BINDER TRANSACTIONS IN THE BOWELS OF THE LINUX KERNEL

Written by Jean-Baptiste Cayrou - 14/12/2018 - in Système

Binder is the main IPC/RPC (Inter-Process Communication) system in Android. It allows applications to communicate with each other and it is the base of several important mechanisms in the Android environment. For instance, Android services are built on top of Binder.
Message exchanged with Binder are called binder transactions, they can transport simple data such as integers but also process more complex structures like file descriptors, memory buffers or weak/strong references on objects.

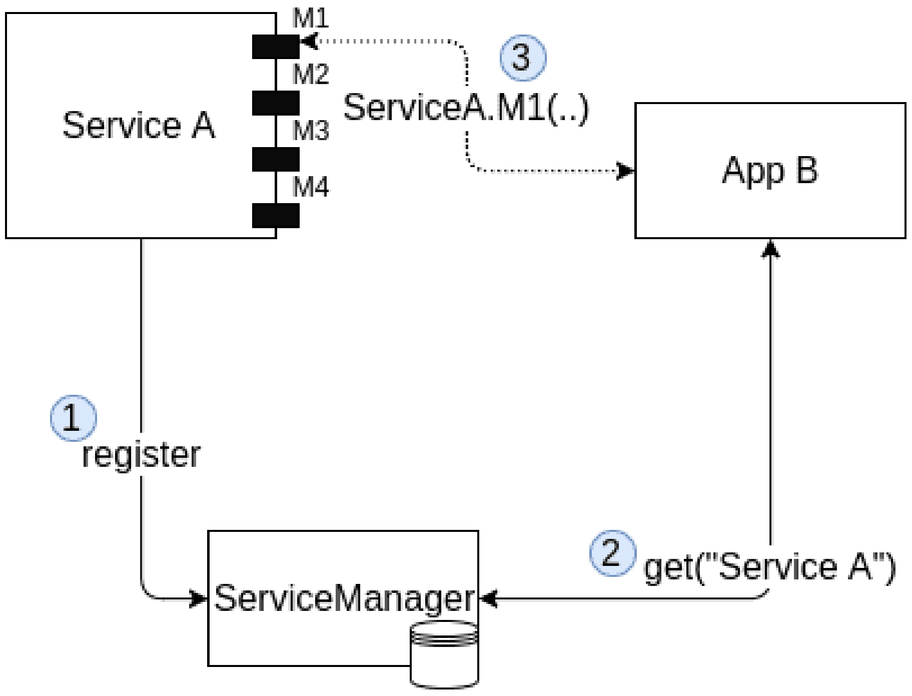
There are a lot of interesting Binder documentations available on the Internet but quite few details on how messages are translated from a process to another. This article tries to describe how Binder handles messages and performs translations of complex objects (file descriptors, pointers) between different processes. For this, a binder transaction will be followed from userland to the binder kernel.

BINDER IN USERLAND

Before exploring how the Binder kernel module works, let's see how a transaction is prepared in the userland, in the case of a call to an Android Service.

ANDROID SERVICE OVERVIEW

Services are Android components that run in background and provide features to others applications. Some of them are part of the Android framework, but installed applications can expose their own features as well. When an application wants to expose a new service, it first registers to the "Service Manager" **(1)** which contains and updates a list of all running services. Later, a client asks an handle to the ServiceManager **(2)** to communicate with this service and be able to call exposed functions **(3)**.



Service Interaction

Since Android 8.0, there are three differents Binder domains. Each domain has its own service manager and is accessible with the corresponding device located in `/dev/`. There are one device per Binder domain as described in the following table :

IPC Domain	Description
`/dev/binder`	IPC between framework/app processes with AIDL interfaces

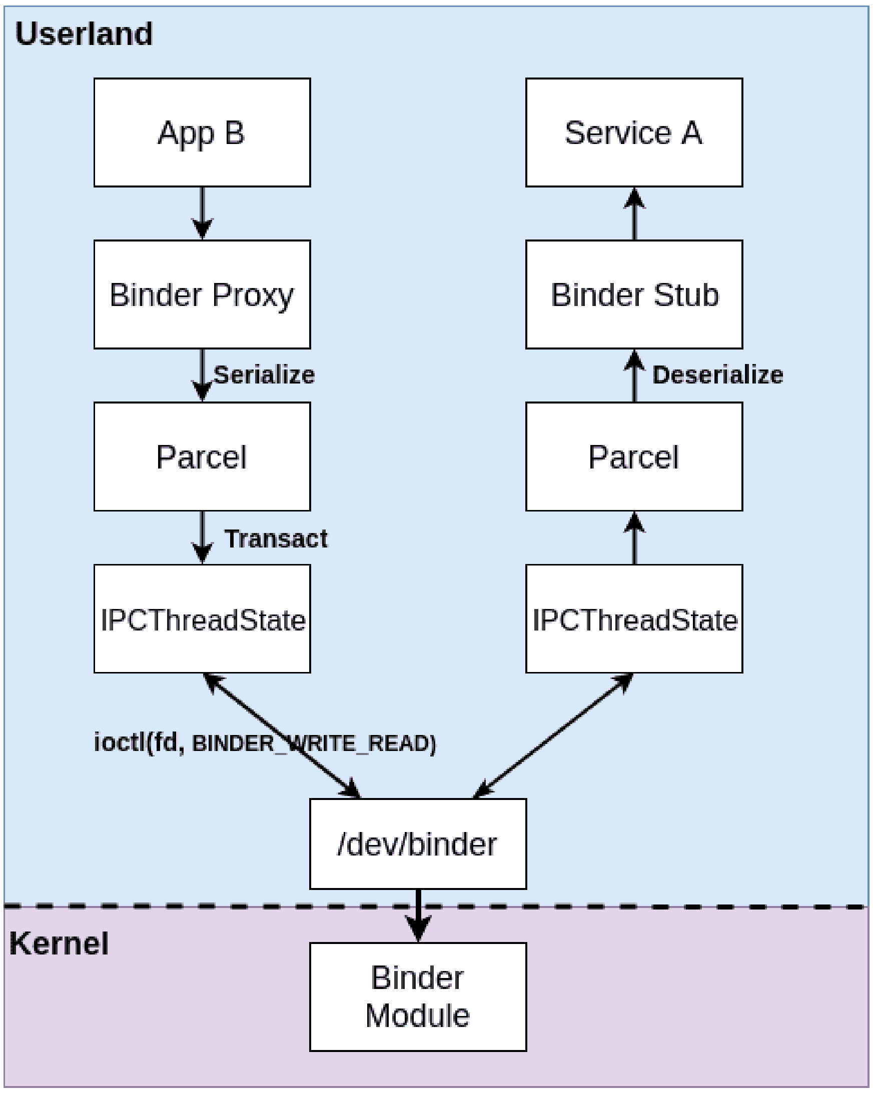
`/dev/hwbinder`	IPC between framework/vendor processes with HIDL interfaces IPC between vendor processes with HIDL interfaces
`/dev/vndbinder`	IPC between vendor/vendor processes with AIDL Interfaces

Binder domains, extract from Android documentation website

To use the binder system, a process needs to open one of these devices and performs some initialization steps before sending or receiving binder transactions.

PREPARATION OF A BINDER TRANSACTION

The Android Framework contains several abstraction layers on top of the binder device. Usually when developpers implement new services they describe interfaces exposed in a high level language. In the case of framework application, descriptions are written with the AIDL language, while hardware services developped by vendors, have interface descriptions written in HIDL language. Theses descriptions are compiled into Java/C++ files where parameters are de/serialized using **Parcel** component. Generated code contains two classes, a **Binder Proxy** and a **Binder Stub**. The proxy class is used to request a distant service and the stub to receive incoming call as described on the following diagram.



Binder Layers

At the lowest level, applications are connected to the Binder kernel module using the domain corresponding device. They use the `ioctl` syscall to send and receive binder messages.

The serialization step is done with the Parcel classes which provides functions to read and write data in a Binder message. There are two differents Parcel classes :

- The `/dev/binder` and `/dev/vndbinder` domains are based on AIDL description language and use the Parcel defined in `frameworks/native/include/binder/Parcel.h`. This Parcel allows to send **basic types** and **file descriptors**. As example, the following code is an extract of the default proxy implementation of the command `SHELL_COMMAND_TRANSACTION`. The command prepares and writes file descriptors of standard input, output and error streams which are used by remote services.

```
// Extract from frameworks/base/core/java/Android/os/Binder.java
public void shellCommand(FileDescriptor in, FileDescriptor out, FileDescriptor err,
        String[] args, ShellCallback callback,
        ResultReceiver resultReceiver) throws RemoteException {
```

```
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        data.writeFileDescriptor(in);
        data.writeFileDescriptor(out);
        data.writeFileDescriptor(err);
        data.writeStringArray(args);
        ShellCallback.writeToParcel(callback, data);
        resultReceiver.writeToParcel(data, 0);
        try {
            transact(SHELL_COMMAND_TRANSACTION, data, reply, 0);
            reply.readException();
        } finally {
            data.recycle();
            reply.recycle();
        }
    }
```

- The `/dev/hwbinder` domain uses another Parcel implemented in `system/libhwbinder/include/hwbinder/Parcel.h` and based on the previous one. This Parcel implementation allows to send **data buffers** like C structures. Data buffers can be nested and contain pointers to other structures. In the following example the structure `hild_memory` structure contains an embedded structure (`hild_string`) and a memory pointer (`mHandle`):

```
    // Extract from  system/libhidl/transport/include/hidl/HidlBinderSupport.h

    // ---------------------- hidl_memory
    status_t readEmbeddedFromParcel(const hidl_memory &memory, const Parcel &parcel, size_t pa
rentHandle, size_t parentOffset);

    status_t writeEmbeddedToParcel(const hidl_memory &memory, Parcel *parcel, size_t parentHan
dle, size_t parentOffset);


    // [...]

    // Extract from system/libhidl/base/include/hidl/HidlSupport.h
    struct hidl_memory {
        // ...
        private:
            hidl_handle mHandle     __attribute__ ((aligned(8)));
            uint64_t    mSize        __attribute__ ((aligned(8)));
            hidl_string mName       __attribute__ ((aligned(8)));
    };
```

These two kinds of Parcel are able to send file descriptors and complex data structures with memory addresses. Because these elements contain data which are specific to the caller process, Parcel components write **binder objects** in the transaction message.

BINDER OBJECTS

In addition of simple types (String, Integer, etc) it is possible to send binder objects. Binder objects are structures with a type value among one of these :

```
// Extract from : drivers/staging/Android/uapi/binder.h
enum {
    BINDER_TYPE_BINDER      = B_PACK_CHARS('s', 'b', '*', B_TYPE_LARGE),
    BINDER_TYPE_WEAK_BINDER = B_PACK_CHARS('w', 'b', '*', B_TYPE_LARGE),
    BINDER_TYPE_HANDLE      = B_PACK_CHARS('s', 'h', '*', B_TYPE_LARGE),
    BINDER_TYPE_WEAK_HANDLE = B_PACK_CHARS('w', 'h', '*', B_TYPE_LARGE),
    BINDER_TYPE_FD          = B_PACK_CHARS('f', 'd', '*', B_TYPE_LARGE),
    BINDER_TYPE_FDA         = B_PACK_CHARS('f', 'd', 'a', B_TYPE_LARGE),
    BINDER_TYPE_PTR         = B_PACK_CHARS('p', 't', '*', B_TYPE_LARGE),
};
```

Below an example of a binder object with the type `BINDER_TYPE_PTR` :

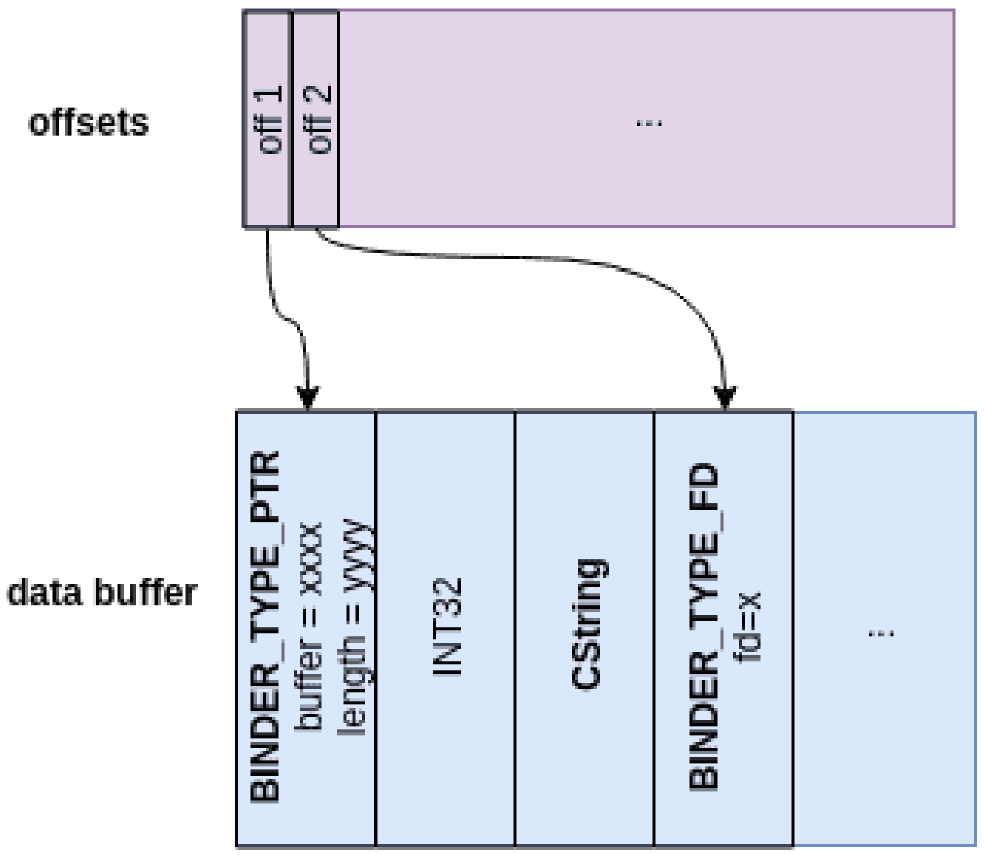
```
struct binder_object_header {
    __u32        type;
};

struct binder_buffer_object {
    struct binder_object_header hdr;
    __u32                   flags;
    binder_uintptr_t        buffer;
    binder_size_t           length;
    binder_size_t           parent;
    binder_size_t           parent_offset;
};
```

Attributes below the `hdr` are specifics to a type.

The different binder objects can be described as following :

- **BINDER_TYPE_BINDER** and **BINDER_TYPE_WEAK_BINDER** : These types are strong and weak references to a local object.
- **BINDER_TYPE_HANDLE** and **BINDER_TYPE_WEAK_HANDLE** : These types are strong and weak references to a remote object.
- **BINDER_TYPE_FD** : This type is used to send a file descriptor number. This is often used to send a ashmem shared memory to transfert a large amount of data. Indeed, binder transaction messages are limited to 1 MB. However any file descriptor types can be used (File, Sockets, stdin, etc).
- **BINDER_TYPE_FDA** : Object describing an array of file descriptors.
- **BINDER_TYPE_PTR** : Object used to send a buffer using a memory address and its size.

When the Parcel class writes a buffer or a file descriptor it adds the binder object in the data buffer (blue on the diagram). Binder objects and simple types are mixed in the data buffer. Each time an object is written, its relative position is inserted in the offsets buffer (in purple).



Binder message buffer and offsets

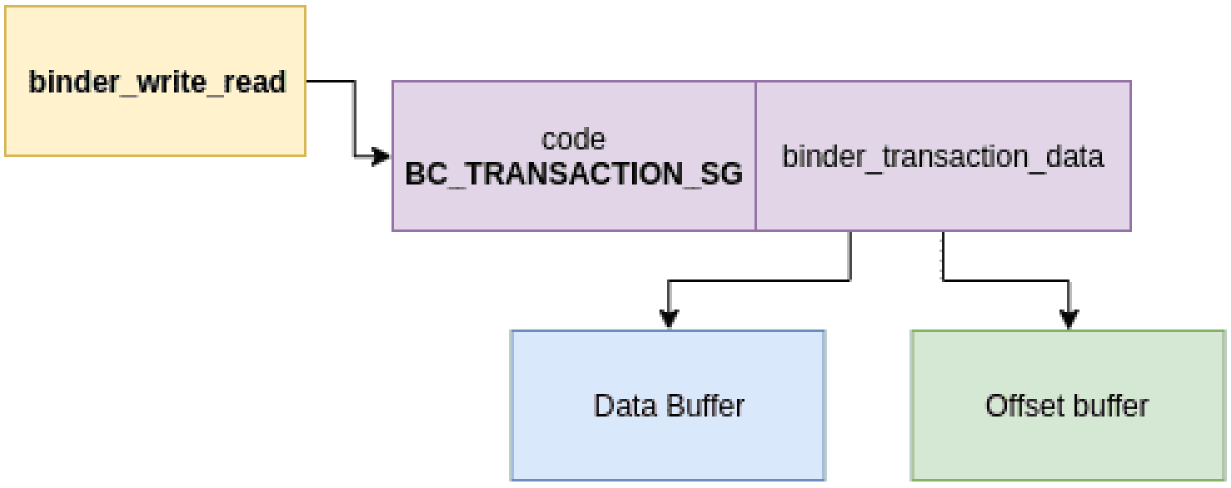
Once the `data` and `offsets` buffers are filled, a `binder_transaction_data` is prepared to be passed to the kernel. We can notice it contains pointers and sizes of data buffer and offsets arrays described above. The field `handle` is used to set the target process which was previously retrieved with the service manager. Another interesting attribute is the `code` on which contains the method id of the remote service to execute.

```
// file : development/ndk/platforms/android-9/include/linux/binder.h

struct binder_transaction_data {
   union {
       size_t handle;
       void *ptr;
   } target;
   void *cookie;
   unsigned int code;

   unsigned int flags;
   pid_t sender_pid;
   uid_t sender_euid;
   size_t data_size;
   size_t offsets_size;

   union {
       struct {
           const void *buffer;
           const void *offsets;
       } ptr;
       uint8_t buf[8];
   } data;
};
```

A last structure (`binder_write_read`) must be filled before calling the ioctl. It contains read and write command buffers and points to the previous one :

```
// file : development/ndk/platforms/android-9/include/linux/binder.h

struct binder_write_read {
    signed long write_size;
    signed long write_consumed;
    unsigned long write_buffer;
    signed long read_size;
    signed long read_consumed;
    unsigned long read_buffer;
};
```

Data structures needed to send a binder transaction can be summed up with this diagram :



binder_write_read structures

We can notice that the `write_buffer` does not point directly on the `binder_transaction_data` structure. Its is prefixed by the command identifier. In the case of a transaction the value is `BC_TRANSACTION_SG` .

Note that many commands exist in addition of `BC_TRANSACTION_SG` for instance `BC_ACQUIRE` and `BC_RELEASE` to increase or decrease a strong handle or `BC_REQUEST_DEATH_NOTIFICATION` to be noticed when the remote service is stopped.

Now all is ready to perform a binder transaction, the caller needs to invoke an `ioctl` with command `BINDER_WRITE_READ` and the kernel module will process the message and translate all binder objects for the target process : strong/weak handles, file descriptors and buffers.

Let's continue the analysis on the Kernel side in the next part !

BINDER KERNEL MODULE

Now the caller process has prepared its data and performed an ioctl to send the transaction. All binder objects will be translated and the message will be copied in the memory of the target.

The command used for the `ioctl` is processed by the `binder_ioctl_write_read` function which performs secure copy of data arguments.

```
// file : drivers/android/binder.c

static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    // [...]
    switch (cmd) {
    case BINDER_WRITE_READ:
        ret = binder_ioctl_write_read(filp, cmd, arg, thread);
        if (ret)
            goto err;
        break;
```

```
// file : drivers/android/binder.c

static int binder_ioctl_write_read(struct file *filp,
                unsigned int cmd, unsigned long arg,
                struct binder_thread *thread)
{
     // [...]
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
            ret = -EFAULT;
            goto out;
    }
    // [...]
    if (bwr.write_size > 0) {
        ret = binder_thread_write(proc, thread,
                      bwr.write_buffer,
                      bwr.write_size,
                      &bwr.write_consumed);
```

In the case of a writing transaction, the function `binder_thread_write` is called and then dispatches the command associated with the transaction to the corresponding handle.

```
// file : drivers/android/binder.c

switch (cmd) {
        case BC_INCREFS:
        case BC_ACQUIRE:
        case BC_RELEASE:
        case BC_DECREFS:
        // [...]
        case BC_TRANSACTION_SG:
        case BC_REPLY_SG: {
            struct binder_transaction_data_sg tr;

            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
            binder_transaction(proc, thread, &tr.transaction_data,
                       cmd == BC_REPLY_SG, tr.buffers_size);
            break;
        }
        // [...]
```

For a command `BC_TRANSACTION_SG`, the binder_transaction_data buffer prepared in the userland is processed by the `binder_transaction` function.

BINDER TRANSACTION

The `binder_transaction` function is located in file `drivers/staging/Android/binder.c`.

This big function performs several tasks: Allocate a buffer in the destination process (in the binder reserved memory), validate all data objects and perform translations, copy data and offsets buffers in the destination memory process.

To validate binder objects, the kernel look over the `offsets` buffer containing the relative positions of all objects. Depending on the object type, the kernel perform different translations.

```
// file : drivers/android/binder.c

static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply,
                   binder_size_t extra_buffers_size){
    // [...]
    // Object validation in binder_transaction function.
    // offp is a pointer to the offsets buffer

    for (; offp < off_end; offp++) {
        struct binder_object_header *hdr;
        size_t object_size = binder_validate_object(t->buffer, *offp);

        if (object_size == 0 || *offp < off_min) {
            binder_user_error("%d:%d got transaction with invalid offset (%lld, min %lld max %lld) o
r object.\n",
                      proc->pid, thread->pid, (u64)*offp,
                      (u64)off_min,
                      (u64)t->buffer->data_size);
            return_error = BR_FAILED_REPLY;
            return_error_param = -EINVAL;
            return_error_line = __LINE__;
            goto err_bad_offset;
        }

        hdr = (struct binder_object_header *)(t->buffer->data + *offp);
        off_min = *offp + object_size;
        switch (hdr->type) {
        case BINDER_TYPE_BINDER:
        case BINDER_TYPE_WEAK_BINDER: {
                // [..] Validation and Translation
        case BINDER_TYPE_HANDLE:
        case BINDER_TYPE_WEAK_HANDLE: {
                // [..]  Validation and Translation
        }
        case BINDER_TYPE_FD:{
                // [..]  Validation and Translation
        }
        case BINDER_TYPE_FDA:{
                // [..]  Validation and Translation
        }
        case BINDER_TYPE_PTR: {
                // [..]  Validation and Translation
        }
```

WEAK/STRONG BINDER/HANDLE

A binder object reference can be either a virtual memory address that refers to a local object (a binder reference) or a handle that identifies a remote object of another process (a handle reference).

When the kernel gets an object reference (local or remote), it updates an internal table which contains for each process a mapping between real virtual memory addresses and handles (binder <=> handle).

There are two kinds of translations:

- Convert a virtual memory address to an handle : `binder_translate_binder`
- Convert an handle to a virtual memory address : `binder_translate_handle`

The Binder kernel module keeps reference counts of shared objects. When a reference is shared with a new process, its counter value is incremented. When a reference is no longer used, the owner is notified and may release it.

BINDER -> HANDLE TRANSLATION

```c
// file : drivers/android/binder.c

static int binder_translate_binder(struct flat_binder_object *fp,
                        struct binder_transaction *t,
                        struct binder_thread *thread)
{
    // [...]
    node = binder_get_node(proc, fp->binder);
    if (!node) {
        node = binder_new_node(proc, fp);
        if (!node)
            return -ENOMEM;
    }
    if (fp->cookie != node->cookie) {
        // [...] ERROR
    }
    // SELinux check
    if (security_binder_transfer_binder(proc->tsk, target_proc->tsk)) {
        // [...] ERROR
    }

    ret = binder_inc_ref_for_node(target_proc, node,
            fp->hdr.type == BINDER_TYPE_BINDER,
            &thread->todo, &rdata);
    if (ret)
        goto done;

    if (fp->hdr.type == BINDER_TYPE_BINDER)
        fp->hdr.type = BINDER_TYPE_HANDLE;
    else
        fp->hdr.type = BINDER_TYPE_WEAK_HANDLE;
    fp->binder = 0;
    fp->handle = rdata.desc;
    fp->cookie = 0;
    // [..]
}
```

The function gets the node corresponding to the binder value (virtual address) or creates a new one if it does not exist. This node has a correlation between the local object and the remote object (`rdata.desc`). After a SELinux security check, the reference counter is incremented and the reference value is changed in the binder object and replaced by the reference handle.

HANDLE -> BINDER TRANSLATION

```c
// file : drivers/android/binder.c

static int binder_translate_handle(struct flat_binder_object *fp,
                        struct binder_transaction *t,
                        struct binder_thread *thread)
{
    // [...]
    node = binder_get_node_from_ref(proc, fp->handle,
            fp->hdr.type == BINDER_TYPE_HANDLE, &src_rdata);
    if (!node) {
         // [...] Error
    }
    // SELinux security check
    if (security_binder_transfer_binder(proc->tsk, target_proc->tsk)) {
        ret = -EPERM;
        goto done;
    }

    binder_node_lock(node);
    if (node->proc == target_proc) {
        if (fp->hdr.type == BINDER_TYPE_HANDLE)
            fp->hdr.type = BINDER_TYPE_BINDER;
        else
            fp->hdr.type = BINDER_TYPE_WEAK_BINDER;
        fp->binder = node->ptr;
        fp->cookie = node->cookie;
        // [...]

        binder_inc_node_nilocked(node,
                     fp->hdr.type == BINDER_TYPE_BINDER,
                     0, NULL);
        // [...]
    } else {
        struct binder_ref_data dest_rdata;

        ret = binder_inc_ref_for_node(target_proc, node,
                fp->hdr.type == BINDER_TYPE_HANDLE,
                NULL, &dest_rdata);
        // [...]
        fp->binder = 0;
        fp->handle = dest_rdata.desc;
        fp->cookie = 0;
    }
done:
    binder_put_node(node);
    return ret;
}
```

This translation function is quite similar to the previous one. However, we can notice that an handle reference can be shared accross differents process. A handle reference is only translated in binder reference if the target process matched with the node.

FILE DESCRIPTOR

When a binder object type is BINDER_TYPE_FD or BINDER_TYPE_FDA the kernel needs to check if the file descriptor is correct (associated with an opened struct file) and copy it in the target process. Translation is done by the `binder_translate_fd` function. Details below :

```c
// file : drivers/android/binder.c

static int binder_translate_fd(int fd,
                        struct binder_transaction *t,
                        struct binder_thread *thread,
                        struct binder_transaction *in_reply_to)
{
    // [...]
    // 1 : Check if the target allows file descriptors
    if (in_reply_to)
        target_allows_fd = !!(in_reply_to->flags & TF_ACCEPT_FDS);
    else
        target_allows_fd = t->buffer->target_node->accept_fds;
    if (!target_allows_fd) {
        binder_user_error("%d:%d got %s with fd, %d, but target does not allow fds\n",
                  proc->pid, thread->pid,
                  in_reply_to ? "reply" : "transaction",
                  fd);
        ret = -EPERM;
        goto err_fd_not_accepted;
    }
    // 2 : Get file struct corresponding to the filedescriptor number
    file = fget(fd);
    if (!file) {
        binder_user_error("%d:%d got transaction with invalid fd, %d\n",
                  proc->pid, thread->pid, fd);
        ret = -EBADF;
        goto err_fget;
    }
    // 3 : SELinux check
    ret = security_binder_transfer_file(proc->tsk, target_proc->tsk, file);
    if (ret < 0) {
        ret = -EPERM;
        goto err_security;
    }

    // 4 : Get a 'free' filedescriptor number in the target process.
    target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
    if (target_fd < 0) {
        ret = -ENOMEM;
        goto err_get_unused_fd;
    }

    // 5 : This inserts the 'file' into the target process with the target_fd filedescriptor n
umber.
    task_fd_install(target_proc, target_fd, file);

    return target_fd;
    // [...]
}
```

After some checks, the last call to `task_fd_install` adds the file associated to the caller file descriptor in the target process. Internally it uses the kernel API function `__fd_install` that installs a file pointer in the process fd array.

BUFFER OBJECT

Buffer objects are the most interesting. They are used by the Parcel class of hardware services and allow to transfert a memory buffer. Buffer objects have a hierarchy mechanism that allows to patch an offset of the parent. This is very useful to send a structure containing pointers. Binder buffer objects are defined by the following structure:

```
// file : include/uapi/linux/android/binder.h

struct binder_buffer_object {
    struct binder_object_header hdr;
    __u32                       flags;
    binder_uintptr_t            buffer;
    binder_size_t               length;
    binder_size_t               parent;
    binder_size_t               parent_offset;
};
```

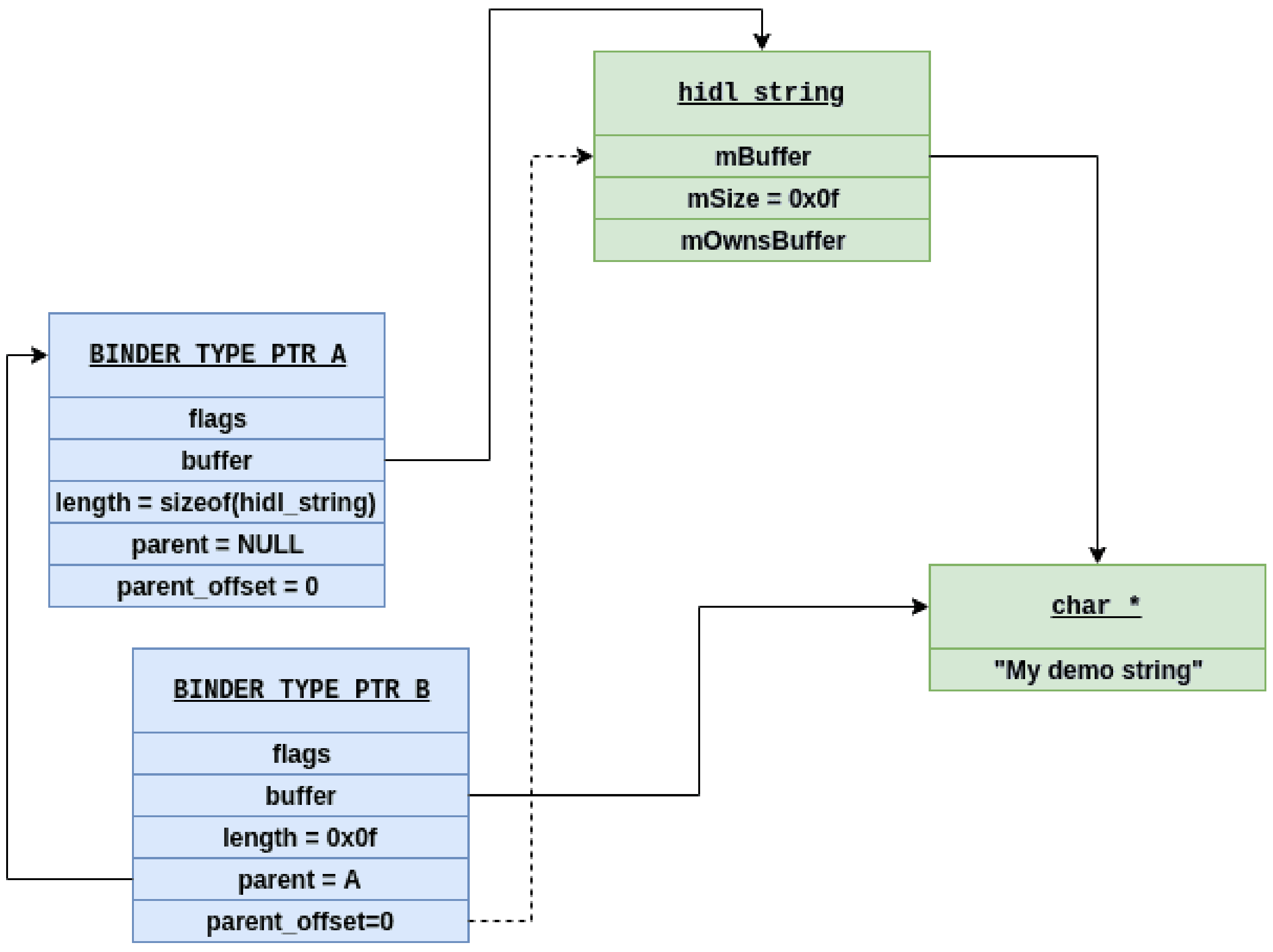
Let's see an example : We have the following code and we want to send an instance of the `hidl_string` structure using Binder.

```
struct hidl_string {
    // copy from a C-style string. nullptr will create an empty string
    hidl_string(const char *);
    // ...
private:
    details::hidl_pointer<const char> mBuffer;      // Pointer to the real char string
    uint32_t mSize;  // NOT including the terminating '\0'.
    bool mOwnsBuffer; // if true then mBuffer is a mutable char *
};

hidl_string  my_obj("My demo string");
```

When my_obj is created, a heap allocation is performed to store the given string and the attribute `mBuffer` is set. To send this object to another process, two `BINDER_TYPE_PTR` objects are needed:

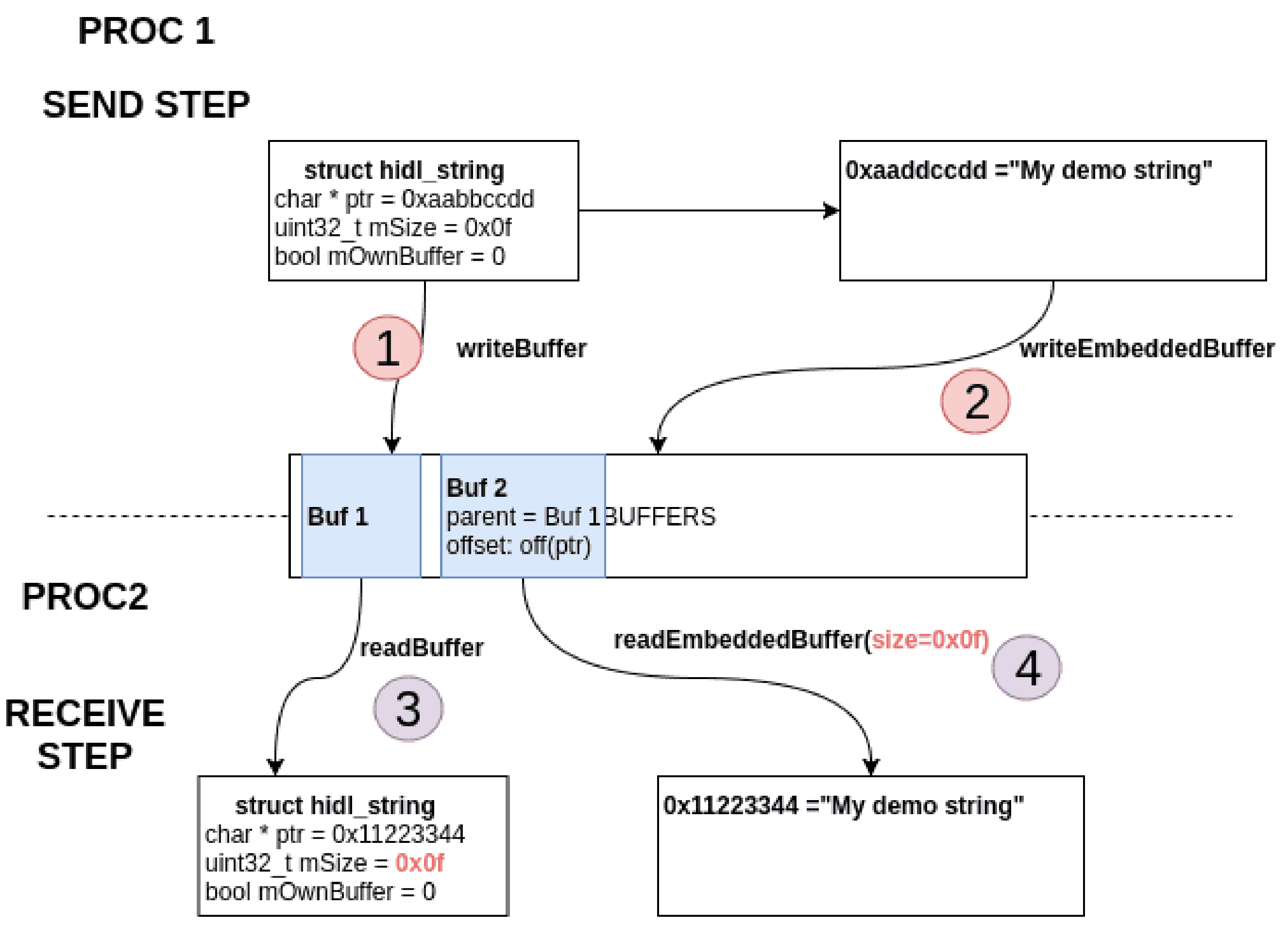
- A first `binder_buffer_offset` with buffer field pointing to the `my_obj` structure
- A second one that points to the string in the heap. This object must be a child of the previous and set the parent_offset attribute to the position of `char * str` in the structure

The following diagram details the configuration of the two binder objects needed:



Binder message buffer

When the kernel translates these objects it patches offsets described in children buffer and copy the different buffers ([object.buffer, object.buffer + object.length]) into the target memory process. In our case, the offset corresponding to the attribute `mBuffer` is patched with the pointer where the string is stored in the target memory process.



Binder message buffer

To parse `my_obj` data, the target process read the first buffer to get the `hidl_struct` (**3**) and next buffer with an expected size of `mSize` to ensure the size described in the structure (`mSize`) is the same that the size of the buffer containing the string (**4**).

CONCLUSION

Binder is a complex and powerfull IPC/RPC system which makes the whole Android ecosystem work. Even though the kernel component is old, there are few documentation on how it works. Moreover, the interesting objects types `BINDER_TYPE_FDA` and `BINDER_TYPE_PTR` were recently added in the Android kernel. These new types are the base of communications (HIDL) in the new HAL architecture introduced in Android 8.0 with the project `Treble`.

REFERENCES

- Andevcon-Binder of Jonathan Levin
- HIDL Android Documentation
- Android 9.0.0_r3 source code
- Binder source code