Why work on *Binder* and *KDBus*?

**High level thoughts:**

- *Could we have the same code running on distros and Android™?*
- *Can Android gain from KDBus?*
- *Are we duplicating work?*

**But also:**

- *Binder* is used everywhere in Android.
- *KDBus* can potentially become widely used.
- We can learn a lot.
What we want to achieve.

Investigate *KDBus* as a replacement for *Binder*:
- Understand if it can be done.
- Build a proof-of-concept.
- Identify potential blockers and difficult problems.

Things we haven’t looked at:
- Any sort of measurement / profiling.
- Comparing security mechanisms.
Our work in a nutshell

*libbinder* provides the API to the rest of the system.
Our work in a nutshell

Let’s make a drop-in replacement which talks to KDBus!
Features covered here

- Abstraction around services.
- Services discovery.
- Remote procedure calls.
- Thread pool management.
- Marshalling.
Agenda

- *Binder* API: Remote interfaces and objects
- Notes about *Binder* internals
- Overview of *KDBus*
- Implementing *Binder’s* API with *KDBus*
- Current state and future work
Binder API: Remote interfaces and objects
Binder is heavily object oriented

- A service is defined by an interface.
- We use a service with an instance object.
- We issue transactions by calling methods.
- Service instances can be passed around.
- A service has a lifetime.

We refer to these special objects as Binders.
Binder: Remote interfaces in C++

A system service provides an interface:

```cpp
class IAdder : IInterface {
    enum Code {
        ADD;
    }

    virtual int add(int a, int b) = 0;
};
```
Binder: Remote calls

No need to know where the transaction will be handled, remotely or locally:

```cpp
sp<IBinder> proxy = ...
sp<IAdder> service = interface_cast<IAdder>(proxy);

int answer = service->add(20, 22);
```
Binder: Servers and clients

Or Proxies and Stubs.

Remote proxy:

```c++
class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        // Issue a blocking transaction.
        return result;
    }
};
```

Native stub:

```c++
class BnAdder : BnInterface<IAdder> {
    int add(int a, int b) override {
        return a + b;
    }
};
```
Binder object abstraction: Remote proxy

class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        Parcel data;
        Parcel reply;

        data.writeInt(a);
        data.writeInt(b);

        remote()->transact(ADD, data, &reply);

        return reply.readInt();
    }
};
Binder object abstraction: Remote proxy

- Package the data.

```cpp
class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        Parcel data;
        Parcel reply;

        data.writeInt(a);
        data.writeInt(b);

        remote()->transact(ADD, data, &reply);

        return reply.readInt();
    }
};
```
Binder object abstraction: Remote proxy

- Package the data.
- Send it with code ADD.

```cpp
class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        Parcel data;
        Parcel reply;

        data.writeInt(a);
        data.writeInt(b);

        remote()->transact(ADD, data, &reply);

        return reply.readInt();
    }
};
```
Binder object abstraction: Native stub

- Implements a callback on request.

```cpp
class BnAdder : BnInterface<IAdder> {
    int add(int a, int b) override {
        return a + b;
    }

    status_t onTransact(uint32_t code, const Parcel& data, Parcel *reply) override {
        switch (code) {
            case ADD: {
                int a = data.readInt();
                int b = data.readInt();
                int result = add(a, b);
                reply->writeInt(result);
                return NO_ERROR;
            }
        }
        return NO_ERROR;
    }
};
```
Binder object abstraction: Native stub

- Implements a callback on request.
- Interpret the transaction code.

```cpp
class BnAdder : BnInterface<IAadder> {
    int add(int a, int b) override {
        return a + b;
    }

    status_t onTransact(uint32_t code,
                          const Parcel& data,
                          Parcel *reply) override {
        switch (code) {
            case ADD: {
                int a = data.readInt();
                int b = data.readInt();
                int result = add(a, b);
                reply->writeInt(result);
                return NO_ERROR;
            }
            
        }
    }
}
```
Binder object abstraction: Native stub

- Implements a callback on request.
- Interpret the transaction code.
- Unpackage incoming data.

```cpp
class BnAdder : BnInterface<IAdder> {
    int add(int a, int b) override {
        return a + b;
    }

    status_t onTransact(uint32_t code, const Parcel& data, Parcel *reply) override {
        switch (code) {
            case ADD: {
                int a = data.readInt();
                int b = data.readInt();
                int result = add(a, b);
                reply->writeInt(result);
                return NO_ERROR;
            }
        }
    }
};
```
Binder object abstraction: Native stub

- Implements a callback on request.
- Interpret the transaction code.
- Unpackage incoming data.
- Native call.

```cpp
class BnAdder : BnInterface<IAdder> {
    int add(int a, int b) override {
        return a + b;
    }

    status_t onTransact(uint32_t code, const Parcel& data, Parcel *reply) override {
        switch (code) {
            case ADD: {
                int a = data.readInt();
                int b = data.readInt();
                int result = add(a, b);
                reply->writeInt(result);
                return NO_ERROR;
            }
        }
    }
};
```
Binder object abstraction: Native stub

- Implements a callback on request.
- Interpret the transaction code.
- Unpackage incoming data.
- Native call.
- Package the reply.

```cpp
class BnAdder : BnInterface<IAadder> {
    int add(int a, int b) override {
        return a + b;
    }
    status_t onTransact(uint32_t code,
        const Parcel& data,
        Parcel *reply) override {
        switch (code) {
            case ADD: {
                int a = data.readInt();
                int b = data.readInt();
                int result = add(a, b);
                reply->writeInt(result);
                return NO_ERROR;
            }
            }
        }
    }
}
```
Binder object abstraction: Remote proxy

- Package the data.
- Send it with code ADD.

```cpp
class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        Parcel data;
        Parcel reply;

        data.writeInt(a);
        data.writeInt(b);

        remote()->transact(ADD, data, &reply);

        return reply.readInt();
    }
};
```
Binder object abstraction: Remote proxy

- Package the data.
- Send it with code ADD.
- Unpackage the reply.

```cpp
class BpAdder : BpInterface<IAdder> {
    int add(int a, int b) override {
        Parcel data;
        Parcel reply;

        data.writeInt(a);
        data.writeInt(b);

        remote()->transact(ADD, data, &reply);

        return reply.readInt();
    }
};
```
**Binder: Searching and registering services**

We have a *special* and unique *Binder* object for this: *ServiceManager*.

- Accessing this special object:
  
  ```java
  sp<IServiceManager> service_manager = defaultServiceManager();
  ```

- Registering our new service with it:
  
  ```java
  sp<IBinder> adder = new BnAdder();
  service_manager->addService("org.compute.adder", adder);
  ```

- Finding the service:
  
  ```java
  sp<IBinder> adder = service_manager->getService("org.compute.adder");
  ```
Outline

This was the *Binder* API in a nutshell.

- Object abstraction.
- Transaction codes.
- Marshalling different kinds of data.
- A special process keeps track of services.

*This API is used by services only. We could change it!*
Notes about *Binder* internals
Binder kernel driver

Kernel driver export a device node: `/dev/binder` and implements a two-way protocol:

\[ BC_* \leftrightarrow BR_* \]

- Maintain per-process memory pools.
- Manages worker threads.
- Dispatch data from one process to another.
Binder kernel driver: Managing a thread pool

Worker threads are managed by the kernel.
Binder kernel driver overview: Object lifetime

The kernel keeps track of who uses a service with reference counting.

- BC_ACQUIRE / BC_RELEASE: Acquire and release a service.
- BC_REQUEST_DEATH_NOTIFICATION / BC_CLEAR_DEATH_NOTIFICATION / BR_DEAD_BINDER: Manage the death of services.
ServiceManager

A special user-space process keeps track of services.

- All clients register themselves with it.
- There can only be one.
- The kernel driver implements a `BINDER_SET_CONTEXT_MGR` ioctl to identify this special service.
Outline: *Binder* internals

This is all abstracted in *libbinder*:

- *Binder* object abstraction.
- Marshalling: Packaging data into *Parcels*.
- Per process thread pool for handling incoming transactions.
- Accessing *ServiceManager*.

*We can see the kernel driver and Binder’s API are tightly coupled.*
Overview of KDBus
KDBus’s kernel interface

$ mount -t kdbusfs kdbusfs /sys/fs/kdbus
$ tree /sys/fs/kdbus
    /sys/fs/kdbus/                      ; mount-point
    |-- 0-system                       ; bus directory
    |   |-- bus                        ; default endpoint
    |   `-- 1017-custom                 ; custom endpoint
    |-- 1000-user                      ; bus directory
    |   |-- bus                        ; default endpoint
    |   `-- 1000-service-A              ; custom endpoint
    |   `-- 1000-service-B              ; custom endpoint
    `-- control                       ; control file
We built a small abstraction library around this and will use it in this talk.
Hello *KDBus*

- Creating a bus:
  We hold a file descriptor open for the lifetime of the bus.

  ```cpp
  // Running as PID 42:
  auto bus = Bus::make("myname");
  assert(bus->name == "42-myname")
  ```

- Connecting to a bus:
  Each connection to the bus gets assigned a unique 64 bit ID.

  ```cpp
  auto c = Connection::hello("42-myname");
  ```

  We can also give it a unique name in the bus’s name registry.

  ```cpp
  c->acquire_name("foo.bar");
  ```
Finding other Connections

Connections can probe the bus:

```cpp
enum ListFlags {
    Unique, // Get all Connection IDs.
    Names,  // Get all Connection IDs with a name.
    Queued, // Get all Connection IDs waiting for a name.
};

auto c = Connection::hello("42-mynname");
for (const auto& name : c->list(Names)) {
    // (...)
}
```
Everything sent to/from *KDBus* is an *Item*:

- Plain old data: copied, shared with *memfd* or file descriptor.
  ```cpp
define payload = ItemPayloadVec(&some_data, sizeof(some_data));
```
  - Identifiers: Name of a bus, a connection, ...etc.
  ```cpp
define name = ItemName("org.compute.name");
```
  - Misc information: Timestamps, credentials, capabilities ...etc.
  - Notifications from the kernel: Dead *Connection*, new *Connection*, timeout ...etc.
Messages

- They have a destination and a source.
- Messages are asynchronous by default but...
- They can expect a reply, identified with a *cookie*.
- *Messages* contain a chain of *Items*.

```c
MessageSync message(42,       // Source ID.
    12,         // Destination ID.
    123456789,  // Unique cookie.
    1000,       // One second timeout.
    ItemPayloadVec(&some_data, sizeof(some_data)),
    ItemPayloadVec(&more_data, sizeof(more_data)));
```
Subscribing to notifications

*KDBus* gives us Items describing rules that can be bundled together to form a *match*.

For example, if we apply the following rules:

- `KDBUS_ITEM_NAME_ADD`
- `KDBUS_ITEM_NAME_REMOVE`

A given connection will receive messages every time a connection acquires or releases a well-known name.
Outline

- **KDBus** gives us a transport layer.
- Provides synchronisation guarantees.
- Notification and monitoring.
- Name registry.

*We have all we need to implement transactions!*

*... KDBus will not manage threads for us.*
Implementing *libbinder’s* API with *KDBus*
Introducing *libkdbinder*
Registering a service with *KDBus*

*Binder* relies on the *ServiceManager*, we don’t!

```cpp
sp<IServiceManager> service_manager = defaultServiceManager();
```

- Create a per process object implementing the *ServiceManager* API.

```cpp
sp<IBinder> adder = service_manager->getService("org.compute.adder");
```

- Send a list command to *KDBus* and find the *Connection* with this name.
- Create a *Binder* object around this *Connection*’s ID.
Registering a service with \textit{KDBus}

\begin{verbatim}
sp<IBinder> adder = new BnAdder();

service_manager->addService("org.compute.adder", adder);
\end{verbatim}

\begin{itemize}
  \item Create a \textit{Connection} to \textit{KDBus} with a well-known name.
  \item Register it in a local per process table:
\end{itemize}

\begin{tabular}{|c|c|}
  \hline
  Connection to \textit{KDBus} & Binder object \\
  \hline
  "org.compute.adder" & sp<IBinder> adder \\
  \hline
\end{tabular}
Handling requests: Per process thread pool

Reminder: The *Binder* driver manages threads for us.
Handling requests: Per process thread pool

*KDBus* does not, let’s simply spawn X threads.
Handling requests: Per process thread pool

And let them handle transactions concurrently.
Final step: Issuing a *Binder* transaction with *KDBus*

- We have a *Parcel* and a transaction code as input.

- Create a *KDBus* synchronous *Message*:

<table>
<thead>
<tr>
<th>MessageSync</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
</tr>
<tr>
<td>Destination</td>
</tr>
<tr>
<td>Cookie</td>
</tr>
<tr>
<td>Timeout</td>
</tr>
<tr>
<td>Item Code</td>
</tr>
<tr>
<td>Item Parcel</td>
</tr>
</tbody>
</table>

- We get a *Message* back:

<table>
<thead>
<tr>
<th>MessageReply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
</tr>
<tr>
<td>Destination</td>
</tr>
<tr>
<td>Cookie</td>
</tr>
<tr>
<td>Item Parcel</td>
</tr>
</tbody>
</table>

- Unpack the *Item* in a *Parcel*
Current state and future work
Covering a subset a *Binder* with tests

We have a working proof-of-concept for isolated test cases.

- `BinderAddInts` benchmark functional.
- `binderLibTests` test cases pass with *KDBus*.

*It's too early for optimisations and profiling.*
Future work: memfd as a replacement for ashmem?

*KDBus* does not recognize *ashmem* file descriptors.

- Is replacing *ashmem* with *memfd* possible in Android™?
- *KDBus* forces us to pass sealed *memfd* descriptors, is it OK?
- Should *KDBus* support *ashmem* or should *Binder* support *memfd*.

*We need to pass big amount of data (frames). This is a potential blocker.*
Future work: Boot2anim?

The next milestone will be displaying the Android™ logo with KDBus.

- Enough of the API is implemented to build SurfaceFlinger!
- ashmem is a blocker.
- Other issues will likely be uncovered.
Future work: find better ways!

Our current implementation is purposely simple.

- Using more than one bus?
- Should services be connections or endpoints?
- We need to look at security as soon as possible.
- ... etc.
Conclusion: Can it work?

Short answer is: Yes of course!

Long answer:

- Feature parity is feasible.
- It will involve implementing *Binder* specific features in user-space.
- Be at least as efficient as *Binder*.
  → It needs more work and investigation.
Thank You

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.
**KDBus: Asynchronous example**

// Create two clients on the bus
auto c1 = Connection::hello("42-myname", "a-client");
auto c2 = Connection::hello("42-myname", "another-client");

// Data to send across the bus.
uint8_t data = 42;

// Create a message from c1 to c2.
Message msg(c2->id, c1->id, 0, 0, ItemPayloadVec(&data, sizeof(uint8_t)));

// Queue the message on c2's memory pool.
c1->queue_message(msg);

// Block until a message is on c2's pool.
auto reply = c2->dequeue_message_blocking();
**KDBus: Synchronous example**

```cpp
auto c1 = Connection::hello("42-myname", "a-client");
auto c2 = Connection::hello("42-myname", "another-client");
uint8_t data_in = 42;
// We pass this value to identify the transaction.
uint64_t cookie = 123456789;

// Create a server thread. Gets a message and replies 1.
std::thread server([&c1, &c2] {
    uint8_t data_out = 1;
    auto reply = c2->dequeue_message_blocking();
    MessageReply message(c1->id, c2->id, reply.cookie, ItemPayloadVec(&data_out, sizeof(uint8_t)));
    c2->reply(message);
});

MessageSync message(c2->id, c1->id, cookie, 1000, ItemPayloadVec(&data_in, sizeof(uint8_t)));

auto reply = c1->transact(message);
```
Handling requests: Worker thread execution

We have done this the simplest we could think of:

1:   for all \textbf{entry} in the service table do
2:       \textbf{entry} $\leftarrow$ Copy \textbf{entry}, protected by a per process mutex.
3:       \textbf{message} $\leftarrow$ Dequeue a message from the connection’s memory pool.
4:       if not time out then
5:           \textbf{cookie} $\leftarrow$ Read cookie value from \textbf{message}.
6:           \textbf{code} $\leftarrow$ Read transaction code from \textbf{message}.
7:           \textbf{Parcel in} $\leftarrow$ Read data from \textbf{message}.
8:           \textbf{Parcel out} $\leftarrow$ Call the \textit{Binder} object with \textbf{code} and \textbf{in}.
9:       \textbf{message} $\leftarrow$ Write \textit{out} into a \textit{KDBus} message.
10:      Send the \textit{msg} reply with the same \textbf{cookie}.
11:      end if
12:     end for
Sending a request: *Parcel in / Parcel out*

We have a remote *Binder* object → we know the *KDBus* connection ID.

1. `connection ← Create a new temporary connection to the bus.``
2. `cookie ← Create a unique transaction cookie.``
3. `item_code ← Bundle the transaction code into an item.``
4. `item_data ← Bundle the *in Parcel* into an item.``
5. `message ← Create a synchronous message with `item_code` and `item_data`.``
6. `reply ← Send `message` with `cookie`. Receive another message back.``
7. `out ← Copy the `reply` message data.`
Packaging data into *Parcels*

Just a matter of copying data from *KDBus’s Items* to *Binder’s Parcels*... Except we can send/receive Binder objects!

Example taken for *SurfaceFlinger*:

```cpp
def createConnection()
{
    Parcel data, reply;
    remote()->transact(BnSurfaceComposer::CREATE_CONNECTION, data, &reply);
    return interface_cast<ISurfaceComposerClient>(reply.readStrongBinder());
}
```
Packaging *Binder* objects into *Parcels*

We can pass *Binder* objects by sending their *KDBus* connection ID over the bus.

- Sending a service reference:

  ```cpp
  status_t Parcel::writeStrongBinder(const sp<IBinder>& val);
  ```

  - Remote: send the connection ID.
  - Local: get the connection ID from the table and send it.

- Receiving a service reference:

  ```cpp
  sp<IBinder> Parcel::readStrongBinder() const;
  ```

  - Remote: create a new remote *Binder* object from the ID.
  - Local: return the local *Binder* object with this ID.
Binder: Getting notified when a service dies

A client can register an object with a remote Binder:

```cpp
class WhatToDo : public IBinder::DeathRecipient {
    public:
        virtual void binderDied(const wp<IBinder>& who) override {
            // Complain.
        }
};
```

If the service dies, the `binderDied` method will be called.

```cpp
sp<IBinder> adder = service_manager->getService("org.compute.adder");
adder->linkToDeath(new WhatToDo);
```
**Binder: linkToDeath**

*Binder* defines a way to execute code when a given service dies. *KDBus* provides this with an `KDBUS_ITEM_ID_REMOVE`. The client will receive a notification in its memory pool.

- We can do this in the exact same way we handle services.
- Add a local per process table in the client:
  
<table>
<thead>
<tr>
<th>Connection to <em>KDBus</em></th>
<th>DeathRecipient object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID 99</td>
<td><code>sp&lt;IDeathRecipient&gt; whatToDo</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Spawn threads handling notifications.