Priority Inheritance during Remote Procedure Calls in Real-Time Android using Extended Binder Framework

Igor Kalkov, Alexandru Gurghian, Stefan Kowalewski
Informatik 11 - Embedded Software
Ahornstr. 55
52074 Aachen, Germany
{kalkov, gurghian, kowalewski}@embedded.rwth-aachen.de

ABSTRACT

Different approaches for improving process scheduling, memory management and generic message passing have been introduced to Android in the past, in order to extend it with support for real-time applications. One of Android's most important security features is sandboxing and strict isolation of running processes. Through its highly modular application framework, an RPC-based interprocess communication using the Binder driver plays a crucial role in the Android platform. This paper presents an extended Binder module for preserving the priority of the calling thread across process boundaries. Introduced modifications affect the low level Linux driver as well as its Java / C++ wrapper in the Android framework. Instead of the invocation of the call with the default priority, an additional pool of real-time threads is created on the remote side and automatically used to handle high priority requests. Empirical evaluation results provide experimental evidence of the effectiveness and scalability of the proposed approach and show bounded execution time in different test configurations. Furthermore, the improved Android platform remains fully backward compatible with existing components and third-party applications.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

Keywords

Android, Binder, Linux, Real-time, Remote Procedure Calls

1. INTRODUCTION

Android is one of the most successful platforms for mobile devices with growing market share[1]. By utilizing a Linux kernel optimized for hardware with limited resources, Android can run on a wide range of different devices. Starting with smartphones and tablet PCs, Android is also available for TVs, entertainment systems and smartwatches.

Its original implementation does not provide any support for applications with real-time requirements. But several approaches for extending [8, 11, 14, 15] or rebuilding [29] the platform to reach this goal have been presented in the past. They all rely on a real-time capable Linux kernel with improvements in memory management and generic messaging. Additionally, presented extensions elaborate more predictable methods for inter-process communication (IPC). Yan et al. enhanced the Android components Handler and Looper for a more deterministic handling of Binder messages on the platform level [29]. Intents, as high level communication objects, were presented in the work of Kalkov et al. [12]. Underneath the Intent architecture Android utilizes the Binder driver for the invocation of remote procedure calls (RPC) and transmission of the parameter data. This driver is loaded into the Linux kernel and provides its functionality to the Android core through the middleware. To the best of our knowledge, none of the low level Binder components were evaluated or modified during the previous research.

To improve the Binder framework in RTAndroid, the platform itself, but also the Linux kernel module and the corresponding middleware wrapper have to be modified. In its original implementation, the Binder selects a random regular thread on the remote side even when processing an RPC invoked by a real-time process. The scheduling class or the effective priority of the calling thread are not taken into account during the execution of a remote call. Additionally, large parts of the kernel module are protected by a mutex which ignores the priority of waiting processes, which may also introduce nondeterministic behavior.

The proposed implementation relies on a priority-aware matching between calling and callee threads. It also makes use of a mutex that grants access to the Binder module based on the priority of waiting threads. The introduced modifications to Android preserve the compatibility of the new platform with already existing software components. Specifically, the key contributions of this work are:

- Investigation of Binder-related components in Android for their suitability in real-time applications.
- Concept design and implementation of predictable remote procedure calls with bounded invocation delays.
- Detailed performance evaluation of the modified system components and discussion of possible improvements.

[1]http://www.gartner.com/newsroom/id/2996817

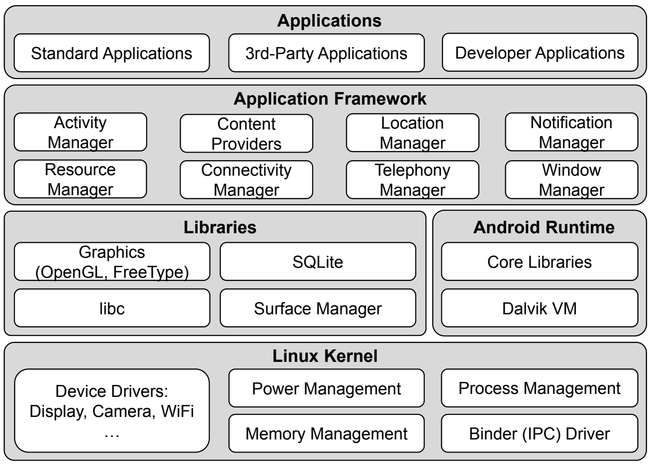

Figure 1: Android Architecture [27]

The remainder of this paper is organized as follows: A brief introduction to the Binder driver and the fundamentals of RTAndroid is given in Section 2. Section 3 presents the implemented approach for extending the Android platform and the Linux Binder component. Empirical evaluation of the resulting system is discussed in Section 4. Section 5 summarizes current research related to RPCs and interprocess communication in Android. This paper is concluded in Section 6 with final remarks.

2. BACKGROUND

This section begins with a brief introduction to Android's architecture and its main components. Following, relevant components of the RPC system and RTAndroid extensions are explained in detail.

2.1 Android

As shown in Figure 1, Android relies on the Linux kernel for core OS functionality like networking, power or process management. A set of C/C++ libraries including Android's own libc called Bionic, WebKit and OpenGL provides additional functionality to the upper layers. The Dalvik virtual machine is the main component of the Android runtime and it is optimized for execution of multiple VM instances on mobile devices with limited resources. The main system services (e.g. connectivity and window management) are contained in the Application Framework layer. They are exposed to user applications as a set of platform APIs. Applications are set on the top of this stack and include both system and user applications currently installed on the device. Android applications are written in the Java programming language, but converted to the DEX format after compilation to be executed inside DVM [16]. They may also contain native code which is executed via the Java Native Interface (JNI) [5].

2.2 Android Components

The four main types of components used in Android application development are Activities, Services, Broadcast Receivers and Content Providers. Activities form the application's user interface with control widgets like buttons and text fields. Services enable the application to perform long-term background operations without user interaction. Broadcast receivers are high-level endpoints for handling system-wide messages called Intents. Content providers allow data storage and data sharing between different applications (queryable SQL-like databases).

By combining these building blocks every Android application is implemented in a modular fashion with a loose component coupling. The platform itself is designed to allow simple interaction or data exchange between components of different applications. As different parts of the same application may work independently, the lifecycle model of Android enables them to be executed in separated user space processes. This introduces a need for safe and lightweight interprocess communication mechanisms.

A prominent example of process separation within the same application is a music player. If the entire app is executed in a single process, the playback will be stopped due to the Android application lifecycle model as soon as the user switches to other tasks. To prevent this, the user interface and the logic of playback control are separated and run in different processes. This allows the user to close the UI and continue working with other apps while the music playback is still running in the background.

The security of the system is achieved by sandboxing user applications and strict isolation of the running processes. Every application (component) is executed in a dedicated instance of the Dalvik VM, taking advantage of the Linux user-based protection. Whenever an application tries to interact with a component outside of its own process, it is forced to take the way through the application framework layer. This ensures a high level of sandboxing and prevents processes from gaining unauthorized access to private data.

The communication between processes in Android can be done in different ways. Since Android is based on Linux, it natively supports standard Linux inter-process communication mechanisms like pipes (unidirectional streams) or domain sockets (bidirectional streams). Although these communication mechanisms are available in the Linux kernel out of the box, the Android framework provides two additional IPC approaches:

- Anonymous shared memory (ASHMEM)
- Remote procedure calls using the Binder driver

ASHMEM is a shared memory allocator with a behavior different from POSIX SHM. It is designed to provide better support for low memory devices, as shared regions can be easily discarded if the system is running out of memory.

The Binder driver is another important component of the Android platform. It is involved in nearly every action that takes place in the system, like starting applications, handling user input events and generic message passing between applications. By forwarding method calls and corresponding argument data between different execution environments the Binder framework provides an invisible layer for making use of the platform functionality encapsulated by system services.

2.3 Android Remote Procedure Calls

Before presenting the internals of the driver itself, the following use case illustrates a simple example of how the Binder is involved in the interaction between separate applications. In our example, one application collects and preprocesses sensor data, which is then offered to the rest of the system. Using the mentioned Service component in Android, a public

```
package com.example.sensorservice;

interface ISensorService
{
    void getSensorData();
}
```

Listing 1: Example of an AIDL file

class `SensorService` with a single method `getSensorData()` shall be accessible from other applications. In order to expose methods to the system, the Android Interface Definition Language (AIDL) is used. Listing 1 shows an example of a simple AIDL file.

With the help of this interface description, Android is able to autogenerate internal structures that make it possible for a client to invoke the method `getSensorData()` across application boundaries, i.e. execute a remote procedure call. The client application first *binds*, i.e. establishes a connection to, the running sensor service. The Android system finds the requested service among all running processes and returns an object of the type `ISensorService`. The provided object is an autogenerated class called a *Proxy*, which exposes the same interface as the original class. The client can now call the method `getSensorData()` on this Proxy object, as if it was implemented in his local context. However, the actual call is not executed in the context of the client. Instead, the Proxy object serializes the function invocation data and transmits it using the Binder driver to the target process. On the Service side, an autogenerated *Stub*, the counterpart to Proxies, deserializes the information and invokes the respective method in the context of the Service process. The result is returned to the caller. For the client connected to the sensor service, the whole interaction with its loca Proxyl object is synchronous and indistinguishable from calling other methods in its own context. All internal actions and data processing is encapsulated by the low level driver. This construct offers a powerful way of exposing methods to other applications or components in separate processes.

2.4 Binder Driver

In Android, the Binder framework is the main component responsible for performing remote procedure calls. By mediating the interaction between separate processes, the Binder driver helps to reconcile sandboxing and interconnectivity between applications. The Binder was originally developed as the OpenBinder project by Palm Inc[2]. The Android team at Google has adopted this implementation and included its customized version into the Android platform[3].

The Binder framework in Android consists of three layers:

- API: High-level layer to be used on the application side. Is usually written in the Java programming language. It is used to provide convenient access to the middleware implementation, mainly through auto-generated Stub and Proxy classes.
- Middleware: Method calls from the Java API are passed via JNI to the Binder C++ middleware. This layer controls the Binder threads inside an application and takes care of marshalling and unmarshalling the passed arguments.
- The Binder driver is implemented in the C programming language. It is not part of the Android framework, as it is implemented as a Linux kernel module. This layer is accessible through `ioctl` system calls, which allow transmission and reception of commands from the driver.

[2]http://www.angryredplanet.com/˜hackbod/openbinder
[3]http://elinux.org/Android_Binder

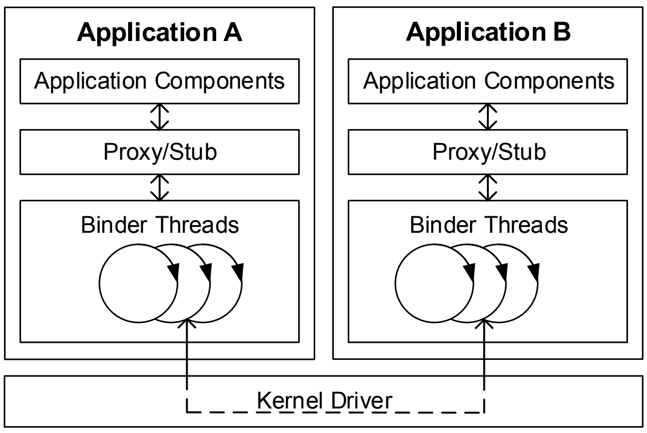

Figure 2: Remote Procedure Call Architecture

The default Binder modules used in an application are depicted in Figure 2. Whenever a new Android process is started, the platform automatically creates a new Binder node including internal management data structures and spawns multiple Binder threads in the context of this application. These threads, which will be referred to as *Looper* threads, usually remain unnoticed by the user and are dedicated to handle external commands like life cycle management or remote procedure calls. For executing a remote procedure call, the caller passes the serialized method invocation to the Binder driver, including the information required to identify the target node. The driver copies the serialized object to the callee, where it is deserialized and executed by one of the local Looper threads. The computed result is returned to the calling application in the same way through the Binder driver.

2.5 Real-Time Android

RTAndroid extends the original Android platform with support for basic real-time requirements. It utilizes the `RT_PREEMPT`[4] patch on the underlying Linux kernel in order to enhance its preemptibility and priority-based process scheduling. As every Android application is encapsulated by one or multiple Linux processes, real-time sensitive logic can be scheduled based on the priority value of the corresponding process, having a precedence over unimportant tasks.

The original implementation of the garbage collection in Android can introduce non-deterministic execution freezes due to the native mark-and-sweep GC algorithm [8]. RTAndroid provides a concurrent reference counting GC in order to avoid undesired process suspensions, meet soft real-time

[4]http://rt.wiki.kernel.org

requirements of running applications and improve predictability of the process behavior.

Furthermore, RTAndroid has a predictable message broadcasting system. The Intent handling is improved by using a priority-based delivery with bounded transmission delays, instead of the original first in – first out processing.

The introduced real-time extensions are encapsulated in the standard Android manner and can be accessed via the additional RTAndroid API[5]. Incorporating these features in user applications enables a reliable execution environment, which prevents the Android platform from suspending or terminating real-time applications. Maximal scheduling latencies of such processes were shown to be bounded by just a few milliseconds [11]. Recent optimizations of the RTAndroid framework show even further improvement, reducing the maximum latency values below 100 microseconds.

The RTAndroid platform does not change the original Android API. System modifications are introduced as additional functionality to ensure full compatibility to the official API and already existing software components [12]. Implementation is kept as modular as possible, such that it can be easily ported to new Android releases.

3. IMPLEMENTATION

This section presents our contribution to improve Android's Binder driver for a more predictable and deterministic remote procedure calls. The first part evaluates the current Binder implementation and shows possible problems if it is used in the context of real-time systems. After that, required modifications to both the Linux driver and its Android wrapper framework are presented in detail.

3.1 Design Considerations

All internal delays of a real-time system must be reliably predictable. Especially when it comes to critical sections [17] or communication between separate processes with different priorities. In Android this means that the maximum time to transfer a remote procedure call to the target process and return the result back to the calling application has to be bounded. The actual execution time of the invoked method depends on the specific implementation and is not part of the system. Therefore it can be excluded from the consideration. As our evaluation shows, the original implementation of Android does not meet this requirement. Elevating the priority of the calling thread and using a real-time scheduler does not improve the system behavior of RPCs either.

The default Binder implementation in the Linux kernel as well as its high-level wrapper behave nondeterministically for several reasons. For a better understanding, a closer examination of the way remote procedure calls are implemented in the Binder driver is presented in the following.

Steps executed by the system during an RPC in Android are depicted in Figure 4. When spawned by the middleware, a Binder thread calls into the kernel driver and registers itself on the low level for performing incoming method invocations (1). As long as no work has to be done, the thread sleeps and waits on a node-specific wait-queue (2). This means that the thread is not scheduled until another Binder thread executes a wake up operation on this wait-queue (8). A remote procedure is invoked from the client side by using a Proxy object (3). This object serializes the function call with

[5]http://rtandroid.embedded.rwth-aachen.de/

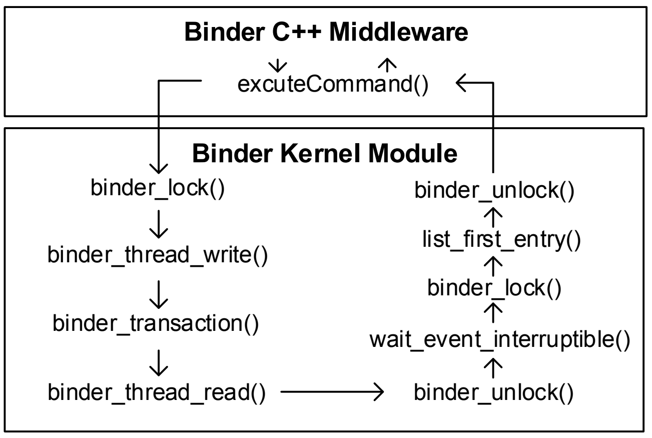

Figure 3: Lifetime of a Binder Looper Thread

corresponding arguments (4) and forwards it into the kernel driver by performing an `ioctl()` call. After determining the required target node (5), the serialized information is copied into a FIFO queue (6), which contains all transactions to be executed by this node. In the next step a wake up operation is performed on the wait-queue of the target node (7). This wakes up all waiting threads on the remote (service) side (8). While waiting for the response, the calling client thread puts itself into sleep (9). One of the services' own Binder threads will poll the transaction from the FIFO queue and return the control flow to the application level (10). All other threads will return to sleep if there is no further work to be done. With help of the Stub object, the Binder thread deserializes the original information (11), invokes the specified method (12), serializes the returned value (13) and returns to the kernel module (14). Here, the returned method result is copied back to the client node. As soon as the execution of the actual call is finished on the remote (service) side, the executing thread puts itself into sleep (17) until new transactions arrive. Remaining steps are performed on the client side. The original calling thread which has been put to sleep in (9) is woken up (16). It returns to the application level on the client side (18), deserializes the result of the remote call (19) and continues its execution (20).

It has to be mentioned that the client-side thread is a regular user-created thread and is not one of the Looper threads. In this example, Looper threads are used only on the service side for the sake of simplicity. Real applications utilize this communication concept in both directions, making RPCs from service to client possible.

Figure 3 shows the steps executed by a Looper thread in an endless loop. In order to avoid the corruption of internal Binder data structures, large parts of the kernel driver are protected by a mutex. When entering the kernel module, the thread first acquires this mutex by calling `binder_lock()`. This prevents two concurrent threads from entering the critical section at the same time. In the next step, the method `binder_thread_write()` is executed, which copies the Looper's transaction data to the target Binder node. After that, the Looper thread calls `binder_thread_read()`, which checks whether there are any transactions to be processed for the node this Looper belongs to. If there are no transactions available, the Looper

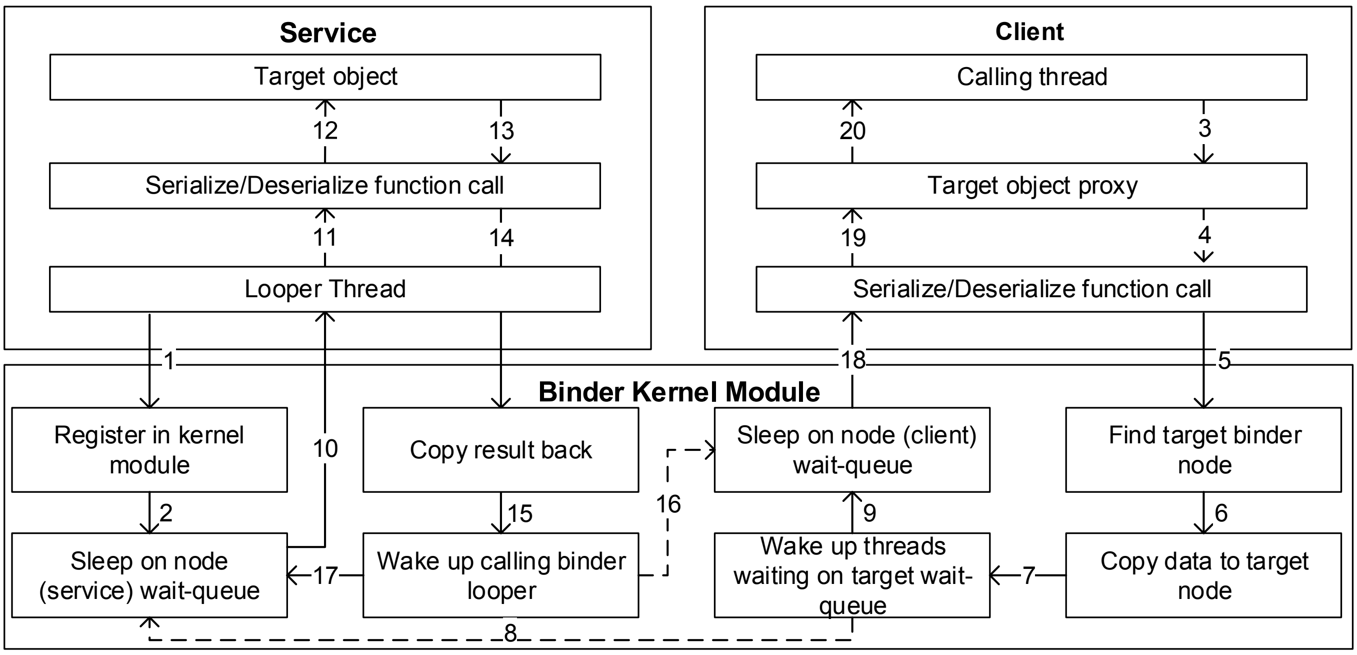
Service
Target object
12
13
Serialize/Deserialize function call
11
14
Looper Thread
Client
Calling thread
20
3
Target object proxy
19
4
Serialize/Deserialize function call
1
18
5
Binder Kernel Module
Register in kernel module
10
Copy result back
16
Sleep on node (client) wait-queue
Find target binder node
2
15
9
6
Sleep on node (service) wait-queue
17
Wake up calling binder looper
Wake up threads waiting on target wait-queue
7
Copy data to target node
8

Figure 4: Binder Execution Steps During a Remote Procedure Call

will release the lock and sleep on a wait queue by calling `wait_event_interruptible()`. As mentioned before, the Looper will be woken up by the system as soon as new transaction data arrives. After reacquiring the lock, the transaction data will be dequeued from the FIFO queue, the lock will be released again and the Looper returns to the C++ middleware, in order to call the method `executeCommand()`. This method consists of a large switch-case statement for executing commands that have been received from the Binder kernel module. Depending on the received command, the Looper may return to higher abstraction levels in order to prepare or perform remote procedure calls.

The kernel driver can also generate other commands, e.g. for creation of additional Looper threads. If a specific Binder node is used extensively, additional local thread can be created by returning a `BR_SPAWN_LOOPER` command to the middleware. After receiving this command, the Looper will spawn a new regular thread with the default priotity as a response to increased load on the Binder node.

The Binder framework in the Linux kernel uses various data structures for node management. The struct `binder_node` holds information about every process that interacts with the Binder, i.e. every Binder node. A part of this datatype is the struct `binder_proc`, which contains the wait-queue `wait` used by idle Binder Loopers to sleep on. The `binder_proc` structure is shared among all threads within one Binder node. The FIFO queue with pending jobs is also implemented as a member of the `binder_proc` struct.

Several problems arise from using the original implementation in context of real-time applications. First of all, the mutex used in the Binder internals does not consider the priority of the threads waiting to acquire the lock. If a mutex is released, the ownership will be given nondeterministically to one of the waiting threads. As Binder is one of the most important and heavily used Android components, e.g. almost every system event triggers a Binder transaction, such concurrency situations occur frequently. The driver will automatically spawn additional (regular) Binder threads in high workload situations, which increases the number of competitors for acquiring the lock. This observation leads to the conclusion that the described mechanism is not suited for real-time systems. The time a real-time thread would have to wait for the mutex is not reliably predictable and its average significantly increases if more regular threads compete for acquiring the lock. Access to such critical sections has to be granted based on the priority of the waiting threads.

Furthermore, RPCs which are invoked from a real-time thread on the client side are executed by a regular thread on the remote side by default. Any thread with a higher priority on the remote is able to preempt this execution and lead to priority inversion.

Another potential problem is identified in the way transactions are picked up by the Binder threads. If an addressed node is already under heavy load, i.e. it has to handle calls from multiple applications, all of its Binder threads will be continuously occupied. Therefore, there is no guarantee that the number of Binder threads will match the number of calling threads. In this case a single Binder Looper will handle transactions from multiple calling threads sequentially. Additionally, because the FIFO data structure is used, a privileged transaction will only be processed after all previous, potentially low-priority transactions have been handled.

To overcome these disadvantages, we propose several major system modifications:

1. If a Binder node is accessed by a real-time thread, it spawns an additional Looper thread with the same priority as the caller. This Looper thread is dedicated for handling privileged transactions only. It will only be spawned once and will be reused for future invocations. In order to keep the high-level Android framework un-

modified and hide complexity from the user, the driver framework will take care of spawning the additional real-time thread on demand. This ensures that every real-time thread on the client side will always have a corresponding privileged Binder thread on the remote side to immediately process the incoming transaction with the same priority.

2. Original mutexes and non-preemptible spinlocks are replaced with their real-time capable equivalents from `RT_PREEMPT` patch.

3. Every Binder node is equipped with one additional queue, which is dedicated to hold real-time transactions only. This will ensure that high-priority transactions do not have to wait for the low-priority transactions to be dequeued first.

All the measures presented in this section aim at establishing priority inheritance between threads involved in a remote procedure call. The priority value of the calling thread has to be inherited on the remote side in order to preserve the execution order. Concrete implementation details of these steps are described in the following sections.

3.2 Linux Binder Driver

The low level Binder driver is implemented as a Linux kernel module in the C programming language. As RTAndroid is based on a preemptible kernel, we make use of the `rt_mutex` API to replace the default mutex in the original Binder implementation. The `rt_mutex` has two main advantages in comparison to the traditional mutex. When the `rt_mutex` is released, its ownership will be deterministically transferred to the waiting thread with the highest priority. Furthermore, the priority inheritance mechanism implemented in the `rt_mutex` raises the priority of the thread currently holding the lock to the highest priority value out of all waiting threads. This will decrease the time a high priority thread has to wait for the lock to be released. As acquiring and releasing the lock is encapsulated in functions `binder_lock()` and `binder_unlock()`, exchanging the mutex type does not require extensive code changes.

For integration of dedicated high priority Looper threads we extend the `binder_proc` structure. Additional boolean flag `has_rt_looper` indicates whether the current node has already spawned a Binder thread for real-time transactions. This thread will inherit the priority of the caller in order to avoid priority inversion. We also introduce a new wait queue `rt_wait_queue` for sleeping and waking up the real-time Looper on the remote side. Only one instance of the `rt_wait_queue` is created per node, which is shared among all real-time Loopers on the remote side. This is required, as the originally implemented wake up operation (see Step 8 in Figure 4) does not provide a possibility to wake up only one specific thread. Finally, we insert an additional job queue `rt_todo` which holds real-time transactions to be processed by the node and that is shared among all Looper threads of a node. This job queue and the `has_rt_looper` flag do not distinguish between real-time threads with different priorities. The current implementation provides only a proof-of-concept and has to be extended in order to deal with complex setups.

The default steps performed in the original Binder kernel module, as depicted in Figure 4, are modified as follows:

1. The real-time thread invoking the remote transaction first checks if the target node has already spawned a dedicated real-time Looper to correspond with the current thread.

2. If no dedicated thread exists yet, the target node is instructed to spawn it by passing a `BR_SPAWN_RT_LOOPER` command to the target node middleware.

3. The priority of the remote real-time Looper is set to the priority of the calling thread.

4. The actual transaction data including provided arguments is stored in the `rt_todo` queue.

5. As soon as the dedicated Looper is available, it will automatically poll the data from the real-time queue and perform the remote procedure call.

The new Looper has the same lifecycle model as original Looper threads. It will be automatically put to sleep on the `rt_wait_queue` as soon as no real-time RPC transactions are available. All allocated resources are automatically released by the Binder framework upon the process termination.

3.3 Android Application Framework

Modifications to the Application Framework of Android are kept minimal and do not alter any existing method signatures. The method `executeCommand()` is extended such that it is able to process the new `BR_SPAWN_RT_LOOPER` command. On receiving this command from the Binder driver, an additional Looper thread will be spawned and added to the pool by calling the already existing method `joinThreadPool()`. After the priority value of the new thread is raised to the real-time value of the calling thread, the new thread registers itself in the Linux Binder driver by transmitting a `BC_REGISTER_RT_LOOPER` message to the Linux kernel driver. This way the new thread is available to the Binder and is ready to process privileged RPC transactions.

4. EVALUATION

The introduced system modifications are evaluated in multiple tests. This section begins with a presentation of the evaluation setup and used definitions. The second part shows the direct comparison between the new approach and the original system in different load scenarios. In the last part the scalability of the new implementation is evaluated and discussed.

For all of our performance tests we use the Google Nexus 10 tablet computer running the latest version of RTAndroid based on Android 4.2.2. This device has a Samsung Exynos 5250 SoC built in with a 1.7 GHz dual-core Cortex-A15 CPU and 2 GB of RAM.

4.1 Experimental Setup

Two separate applications A and B make use of Binder RPCs as depicted in Figure 5. Both of them are implemented as conventional Android applications using real-time extensions as described by Kalkov et al. [8, 11, 12]. Application B provides a public service with a single method `getTimestamp()`, which returns the current timestamp using the high resolution timer. After the Application A is started, it binds this service and starts k_{rt} real-time threads and k_{nrt} conventional threads, used to execute the remote

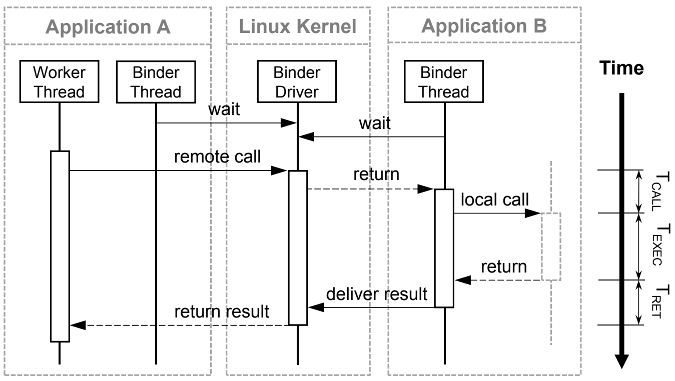

Figure 5: Measuring the Time to Invoke a Method Call in the Remote Process and Receive the Result

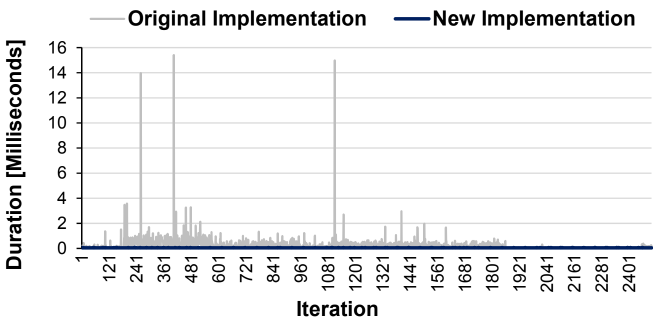

Figure 6: Comparison of T_{CALL} for $k_{nrt} = 20$

procedure call in the Application B. Each thread saves timestamps t_{pre} immediately before and t_{post} immediately after the invocation of `getTimestamp()`. Using the value t_{res} returned by the remote service, we calculate the invocation delay $T_{CALL} = t_{res} - t_{pre}$. The return delay is calculated in a similar way: $T_{RET} = t_{post} - t_{res}$. As the method `getTimestamp()` consists of one single return statement, we assume the execution time T_{EXEC} to be negligible, such that it is not considered during the test. Non real-time threads are only used to generate system load during the test. Although they also calculate T_{CALL} and T_{RET} during the execution, their values are neither recorded nor used in the following evaluation.

4.2 Performance Under Low Load

The first test is performed with a single real-time thread and $k_{nrt} = 20$ non real-time threads. Figure 6 shows invocation delays T_{CALL} in the original implementation and in the modified system for 2500 iterations.

The Binder framework obviously does not provide any privileged execution of RPCs invoked from a high priority thread. Although the measured delays stay below 2 ms most of the time, there are outliers up to 15 ms. In contrast to this, the predictability of the modified system is significantly improved. All measured invocation delays stay in the range between only $T_{CALL}^{MIN} = 52$ μs and $T_{CALL}^{MAX} = 75$ μs.

In order to improve the comparability and include the corresponding return delays, two separate diagrams will be used in the following. Figure 7 shows both Binder delays from the same experiment on the original system. Transmitting

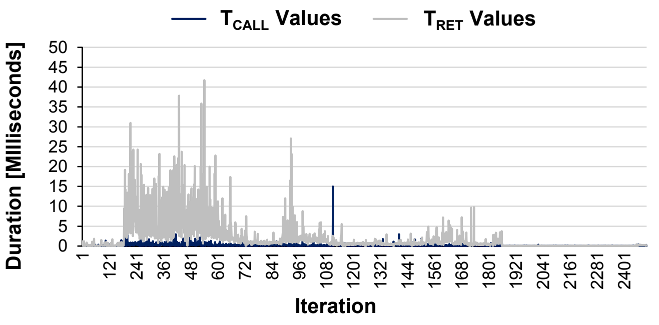

Figure 7: Binder Delays in the Original Android with $k_{nrt} = 20$ Threads

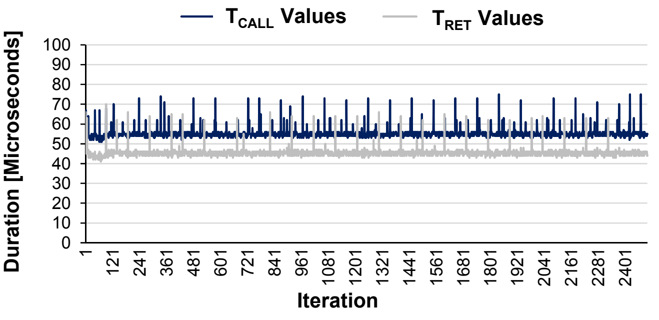

Figure 8: Binder Delays in the Modified Android with $k_{nrt} = 20$ Threads

the result to the caller seem to perform even worse, than the invocation of the method call. Return delays are distributed in a nondeterministic manner and reach up to 42 ms. The original implementation of the Binder driver is not capable of inheriting the priority of the calling thread on the remote side or ensuring privileged execution of RPCs from real-time applications. Incoming calls are blocked by low priority threads until the execution of previous invocations is finished.

This behavior is improved by the proposed approach as shown in Figure 8. For better readability, the plot units were changed from milliseconds to microseconds. Inheriting the priority of the calling thread results in a more deterministic system behavior. Real-time calls are executed by a privileged thread on the remote side, avoiding being blocked by other processes. Both, the invocation delay and the return delay, are bounded by about 80 μs without excessive outliers. The extended Binder driver is concluded to be able of serving real-time RPCs under low system load.

4.3 Performance Under High Load

The next test evaluates the performance of the new approach under high system load. For this purpose we reuse the setup described in Section 4.2, but start $k_{nrt} = 100$ regular threads trying to execute the same method in the remote process. Binder delays are only recorded for the single real-time thread.

Measurement results of the original Binder framework are presented in Figure 9. In comparison to the execution with only 20 regular threads as presented in Figure 7, increased values for both the invocation delay and return delay can be observed. While T_{CALL} is fluctuating around 2 ms with outliers up to 42 ms, T_{RET} stays high most of the time. This

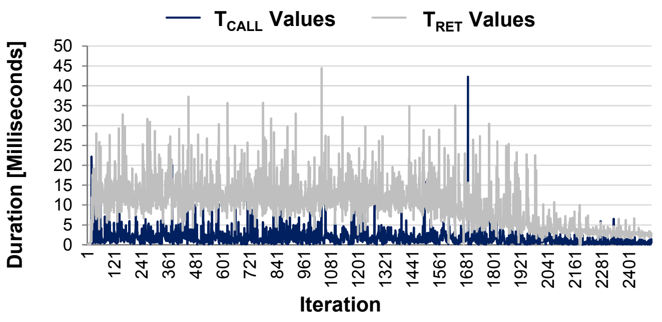

Figure 9: Binder Delays in the Original Android with $k_{nrt} = 100$ Threads

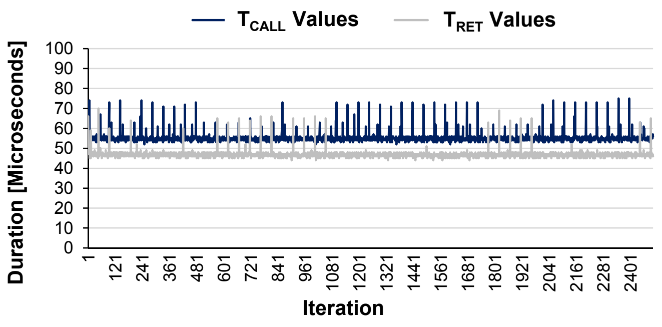

Figure 10: Binder Delays in the Modified Android with $k_{nrt} = 100$ Threads

confirms our previous observation that the unmodified Binder driver is not able to provide a bounded RPC execution time.

In contrast to the original implementation, recorded values for the modified system do not considerably change under load, as depicted in Figure 10. The maximum invocation delay under high system load $T_{CALL}^{MAX} = 75$ μs remains the same as under low load. Values $T_{RET}^{MIN} = 43$ μs and $T_{RET}^{MAX} = 71$ μs for the return delays are bounded and more deterministic than in the original system. The increased number of regular threads seem to have no significant negative effects on RPCs when running using the proposed approach. The only remarkable difference between Figure 8 and Figure 10 is the number of values above 70 μs. While there are approximately 30 such outliers for $k_{nrt} = 20$, executing $k_{nrt} = 100$ regular threads increases this value to over 45. This was expected, as introducing more threads leads to more RPCs per time period and higher chance for the real-time thread to be blocked. As the maximum invocation delay does not change in this test and the frequency increase is low, it is not considered problematic.

To verify the scalability of the new method, this test was additionally repeated for the intermediate number of $k_{nrt} \in \{0, 40, 60, 80\}$ non real-time threads. Figure 11 summarizes all resulting values for the original implementation and the new implementation, as well as the results of the previous runs. For better visualization of small values, the scaling of the time axis is logarithmic. The biggest increase of the maximum value can be observed in the unmodified system, when executing the first 20 non real-time threads. In this case the invocation delay increases from approximately 3.5 ms to 31.4 ms. The return delay is also subject of a significant

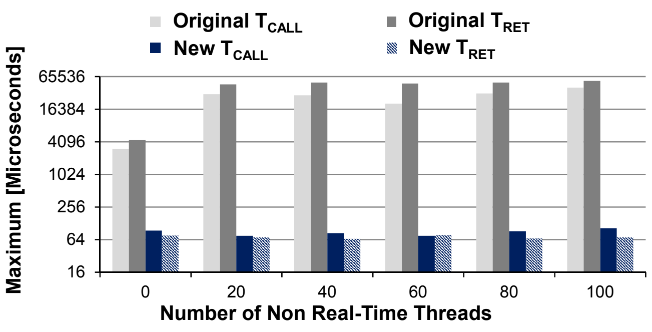

Figure 11: Maximum Values for T_{CALL} and T_{RET} for Rising Number of Regular Threads

increase from 4.3 to 47.1 ms. Interestingly, the impact of adding additional threads does not rise much, when the number of threads is set to $k_{nrt} = 100$. The value of T_{CALL}^{MAX} increased to 40.8 ms and T_{RET}^{MAX} increased to 54.2 ms.

In contrast to this, the worst case invocation delay on the new implementation increases from 94 μs for $k_{nrt} = 0$ to only 103 μs for $k_{nrt} = 100$. The maximum value of the return delay for $k_{nrt} = 100$ with 72 μs is even lower, than it was for $k_{nrt} = 0$ regular threads (76 μs). Inheriting the real-time priority of the caller on the remote side prevents its execution from being blocked by regular threads and assures predictable process behavior. The extended Binder framework is shown to be able to provide a bounded invocation time for RPCs even under heavy system load.

4.4 Multiple Real-Time Threads

This experiment evaluates the scalability of the new implementation in a setup with multiple real-time threads ($k_{rt} \in \{2, ..., 7\}$, same priority) periodically executing the same RPC. In every test run, the system load is additionally kept high by executing $k_{nrt} = 100$ regular threads as described in the previous test. For the sake of simplicity only T_{CALL} values of all real-time threads are considered in the evaluation. This means that the presented statistics are created based on the total number of $2500 \times k_{rt}$ values in each step. Figure 12 summarizes the new results and measurements from the last test (see Figure 11).

The average values show a clearly linear trend with an almost straight line between 55 μs for $k_{rt} = 1$ and 319 μs for $k_{rt} = 7$. Every additional real-time thread increases the average duration of T_{CALL} by about 44 μs. As known from the previous tests, this is approximately the overhead of executing the method with a single thread. This shows that the usage of the `rt_mutex` assures sequential processing of the wait queue while preserving the correct execution order.

The maximum value stays within a constant distance of about 30 – 40 μs from the corresponding average during all tests. This indicates that the execution time is bounded and predictable to be used in a real-time system. The variance shows a slow growth and reach 56 μs only for $k_{nrt} = 7$ competing real-time threads.

5. RELATED WORK

A number of different applications for Android-based devices have been introduced in the last years. Smartphones and tablets are used in the medical field and health-care sec-

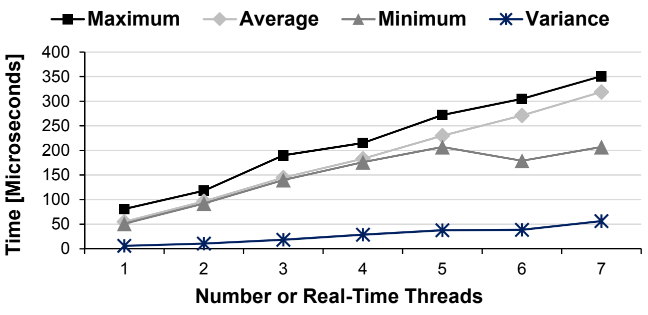

Figure 12: Invocation Delays T_{CALL} for Rising Number of Real-Time Threads

tor for real-time patient monitoring [2] as well as for analysis and visualization of the patient's data [9].

There are examples of mobile devices being utilized in automotive applications, e.g. in infotainment and navigation systems [3]. The suitability of Android was also evaluated in more advanced automotive setups like controlling [13, 25], energy management [4, 6] and on-board diagnostics [26].

Simple visualization and user interaction tasks usually do not require strictly deterministic behavior of the operating system, but missing deadlines in an industrial environment may result in serious harm to humans or machinery. Modern mobile devices are deployed in cyber-physical systems [24], wheelchairs [22], robotics platforms [21] and satellites[6]. Researchers also identify the need for a real-time capable mobile platform for automation purposes [19].

Studies have shown that the original Android distribution is not capable of performing critical tasks in a deterministic manner [11, 14, 18, 23, 29]. But the growing interest into open source mobile platforms facilitates new research to extend Android with real-time support. Kalkov et al. presented RTAndroid [11] as the first framework for Android applications with basic real-time requirements. RTAndroid utilizes a modified `RT_PREEMPT` patch, implements non-blocking garbage collection [8] and introduces new APIs to enable reliable program execution of Android applications. The effort for applying this patch to Android's Linux kernel was estimated by Mauerer et al. [15], where the authors also perform a performance evaluation of the resulting system. The suitability of the original Dalvik VM in a real-time setup was analyzed by Oh et al. [20].

A different approach of real-time integration in Android by building upon an off-the-shelf real-time kernel and the Fiji VM was presented by Yan et al. [29]. In the subsequent research the authors examine Android's sensor architecture and re-design it in order to improve the predictability of the sensor data management [28]. The authors do not evaluate approaches for real-time inter-process communication, but address the delivering of sensor information to Android applications. The Binder framework, which is implicitly involved in the presented approach is not evaluated, although it can dramatically affect the behavior of message delivery.

RTAndroid also presents an approach to improve intra- and inter-process communication based on Intent broadcasting [12]. The improved components are responsible for the high-level processing of Intent objects inside of Android's application framework layer. The Binder IPC, which is actually used by the broadcasting mechanism for message delivery across process boundaries is not evaluated or modified.

[6]http://www.phonesat.org

Intent broadcasting, as only one aspect of the Binder-based IPC, was already covered by a number of different publications in the past as summarized in our previous work [12]. But the analysis of its low-level functionality (e.g. information flow and IPC transactions) was limited to the identification and improvement of security issues. RiskMon [10] implements a reference monitor and tracks IPC transactions for risk assessment by adding new hooks in the Binder device driver. Another application for real-time privacy monitoring using a customized Binder is TaintDroid [7]. Backes et al. introduced Scippa [1], a Binder extension for preventing security attacks.

To the best of our knowledge, Binder's internal thread management and remote procedure calls have not been evaluated in the context of real-time applications yet.

6. CONCLUSION

This paper presents a detailed analysis of the Binder RPC framework, evaluation of its suitability in the context of real-time applications and an approach to improve RPC-based interprocess communication in Android. Section 3 investigates the Binder-related components of Android and introduces an extension of the original implementation. These modifications are designed for preserving the priority of the calling thread across process boundaries. Instead of the execution of the RPC by a regular thread on the remote side, a dedicated real-time thread is created and automatically used to handle high priority requests. Additionally, the mechanism protecting critical sections has been modified to grant access based on the priority of the waiting processes.

The resulting RPC platform is evaluated in multiple tests in Section 4. Empirical evaluation of the original system highlights its unsuitability for real-time applications. Even without additional load, standard implementation of the Binder driver introduces delays up to 4 ms for calling a remote method or returning the result value. The new approach is shown to provide bounded execution time in different test configurations. This makes the usage of RPCs in real-time applications possible even under significant system load.

As the Binder driver mediates interactions between Android system components and user applications, the impact of the introduced modifications has to be further evaluated. The processing mechanism for Intents, Android's high-level message passing construct, is a notable Android feature that may benefit from the new Binder modifications.

In the future, the feasibility of reducing the critical section that guards the Binder has to be analyzed. Furthermore, a pool of dedicated real-time threads with different priorities may show better performance than the current approach.

7. REFERENCES

[1] M. Backes, S. Bugiel, and S. Gerling. Scippa: System-Centric IPC Provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14, 2014.

[2] Bin Xiao, M. Z. Asghar, T. Jamsa, and P. Pulii. Canderoid: A mobile system to remotely monitor travelling status of the elderly with dementia. In *Proceedings of the International Joint Conference on Awareness Science and Technology and Ubi-Media Computing*, iCAST-UMEDIA '13, pages 648–654, 2013.

[3] R. Bose, J. Brakensiek, Keun-Young Park, and J. Lester. Morphing Smartphones into Automotive Application Platforms. *Computer*, 44(5):53–61, 2011.

[4] A. Cela, A. Hrazdira, A. Reama, R. Hamouche, S. I. Niculescu, H. Mounier, R. Natowicz, and R. Kocik. Real time energy management algorithm for hybrid electric vehicle. In *Proceedings of the 14th International IEEE Conference on Intelligent Transportation Systems*, ITSC '14, pages 335–340, 2011.

[5] O. Cinar. *Pro Android C++ with the NDK*. Apress, 1st edition, 2012.

[6] A. Dardanelli, M. Tanelli, B. Picasso, S. M. Savaresi, O. Di Tanna, and M. D. Santucci. A smartphone-in-the-loop active state-of-charge manager for electric vehicles. *IEEE/ASME Transactions on Mechatronics*, 17(3):454–463, 2012.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[8] T. Gerlitz, I. Kalkov, J. Schommer, D. Franke, and S. Kowalewski. Non-Blocking Garbage Collection for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 108–117, 2013.

[9] S. Gradl, P. Kugler, C. Lohmuller, and B. Eskofier. Real-time ECG monitoring and arrhythmia detection using Android-based mobile devices. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC '12, pages 2452–2455, 2012.

[10] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. RiskMon: Continuous and Automated Risk Assessment of Mobile Applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 99–110, 2014.

[11] I. Kalkov, D. Franke, J. F. Schommer, and S. Kowalewski. A Real-time Extension to the Android Platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 105–114, 2012.

[12] I. Kalkov, A. Gurghian, and S. Kowalewski. Predictable Broadcasting of Parallel Intents in Real-Time Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 57–66, 2014.

[13] K. Kołek. Application of Android OS as Real-Time Control Platform. In *Automatyka / Automatics*, volume 17, pages 197–206, 2013.

[14] C. Maia, L. Nogueira, and L. M. Pinho. Evaluating Android OS for Embedded Real-Time Systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '10, pages 62–70, 2010.

[15] W. Mauerer, G. Hillier, J. Sawallisch, S. Hönick, and S. Oberthür. Real-Time Android: Deterministic Ease of Use. In *Proceedings of Embedded Linux Conference Europe*, ELCE '12, 2012.

[16] R. Meier. *Professional Android 4 Application Development*. John Wiley & Sons, 2012.

[17] C. W. Mercer and H. Tokuda. Preemptibility in Real-Time Operating Systems. In *Real-Time Systems Symposuim*, pages 78–87, 1992.

[18] B. S. Mongia and V. K. Madisetti. Reliable Real-Time Applications on Android OS, 2010.

[19] M. Obster, I. Kalkov, and S. Kowalewski. Development and Execution of PLC Programs on Real-Time Capable Mobile Devices. In *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA '14, 2014.

[20] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of Android Dalvik Virtual Machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 115–124, New York, NY, USA, 2012. ACM.

[21] N. Oros and J. L. Krichmar. Neuromodulation, attention and localization using a novel Android$^{\text{TM}}$ robotic platform. In *ICDL-EPIROB*, pages 1–6. IEEE, 2012.

[22] S. Park, T. T. T. Ha, J. Y. Shivajirao, M. Hahn, J. Park, and J. Kim. Smart Wheelchair Control System Using Cloud-Based Mobile Device. In *Proceedings of the International Conference on IT Convergence and Security*, ICITCS '13, pages 1–3, 2013.

[23] L. Perneel, H. Fayyad-Kazan, and M. Timmerman. Can Android be Used for Real-Time Purposes? In *Proceedings of the International Conference on Computer Systems and Industrial Informatics*, ICCSII '12, pages 1–6, 2012.

[24] R. Podorozhny. Design and Verification of Cellphone-based Cyber-Physical Systems: A Position Paper. In *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 603–607, 2013.

[25] P. D. Urbina Coronado, V.-B. Sundaresan, and H. Ahuett-Garza. Design of an Android Based Input Device for Electric Vehicles. In *Proceedings of the International Conference on Connected Vehicles and Expo*, ICCVE '13, pages 736–740, 2013.

[26] J. Wideberg, P. Luque, and D. Mantaras. A smartphone application to extract safety and environmental related information from the OBD-II interface of a car. *International Journal of Vehicle Systems Modelling and Testing*, 7(1):1–11, 2012.

[27] K. Yaghmour. *Embedded Android*. O'Reilly Media, 2013.

[28] Y. Yan, S. Cosgrove, E. Blantont, S. Y. Ko, and L. Ziarek. Real-Time Sensing on Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 67–75, 2014.

[29] Y. Yan, S. H. Konduri, A. Kulkarni, V. Anand, S. Y. Ko, and L. Ziarek. RTDroid: A Design for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 98–107, 2013.