Integrating *KDBus* in Android™

Pierre Langlois <pierre.langlois@arm.com>

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.

ARM

What we want to achieve.

Investigate *KDBus* as a replacement for *Binder*:

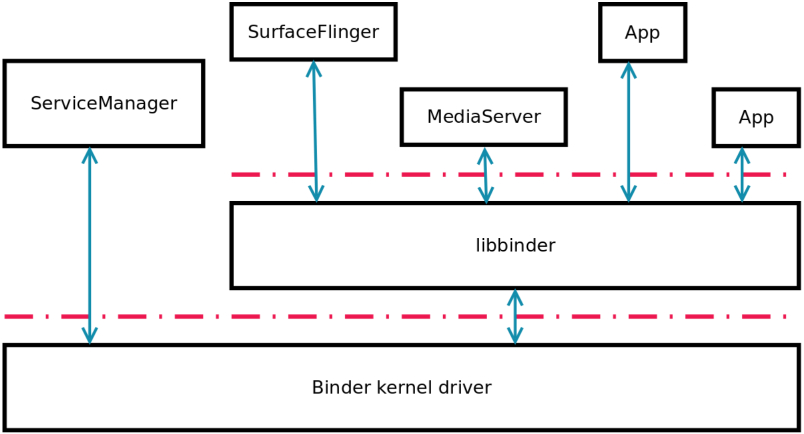
- Understand if it can be done.
- Build a proof-of-concept.
- Identify potential blockers and difficult problems.

Things we haven't looked at:

- Any sort of measurement / profiling.
- Comparing security mechanisms.

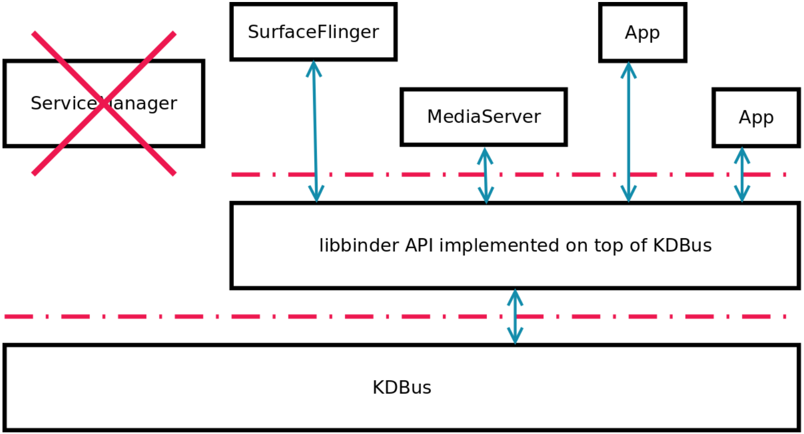
Our work in a nutshell

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:

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

  virtual int add(int a, int b) = 0;
};
```

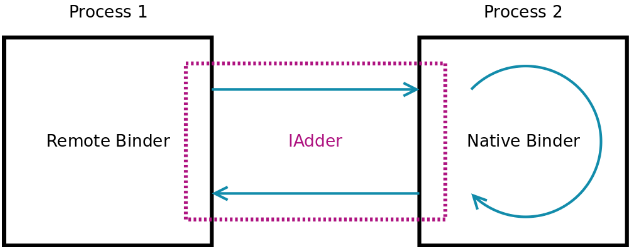
ARM

Binder: Remote calls

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

```
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:

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

Native stub:

```
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();
  }
};
```


ARM

Binder object abstraction: Remote proxy

- Package the data.

```
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.

```
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.

```
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.

```
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;
      }
    }
  }
}
```


ARM

Binder object abstraction: Native stub

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

```
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.

```
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.

```
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;
      }
    }
  }
}
```


ARM

Binder object abstraction: Remote proxy

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

```
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();
  }
};
```

ARM

Binder object abstraction: Remote proxy

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

```
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();
  }
};
```


ARM

Binder: Searching and registering services

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

- Accessing this special object:

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

- Registering our new service with it:

```
sp<IBinder> adder = new BnAdder();
service_manager->addService("org.compute.adder", adder);
```

- Finding the service:

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


ARM

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

ARM

Binder kernel driver

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

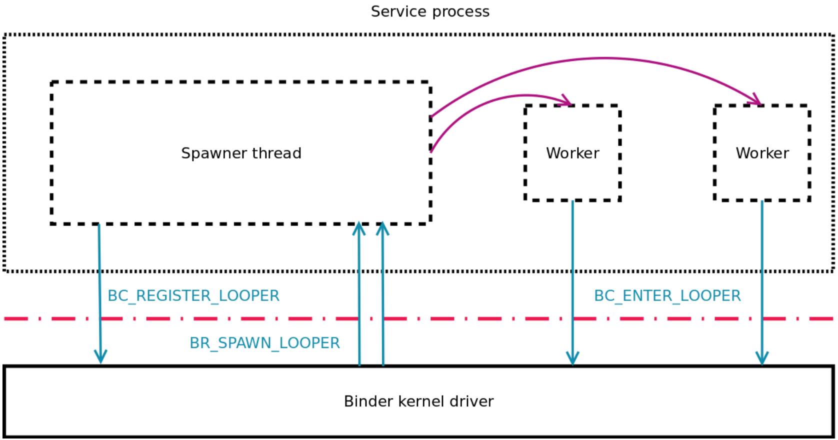
$$BC\_* \leftrightarrow BR\_*$$

- Maintain per-process memory pools.
- Manages worker threads.
- Dispatch data from one process to another.

ARM

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.

ARM

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.

ARM

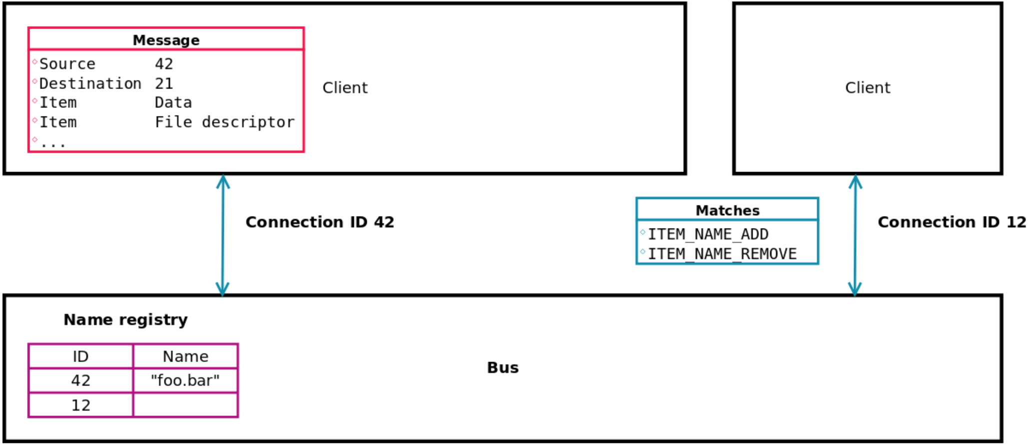
Overview of *KDBus*

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
```


ARM

KDBus: Overview



We built a small abstraction library around this and will use it in this talk.

ARM

Hello *KDBus*

- Creating a bus:

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

  ```
  // 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.

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

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

  ```
  c->acquire_name("foo.bar");
  ```

Finding other *Connections*

Connections can probe the bus:

```
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-myname");
for (const auto& name : c->list(Names)) {
  // (...)
}
```

Items

Everything sent to/from *KDBus* is an *Item*:

- Plain old data: copied, shared with *memfd* or file descriptor.

```
auto payload = ItemPayloadVec(&some_data, sizeof(some_data));
```

- Identifiers: Name of a bus, a connection, ...etc.

```
auto 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*.

```
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.

ARM

Outline

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

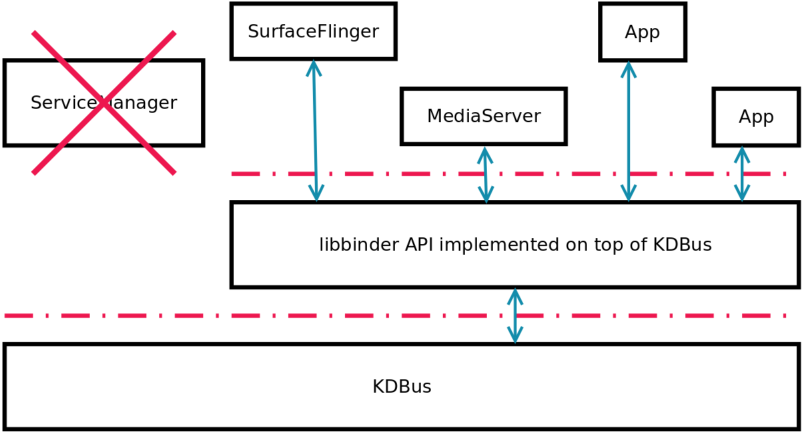
We have all we need to implement transactions!

… KDBus will not manage threads for us.

Implementing *libbinder*'s API with *KDBus*

ARM

Introducing *libkdbinder*



ARM

Registering a service with *KDBus*

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

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

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

```
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.

ARM

Registering a service with *KDBus*

```
sp<IBinder> adder = new BnAdder();
service_manager->addService("org.compute.adder", adder);
```

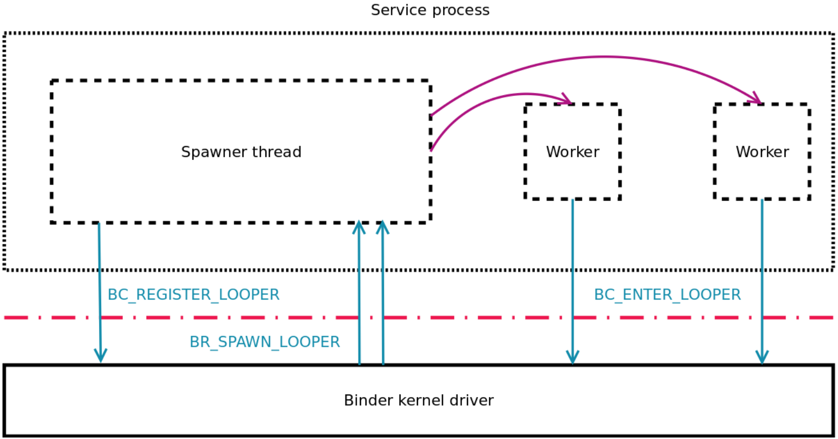
- Create a *Connection* to *KDBus* with a well-known name.
- Register it in a local per process table:

Connection to *KDBus*	*Binder* object
"org.compute.adder"	sp<IBinder> adder
…	…

ARM

Handling requests: Per process thread pool

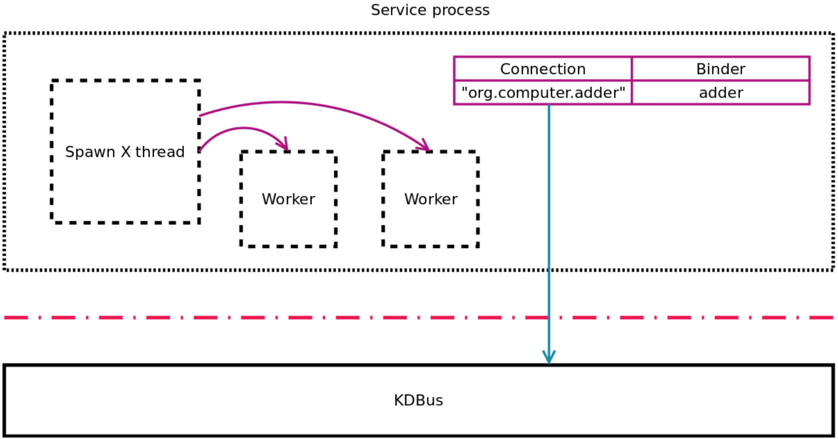
Reminder: The *Binder* driver manages threads for us.



ARM

Handling requests: Per process thread pool

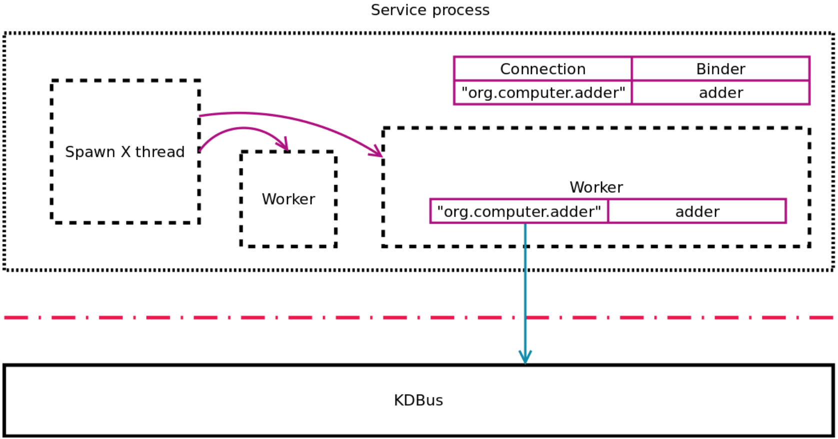
KDBus does not, let's simply spawn *X* threads.



ARM

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*:

MessageSync	
Source	42
Destination	12
Cookie	0x123456789
Timeout	1000
Item	Code
Item	Parcel data

- We get a *Message* back:

MessageReply	
Source	12
Destination	42
Cookie	0x123456789
Item	Parcel data

- Unpack the *Item* in a *Parcel*

ARM

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.

ARM

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.

ARM

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.

ARM

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.

ARM

Thank You

Backup slides

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();
```


ARM

KDBus: Synchronous example

```
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:

```
for all entry in the service table do
    entry ← Copy entry, protected by a per process mutex.
    message ← Dequeue a message from the connection's memory pool.
    if not time out then
        cookie ← Read cookie value from message.
        code ← Read transaction code from message.
        Parcel in ← Read data from message.
        Parcel out ← Call the Binder object with code and in.
        message ← Write out into a KDBus message.
        Send the msg reply with the same cookie.
    end if
end for
```

Sending a request: *Parcel* in / *Parcel* out

We have a remote *Binder* object $\longrightarrow$ we know the *KDBus* connection ID.

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

ARM

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*:

```
virtual sp<ISurfaceComposerClient> 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:

  ```
  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:

  ```
  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*:

```
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.

```
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:

Connection to *KDBus*	DeathRecipient object
ID 99	sp<IDeathRecipient> whatToDo
...	...

- Spawn threads handling notifications.