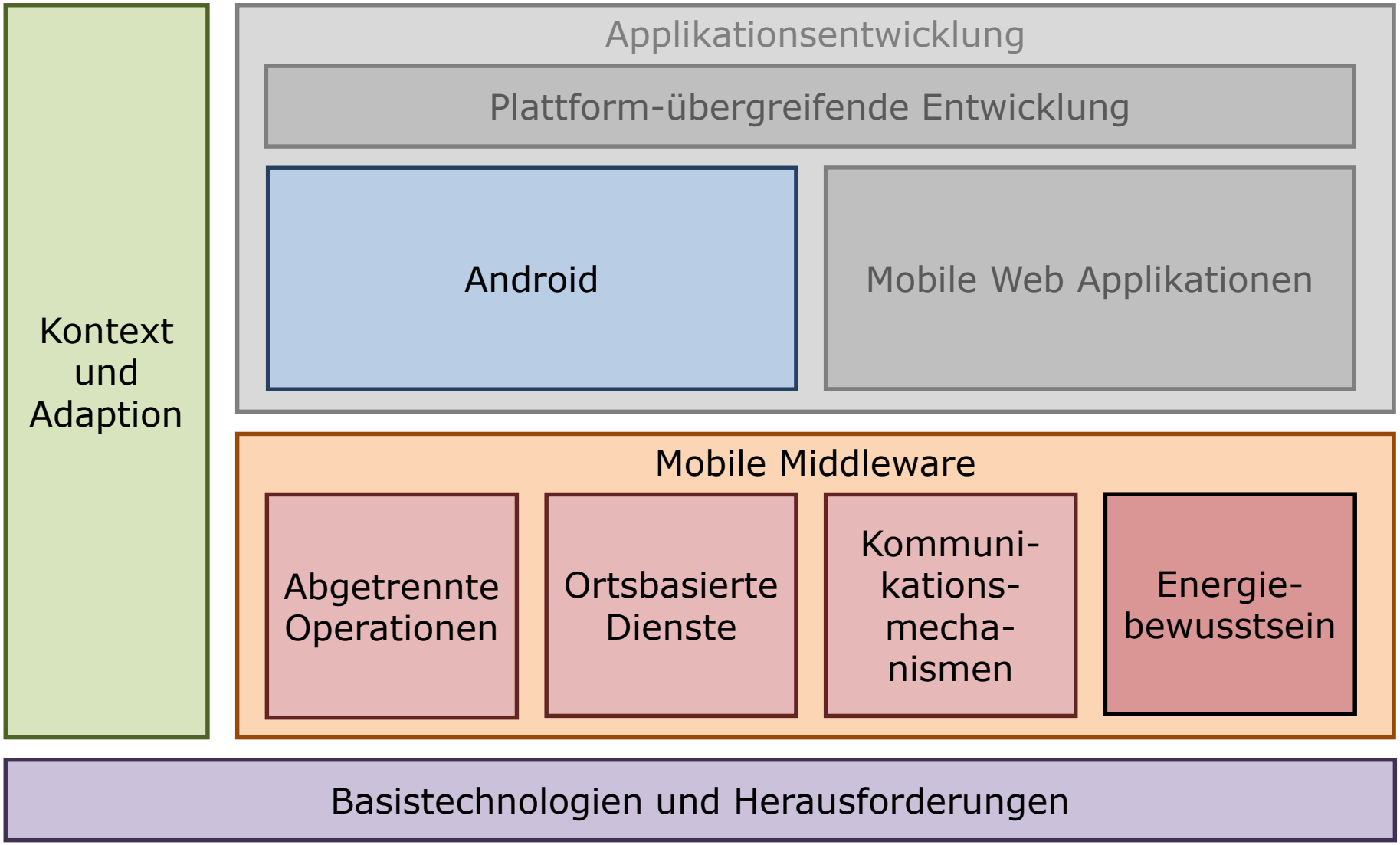
Web- und App-Programmierung

Energie-herausforderung

mit Skriptmaterial von Christin Groba, Ph.D.

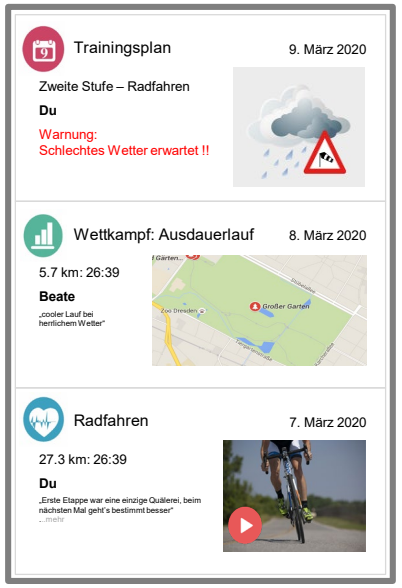
Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

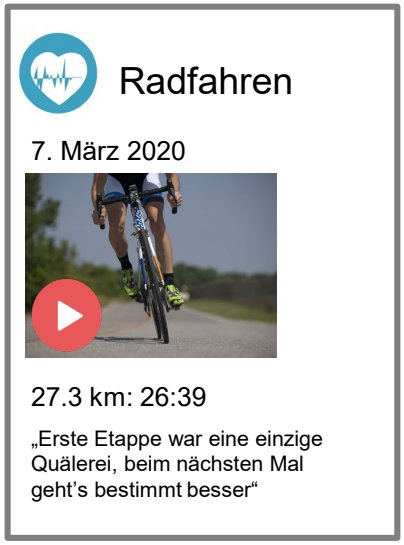
Aufbau der Lehrveranstaltung



Rückblick: Social-Fitness-App

- Armband/Smartwatch mit Smartphone/Tablet-App
- Server-Komponente für Datenhaltung und Nutzerverwaltung
- Funktionalität
  - automatische Aktivitätserkennung, -aufzeichnung und –(um)planung
  - Aufzeichnung der Aktivitätszustände und korrelierter Medien- und Sensordaten (Bilder, Videos, Pfade, Pulskurven, …)
  - Zeitlinie der Aktivitäten
    → Posten der eigenen Aktivitäten
    → Sehen der Aktivitäten anderer
    → integrierte Zeitlinie mit Medien
  - Verwaltung des Trainingsplans
  - Verwaltung von Wettkämpfen und Platzierungen





Social-Fitness-App – Energieherausforderung

- Lokationsverfolgung und Aktivitätsaufzeichnung verbrauchen Energie
  - Wann soll das Tracking aktiviert werden?
  - Welche Genauigkeit ist notwendig?
  - Welche Sensoren sollen benutzt werden?
- Kommunikation verbraucht Energie
  - Wann sollen Informationen veröffentlicht werden?
  - Mobilfunk benutzen oder auf WiFi warten?
- Entwicklung
  - Verbraucht die Applikationslogik mehr Energie als notwendig?
  - Was sind verbreitete Programmierfallstricke?

Übersicht

- Energieverbrauch
  - Gegenstand der Messung
  - Messung
  - Verbraucher
- Drahtloskommunikation
  - WiFi und 3G/LTE
  - WiFi Direct und Bluetooth
- Lokationsverfolgung
  - GPS, WiFi und GSM
  - dynamische Anbieter/Frequenz-Auswahl
- unerwartete Energieverbraucher
  - Energiefehler (Energy Bugs)
  - Speicherleck (Memory Leak)

Energieverbrauch

Energieverbrauch – Gegenstand der Messung

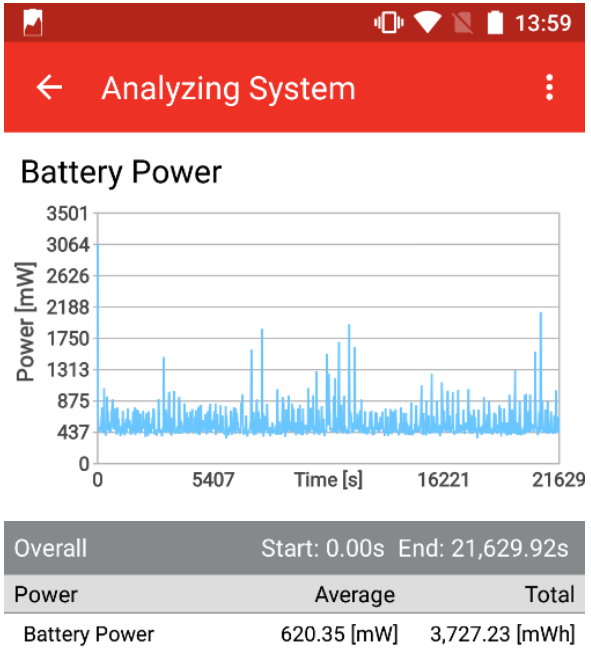
- Energieverbrauch ist lose definiert als
  a) Bedarf an Energie, um eine bestimmte Arbeit in einer Zeitspanne zu verrichten, oder
  b) allgemeine Leistung (in einer Zeitspanne umgesetzte Energie)

  $P = \frac{E}{t}$, mit $[P] = \frac{[E]}{[t]} = \frac{\mathrm{Ws}}{\mathrm{s}} = \mathrm{W}$

- Beispiel
  - Batterie mit einer Kapazität $C = 11.400\mathrm{mWh} = 3.000\mathrm{mAh} \times 3{,}8\mathrm{V}$
  - durchschnittliche Leistung eines 3G-Telefonats: $542\mathrm{mW}$
  - Zeit bis zur Entleerung der Batterie: ca. 21 Stunden
- Leistung erlaubt normalisierten Vergleich des Verbrauchs unabhängig von tatsächlicher Messdauer

Energieverbrauch – Messung

- elektrischer Schaltkreis
  - Smartphone wird an externes Messgerät angeschlossen, bspw. Oszilloskop oder Stromzähler
  - sehr akkurat, aber unpraktikabel für Geräte mit fest verbautem Akku
- in Hardware integrierte Aufzeichnung (Profiler)
  - tatsächlicher momentaner Strom und anliegende Spannung werden aus Daten des Energieverwaltungs-Chips ermittelt und in Leistung umgerechnet
  - akkurat für Zielplattform
    → bspw. Qualcomm Trepn Power Profiler für Snap-Dragon-Plattform



Energieverbauch – Messung

- Energieprofile und Statistiken
  - Gerätehersteller veröffentlichen Energieprofile (Power Profile), i.d.R. Strom I_C, den eine Komponente unter Spannung U_R zieht
  - Statistiken protokollieren die Nutzungszeit $t_{C,u}$ einer Komponente C
  - Energie ist dann

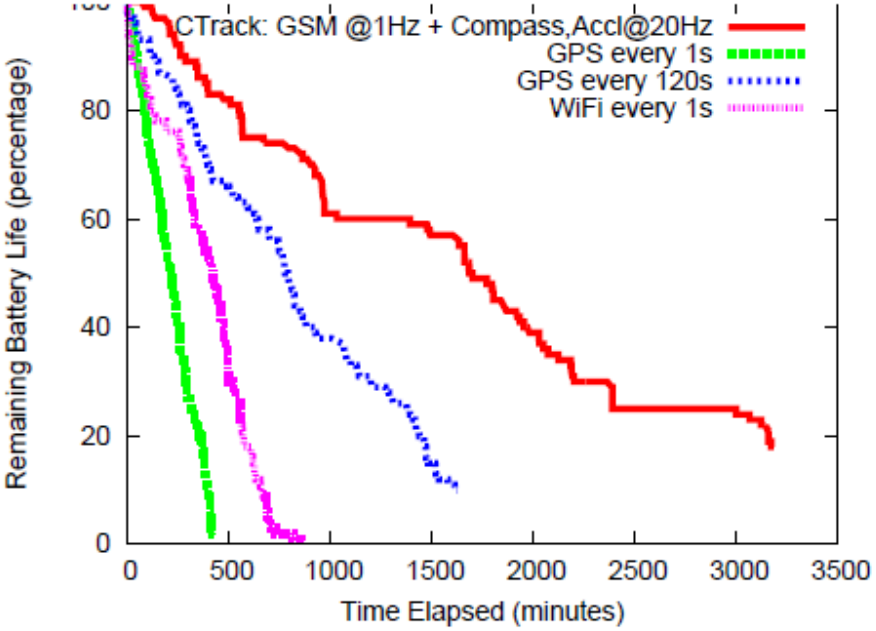
$$E = \sum_{\forall C} \left(I_C \times t_{C,u} \right)$$

mit $[E] = \mathrm{mAh}$ – eigentlich Einheit der elektrischen Ladung ($1As = 1C$), dient aber wegen der allgemein konstanten Betriebsspannung als Energieindikator

  - Statistiken können aber ggf. nicht detailliert genug sein!

Energieverbrauch – Messung

- Batterielaufzeit
  - nutzt API um Batterielaufzeit als Zeitfunktion zwischen zwei Apps zu vergleichen
    [Thiagaranjan et al.; 2011]
  - bei identischem Wertebereich ist die App mit größerer Ursprungsmenge die energieeffizientere



Energieverbrauch – Messung

- Zugriff in Android

```
IntentFilter ifilter =
   new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
int level = batteryStatus.getIntExtra(
   BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(
   BatteryManager.EXTRA_SCALE, -1);
float batteryPct = level / (float)scale;
Log.d(TAG, "Battery in percent: " + batteryPct);
```

- `BatteryManager` propagiert Batteriedetails via persistentem `Intent`
  → wegen persistentem `Intent` kein `BroadcastReceiver` nötig

Energieverbrauch – Verbraucher

- CPU
  - Taktrate bestimmt Energieverbrauch
  - alleinstehende Messung des Energieverbrauchs der CPU nicht sinnvoll → Taktrate hängt von Systemlast ab
    → besser: CPU als Teil modellbasierter Energiemessungen
- Anzeige
  - eingeschaltet hoher Verbrauch, aber essentiell für Nutzerinteraktion
  - Energieverbrauch steigt mit zunehmender Helligkeit
  - Abblenden während Nutzerinaktivität reduziert Verbrauch enorm

Energieverbrauch – Verbraucher

- Funkschnittstellen
  - 3G/LTE, WiFi, BT
  - Standby
    - benötigt Energie, um nach Beacons von Funkmasten und Access Points zu lauschen
    - Beacons sind essentiell, bspw. zeigen sie Zellwechsel an
  - Datenübertragung
    - benötigt Energie über reine Übertragung hinaus
      → Paketierung, Empfang von Bestätigungen, …
    - Verbrauch proportional zu Größe und Übertragungszeit
- Positionierung
  - GPS, WiFi, GSM, Beschleunigungssensoren
  - Verbrauch steigt mit zunehmender Genauigkeit und/oder Abtastrate

Drahtlos-kommunikation

Drahtloskommunikation – WiFi und 3G/LTE

Problem: Nutzeraktivität jetzt über 3G/LTE mit dem Server synchronisieren, oder auf WiFi-Verfügbarkeit warten?

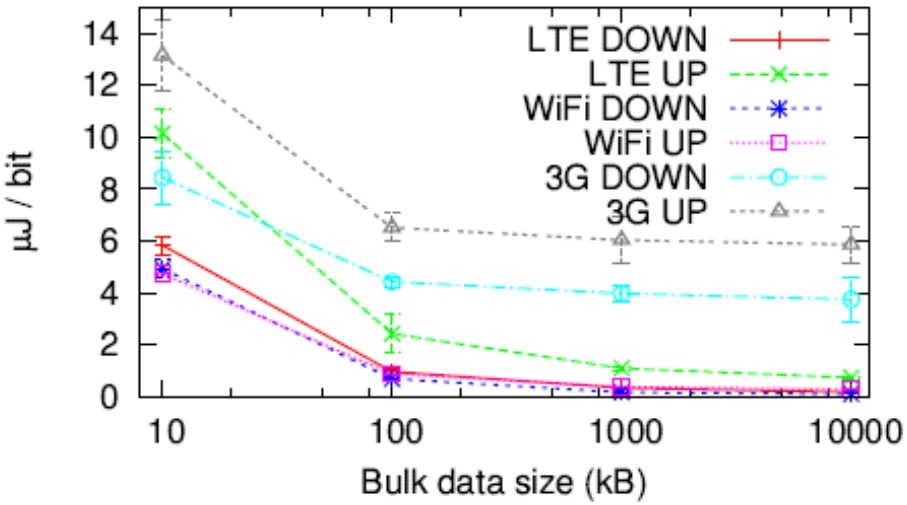
- energetisch ist auf WiFi zu warten sinnvoller
  → Energiebedarf pro Bit sinkt mit zunehmender Datenmenge [Huang et al.; 2012]
- Durchsatz erreicht volle Leitungsbandbreite nicht wegen TCP Slow Start
- Energieverhältnisse für 10MB Daten (Daten aus 2012)
  - Download

    $E_{\text{LTE,down}} = E_{\text{WiFi,down}} \times 1{,}6$

    $E_{\text{3G,down}} = E_{\text{WiFi,down}} \times 35$
  - Upload

    $E_{\text{LTE,up}} = E_{\text{WiFi,up}} \times 2{,}5$

    $E_{\text{3G,up}} = E_{\text{WiFi,up}} \times 20$



Drahtloskommunikation – Wifi und 3G/LTE

- schwaches Signal
  - Netzwerkschnittstelle kompensiert schwache Signale durch Erhöhung der Sende- und/oder Empfangsleistung (TX/RX Power)
  - wiederholte Übergaben im Netzwerk (bspw. beim Autofahren) entleeren Batterie
- Optionen für Entwickler
  - warten auf WiFi bevor mit Server synchronisert wird
  - Menge von Netzwerkanfragen in einer Warteschlange reihen und gemeinsam abarbeiten
  - Anpassen der Rate an Netzwerkanfragen
  - nächste Nutzeraktionen vorhersehen und benötigte Daten als Masse vorladen (Bulk Prefetching)

Drahtloskommunikation – WiFi Direct und Bluetooth

Problem: Synchronisieren zwischen zwei Smartphones bei intermittierendem Internetzugriff. – WiFi Direct oder Bluetooth als Alternative nutzen?

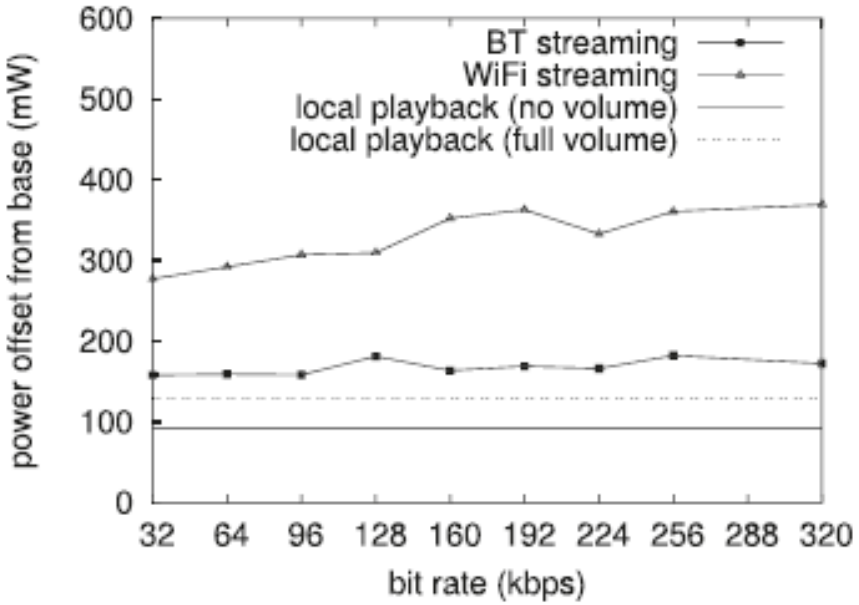
- Wahl hängt vom Durchsatz ab
- Durchsatzmaximierung
  - BT verbraucht zwei- bis dreimal weniger Energie als WiFi, aber
  - WiFi überträgt mehr Daten bei gleicher Arbeit

[Friedman et al.; 2013]

		Samsung Omnia M (S7530)				
		BT	WiFi			
			Access Point		ad hoc	
		RFCOMM	TCP	UDP	TCP	UDP
sendend	kB/s	137	1.186	1.136	1.232	1.075
	mW	520	1.568	1.544	1.600	1.560
	kB/mWs	**0,26**	0,76	0,74	**0,77**	0,69
	% Verlust			0,4		0
empfangend	kB/s	128	1.201	627	1.336	618
	mW	456	1.504	1.496	1.496	1.448
	kB/mWs	0,28	0,80	0,42	0,89	0,43
	% Verlust			58		55,9

Drahtloskommunikation – WiFi Direct und Bluetooth

- für langsamen, nahezu konstanten Durchsatz (Audiostreaming, VoIP, …)
  - BT verbraucht weniger Energie als WiFi
  - Energieverbrauch durch WiFi steigt mit dem Durchsatz
  - Energieverbrauch durch BT hängt (fast) nicht vom Durchsatz ab



(Vergleich: Streamen einer MP3-Datei per BT und WiFi sowie lokales Abspielen)

[Friedman et al.; 2013]

Drahtloskommunikation – WiFi Direct und Bluetooth

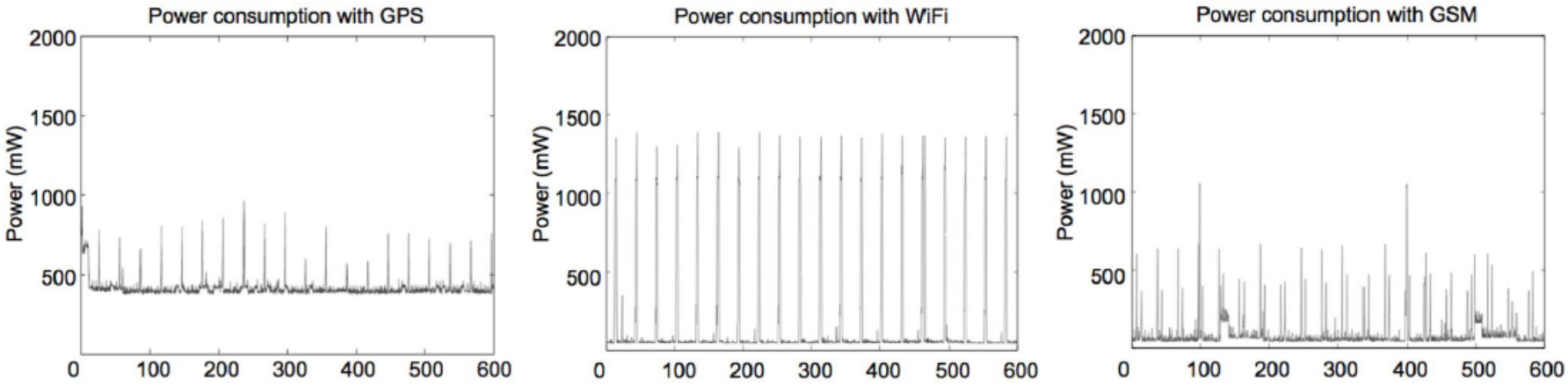
- Vergleich
  - WiFi Direct ist anfälliger für Interferenzen; kein dynamisches Kanalwechseln wie bei BT
  - WiFi Direct hat keinen Energiesparmodus wie bei WiFi (Leerlaufverbrauch 20mal höher als bei WiFi)
  - Bluetooth hat hohen Energieverbrauch beim Suchen und Gefunden-werden → sollte nur selektiv eingesetzt werden
- Option für Entwickler: Auswahl entsprechend Einsatzfall

Lokationsverfolgung

Lokationsverfolgung – GPS, WiFi und GSM

Problem: Lokationsbewusste App. Regelmäßige Positionsupdates per GPS, WiFi oder GSM durchführen?

- Energieverbrauch unterschiedlicher Lokationsanbieter



(Energieverbrauch auf Nokia N95)

[Constandache et al.; 2009]

Lokationsverfolgung – GPS, WiFi und GSM

- Abwägen von Genauigkeit gegen Batterielaufzeit [Constandache et al.; 2009]
  - GPS: 10m gegen 9 Stunden
  - WiFi: 40m gegen 40 Stunden
  - GSM: 400m gegen 60 Stunden
- Gründe für hohen Energieverbrauch bei GPS
  - Empfang von Daten von 4 oder mehr Satelliten auf sehr langsamen Kanal (50 Bit/s) über längeren Zeitraum
  - während GPS aktiv ist kann Lokationssystem nicht in Ruhezustand wechseln, um Energie zu sparen
  - Lokationsanwendungen sind berechnungs- und damit energieintensiv

Lokationsverfolgung – GPS, WiFi und GSM

- Optionen für Entwickler
  - Prioritäten (bspw. Android Google Location Services API)
    - `PRIORITY_HIGH_ACCURACY`: höchste Präzision, vornehmlich GPS
    - `PRIORITY_BALANCED_POWER_ACCURACY`: ca. 100m, WiFi & GSM
    - `PRIORITY_NO_POWER`: App löst keine Lokalisierungen aus, empfängt aber Lokationsaktualisierungen vom Location Service
  - zu beachtende Beschränkungen (bspw. in Android 8.0)
    - Lokation für Hintergrundanwendungen nur einige Male pro Stunde verfügbar → Energiespardiktat vom Betriebssystem
    - Vordergrundanwendungen sind nicht betroffen
  - dynamische Auswahl des Lokationsanbieters und der Abfragefrequenz

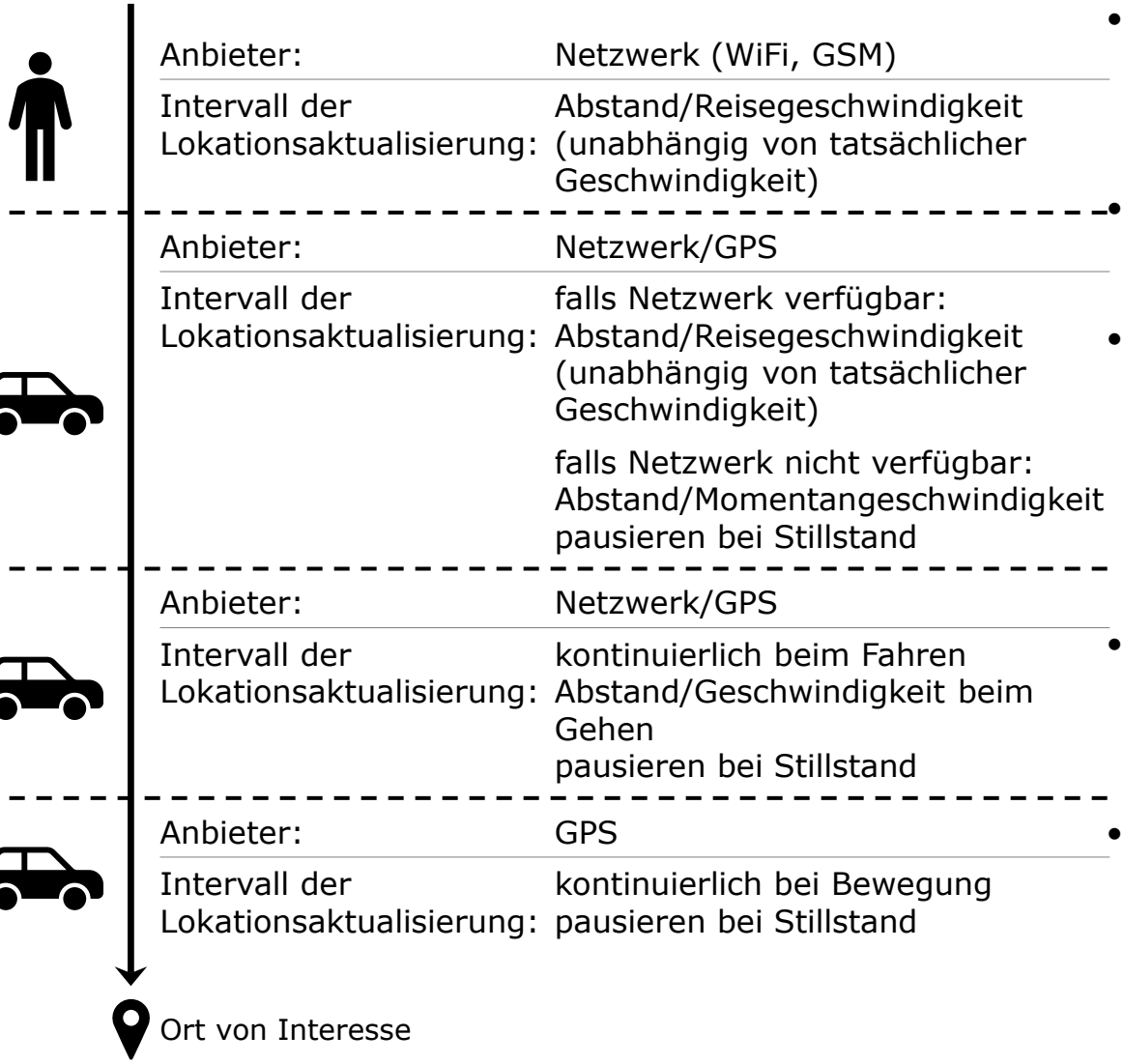
Lokationsverfolgung – dynamische Anbieter/Frequenz-Auswahl

Beispiel: Benachrichtigung bei Nähe zu Ort von Interesse (POI)

- Grundidee
  - umso näher am POI, umso höher die Präzision und Abfragefrequenz
  - umso schneller in Bewegung, umso schneller am POI
    → höhere Präzision und Abfragefrequenz
- dynamische Bestimmung der Abfragefrequenz und des Lokationsanbieter basierend auf [Bulut & Demirbas; 2013]
  - Abstand zum POI und
  - Fortbewegungsmittel
- kann durch Vermeidung von GPS Batterielaufzeit um bis zu 75% erhöhen [Bulut & Demirbas; 2013]

Lokationsverfolgung – dynamische Anbieter/Frequenz-Auswahl

zurückzulegende Strecke



- auf den POI zubewegend feststellbar:
- Änderung des Lokationsanbieters
- selektive Nutzung des Beschleunigungs-sensors, bspw. beim Bestimmen der Momen-tangeschwindigkeit
- selektive Nutzung von GPS, bspw. in Nähe vom POI
- Änderung der Aktuali-sierungsfrequenz, bspw. bei Stillstand pausierend

unerwartete Energieverbraucher

Definition: Energy Bug

Ein Fehler in einer Applikation, dem Betriebssystem oder der Firmware, der zu unerwartetem Energieverbrauch im Gesamtsystem führt.

[Pathak et. al.; 2011]

⇒ Der so definierte Energy Bug umfasst Hardware-Fehler explizit nicht!

No-Sleep Bug (1/2)

No-Sleep Bug (NSB; Insomnie)

- normalerweise friert das Betriebssystem nach kurzer Nutzerinaktivität das System ein, um Energie einzusparen
- über API bereitgestellte Bereitschaftssperre (Wake Lock) erlaubt Komponenten von Nutzeraktivität unabhängig aktiv zu bleiben
- App nimmt Wake Lock und gibt diesen nicht mehr frei

- Beispiel: PARTIAL_WAKE_LOCK in Android

```
PowerManager pm = (PowerManager)
   getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl =
   pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "My Tag");
wl.acquire();
/* Bildschirm aus
   ... */
wl.release();
```

Falls PARTIAL_WAKE_LOCK nicht freigegeben wird, wird die CPU durchgehend aktiv bleiben, selbst wenn der Nutzer die Applikation zurückstellt.

No-Sleep Bug (2/2)

- NSB lassen sich grob in drei Kategorien einteilen
  - Code Path NSB: Entwickler setzt Wake Lock, aber vergisst diesen später wieder freizugeben (siehe vorherige Folie)
  - Race Condition NSB: Beziehen und Freigeben von Wake Lock folgt einer inkorrekten Ablaufreihenfolge wegen Thread-Wettlauf und fehlendem Semaphor
  - Sleep Dilation NSB: Freigabe eines Wake Locks wird durch langen Kodeblock unnötig verzögert

To Wake Lock or not to Wake Lock?

- Google-Empfehlung: gänzlich auf Wake Locks verzichten!
  - in Activities mit OS-Flags arbeiten
    - Beispiel: `FLAG_KEEP_SCREEN_ON`
  - Scheduled Jobs statt Services verwenden
    - Job Scheduler setzt und entfernt Wake Locks automatisch
  - für Zeit-Trigger und Alarme: `AlarmManager` nutzen
    - hält *einen* Wake Lock für alle an ihn delegierten Aufgaben
    - opportunistische, wohlgesonnene Wake-Lock-Unterbrechung in Abstimmung mit anderen Managern und MutEx zu vermeiden
- außer `PARTIAL_WAKE_LOCK` sind alle Wake Locks in Android inzwischen *deprecated* und sollen nicht mehr verwendet werden!

Wake Lock korrekt nutzen

- nur entsprechend der benötigten Eskalation
  (falls andere Wake Lock als `PARTIAL_WAKE_LOCK` verwenden müssen)
- nur notwendige Anweisungen zwischen Bezug und Freigabe
  (d.h. kurze Kodeblöcke schreiben!)
- nur Thread-sicher arbeiten und von Prüfvariablen abhängige Wake-Lock-Freigabe mit Semaphor schützen
- Timeout in `wakeLock.acquire()` setzen
- Wake Lock statischen, deskriptiven Tag geben
  (Android referenziert Wake Locks über Namen → einfacheres Debuggen)
- Sicherheitnetz verwenden:

```
try { … }
finally {
  wakeLock.release();
}
```

Achtung vor No-Sleep Bug v2

- entsteht durch exzessive Nutzung von `AlarmManager*_WAKEUP`
- durch ständiges Aufwachen und Scheduling vieler Apps geht Vorteil ggü. Wake Locks verloren → Batterie leert sich trotzdem schnell
- Vermeidungsstrategien
  - `WAKEUP`-Frequenz auf das nötigste reduzieren (ab Android 11 begrenzt auf 1 Trigger per 15 Minuten)
  - generell durch Alternativen (Firebase Cloud, Messaging, WorkManager API, JobScheduler oder SyncManager) ersetzen
    - automatisches Backoff und Retry
    - Scheduling kann an Bedingungen geknüpft werden (Kontext wie „Netzwerk", „Ladend", …)
    - befolgt Vorgaben der Zustände „Doze" und „Standby"
    - automatische Wake-Lock-Behandlung im Hintergrund

Loop Bug (1/2)

Loop Bug (Schleifenfehler)

- Thread wartet auf ein Ereignis um fortzusetzen, aber Erkennen des Ereignisses ist an Prüfvariable gebunden
- Thread liest Prüfvariable regelmäßig bis Änderung eintritt
  → verbraucht unnötig CPU-Zyklen (klassisches Busy Waiting)
- Beispiele
  - Kalender-Server stürzt ab → Applikation synchronisiert ständig eigene Änderungen mit sich selbst, da Persistierung nicht bestätigt wurde
  - Server-API wurde aktualisiert → Applikation arbeitet mit alter API-Definition und wartet auf Ereignisse an der falschen Schnittstelle
  - allgemeines Beispiel:

```
public void longRunningMethod (Object o) {
   while (someVariable != certainWakeLockValue)
      someVariable = PowerManager.WakeLock;
   doSomething();
}
```

Schleifen – Loop Bug (2/2)

- Android, Fuchsia, HarmonyOS und iOS sind Ereignis-getrieben
  → Nutzung der Ereignisbehandlung des Betriebssystems

- Optionen für Entwickler
  - Futures, Promises und Rückrufe
  - präemptive Ausführung
    → Vorsicht: präemptive Ausführung* kann unsicher sein
  - defensive Freigabe und/oder optimistisches Warten

*: z.B. CVE-2017-5715, CVE-2017-5753, CVE-2017-5754, CVE-2017-6289, CVE-2018-3615, CVE-2018-3620, CVE-2018-3639, CVE-2018-3640, CVE-2018-3646, CVE-2018-3665, CVE-2018-3693, CVE-2018-9056, CVE-2022-23825, CVE-2022-29900, CVE-2022-40982, CVE-2023-20569, CVE-2023-32543
(auch bekannt als Spectre, Meltdown, ZombieLoad, Inception, RETbleed, Zenbleed, Downfall, etc. pp.)

Immortality Bug

Immortality Bug (Unsterblichkeit)

- Applikation lässt sich nicht im Nutzerkontext beenden
- Applikation startet nach Beendigung einfach erneut
- oft auf Grund falsch gesetzter Rechte oder Eigenschaften
  - System Service
  - User Service
  - persistente Rückrufe
  - Self-Intent oder Protected Execution

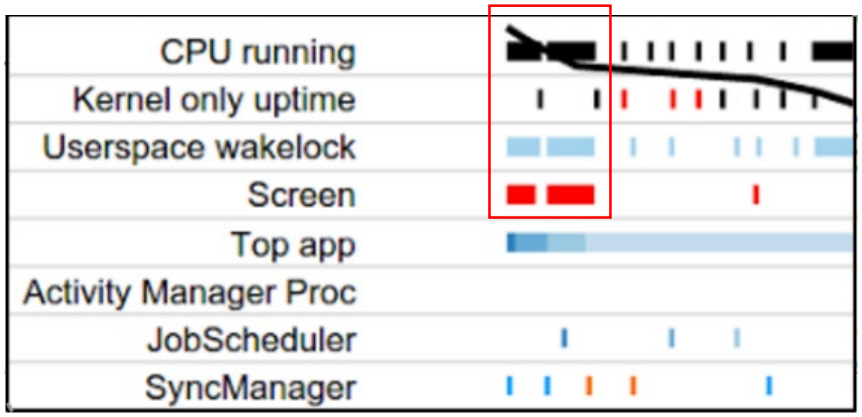
Optionen für Anwender

- nicht verwendete Apps terminieren (ggf. Force Close)
- App-Aktivität genau beobachten, z.B. unter Android 9
  - Settings → Accounts → Synchronisation
  - in Applikationen selbst:  oder 
- ggf. Fehlerbericht an Entwickler senden

Optionen für Entwickler

- Wake Locks so sparsam wie möglich verwenden
- Wake Lock früh wie möglich freigeben
- Freigabe von Wake Locks nicht an Bedingungen knüpfen
- falls bedingte Freigaben eingesetzt werden:
  - Zwei-Generäle-Problem
  - Idempotenz und Isoliertheit beachten
- Analyse der Energieereignisse, bspw. mit Android Power Historian

Battery Historian



- starker Abfall der Batteriekapazität (schwarze Linie)
- Grund: CPU ist aktiv, eine App hat einen Wake Lock und der Bildschirm ist an

⇒ Battery Historian hilft zu erkennen, welche Ereignisse eintreten, wie sie korreliert sind und wie sie zum Energieverbrauch beitragen

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

Von einer Applikation belegter, aber nicht mehr verwendeter Speicher.

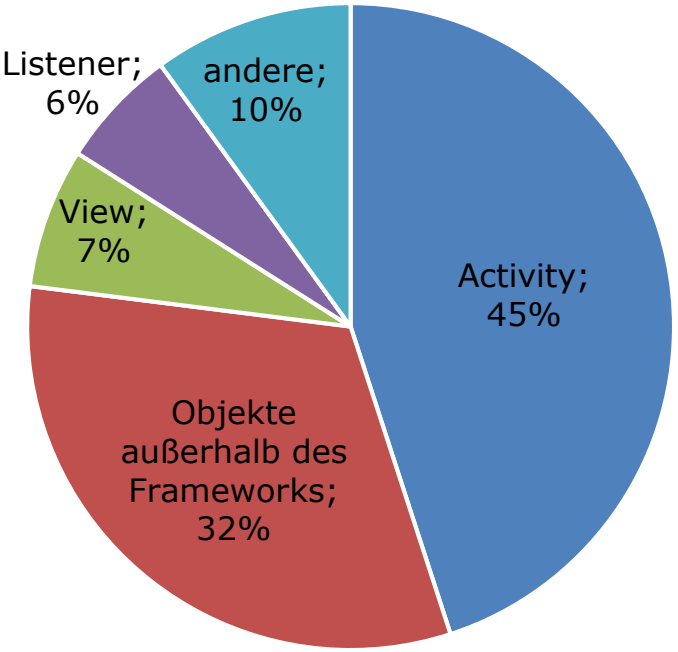
⇒ Der so definierte Memory Leak umfasst Hardware-Fehler explizit nicht!

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

- typischerweise interagieren Nutzer nur kurz mit einer Applikation bevor sie zur nächsten wechseln
- wiederholtes grundständiges Laden einer Applikation dauert
- Betriebssystem hält Applikationen ohne aktuelle Nutzerinteraktion im Hintergrund, bspw. im Cache
  - feste Queue-Größe
  - LRU-sortiert
- sobald Speichergrenzen erreicht: Applikationen nach LRU terminieren
- umso mehr Speicher eine App nutzt/leckt, umso weniger Apps passen in den Hintergrund-Cache, umso häufiger müssen Apps terminiert und neugestartet werden, umso mehr Energie wird verbraucht

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

- Android: Activity leckt am meisten
  [Xia et al.; 2012]
- Grund: asynchrone Komponente wird gestartet, aber Activity wird terminiert bevor Komponente fertig wird
- Beispiel: Handler, der eine Nachricht nach einer definierten Verzögerung sendet



nach [Pathak et al.; 2011]

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

```
public class HandlerExample extends AppCompatActivity {
    private Handler mLeakyHandler = new Handler();
    private TextView myTextBox;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_samples);
        myTextBox = (TextView) findViewById(R.id.tv_handler);
        // Nachricht setzen, aber Ausführung 10 Sekunden verzögern
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                myTextBox.setText("fertig");
            }
        }, 1000 * 10);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}
```

Leckender Handler

- anonymer Kode hält Referenz auf Activity
- falls Activity während Rückhaltung der Nachricht zerstört wird, kann sie nicht vom Garbage Collector bereinigt werden

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

```
public class HandlerExample extends AppCompatActivity {
    private Handler mLeakyHandler = new Handler();
    private TextView myTextBox;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_samples);
        myTextBox = (TextView) findViewById(R.id.tv_handler);
        // Nachricht setzen, aber Ausführung 10 Sekunden verzögern
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                myTextBox.setText("fertig");
            }
        }, 1000 * 10);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        mLeakyHandler.removeCallbacksAndMessages(null);
    }
}
```

Lösung A: Effekte beseitigen

`removeCallbacksAndMessages( )` in der `onDestroy( )`-Methode der Activity beseitigt die Effekte des Lecks, nicht aber das Leck selbst.

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

```
public class HandlerExample extends AppCompatActivity {
    private Handler mLeakyHandler = new Handler();
    private TextView myTextBox;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_samples);
        myTextBox = (TextView) findViewById(R.id.tv_handler);
        // Nachricht setzen, aber Ausführung 10 Sekunden verzögern
        mLeakyHandler.postDelayed(new Runnable() {
            @Override
            public void run() {
                myTextBox.setText("fertig");
            }
        }, 1000 * 10);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}
```

Lösung B: nichtleckender Handler

Konflikt vorab vermeiden und Kodeblock durch innere, statische Methode mit schwacher Referenz ersetzen

(Kode in blauer Box durch Kode auf nächster Folie ersetzen!)

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

```
...
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    ...
    // Nachricht setzen, aber Ausführung 10 Sekunden verzögern
    mLeakyHandler.postDelayed(new MyRunnable(myTextBox), 1000 * 10);
}
private static class MyRunnable implements Runnable {
    WeakReference<TextView> myTextBox;
    public MyRunnable(TextView myTextBox) {
        this.myTextBox = new WeakReference<TextView>(myTextBox);
    }
    @Override
    public void run() {
        TextView mText = myTextBox.get();
        if (mText != null) {
            mText.setText("fertig");
        }
    }
}
...
```

statische, innere Klasse

schwache Referenz

Speichern des TextView in lokale Variable da schwach referenziertes Objekt jederzeit geleert oder bereinigt werden kann

unerwartete Energieverbraucher – Speicherleck (Memory Leak)

- Optionen für Entwickler
  - prüfen auf Memory Leaks, bspw. mit Android Studio Memory Profiler
  - kennen des Laufzeitzyklus' einer Activity bspw. bei Änderung der Bildschirmausrichtung oder beim Wechseln zwischen Apps
  - befolgen des Leitfadens für speichereffizienten Kode

Zusammenfassung und Aufgaben

Zusammenfassung

- Während Energieverbrauch für sich wichtig ist, muss er gegen Funktionalität, Nutzungserlebnis und Performanz abgewogen werden.
- Der Entwickler muss sich der energetischen und funktionalen Implikationen seiner Entscheidungen bewusst sein.

Aufgaben

- Diskutieren Sie mit Ihren Kommilitonen, weshalb beim Senden und Empfangen von Daten unterschiedlich viel Energie benötigt wird.
- Diskutieren Sie mit Ihren Kommilitonen Szenarien in denen Applikationen Wake Locks zwingend benötigen.
  → Wann kann auf sie verzichtet werden?
- Diskutieren Sie mit Ihren Kommilitonen die Vor- und Nachteile der Strategien zur Vermeidung von Wake Locks!
- Implementieren Sie in Android Studio ein einfache App zum Ausprobieren der vorgestellten Vermeidungsstrategien bzgl. Energy Bugs. Beispielsweise können Sie einen Wecker, Countdown o.ä. implementieren.

Referenzen

Friedman et. al. (2013). On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones. IEEE Transactions on Mobile Computing, 12(7), 1363–1376.

Qualcomm Trepn Power Profiler, https://developer.qualcomm.com/software/trepn-power-profiler

Thiagarajan et. al. (2011). Accurate, low-energy trajectory mapping for mobile devices. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (pp. 267--280). USENIX Association.

Monitoring the Battery Level and Charging State, https://developer.android.com/training/monitoring-device-state/battery-monitoring.html

Huang et. al. (2012). A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (pp. 225--238). ACM.

Referenzen

Constandache et.al (2009). EnLoc: Energy-efficient localization for mobile phones. In Proceedings - IEEE INFOCOM (pp. 2716–2720).

Changing Location Settings, https://developer.android.com/training/location/change-location-settings.html

Bulut, M. F., & Demirbas, M. (2013). Energy Efficient Alert On Android. GLOBECOM - IEEE Global Telecommunications Conference, (March), 2816–2821.

Pathak et. al. (2011). Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In HotNets-X: Proceedings of the 10th ACM Workshop on Hot Topics in Networks (p. 5:1--5:6). ACM.

Analyzing Power Use with Battery Historian, https://developer.android.com/topic/performance/power/battery-historian.html

Referenzen

Analyzing Power Use with Battery Historian, https://developer.android.com/topic/performance/power/battery-historian.html

Overview of Android Memory Management, https://developer.android.com/topic/performance/memory-overview.html

Xia, M., He, W., Liu, X., & Liu, J. (2013). Why Application Errors Drain Battery Easily? A Study of Memory Leaks in Smartphone Apps. In Proceedings of the Workshop on Power-Aware Computing and Systems - HotPower '13 (pp. 1–5).

Memory leak example, http://www.codexpedia.com/android/memory-leak-examples-and-solutions-in-android/

Android Studio Memory Profiler, https://developer.android.com/studio/profile/memory-profiler.html

Memory-efficient code, https://developer.android.com/topic/performance/memory.html

Optimizing Battery Life in Android, https://developer.android.com/training/monitoring-device-state/index.html