CSE 486/586: Distributed Systems
Android Architecture and Development (Part 1)

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo
Moments Ago…

The **end-to-end principle** is powerful.

The **transport layer** provides additional functionality:
- **TCP**: App. differentiation, flow control, reliability
- **UDP**: Application differentiation

TCP guarantees are **end-to-end at the host level**.
Introduction

Today we will discuss:

- The architecture of an Android app
- Concurrency and races
- Event-driven techniques
- The handout code for Project 1

...or however much of that we accomplish!
Android Architecture

An Android application is an **event-driven, loosely-coupled** system of inter-operating components.

*It’s almost a distributed system all by itself!*

Each application provides a set of:

- Activities
- Services
- BroadcastReceivers
- ContentProviders

It may also have other components.
Callbacks

Each of those major Android components consists of a set of **callbacks** that determine its behavior.

There is no `main()`-style function or method!\(^1\)

A designated **Activity** provides the application entry point.

Each callback must **return quickly** to avoid hanging the application!

This has implications on design.

\(^1\)Sort of…
Callbacks Example

```java
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);
    protected void onStart();
    protected void onRestart();
    protected void onResume();
    protected void onPause();
    protected void onStop();
    protected void onDestroy();
}
```
Metadata

The file AndroidManifest.xml describes:

- The main Activity
- Special permissions required
- Services provided

This is a declarative description of the application.
Manifest Example (abbreviated)

<manifest xmlns:android="http://..."
    package="edu.buffalo.cse.cse486586.simplemessenger"
    <uses-permission
        android:name="android.permission INTERNET"/>
    <application>
        <activity
            android:name=".SimpleMessengerActivity">
            <intent-filter>
                <action android:name="..."/>
                <category android:name="..."/>
            </intent-filter>
        </activity>
    </application>
</manifest>
Activity Lifecycle

Copyright Google Developer Training Team, CC BY-NC 4.0
Event-Driven Execution

Many Android components, and in particular all UI Activities, are event-driven.

This means that:

- There is only one thread of execution
- The Activity responds serially to events
- Event durations must be finite

Each event is handled by a callback.

There is no main() because it’s an event loop!
Event Loops

The main event loop proceeds as follows:

1. Block waiting for an event to occur
2. Identify the event, find a callback to handle it
3. Invoke the callback
4. Go to 1
Many Android events are messages called **Intents**.

Remember this?

*We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.* [1]

Intents can:

- Notify any interested parties of a condition/action
- Request a specific action from another Activity/Service
Blocking

Facts:

1. Events cannot block indefinitely
2. Java I/O is blocking

So ... how do we do I/O? (E.g., Socket.accept()!)

Android also has AsyncTasks, which are threads.

- Create an AsyncTask
- Block in it
- Notify the event loop via a message (e.g., publishProgress())

\[ ^2 \text{Channels provide a non-blocking API, but the program must still block for the next Channel activity.} \]
The AsyncTask Interface

AsyncTask provides:

- A wrapper for Java thread
- A way to communicate with the UI thread
- Some state transition callbacks

In particular, calling publishProgress() causes:

- A Progress message to be sent to the UI thread
- The UI thread to invoke onProgressUpdate(Progress...)

... While doInBackground(Params...) is invoked on the thread.
Flavors of AsyncTask

Instances of AsyncTask can be run:

- **Serially on a shared thread**, with SERIAL_EXECUTOR
- **On a dedicated thread**, with THREAD_POOL_EXECUTOR

Tasks that are expected to block should use a dedicated thread!

There are *about* 5 dedicated threads available per App [3].
Concurrency

There are two major types of concurrency in an Android application.

- Interleaved events
- AsyncTasks

Events on a single thread are mutually exclusive.
E.g., all UI thread events.
Data Races

```c
int x, y;

void addx() {
    x += 2;
}

void swapXY() {
    int tmp = x;
    x = y;
    y = tmp;
}
```

What happens if `addx()` is called during execution
- Before line 7
- Between lines 7 and 8?
- Between lines 8 and 9?
Protecting from races

```java
int x, y;
synchronized void addx() {
    x += 2;
}

tsynchronized void swapXY() {
    int tmp = x;
    x = y;
    y = tmp;
}
```

What happens if `addx()` and `swapXY()` are synchronized?

What happens if they’re unsynchronized event callbacks?
Example Application — Event Loop
Mutual Exclusion

Mutual exclusion is a property of code regions.\(^3\)

Two regions are mutually exclusive if they cannot run simultaneously.

This property is pairwise between regions.

Actions are atomic if they appear to be mutually exclusive with all other regions.

\(^3\)It’s more complicated than that, but for now.
Defending Against Races

Mutual exclusion can be used to prevent data races. Recall `swapxy()` and `addx()`.

Regions of code that:

- **mutate** shared state, or
- rely on **stability** of shared state

should be **mutually exclusive**.
Event Loops and Concurrency

We said that events are **mutually exclusive**. What does that mean?

---

4 On the same event loop, that is!
Event Loops and Concurrency

We said that events are mutually exclusive. What does that mean?

Each event starts and runs to completion alone.

Therefore, no synchronization is required between events.\(^4\)

Note: You can’t even add synchronized to a callback!

\(^4\)On the same event loop, that is!
Event Loops and Concurrency

We said that events are mutually exclusive. What does that mean?

Each event starts and runs to completion alone.

Therefore, no synchronization is required between events.\(^4\)

Note: You can’t even add synchronized to a callback!

Question: What about synchronizing between events and AsyncTasks?

\(^4\)On the same event loop, that is!
Java Memory Model


Doug Lea has some good explanations of the JMM.

Explicit mutual exclusion is provided by Java monitors.

Monitors are entered via the synchronized keyword.
Development Guidelines

- Learn the APIs: read the API documentation!
- Learn the tools
  - `adb`, `emulator`
  - The Android Studio debugger
- Become familiar with the environment
  - Environment variables
  - The shell
- Use good practices
  - Write clean, documented code
  - Maintain loop invariants
  - Iteratively refine your application
Summary

Android Architecture:
- Android is **event-driven**.
- The **manifest** describes the application.
- Threads are provided by **AsyncTask**.

Concurrency:
- The JMM defines **Java monitors**.
- Events on an event loop are **mutually exclusive**.
- Shared state must be protected.
Optional Readings

