Writing OS Updater App for Android

Aayush Gupta

Agenda

OS updates & updater app

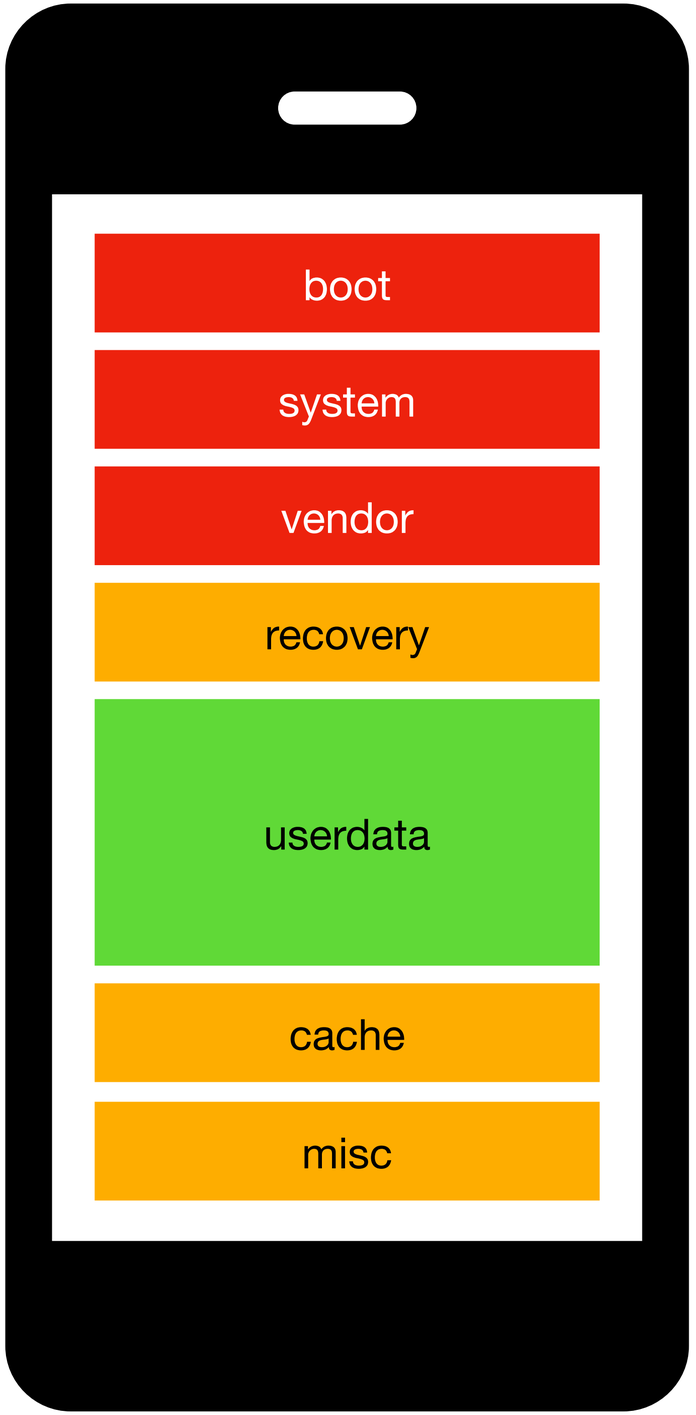
- Legacy system updates
- Seamless updates
- Writing an OS updater app

Legacy System Updates

Legacy system updates

Non-A/B devices

- Devices launched before Android 8.0 are known as non-A/B devices
- Recovery handles the installation of updates, while cache and misc partitions store commands and logs



Legacy system updates (contd.)

- Device checks and downloads update (either in cache or userdata)
- Update’s signature is verified with system
- Device reboots into recovery mode while loading commands stored in cache
- Recovery (binary) verifies signature of update and installs it
- Device reboots, system updates recovery partition with new image if required

Issues with legacy system updates

- Updater app needs to handle storage space issues
- User is unable to use the device while updating
- After update, rebooting takes longer compared to normal boot
- On update failure, device is affected and user needs to visit service centre in most cases

Seamless Updates

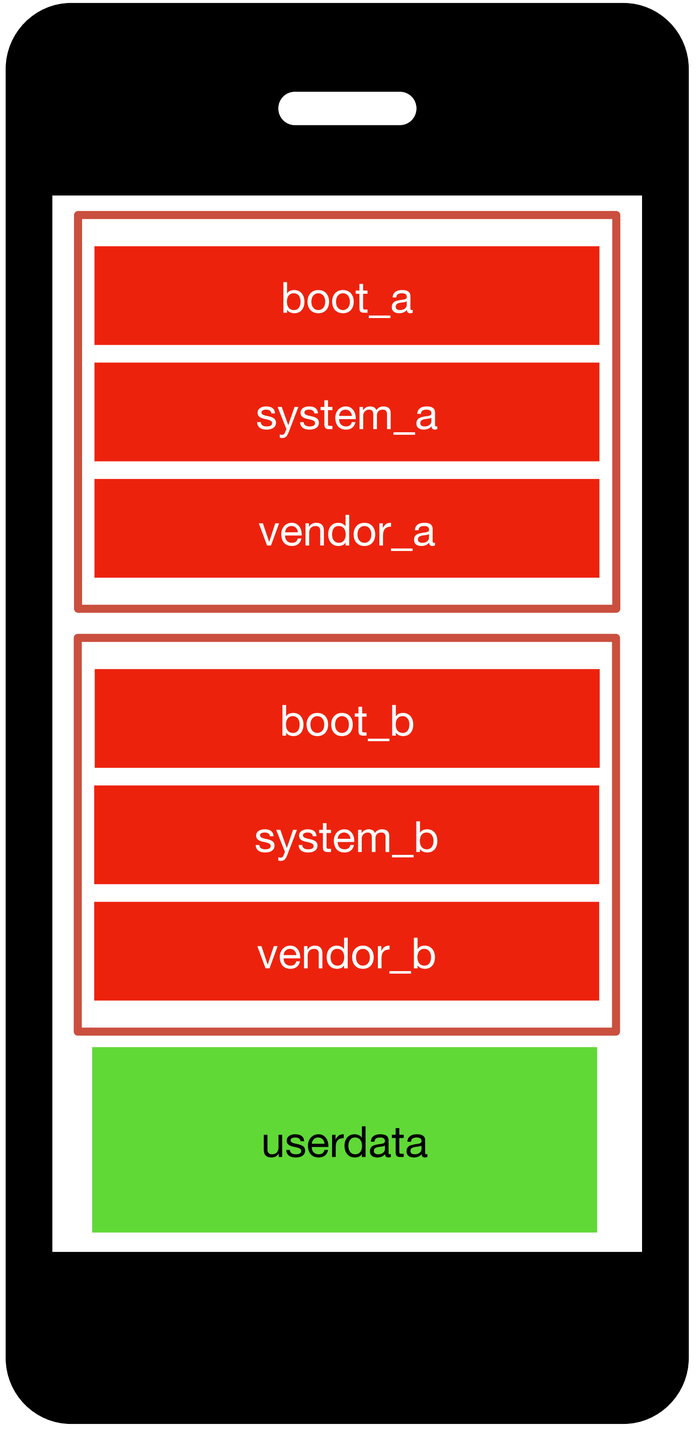
Seamless updates

A/B devices

- New way to update OS on device since Android 8.0+
- Ensures device has a working bootable system in case of update failure
- Updates are installed in background without affecting device usability
- Storage space issues can be avoided with streaming updates

Seamless updates (contd.)

- A/B system updates contains two set of partitions for running system
- Partitions are also known as slots (slot_a, slot_b)
- System runs on *current* slot while other slot remains *unused*
- Updates are installed by Updater app in *unused* slot



Seamless updates (contd.)

- A/B updates uses a background daemon called *update_engine*
- *update_engine* is responsible for streaming/installing the update
- *update_engine* can either install update from a local file or stream it directly from remote server
- Once update is installed in unused slot, *update_engine* marks it as active
- System switches to active slot on reboot

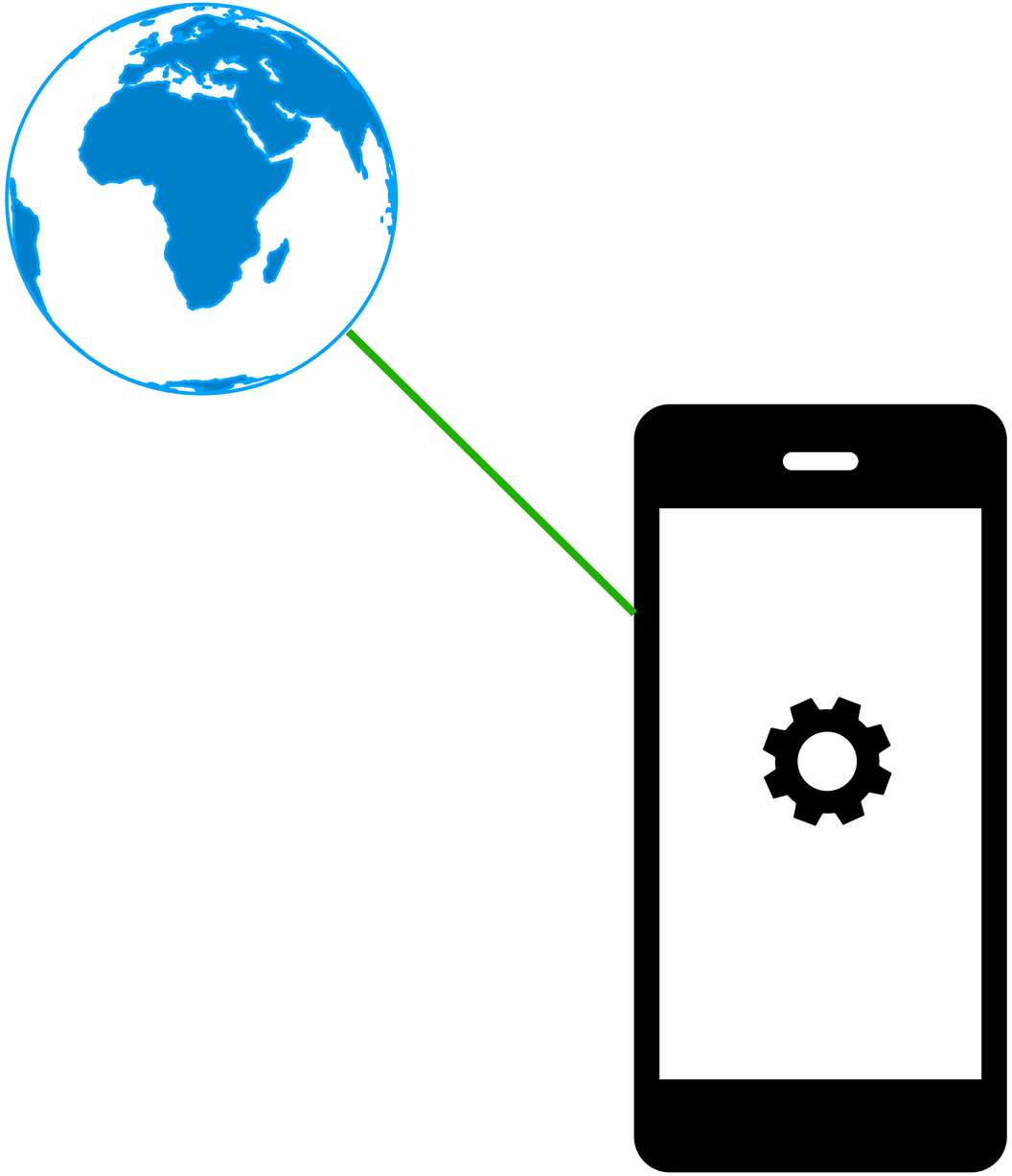
Writing an OS Updater App

AOSP sample app

- Exists in AOSP's ***bootable/recovery*** repository
- Showcases basic APIs usage
- Provides python script to generate update config
- Not compatible with Gradle build system
- Written in Java and last updated 4+ years ago

Real world use-case

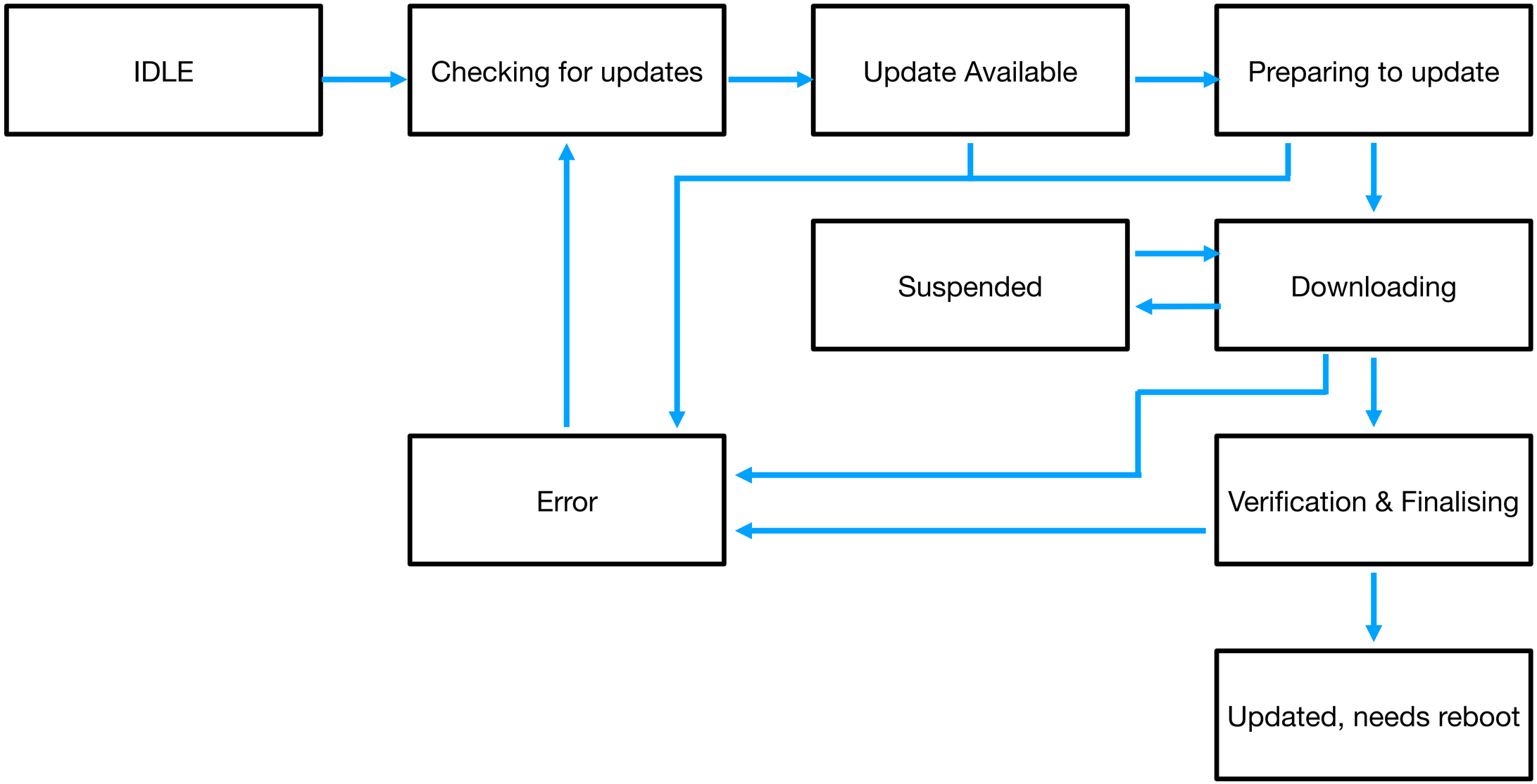
- Updater needs to connect with remote server for downloading and installing updates
- Periodic automatic updates check and install is required
- Need to handle different kind of errors



CalyxOS's SystemUpdater

- Written in Kotlin and Material3
- UI shows different elements based on **status code** emitted in a StateFlow
- Codes emitted in flow contains both custom and update_engine codes to accommodate operations in different stages

Update stages



Update status (API & App)

```
/**
 * Possible status of the Update
 *
 * This enum holds a combination of both status supplied by update_engine and
 * custom ones required for this app. Status from update_engine are followed by custom.
 */
// Keep in sync with: system/update_engine/client_library/include/update_engine/update_status.h
enum class UpdateStatus {
    IDLE, // 0
    CHECKING_FOR_UPDATE, // 1
    UPDATE_AVAILABLE, // 2
    DOWNLOADING, // 3
    VERIFYING, // 4
    FINALIZING, // 5
    UPDATED_NEED_REBOOT, // 6
    REPORTING_ERROR_EVENT, // 7
    ATTEMPTING_ROLLBACK, // 8
    DISABLED, // 9
    NEED_PERMISSION_TO_UPDATE, // 10
    CLEANUP_PREVIOUS_UPDATE, // 11
    SUSPENDED, // custom: event when update is suspended
    PREPARING_TO_UPDATE, // custom: event sent during payload verification, fetching props
    FAILED_PREPARING_UPDATE // custom: event when payload verification or fetching props fails
}
```

Listening callbacks (API)

```
/**
 * Callback function for UpdateEngine. Used to keep the caller up to date
 * with progress, so the UI (if any) can be updated.
 */
@SystemApi
public abstract class UpdateEngineCallback {

    /**
     * Invoked when anything changes. The value of {@code status} will
     * be one of the values from {@link UpdateEngine.UpdateStatusConstants},
     * and {@code percent} will be valid [TODO: in which cases?].
     */
    public abstract void onStatusUpdate(int status, float percent);

    /**
     * Invoked when the payload has been applied, whether successfully or
     * unsuccessfully. The value of {@code errorCode} will be one of the
     * values from {@link UpdateEngine.ErrorCodeConstants}.
     */
    public abstract void onPayloadApplicationComplete(
            @UpdateEngine.ErrorCode int errorCode);
}
```

Listening callbacks (contd.)

```
init {
    // restore last update status to properly reflect the status
    restoreLastUpdate()

    // handle status updates from update_engine
    updateEngine.bind(this)
    GlobalScope.launch {
        updateStatus.onEach {
            when (it) {
                UpdateStatus.CHECKING_FOR_UPDATE -> {}
                else -> {
                    sharedPreferences.edit(true) { putString(UPDATE_STATUS, it.name) }
                }
            }
        }.collect()
        updateProgress.collect()
    }
}
```

Checking for updates

```
class UpdateWorker @AssistedInject constructor(
    @Assisted val appContext: Context,
    @Assisted workerParams: WorkerParameters,
) : CoroutineWorker(appContext, workerParams) {

    companion object {
        const val WORK_NAME = "UpdateWork"
    }

    override suspend fun doWork(): Result {
        Intent(appContext, SystemUpdaterService::class.java).also {
            it.action = SystemUpdaterService.CHECK_AND_APPLY_UPDATES
            appContext.startService(it)
        }
        return Result.success()
    }
}
```

Preparing to update (Verifying metadata)

```
metadataFile.createNewFile()
val connection = URL(url).openConnection() as HttpsURLConnection
// Request a specific range to avoid skipping through load of data
// Also do a [-1] to the range end to adjust the file size
connection.setRequestProperty(
    "Range",
    "bytes=${packageFile.offset}-${packageFile.offset + packageFile.size - 1}"
)
connection.inputStream.use { input ->
    metadataFile.outputStream().use { input.copyTo(it) }
}
if (!updateEngine.verifyPayloadMetadata(metadataFile.absolutePath)) {
    _updateStatus.value = UpdateStatus.FAILED_PREPARING_UPDATE
    return@withContext Result.failure(Exception("Failed verifying metadata!"))
}
return@withContext Result.success(true)
```

Downloading

```
val propertiesFile = updateConfig.find { it.filename == payloadProperties }
val payloadFile = updateConfig.find { it.filename == payloadBinary }

if (propertiesFile != null && payloadFile != null) {
    val properties = fetchPayloadProperties(updateConfig.url, propertiesFile)
    if (properties.isSuccess) {
        _updateStatus.value = UpdateStatus.DOWNLOADING
        updateEngine.applyPayload(
            updateConfig.url,
            payloadFile.offset,
            payloadFile.size,
            properties.getOrDefault(emptyArray())
        )
    }
}
```

Suspending & resuming

```
fun suspendUpdate() {
    updateEngine.suspend()
    _updateStatus.value = UpdateStatus.SUSPENDED
}

fun resumeUpdate() {
    restoreLastUpdate()
    updateEngine.resume()
}
```

Errors

- Common errors that can occur are usually related to permissions
- Ensure the OS uses proper CA certificates
- Write SELinux rules for *update_engine* and *app*
- Verify metadata and other files have proper rw permissions
- Allow updater app to run in background for proper operations

Thank You!