Sigma: Android Binder Services over Network

KASTURI RANGAN RAGHAVAN University of California, Los Angeles

Contents

1. INTRODUCTION AND PROBLEM DEFINITION

The Android mobile platform is built on a service-oriented architecture. Yet there are no idiomatic mechanisms to provide local services to remote Android clients. This paper introduces Sigma, an experimental platform that builds upon Android's service-oriented runtime to allow tunneling of Android services over network connections. Sigma makes communication between devices a responsibility of the mobile platform. All binder services are supported over the network, even enabling access to system services. Treating every Sigma-enabled device as a peer in a network, Sigma transforms the Android mobile platform into a platform for distributed applications.

1.1 Mobile Apps as Distributed Apps

As motivation, we now discuss a relevant set of mobile applications which can be implemented via the distributed communication model rather than the traditional client-server model.

1.1.1 *Social Apps.* A niche of social services, especially those that connect small sets of participants together for brief periods of time are good candidates for the distributed communication model. These services require a central server only occasionally–for tasks such as to authenticate or to establish an initial network among participants. Voice and video chat applications are good examples. Even today, these apps typically use a direct peer-to-peer (P2P) connection among participants once a chat session is established [6, 23].

The use of real-time context has gained popularity among social apps. There are a whole crop of location sharing apps that give users a glimpse into a small set of friends' location and activities. Apps like Find My Friends [3] for iOS or Glympse [4] for Android. Although it is not known whether these closed-source apps use a peer-to-peer or client-server approach (probably the latter), from their functionality alone it seems feasible to implement with the former model. The basic usefulness is garnered from the streaming of real-time data from participants' mobile devices, and therefore a prerequisite is that all active participants must be reachable over the network. It follows that participants must be able to establish P2P connections too. Also, such services do not need access to historical data, so we can do without a central server's storage and serving capacity. Most importantly, the set of participants is small and stable.

An emerging class of social apps is characterized by the sharing of ephemeral data. The most popular service being Snapchat–a photo messaging app where recipients see the photos shared with them for a brief period (1 to 10 seconds) and then it disappears [13]. The nature of the service is one that affords participants some privacy.

"Snapchat is a super interesting privacy phenomenon because it creates a new kind of space to communicate, which makes it so that things that people previously would not have been able to share, you now feel like you have place to do so," says Mark Zuckerberg–CEO of Facebook. "That's really important, and that's a big kind of innovation that we're going to keep pushing on and keep trying to do more on, and I think a lot of other companies will, too." [18]

However, Snapchat follows a client-server model today. This much is known, because a security breach [25] at Snapchat established that shared pictures (thought to be privately shared and ephemeral) are stored by Snapchat servers long after they have been viewed by recipients. A truly privacy-aware implementation of Snapchat would use a pure P2P approach. If distributed communication is well-supported, it encourages developers create applications following that model. A simple implementation of a Picture Sharing as a distributed application (with the help of Sigma) is provided in section 9

1.1.2 *Internet of Things.* Distributed communication is a valuable model in the Internet of Things (IoT). It is likely that Android a major player in the IoT. Already consumer products like the $30

Chromecast–a bare-bones Android device that plugs into the HDMI port of TVs–is capable of receiving video streams and display from other Android devices over LAN [19]. Chromecast communicates with other Android devices using webrtc, an emerging W3C standard P2P technology [27]. Future internet-enabled home appliances will probably use Android in some form [29]. And the idea of a "smart home" is taking off. Google–the maintainer of Android–recently acquired Nest [5]–a company that formerly developed smart home thermostats.

The IoT is filled with problems of communication between devices. For example, in the smart home an Android-enabled thermostat might poll for sensor readings like temperature and humidity from a mobile Android device in the home. We see already a crop of frameworks–like the Eclipse M2M stack for distributed communication between embedded devices [20] using the publish-subscribe model. Although publish-subscribe is a simple, easy-to-understand model, the Android service-oriented architecture is more than that (as we will see in later sections). It is interesting to experiment with the Android architecture as a distributed communication technology, especially when concerning ourselves with Android-enabled devices in the IoT (rather than all forms of embedded devices).

1.2 Related work

There are standard implementations of P2P technologies and networking libraries. Naturally, these implementations exist outside of the Android Runtime. Sigma aims to use these standard technologies, but cater distribute communication to better fit within Android. There is prior work in Academia very similar to this paper. To give just one example: the Interdroid [7] project also aims to create a platform for distributed applications on Android. However Interdroid defines a separate API for distributed communication that apps must conform to. Related more to the creation of privacy-aware apps is πBox [24]: a platform for privacy-preserving apps, achieving a sandbox that spans across devices, providing secure channels. Since πBox targets Android, it fits the Android model better than standard third-party libraries. Again distributed communication–the API for creating secure channels between devices–requires the reimplementation of apps using the new API. On the other hand, Sigma is an attempt at generalizing the existing service-oriented architecture. This means refactoring existing apps is a less daunting task. Also for creating new apps, the programmer support provided in the Android SDK for creating service-oriented apps can be leveraged. No previous project, to the best of our knowledge, enables distributed communication on Android using Android's very own service-oriented runtime.

The inspiration for this paper is in part from the well-known X windows system. X is a server that manages display and input-device operations. Applications connect to X and follow a simple protocol (X11) to create and coordinate GUI interactions. It is a fully message-based protocol, and is specifically designed to allow for messages to be tunneled through network connections [17]. This allows applications running on one machine to be displayed and controlled via a remote X server. Thus, anyone with an interactive terminal (e.g. ssh) to a machine can interact with GUI applications via X11-forwarding.

1.3 Problem definition

The goal of Sigma is to provide Android services over the network. Android services are provided with an underlying message-based interprocess-communication (IPC) protocol known as Binder. It is implemented as a linux kernel module. The protocol is similar to X11, in the sense that it is message-based, but messages transit through the kernel module and often contain local descriptors–pointers to other binder services or file descriptors managed by the kernel. As such, Binder is not designed to be tunneled over network connections, and there is no such existing facility for it. Sigma is also a message-based protocol–implementing a subset of Binder messages. Its design and implementation is specifically made to support tunneling over network connections. Sigma inspects and rewrites Binder messages containing descriptors, providing operations on descriptors over the network.

This paper explores the Binder framework in depth, and then provides the design and implementation of Sigma. The rest of the paper is organized as follows. Section 2 is an overview of Binder from its kernel module implementation to its OO abstractions in the Android Runtime. Next, Section 3 is an overview of the implementation of a few Android system services, with a discussion on which services make sense to provide over the network. Then the design and implementation detail of Sigma is presented in Section 4, presenting an overall design idea followed by a systematic implementation detail of each piece. We evaluate overhead and test performance in Section 5: first with some simple binder services and then with system services. As an example of a distributed application built from the ground-up, in Section 9 we develop a simple picture sharing service using purely Binder services– made into distributed app with the help of Sigma. Finally, Section 7 contains discussion and next steps, particularly detailing limitations and drawbacks and what we can do to fix them.

2. THE ANDROID BINDER FRAMEWORK

The Binder framework is Android's primary interprocess-communication (IPC) mechanism, and it is used widely within the Android Runtime. System services are provided through binder. A binder allows a process to present a specific interface as a service which may be called by other threads or processes. A binder call is essentially a remote procedure call (RPC) with communication between linux processes achieved using ioctl calls on a file descriptor. The binder at its most basic level is a kernel module. As such binder maintains isolation between processes, maintaining modularity and privilege separation as implemented by the linux kernel [10].

2.1 Kernel Module

The binder kernel module (BKM) provides the facility to register a default binder service known as the system context manager. It is then the job of the context manager to manage binder objects in the operating system. This context manager can be accessed by arbitrary processes via a call to the BKM. In Android the context manager presents an IServiceManager interface. The ServiceManager starts at boot-up as the root system process, and is registered as the context manager with BKM. It manages a directory of system services,where system services publish themselves to the directory via binder RPC.

Using the lowest-level C API for communicating with the BKM, here are the sequence of commands to create and publish a binder service. Incoming requests to the binder service are implemented via a handler function, not shown here. The arguments to binder_call are described in the Appendix 8.3.

```
binder_state* binder_service = binder_open();
binder_state* publish_txn = binder_open();

// RPC to the ServiceManager to publish the  service, details of
// arguments to binder_call  are described in the appendix.
binder_call(publish_txn, msg, reply, (void*)0, SVC_MGR_ADD_SERVICE);

// loop indefinitely. Incoming transactions  will invoke the handler.
binder_loop(binder_service, (void*)service_handler);
```

Snippet 1: binder_call to publish a service with ServiceManager

Note that msg and reply are binder_io data structures that contain the data to be passed into or returned, respectively, from binder_call. Implementing RPC services through binder_call is a matter of convention. In the above example to publish a service, the binder_io msg contains the following elements, with the data encoded into a flat byte array. The 2 strings and 1 object are read in the same order by ServiceManager's handler. Thus implementing the RPC.

```
msg = {
  ”android.os.IServiceManager” /* string name of the remote interface */,
  ”com.example.service” /* string name of service being published */,
  object /* flat_binder_object struct with pointer to binder_service */
}
```

Snippet 2: contents of msg passed into binder_call

A client process can obtain a handle to the just-published service via a binder RPC to ServiceManager with a different code and binder_io arguments, allowing the service to be queried by its published name. The return value (the reply) will contain a descriptor–the handle to the remote binder service. Invoking a binder_call RPC against this new handle will synchronously invoke its associated service handler on a remote process. The interaction between the kernel module and other processes is visualized below in figure 1.

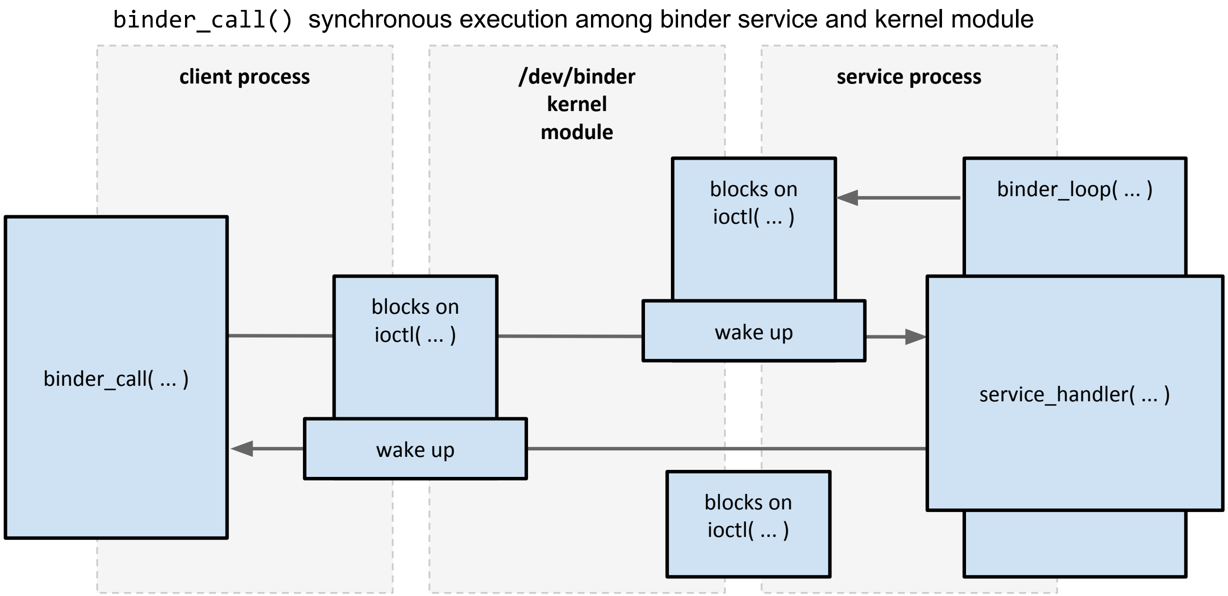


Fig. 1

2.1.1 *binder_io with objects and file descriptors.* An essential feature of binder is the ability to write binder objects and file descriptors as arguments into (or in replies from) RPCs. A process can obtain a handle to a remote object this way, and can invoke a RPC on the object as a callback. Both primitive types and objects are written into the binder_io byte-array. A handle to an object is an descriptor that is tracked by the kernel module, and the Binder driver takes care of rewriting the structure type and data as it moves between processes [2]. This level of understanding of BKM is sufficient in detail for the scope of this paper. Documentation on internals of binder is sparse. The following references [21, 28] provide a useful resource, but the real reference to Binder is in code–its implementation in Android.

2.2 Binder within the Android Runtime

The Android Runtime has several layers of abstractions built on top of the basic BKM. All these layers eventually result in calls either to the lowest-level C API which we have just covered or directly operate on the kernel driver. The main difference is that the C++ and Java APIs are object-oriented (OO), defining an IBinder interface which binder services implement. transact(...) is the method to perform generic operations to implement RPC. It very much resembles the low-level C API's binder_call(...) function. The OO implementation also encapsulates binder_io structures as Parcel objects.

```
interface IBinder {
  // Get the canonical name of the interface supported by this binder.
  String getInterfaceDescriptor();
  boolean transact(int code, Parcel data, Parcel reply, int flags);
  void linkToDeath(DeathRecipient recipient, int flags);
  boolean unlinkToDeath(DeathRecipient recipient, int flags);
}
```

Snippet 3

The OO Binder follows the Proxy Pattern [11]. BBinder is a class that is instantiated as the real implementation of the service. It calls the equivalent of binder_loop and accepts transaction requests. Then there is BpBinder which is a proxy class that is instantiated with a remote handle to an active BBinder service. Analogously, there is a Java implementation that has a mirror IBinder Java interface 3, defining Binder and BinderProxy classes which encapsulate the native BBinder and BpBinder classes, respectively, via JNI.

An invocation of BpBinder.transact() simply invokes a binder_call to its remote handle, and via binder invokes the BBinder.transact() method on the remote process. A service is implemented on top of this structure, by extending the two classes and following a convention to implement RPC via calls to transact(). For instance, a FooBarService is implemented by associating each method with an ID, passed in as the code argument into transact().

```
class BpFooBarService : public BpBinder {
  void foo(int a, int b) { /* ... */ transact(1, encoded_args, ...); }
  void bar(string c) { /* ... */ transact(2, encoded_args, ...); }
  /* transact() invokes BnFooBarService.transact() on remote process
     via binder. Waits for and returns reply parcel. */
}
class BnFooBarService : public BnBinder {
  void foo(int a, int b) { /* Do something useful */ }
  void bar(string c) { /* Another useful fucntion */ }
  void transact(code, msg, reply, flags) {
    switch(code) {
      case 1: /* decode args... */ foo(args); break;
      case 2: /* decode args... */ bar(args); break;
}}}
```

Snippet 4

The creation of binder RPC involves boilerplate code. To make this more convenient, the Android SDK defines a domain-specific language named AIDL to specify interfaces for binder services in a language similar to the language for specifying Java interfaces. The AIDL compiler generate Stub and Proxy classes that implement the specified interface and encapsulate Binder and BinderProxy objects, respectively. A developer simply extends the Stub class, providing an implementation for each service method (like foo() and bar() above). The generated classes do all the necessary work to marshall and unmarshall arguments as Parcels and invoke the appropriate method on the actual service object. For instance, here is the AIDL definition of the FooBarService.

```
interface FooBarService { void foo(int a, int b); void bar(String c); }
```

Snippet 5

A binder object implementing a specified service interface can be created simply by extending the Stub class, providing an implementation for service methods. Remote processes receive a BinderProxy initialized with a handle to the an instance that extends FooBarService.Stub. That way, the actually implemented methods can be invoked by remote processes.

```
IBinder localBinder = new FooBarService.Stub() { void foo(...) { ... }
                                                 void bar(...) { ... } };
```

Snippet 6

2.3 Accessing Binder Services

The Android Runtime provides pathways for components to communicate binder objects between each other. For instance, Android apps can declare a Service component that other components, like GUI Activity components, can bind to. Binding to a service brings up the Service and then returns the result of invoking its onBind method back to the Activity.

```
class FooBarAndroidService extends Service {
  public void onBind() { return new FooBarService.Stub() { ... }; }
}

class FooBarClient extends Activity {
  public void onStart() { bindService(/* name of the service */,
                                      /* callback on connected */ this);
  }
  public void onServiceConnected(
    Context context, IBinder binder) {
    // The binder here is a BinderProxy, not the original Binder object.
    FooBarService.asInterface(binder).foo(1, 3);  // invokes RPC.
  }
}
```

Snippet 7

As we see above, retrieving the binder interface to a service is just an easy-to-understand asynchronous operation from the perspective of the app developer. However, there are a few core services of the Android Runtime that come into play to make it happen. The process is visualized in figure 2 and detailed below.

bindService(...) is invoked with a callback binder object–the ServiceConnection. Once the requested service is up, the ServiceConnection callback is invoked asynchronously with the requested binder service as an IBinder object. The ActivityManager works together with other essential system services to bring up and connect to the specified service component. The PackageManager, which tracks all installed components on the system, validates that the requested service component exists and checks that the caller has permissions to connect to it. If the requested service resides in a separate java package namespace, a low-level component known as Zygote brings up the service as a separate process. When the service is up, it attaches to the ActivityManager via RPC. Afterwards, ActivityManager retrieves the binder service provided by the service component as an IBinder object, and passes it on

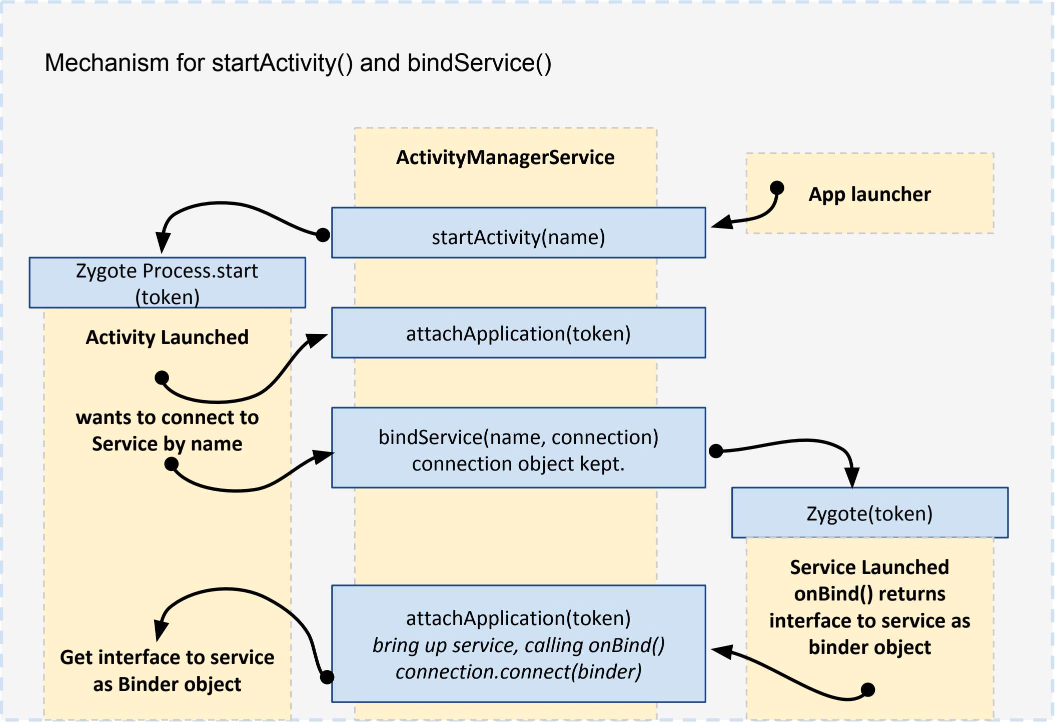

Mechanism for startActivity() and bindService()
ActivityManagerService
App launcher
startActivity(name)
Zygote Process.start (token)
Activity Launched
attachApplication(token)
wants to connect to Service by name
bindService(name, connection) connection object kept.
Zygote(token)
Service Launched onBind() returns interface to service as binder object
attachApplication(token)
bring up service, calling onBind()
connection.connect(binder)
Get interface to service as Binder object

Fig. 2

the caller by invoking the ServiceConnection callback. There are quite a few RPCs involved in this procedure!

2.3.1 *Accessing Android System Services.* bindService(...) cannot be used for everything. For instance, in the course of binding to a service, the ActivityManager brings up the service first and waits for it to attach. The newly-up service cannot use bindService() to get a reference to the ActivityManager, as that is a circular dependence. Instead core system services like ActivityManager are all available through the ServiceManager–the BKM's default context manager. It is straightforward to obtain a reference to the ServiceManager from any process or thread (i.e. make binder_call with target=(void*)0). So it follows that it is always possible to retrieve IBinder references to any currently active service published to the ServiceManager.

The direct method using binder_call() to communicate with ServiceManager (as in Example 1) is not strictly necessary. Rather, ServiceManager has an OO Binder interface, this interface is provided in the Appendix 8.6.

3. SYSTEM SERVICES: PROVIDED OVER THE NETWORK?

Before going into the design and implementation of Sigma, it is useful to understand the kinds of system services offered via binder. This will serve as guide to figure which features of binder are commonly used, and will set a baseline for the features Sigma should support. In particular, we want to answer: which services are useful for remote devices to access? And which cannot be accessed (or don't make any sense to provide access to)? To start, it is reasonable to expose ServiceManager since it is an entry-point into all other system services. There are dozens of system services, but the patterns used repeat. So, four interesting services are covered here.

3.1 LocationManager: location updates via binder callback and PendingIntent

It is most natural to expose system services that are standalone and isolated in a sense. A good example is the LocationManager which “allows applications to obtain periodic updates of the device’s geographical location, or to fire an application-specific Intent when the device enters the proximity of a given geographical location. [8]” The location manager operates as a standalone source of location updates. Here are 2 methods of the LocationManager that we look into further.

```
interface ILocationManager {
  void requestLocationUpdates(
      in LocationRequest request,
      in ILocationListener listener,
      in PendingIntent intent,
      String packageName);

  void removeUpdates(
      in ILocationListener listener,
      in PendingIntent intent,
      String packageName);
  // ... among other methods
}
```

Snippet 8

```
oneway interface ILocationListener {
    void onLocationChanged(in Location location);
    // ... among other callback methods
}
```

Snippet 9

The main requestLocationUpdates() method requires the client to provide either a ILocationListener binder callback or PendingIntent callback. If the former callback is provided, LocationManager holds a BinderProxy initialized with the remote handle to the local ILocationListener. The .onLocationChanged() RPC is invoked periodically with location updates, provided that caller application keeps a the callback object alive (i.e. a keep around a strong reference) and that the caller process remain running.

The latter PendingIntent callback is also implemented via binder. However the callback object is managed by the Android Runtime. The caller first obtains a PendingIntent via RPC to the central ActivityManager. A PendingIntent is constructed with the address of a recipient component to which messages are delivered. Used as a callback, the PendingIntent is usually constructed with the address of a component in the caller’s namespace (or the caller itself). With the PendingIntent provided as argument to requestLocationUpdates(), the LocationManager later can invoke PendingIntent.send() with location updates as contained data. This RPC goes to the Android Runtime (the owner of the PendingIntent), which then via another RPC routes the message to the recipient component. Though the PendingIntent mechanism is inherently more complex, the advantage is that the Android Runtime takes care to brings up the recipient component in case it is not already up. This way an app can register a PendingIntent callback and not have the burden of being active to receive callbacks.

```
interface IIntentSender {
  int send(
    /* The message */
    int code, in Intent intent,
    /* Address of recipient */
    String resolvedType,
    /* Send finished callback */
    IIntentReceiver finishedReceiver,
    ...);
}
```

Snippet 10

```
oneway interface IIntentReceiver {
  void performReceive(
      in Intent intent, int resultCode,
      ...);
}
```

Snippet 11

PendingIntent.send() is implemented in non-blocking fashion. Internally a PendingIntent is an IIntentSender binder object which is owned by (and thus invoked RPCs runs within) the ActivityManager. An additional IIntentReceiver callback can optionally be passed in with each invocation of .send()–a callback which is invoked once the send operation is complete. The LocationManager does make use this additional callback. The LocationManager holds what's known as a WakeLock–an Android Runtime feature used to prevent the device from entering sleep mode. Without the WakeLock, messages would be stuck in-flight and delivered only when the device wakes up from sleep. The WakeLock is acquired while the GPS is active, and is held until all PendingIntent messages are received by recipients. This prevents messages from being stuck in-flight.

Both direct binder object callbacks and PendingIntent callbacks should be supported by Sigma. PendingIntent callbacks are of particular interest since they can be used in the design of "push" services [12]. A client can register a PendingIntent with a service provided by a remote device. Subsequently the local app can receive messages without the need to be always active, as the local Android Runtime will bring up the app. Of course, the Sigma network stack in charge of receiving remote binder messages and the service on the remote device itself have to be active for this to work.

3.2 SensorManager: sensor events via Unix domain sockets

The SensorManager is another good candidate to allow access from remote devices. It is a standalone source of sensor data streams. Recently, sensors services have evolved to to provide the user's activity state–e.g. categories like walking or driving. We image the future of sensor services will include an variety of contextual inferences.

Also interesting is SensorEvents are sent via a BitTube–an OO abstraction on top of Unix domain sockets. This system service uses binder to setup a socket connection between two processes, and subsequently pushes sensor events over the socket. If we are to expose SensorManager over the network, we need to the ability to send file descriptors (and importantly: to support IO operations on those file descriptors) over the network as well.

```
class ISensorServer : public IInterface {
public:
    Vector<Sensor> getSensorList() = 0;
    sp<ISensorEventConnection>
      createSensorEventConnection()
    // ... among other methods
};
```

Snippet 12

```
class ISensorEventConnection : public IInterface {
public:
    sp<BitTube> getSensorChannel()
    status_t enableDisable(int handle, bool enabled)
    status_t setEventRate(int handle, nsecs_t ns)
};
```

Snippet 13

3.3 AlarmManager: not very useful over the network

There are some services which don't make too much sense to expose. For instance, the AlarmManager is "intended for cases where you want to have your application code run at a specific time, even if your application is not currently running." There is no need to access a remote device's alarm manager as it provides no useful data or service more than the local instance of AlarmManager.

3.4 ActivityManager: difficult to expose over network without extensive refactoring

Services that are not standalone or are coupled to the local context of Android applications are difficult to directly expose over the network. This is because the operation of such services depends on records that are local to the device. For instance, ActivityManager is a crucial system service that is in charge of managing all live components running on the Android device. It provides useful operations such as bindService(). However, it is not so straightforward to invoke bindService() on a remote handle to ActivityManager since the call depends ActivityManager's records about the calling process that only the ActivityManager on the caller device knows about.

3.5 A caveat with Android Permissions

A complication, particularly when dealing with system services, is that many of them require permission to access. That is, local apps declare permission to use a service, and the PackageManager grants such permission at install time. It is not trivial to extend the Android permissions system to allow for remote binder objects. As such Sigma as it is implemented disregards Android permissions, allowing remote devices access to all services running on a device. A thorough treatment of the permissions framework and its support in Sigma is a left for future work. A discussion on this topic is found at the end of the paper in Sections 7.0.8 and 7.1.

4. SIGMA: DESIGN AND IMPLEMENTATION

The goal of Sigma is to essentially to follow the Proxy Pattern of binder services one hop out–tunneling binder transactions over the network, see figure 3. The design of Sigma should minimally modify the Android Runtime, while supporting both native and Java-based binder services. And, perhaps most important: Sigma should support binder's powerful feature to allow the passing of binder objects and file descriptors as arguments in a remote RPC call.

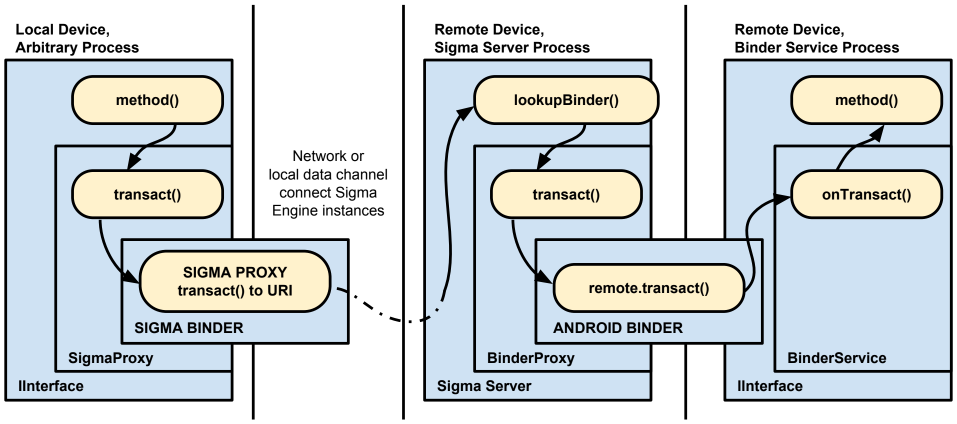


Fig. 3

4.1 Proxy Pattern over the Network

Consider a network consisting of two Android devices, and that there is a pre-established data channel between them. Each device runs a Sigma Engine instance which has server-like and client-like functions. Its function viewed as a server is to associate a URI to local binder services, and subsequently accept binder transaction requests made to a URI–performing the binder transaction on the associated local binder. Its core function as a client is to allow the retrieval of remote binder objects (retrieved as a URI) and implement the Proxy Pattern, passing on the Proxy IBinder to local services. Subsequent binder transactions invoked on the proxy object forward the transaction over the network to the associated URI.

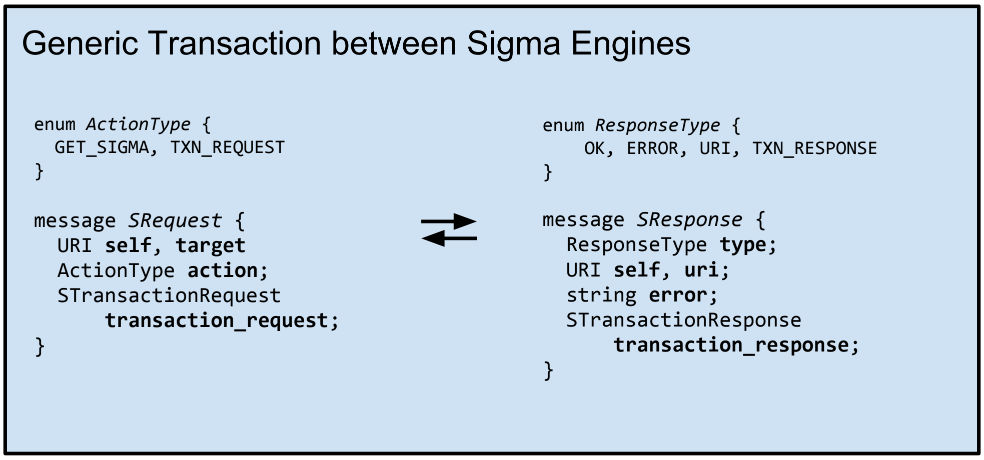
Binder services are often implemented in Java with AIDL-specified interfaces. But not always. Binder services can also be hand-made (in c, c++, or java) following any arbitrary convention for doing RPC. A binder service simply has to use the BKM in some way to qualify as one. Fortunately, all Android binder services follow the IBinder interface. So the goal of Sigma is to proxy all methods of the IBinder interface. Also, fortunately, the Java and C++ versions of IBinder specify identical interfaces that are compatible with one another. In fact the Java implementation is really a JNI-based wrapper of the C++ version. Thus, native services can be made into Java BinderProxy objects by initializing them with a native service.

Sigma Engine needs a networking stack to work with. It makes most sense to implement this part at the Java level–using the features of the Java Android Runtime (and convenience of the SDK) to full benefit. Networking libraries are readily available for Android as pure Java libraries. Moreover, this way Sigma can be packaged as any other third-party app. Distributed and installed through the app store.

Caveat As Sigma is implemented today, it requires some underlying modifications to the Android Runtime, as such it is not compatible with regular Android devices. See Appendix 8.1

4.2 Design of the Sigma Message-based Protocol

Sigma Engine follows a protocol for retrieving remote BinderProxy objects as URIs and then make binder transactions targeted at the URI. Assuming we have established a data channel between two devices, the first decision is to chose a structured format for communication over the network data channel. There is a lightweight implementation of Google's protobuf format for encoding structured data called Wire [16, 22]. It is a open source library created specifically for Android by the mobile app company Square. The protocol is an exchange of Wire messages in transations. Each transaction is a SRequest message followed by a SResponse message. Figure 4 contains the rought protobuf format implementing the generic transaction.



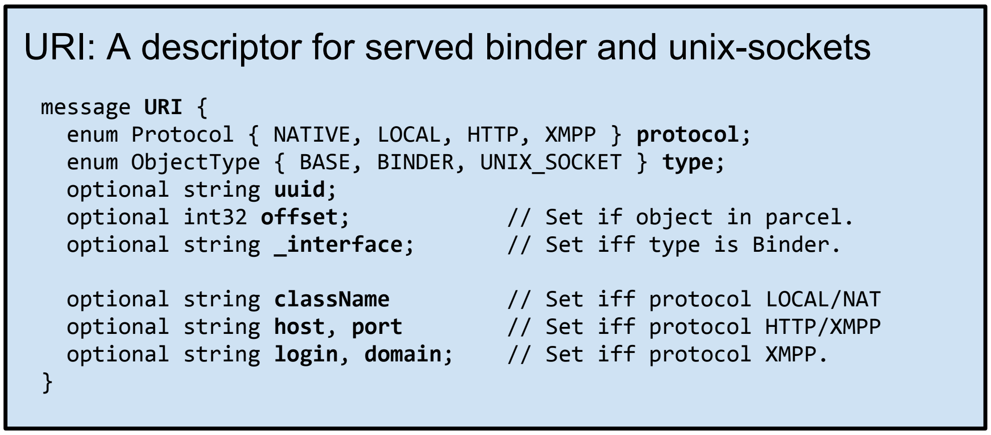
Generic Transaction between Sigma Engines

```
enum ActionType {
  GET_SIGMA, TXN_REQUEST
}

message SRequest {
  URI self, target
  ActionType action;
  STransactionRequest
      transaction_request;
}
```

```
enum ResponseType {
    OK, ERROR, URI, TXN_RESPONSE
}

message SResponse {
  ResponseType type;
  URI self, uri;
  string error;
  STransactionResponse
      transaction_response;
}
```

Fig. 4

The URI is a Wire message (see Figure 5) that contains all the necessary network information about the source or target Sigma Engine. This is known as a BASE URI. For instance, an HTTP-based Sigma Engine has a URI with the host and port fields set. A URI can also reference a remote object. A BINDER URI is a BASE URI with additional fields like the an id of the referent binder and its interface descriptor.

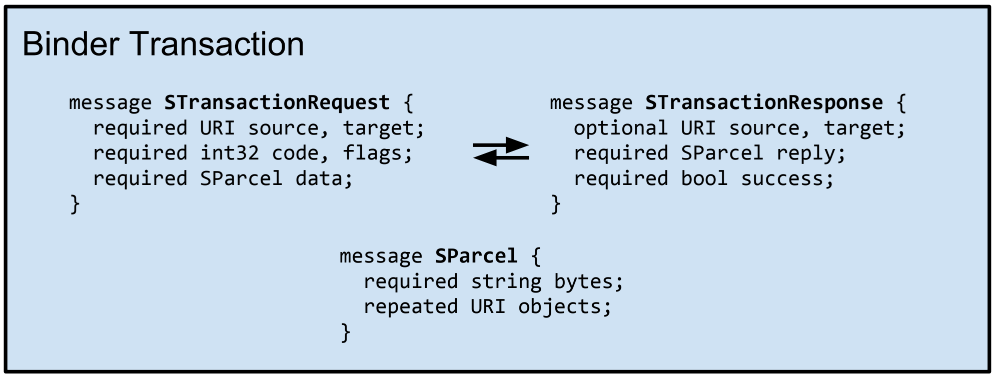


URI: A descriptor for served binder and unix-sockets

```
message URI {
  enum Protocol { NATIVE, LOCAL, HTTP, XMPP } protocol;
  enum ObjectType { BASE, BINDER, UNIX_SOCKET } type;
  optional string uuid;
  optional int32 offset;            // Set if object in parcel.
  optional string _interface;       // Set iff type is Binder.

  optional string className         // Set iff protocol LOCAL/NAT
  optional string host, port        // Set iff protocol HTTP/XMPP
  optional string login, domain;    // Set iff protocol XMPP.
}
```

Fig. 5

A remote service's Binder interface is obtained as a BinderProxy by the remote Sigma Engine and served as a BINDER URI. The local Sigma Engine creates a new Binder object, targeted at the BINDER URI, passing it on to local devices as a BinderProxy. Thus a local client and remote client will see the same interface to a remote service. A transaction invoked on the newly-created local Binder creates a SRequest message with the fields of STransactionRequest populated. The local Sigma Engine then sends the message over the network to the to the target BINDER URI. The remote Sigma Engine performs the transaction and returns a SResponse containing the reply as STransactionResponse. The transaction request and response both contain Parcel objects encoded as SParcel, where the data array is first Base64-encoded to a string. These Wire messages format are in Figure 6.



Binder Transaction

```
message STransactionRequest {
  required URI source, target;
  required int32 code, flags;
  required SParcel data;
}

message STransactionResponse {
  optional URI source, target;
  required SParcel reply;
  required bool success;
}

message SParcel {
  required string bytes;
  repeated URI objects;
}
```

Fig. 6

4.3 Tunneling Binder Descriptors

As we know, binder objects can be contained in the Parcel passed into a binder transaction. These are invariable BinderProxy objects as the actual service object being passed in is owned by a remote process and not by Sigma. In such a situation, the Sigma Engine manages the BinderProxy, assigning to it an unique ID. This ID is transformed into a BINDER URI (taking the local Sigma Engine's BASE URI, adding on the ID of the managed binder object, and adding the offset into the data array that the BinderProxy object is written into). In this way, Parcels containing binder objects are encoded as SParcel messages with containing an array of corresponding BINDER URIs in the objects field.

Conversely, a Sigma Engine can receive remote binder transaction requests with SParcels that have a non-empty array of BINDER URI objects. In this situation, a local Binder is created and managed by Sigma Engine. The local Binder is a specially implemented to forward binder transactions made on it as a SRequest message over the network to the BINDER URI target that it was initialized with.

```
def encode(p)
  encObjects = []
  for (binder, offset) in p.objects:
    uri = engine.serve(binder)
    encObjects.add((uri, offset))
  return (parcel.data, encObjects)
```

```
def decode((data, encObjects))
  parcel = <>
  parcel.data = data
    for (uri, offset) in encObjects:
    binder = engine.proxy(uri)
    parcel.objects.add(binder, offset)
  parcel.data = data
  return parcel
```

4.4 Reference counting

Another detail is reference-counting. Binder objects remain in memory unless all strong references to them are removed. Holding BinderProxy objects does not prevent the target Binder object from being garbage collected. If we hold only a weak reference to a BinderProxy object, then that BinderProxy will also be garbage collected along with the target Binder. This reference-counting is a feature of Android Binder (not the BKM).

A local Sigma Engine creates Binder objects targeted at remote URIs. These objects are not garbage collected by default. On the remote side, Sigma only associates URIs with BinderProxy objects (not Binder objects) as the real Binder object is created by a remote process outside of Sigma. By holding only weak references on the remote side, we can ensure the remote BinderProxy objects are cleaned up automatically by the garbage collector. Using this to our advantage, we also send a notification to the other side when a BinderProxy is finalized. This way, local Binder objects can be nulled out when the corresponding remote BinderProxy objects are removed. Such nulled out objects are eventually garbage collected, thus maintaining no memory leaks in the form of dangling Binder objects. See 5.3.1 for an example of Sigma's reference-counting scheme.

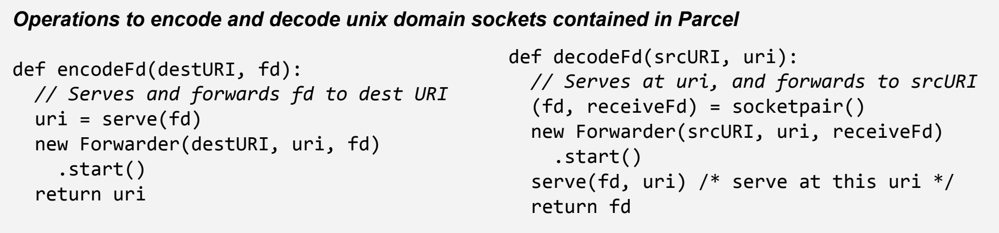
4.5 Tunneling File Descriptors over Network

File descriptors in Parcels are handled analogously to binder objects. However, in Unix-like systems, file descriptors can refer to any Unix file type named in a file system. As well as regular files, this includes directories, block and character devices (also called "special files"), Unix domain sockets, and named pipes. File descriptors can also refer to other objects that do not normally exist in the file system, such as anonymous pipes and network sockets [15].

Although it is theoretically possible to handle each type file descriptor (like the command-line utility nc that exposes each over a socket), each type has to be treated separately since each type has a specific set of supported IO operations. We only want to demonstrate as a proof-of-concept that we can handle file descriptors by focusing on a single type: the Unix domain socket. These special sockets are used to communicate sensor events in SensorManager, and used elsewhere in GUI-related Android services, so there is a valid use case. Such file descriptors are created via socketpair() with the protocol SOCK_SEQPACKET, and provide sequenced, reliable, bidirectional, connection-mode transmission paths for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record [14]. Fortunately, we need to support only a small set of operations: send() and recv(). Also the remote end needs to know when the socket is no longer valid or is otherwise closed so that resources can be cleaned up.

Unlike a binder object where requests go only from proxy to server, the socket is bidirectional. So a when Sigma Engine receives a local transaction with a unix domain socket in the Parcel, it takes ownership of the file descriptor and creates a corresponding a UNIX_DOMAIN_SOCKET URI that is encoded into the SParcel.

At the the remote end, receiving SParcels with UNIX_DOMAIN_SOCKET URIs creates a local unix domain socket which is written into the Parcel in completing the binder transaction.

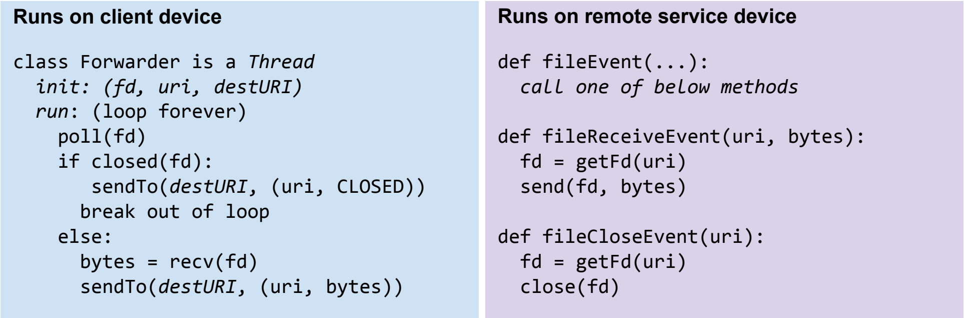


Operations to encode and decode unix domain sockets contained in Parcel

```
def encodeFd(destURI, fd):
  // Serves and forwards fd to dest URI
  uri = serve(fd)
  new Forwarder(destURI, uri, fd)
    .start()
  return uri
```

```
def decodeFd(srcURI, uri):
  // Serves at uri, and forwards to srcURI
  (fd, receiveFd) = socketpair()
  new Forwarder(srcURI, uri, receiveFd)
    .start()
  serve(fd, uri) /* serve at this uri */
  return fd
```

Creating and exchanging the URI for the file descriptor is only the first step. IO Operations need to be forwarded. A Forwarder object is initialized to poll the file descriptor and on receiving data, forwards the data over the network addressed to the appropriate UNIX_DOMAIN_SOCKET URI. On the other side, socket data received at a UNIX_DOMAIN_SOCKET URI is forwarded locally to the local managed file descriptor.



Runs on client device

```
class Forwarder is a Thread
  init: (fd, uri, destURI)
  run: (loop forever)
    poll(fd)
    if closed(fd):
       sendTo(destURI, (uri, CLOSED))
      break out of loop
    else:
      bytes = recv(fd)
      sendTo(destURI, (uri, bytes))
```

Runs on remote service device

```
def fileEvent(...):
  call one of below methods

def fileReceiveEvent(uri, bytes):
  fd = getFd(uri)
  send(fd, bytes)

def fileCloseEvent(uri):
  fd = getFd(uri)
  close(fd)
```

4.6 Choice and Implementation of a Networking Stack

Sigma is first a protocol and then an implementation. As a protocol, it is agnostic to the network connection, assuming only that a reliable underlying data channel is available for atomically sending and receiving messages. For instance, one version of Sigma is implemented locally using Android Binder itself as a data channel. This is useful for testing as we can run two instances of Sigma engine on the same device. To evaluate Sigma between Android devices, we have a gamut of networking technologies to choose from. We choose HTTP- and XMPP- based channel to test evaluate the design and measure performance.

4.6.1 *Sigma Engine over HTTP.* In this mode, each device brings up a HTTP server listening on a pre-determined port. Clients communicate to another device by performing an HTTP POST request to another device's HTTP server. The client needs to know the IP or hostname of the server and the server must be accessible over the network.

4.6.2 *Sigma Engine over XMPP.* In this mode, each device login with distinct handle to a mutually accessible XMPP server. XMPP is a popular protocol for real-time collaboration–messaging and chat. The data channel is through chat threads with other active handles. Each thread serves one transaction, similar to what is available in one HTTP POST request. Chat threads are limited to one request and to one reply. Future (or parallel) transactions occur on separate threads.

4.6.3 *Encoding Sigma (Wire) Messages, Examples.* The Sigma Engine creates Wire messages for each transaction. These can be printed to human-readable format. For instance, here is a URI to an XMPP-based SigmaEngine.

```
URI{_interface=com.example.BinderSigma.samples.chat.ICommentReceiver,
domain=quark, login=rr, protocol=XMPP, type=BINDER,
uuid=c2f980b5-ae50-4f7e-9559-438abc49508f}
```

Snippet 14

Such Wire messages are compactly serialized into byte arrays. Over HTTP, the byte array is sent directly via a POST request as a byte-array entity. Over XMPP, the byte array is first made into a Base64-encoded string and then sent as a chat message. Below is an example of a SRequest sent as an XMPP chat message (note: the SRequest is serialized then Base64-encoded)

```
<message type="chat" id="30e5a4c7-b0d7-4d11-bd2c-14abf4ff1ce0"
        to="kk@quark" from="rr@quark/Smack" >
        <body> <!-- Base-64 encoded string --> </body>
        <thread>877a1f <!-- id of transaction --> </thread>
        </message>
```

Snippet 15

The Appendix 8.7 and 8.8 contains a section with examples of actual Wire messages exchanged in implementing the example at the beginning of this sections.

4.7 Sigma as a binder service itself

The Sigma Engine is implemented as an Android service component. It is brought up with a BASE URI that serves as the Sigma Engine's identity going forward. There are separate services for each different implementation (i.e. LOCAL, HTTP, XMPP). Bringing it up any Sigma Engine service component and binding to it returns a remarkably simple binder service interface. This is the interface which can be used by app developers to get remote services as IBinder objects.

```
interface ISigmaManager {
    URI getBaseURI();
    ISigmaManager getRemoteManager(in URI targetBaseURI);
    IBinder getServiceManager();
    /* synchronous version of bindService(Intent...) */
    IBinder getService(in Intent intent);
}
```

Snippet 16

Consider an example. Assume that there are two distinct sigma engine instances, ΣA and ΣB, running on two devices, named A and B respectively. An app can access its own local Sigma Engine instance and make use of the provided ISigmaManager service. The following pseudo-code walks through the steps taken by an app running on device A to access a remote service from installed on device B. It is assumed ΣB is already up and running, listening for connections.

```
uriB = /* given the URI to ΣB is already known */
sigmaManA = connectToLocalSigmaEngine(ΣA)
sigmaManB = sigmaManA.getRemoteManager(uriB)
remoteBinder = sigmaManB.getService("name.of.remote.service");
remoteBinder.foo() /* Invoke remote binder RPC via Sigma Proxy */
```

Snippet 17

To get a sense of the interaction between the processes on the two devices, figure 7 is a useful visualization.

4.8 Accessing remote system services

All system services are published to the ServiceManager. A client app can access the remote service manager by performing the .getServiceManager() RPC on a remote ISigmaManager. However, the remote service interface is usually not the interface used by client apps. Often the remote interface is wrapped up in local client object that provides the final service to clients. To alleviate this issue, Sigma provides a convenient RemoteContext class that is constructed with reference to a remote ServiceManager. It provides a getSystemService(name) method that implements code to retrieve remote system services and wraps them up in a client object that is more usable.

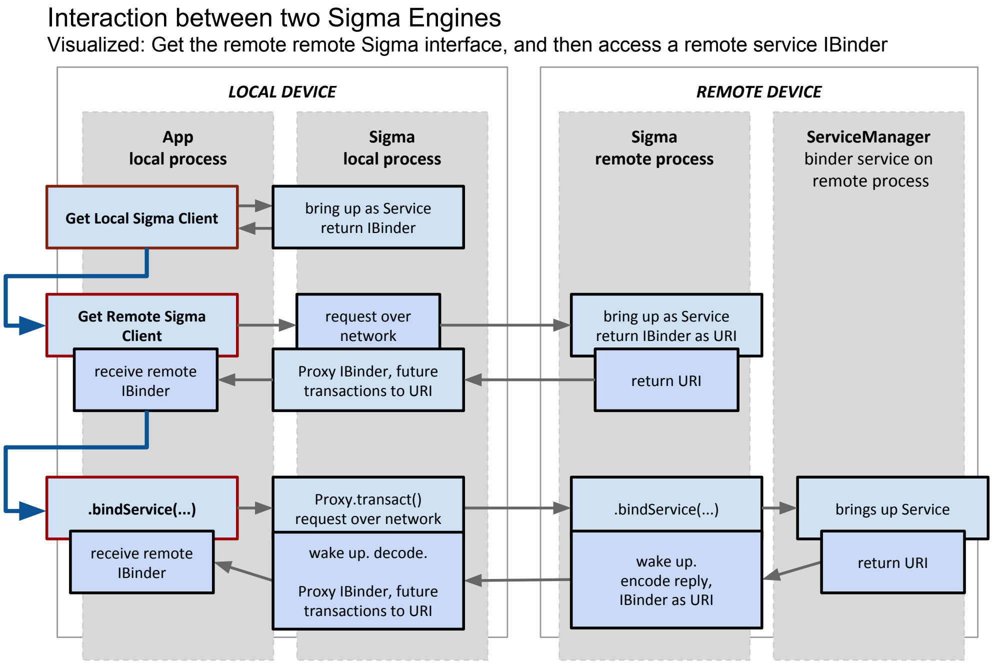

Interaction between two Sigma Engines
Visualized: Get the remote remote Sigma interface, and then access a remote service IBinder
LOCAL DEVICE
REMOTE DEVICE
App local process
Sigma local process
Sigma remote process
ServiceManager binder service on remote process
Get Local Sigma Client
bring up as Service return IBinder
Get Remote Sigma Client
request over network
bring up as Service return IBinder as URI
receive remote IBinder
Proxy IBinder, future transactions to URI
return URI
.bindService(...)
Proxy.transact() request over network
.bindService(...)
brings up Service
receive remote IBinder
wake up. decode.
Proxy IBinder, future transactions to URI
wake up. encode reply, IBinder as URI
return URI

Fig. 7

As a proof-of-concept we have implemented the retrieval of two remote services–the LocationManager and SensorManager. Many services are easily wrapped in local client objects. The local LocationManger client is straightforward to construct using reflection from the internal ILocationManager service. However, SensorService (the internal binder service providing sensor events) has a local client with parts implemented in C++ and other parts in Java (via JNI). This is not usually a issue, but the SensorManager is hard-coded to retrieve only the local SensorService! So it is impossible via reflection or other hacks to construct a SensorManager client with a remote SensorService handle. To enable this, we had to modify the Android runtime–the source code is available online, the link is found in Appendix 8.2.

5. PERFORMANCE EVALUATION

It is useful to know the CPU cost and latency to establish a proxy to a remote binder service and then the cost to make RPCs on that proxy. Overhead is expected due to the additional IPC within each device (there is communication from apps and services to the Sigma Engine). There is also the cost of encoding/decoding parcels during binder transactions, of creating requests and response messages, and naturally there is some cost to have network stack perform IO. We first describe some simple test binder services and then present the execution timing for invoking RPCs on these test services.

5.1 Testing Simple Binder Services

As a first test, consider a trivial binder service: a random number server. The interface is below.

```
class RandomServer extends IRandomServer.Stub() {
  int getRandom() { return (new Random()).nextInt(); }}
```

Snippet 18

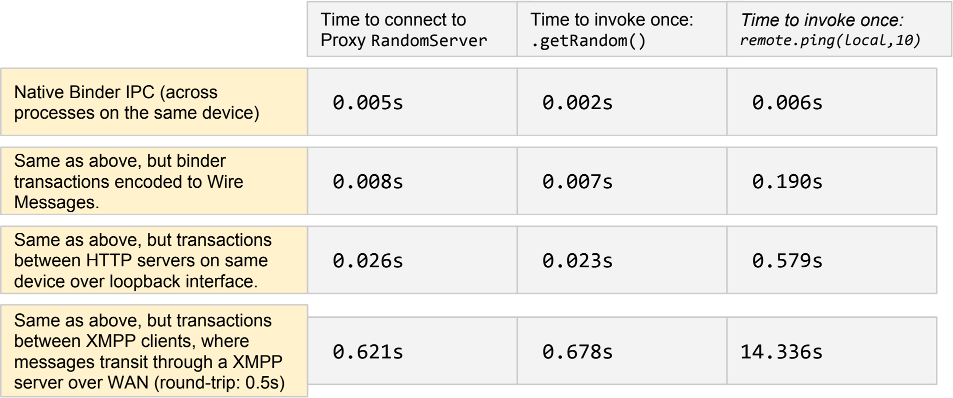
As a second test, consider a binder service that accepts binder object arguments. In particular, using this feature it is possible to specify a callback object that invokes a binder transaction back to the local device. Taking it one step further, recursion can occur back-and-forth as a series of binder calls between the two devices. The following binder service called PingPongService implements such a test. A local instance is passed in to invoke a method on the remote instance, starting in a recursive case like: remote.ping(local, 10). This causes a cascade of recursive calls, each counting down the number argument.

```
class PingPongService extends
    IPingPongService.Stub() {
  void ping(IPingPongService other, int count) {
    if (count > 0) other.pong(this, count - 1);
  }
  void pong(IPingPongService other, int count) {
    if (count > 0) other.ping(this, count - 1);
  }
}
```

Snippet 19

5.2 Performance Measurements

Here is the result of timing various RPCs, from above. These are timed for 3 prototype implementations each using a different type of data channel for communication.



	Time to connect to Proxy `RandomServer`	Time to invoke once: `.getRandom()`	*Time to invoke once:* *`remote.ping(local,10)`*
Native Binder IPC (across processes on the same device)	0.005s	0.002s	0.006s
Same as above, but binder transactions encoded to Wire Messages.	0.008s	0.007s	0.190s
Same as above, but transactions between HTTP servers on same device over loopback interface.	0.026s	0.023s	0.579s
Same as above, but transactions between XMPP clients, where messages transit through a XMPP server over WAN (round-trip: 0.5s)	0.621s	0.678s	14.336s

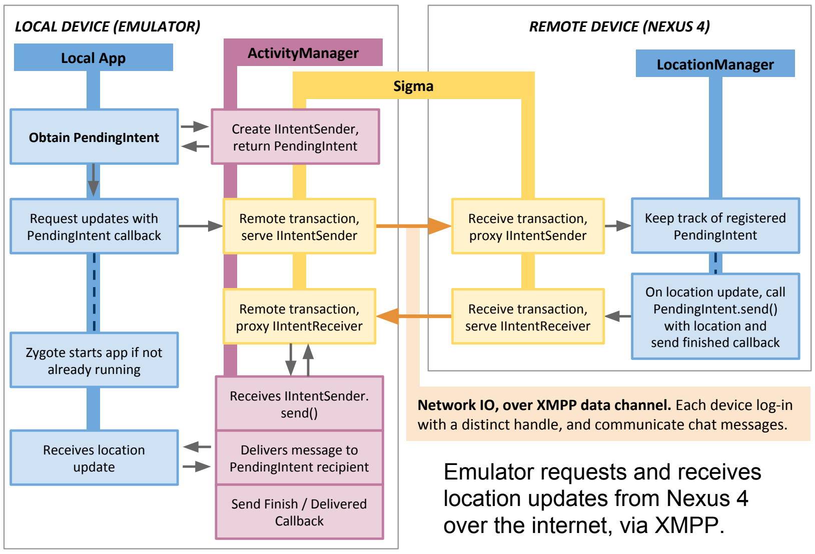
As is apparent, it is 2x-3x as expensive to perform a binder call over a proxy that encodes messages than it is to make a straight, native binder call. However, the recursive PingPong test is quite expensive since each recursive call is a new binder transaction with another round of encoding and decoding, with added network overhead.

5.3 Testing Performance with System Services

The following assumes knowledge of the respective system services, a overview of which was previously provided in section 3.

5.3.1 *Remote Location Updates.* We demonstrate that location updates can be requested from a remote device. As we have already demonstrated that binder object callbacks are supported by Sigma (see the above PingPong example), in this test we request location updates with the PendingIntent approach.

We have implemented the demonstration with location updates streamed from a real device (a Nexus 4) to an emulator running on a PC. The data channel is an XMPP-based channel, with a central XMPP server located on the internet (in Amazon EC2, with a round-trip time from the device to the cloud is 500ms). Both devices login as separate handles into this XMPP server. Below is a diagram detailing the binder objects shared with between the Nexus 4 and emulator.


LOCAL DEVICE (EMULATOR)
REMOTE DEVICE (NEXUS 4)
Local App
ActivityManager
Sigma
LocationManager
Obtain PendingIntent
Create IIntentSender, return PendingIntent
Request updates with PendingIntent callback
Remote transaction, serve IIntentSender
Receive transaction, proxy IIntentSender
Keep track of registered PendingIntent
Remote transaction, proxy IIntentReceiver
Receive transaction, serve IIntentReceiver
On location update, call PendingIntent.send() with location and send finished callback
Zygote starts app if not already running
Receives IIntentSender. send()
Receives location update
Delivers message to PendingIntent recipient
Send Finish / Delivered Callback
Network IO, over XMPP data channel. Each device log-in with a distinct handle, and communicate chat messages.

Emulator requests and receives location updates from Nexus 4 over the internet, via XMPP.

Useful metrics to measure include CPU load and memory usage. The CPU usage of Sigma actively encoding messages and sending them over XMPP is measured at a reasonable average CPU utilization of 10%–using the dumpsys command provide by Android. Memory usage is more interesting. We can track heap allocation and separately track the number of Binder objects and Proxy objects allocated by each process. Remarkably, this is a good way to test and evaluate the reference counting mechanism. Since location updates are sent with via the PendingIntent.send() method, each send operation creates a new "send finished" callback–an IIntentReceiver object. Each such object is a Binder owned by the Android Runtime, and managed and served as BinderProxy URI by the Sigma Engine on the Nexus 4 device. On the emulator, a new Local Binder is created–targeted at the URI–for each new IIntentReceiver. As IIntentReceiver Binders (and associated BinderProxies) are finalized on the Nexus 4, the emulator receives notification messages which causes the corresponding Binder objects to be nulled out. The garbage collection on emulator finalizes these over time. The allocation and deallocation of Binder objects managed by Sigma is visualized in figure 8.

5.3.2 *Remote Sensor Events.* The request to register for sensor events creates a SensorEventConnection which contains a BitTube object that is essentially an OO version of unix domain sockets. In establishing a SensorEventConnection, the file descriptor for the unix domain socket is communicated via binder. Subsequent communication of SensorEvents happens via this socket. We demonstrate that socket events are proxied and forwarded by Sigma. And plot the time difference between the native

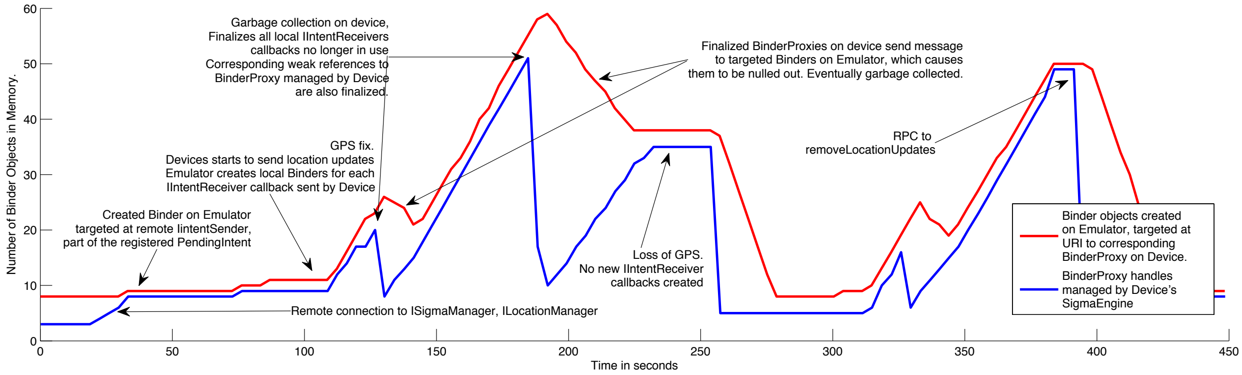
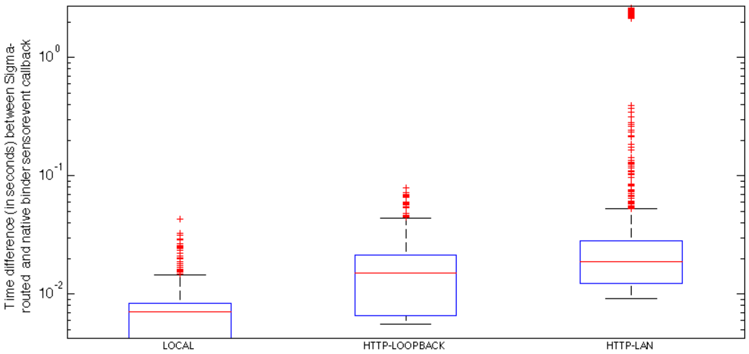


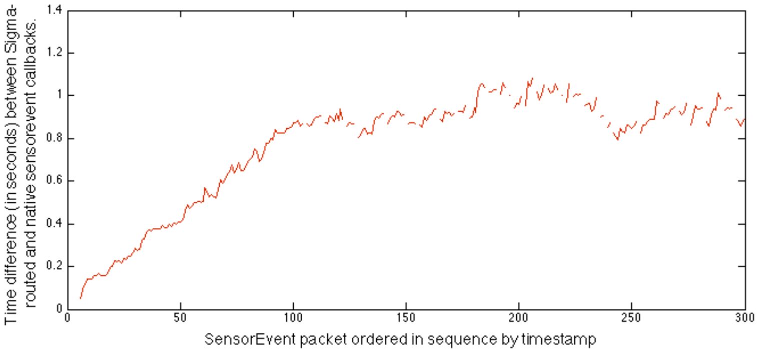
Fig. 8

SensorEventListener callback and the remotely-routed SensorEventListener callback. Of course, the data channel used will have an impact on delay, and so we plot three different versions, each with progressively more latency and jitter.

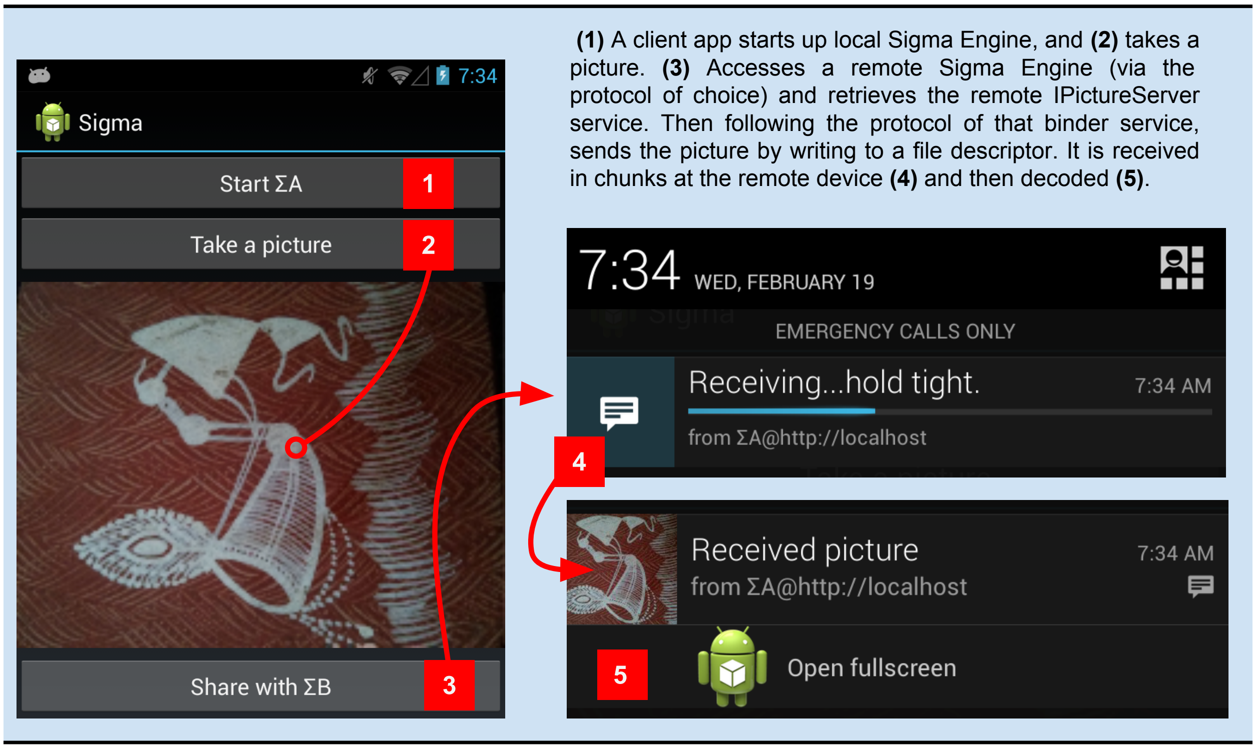
From profiling, it turns out that the implementation of the HTTP server is to blame for the dramatic increase in latency going from a local data channel implementation to one using the HTTP data channel. The local data channel uses an average of 5% CPU on a Nexus 4 device while sending sensor events, whereas the http data channel uses an average of 47% CPU doing the same. The Sigma Engine itself uses the same 10% CPU regardless of choice of data channel. The NanoHTTPD [9] server we have chosen for the HTTP implementation is not very efficient. Each sensor event receives spawns a new thread which causes dozens of context switches each second. Contributing to an increase in latency and CPU.



Because there is jitter in received sensor events, it is necessary to depend on the timestamp provided by the SensorEvent and not the timestamp of when the event is received by the listener callback. Also, depending on network conditions, it may not be possible to send sensor events at the full rate. Below we have simulated a limited bandwidth situation by routing HTTP packets through a rate limited proxy. We see that as packets start to get queued, the latency increases to a point where packets are dropped or never make it back to the device.



6. PICTURE SHARING: AN EXAMPLE DISTRIBUTED APPLICATIONS PURELY IN ANDROID BINDER



(1) A client app starts up local Sigma Engine, and **(2)** takes a picture. **(3)** Accesses a remote Sigma Engine (via the protocol of choice) and retrieves the remote IPictureServer service. Then following the protocol of that binder service, sends the picture by writing to a file descriptor. It is received in chunks at the remote device **(4)** and then decoded **(5)**.

Receiving...hold tight. 7:34 AM
from ΣA@http://localhost

Received picture 7:34 AM
from ΣA@http://localhost

Fig. 9

```
interface IPictureChatServer {
  // Returns a fileDescriptor to which caller writes picture data.
  // Server polls the fileDescriptor to locally receives picture.
  // Sigma takes care of proxying file descriptor over network.
  // Also returns (modifies) PictureEntry with metadata as a new id for picture
  // used for subsequent request to the server.
  ParcelFileDescriptor /* readFrom */ requestPicturePut(
    ISigmaManager caller, inout PictureEntry entry);

  // Client passes in fileDescriptor to which server will write picture data.
  PictureEntry requestPictureGet(
    ISigmaManager caller, String uuid, in ParcelFileDescriptor writeTo);
}

class PictureEntry implements Parcelable {
    String uuid, from;
    int numBytes;
}
```

Snippet 20

The design involves a single Binder service that acts as a picture server, this service can run on each device. Clients connects to a remote picture server to upload (or share) a picture. The service interface could have been one that receives the picture as single large byte array. However, to make the implementation more interesting as a test, the PictureChatServer operates on put- and get- requests which operate on returned or passed-in file descriptors, respectively. Picture data is communicated from one end to the other by performing IO on these file descriptors

The PictureChatServer already (via native binder) allows remote processes to connect to it to put or get pictures. With Sigma an app on a remote device can access the same PictureChatServer service. Thus, a picture sharing app can be implemented simply by writing a regular Android binder service and add make it into a distributed app with Sigma.

By requiring caller Sigma Engine to be passed itself in as an argument (i.e. the "ISigmaManager caller" argument), the PictureChatServer can invoke RPCs to get details about the caller device and credentials. Here in this example, we use it to get the name and protocol of the remote caller. In the future, we expect this is useful to authenticate and restrict callers.

7. DISCUSSION AND NEXT STEPS

Sigma is not a finished product, nor is it the ultimate distributed communication mechanism. Rather it is a good starting point for experimentation with Binder as a distributed communication mechanism. There are still some deficiencies that need to be fixed up, and important features of Binder remain unimplemented.

7.0.3 *Sigma as it is implemented today is not compatible with standard Android.* We have implemented Sigma as an Android APK that can be installed on all devices. First issue: much of the internal Java Android Runtime is hidden in the SDK, but fortunately is accessible to applications by employing reflection. Many features of Sigma are implemented this way, but is unsupported and not a stable API. For instance, here's how we retrieve the internal ServiceManager from Java.

```
IBinder binder = sigma.getServiceManager(remoteURI);
Object serviceManager = Class.forName(”android.os.ServiceManagerNative”)
                    .getMethod(”asInterface”, IBinder.class)
                    .invoke(null, binder);
```

Snippet 21

The second and larger issue is: the OO Binder implementation encapsulates the internal structures that we need to access to correctly manage binder messages. For this we had to minimally modify the Android Runtime, and so Sigma is not compatible with the regular Android Runtime running on devices today–i.e. without resorting to extreme hacks (like poking into memory addresses [26]) to get at the same data. The necessary modifications were covered in Appendix 8.1

7.0.4 *System services not portable across Android versions.* The RPC interface for the vast majority of system services is generated by the AIDL compiler–few like ServiceManager are hand-made. The generated RPC interfaces is not set in stone, it can vary with the different versions of the AIDL compiler or modifications to the AIDL source (like when switching the order of methods). For instance, a method foo() may be assigned code=1 in one version, and code=2 in another. Naturally, when operating between devices, identical RPC interfaces are expected by both ends. Generated AIDL interfaces do not have a notion of “versions” and it is impossible to tell from the Binder level whether two implementations of an interface (that go by the same name) have compatible internally generated structure. In testing, we have used the same version of Android among devices, compiled from the same source. So this issue was circumvented.

This limitation does not apply to RPC interfaces generated by third-party developers since they can fix a generated interface, ensuring backwards compatibility with previous versions by modifying the interface subsequently by hand.

7.0.5 *Parcels not portable between architectures.* Parcels were never implemented with the idea of portability. As we know, Parcels can contain binder objects and file descriptors which are obviously non-portable. Of course, the major implementation work in Sigma involved the managing of such objects, and provide a proxy for operating on them.

There is yet another aspect of Parcels that is non-portable, and this because primitive elements are represented in a long byte array, copied byte-for-byte. This operations happens preserves endianness, and this fact is troublesome since a remote device of the the opposite endianness would incorrectly process primitive data elements when it comes time to read each element out of the parcel. Although this is a seemingly major obstacle to the use of binder as a distributed communication mechanism, in practice we see that most all Android devices are ARM-based, and ARM chips are bi-endian (there is a little switch to toggle to specify the endianness). Android, by specification, is little-endian for all such ARM-based devices [1].

There is a way to make parcels portable, but we leave that to future work. Parcels store offsets to objects contained within, but not offsets to other types of primitive data. We would first have to implement that feature (which will come with additional overhead in space requirements). To send over parcels, we would need to interpret each primitive element written into data array (is it an integer or a string of some length or something else?), marshalling each element in the Parcel separately using a canonical choice for endianness. Then on the other side, unmarshalling back into the data array would need to follow an analogous reverse procedure, decoding each element into the endianness appropriate for that system.

7.0.6 *XMPP and HTTP are not entirely satisfactory as data channels.* HTTP is only suitable over the local network since a HTTP server running on a device behind NAT is not accessible by others without additional configuration. XMPP uses a central chat server (accessible to all other devices) as a middleman. Having such a middleman incurs a performance penalty, and possible privacy concerns.

There are new technologies such as WebRTC [6] that seek to establish fully peer-to-peer connections. A signaling channel such as XMPP is used first to exchange descriptors between devices, and then a central server (typically hosted publicly by a third-party) is involved in establishing a direct peer-to-peer connection. All subsequent communication is done via this connection. This same technology is used by Google Talk, and is soon going to become a W3C standard. Implementing and evaluating binder over network using webrtc data channels is left for future work.

7.0.7 *What happens when a remote binder dies? Or if the network channel disconnects?.* Binder supports via "linkToDeath()" the registering of callback that is invoked when the process hosting a remote binder unexpectedly dies. This is fortunate because we can use the same mechanism to notify of clients of a Sigma-managed Proxy Binder when networks disconnects. A "linkToDeath" is not mandatory for clients to use. Another feature of Android Binder is that a DeadObjectException is thrown when a RPC is invoked against a dead remote binder object. However, these two feature are not currently implemented in the prototype.

A final feature supported by Java Binder is handling exceptions that happen remotely. For instance, during a binder transaction an exception can be thrown by the remote process. This exception is written into the reply Parcel instead of the expected reply. Back at the Proxy, reading an exception throws an RemoteException. Since this is mechanism implemented through Binder messages exclusively, Sigma automatically supports such exception handling.

7.0.8 *Sigma disregards Android permissions model.* Each Android app belongs to a distinct Java package–this is the permissions namespace of the app. The PackageManager keeps track of the set of permissions that are requested by the package and also holds a map of the user-id assigned to each active package. Android activities and services within a package run as one or more processes which share the same user-id.

System services typically run as the root user. When an app (the caller) uses Binder IPC to make a call to another process like a system service, the callee can look up the process-id and user-id of the caller. This facility is baked into the BKM which is the component that handles the context switch from the caller to the callee (i.e. the caller thread sleeps, and wakes up a receiving thread in the callee's process). The callee can figure out (with the help of PackageManager) whether the caller has permission to access that particular system service. The system service in this way can reject unauthorized calls. This is a runtime security feature provided by BKM.

When the caller happens to be a process on a remote device, the Android security model breaks down. As implemented now, binder services are accessed and bound to the Sigma Engine, and not the process on the local device. The latter option does not make sense anyway since the remote ActivityManager does not know about the local process. Right now, Android permissions are effectively circumvented by having the Sigma Engine declaring all permissions–i.e. it essentially runs with root permissions. Any security is implemented separately by the Sigma Engine. And the permissions declared by the local process do not come into play.

There are obvious concerns with running with full permissions. One idea would be the remote device to communicate with the PackageManager on the local device, and grant an app all the permissions it is granted on the local device. However, this solution does not seem entirely satisfactory. Imagine an application requesting permission to access location of the local (its own) device means that it can also access location on all other remote devices.

It makes more sense for an app to be given permission to use a set of local services, and then a separate set of permissions granted to use remote services. For instance, a configuration panel on the remote device, much like a firewall, can be used to allow other devices access to certain services. Whenever the PackageManager is tasked with checking permissions for a binder object owned by Sigma Engine, such requests are redirected to Sigma. The implementation of such a panel and the necessary modifications to PackageManager is left for future work.

7.1 Next Steps

In the long-term, it would be interesting to tightly couple two devices merging their Android Runtime state. Imagine that the local and remote PackageManager merge and present each device's components in their own namespace–one local, the other remote. Permissions to access services would have to be specified with the idea of a local and remote namespace as well. Consider also that the ActivityManager is merged. Sigma Engine would run within the runtime as part of the ActivityManager rather than as a separate package. With such tight coupling established, local components are able to connect to remote components directly through the ActivityManager's bindService(), the only difference is that apps specify a "remote" or "local" namespace to connect with. Tight coupling also solves some issues with notifications of binder service death. Life-cycle events of Android components on one device would automatically notify the other device, etc.

REFERENCES

1 Android native cpu abi management. http://www.kandroid.org/ndk/docs/CPU-ARCH-ABIS.html.

2 https://android.googlesource.com/kernel/goldfish/+/android-goldfish-3.4/drivers/staging/android/binder.h.

3 FindMyFriends. https://itunes.apple.com/us/app/find-my-friends/id466122094/.

4 https://www.glympse.com/".

5 http://www.forbes.com/sites/haydnshaughnessy/2014/01/15/google-and-nest-the-long-and-tetchy-view/, .

6 https://code.google.com/p/libjingle/, .

7 interdroid.net".

8 http://developer.android.com/reference/android/location/LocationManager.html.

9 NanoHttpd. https://github.com/NanoHttpd/nanohttpd.

10 http://en.wikipedia.org/wiki/OpenBinder.

11 http://en.wikipedia.org/wiki/Proxy_pattern.

12

13 Snapchat. http://en.wikipedia.org/wiki/Snapchat/.

14 http://pubs.opengroup.org/onlinepubs/009695399/functions/socket.html.

15

16 Wire: Clean, lightweight protocol buffers for android. https://github.com/square/wire.

17 http://en.wikipedia.org/wiki/X_Window_System.

18 Zuckerberg on Snapchat.

19 https://www.google.com/intl/en/chrome/devices/chromecast/.

20 http://iot.eclipse.org/.

21 M. I. Aleksandar (SaÅąa) Gargenta. Deep dive into android ipc/binder framework. http://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf.

22 S. A. Dan Rice. Introducing wire protocol buffers. http://corner.squareup.com/2013/08/introducing-wire.html.

23 S. Guha and N. Daswani. An experimental study of the skype peer-to-peer voip system. Technical report, Cornell University, 2005.

24 S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. ÏĂbox: A platform for privacy-preserving apps. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 501–514, Berkeley, CA, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/ÏĂbox-platform-privacy-preserving-apps.

25 NICOLE PERLROTH and JENNA WORTHAM. Snapchat Breach Exposes Weak Security.

26 D. Reiss. Under the hood: Dalvik patch for facebook for android. https://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920.

27 D. Ross. How chromecast works: Html5, webrtc, and the technology behind casting".

28 T. Schreiber. Android binder: Android interprocess communication. https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf.

29 A. Vance. Behind the 'internet of things' is androidâĂŤand it's everywhere. http://www.businessweek.com/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere.

8. APPENDIX

8.1 Necessary Modifications to the Android Runtime

Unfortunately, there are a few obstacles to an implementation in the default Android Runtime. The implementation of Java Binder and BinderProxy does not expose native handles to BBinder and BpBinder. In addition, there is no method to create Java Binder objects out of native handles. So if we are to implement Sigma in pure Java as it is, we have no hope of creating proxy interfaces to native binder services. An example of such a service is the SensorService. The remedy is to modify the Android Runtime. The following are signatures of added methods that do the job of converting from native binder to Java binder and vica-versa.

```
class BinderInternal {
  // ...
  native IBinder binderForNativeHandle(int handle);
  native int nativeHandleForBinder(IBinder binder);
}
```

Snippet 22

Binder transactions operate on binder_io arguments which may contain objects and file descriptors. The BKM specially rewrites these objects, and so the binder_io data structure has an offsets array containing information about where each object or file descriptor is in the data array. It follows that Sigma, in proxying binder transactions, also has to specially manage such arguments–these details we will leave for the next section. As a first requirement though, we need to access the offsets array from Java–where Sigma is implemented. Unfortunately, the Java and C++ Parcel objects which encapsulate the binder_io data structure prevents access to this very important offsets array. Without it, we cannot know where and how may objects are in contained in the Parcel. Because this is an essential requirement, and we resort to modify or add methods to the Android Runtime to allow access.

```
class Parcel {
  public static byte[] marshall();
  public static void unmarshall(byte[] data, int offset, int length);
  public static int[] getObjectPositions();
}
```

Snippet 23

The marshall() and unmarshall() methods are already present in the default Parcel implementation, but it cannot handle Parcels that contain objects elements–we have fixed it to not care about that. The getObjectPositions() method is new it simply provides access to the underlying offsets array.

```
parcel.setDataPosition(offset);
IBinder binder = parcel.readStrongBinder();
```

Snippet 24

The source code to the modified Android Runtime is available online, see Appendix 8.2. It is available as a patch for a particular version (4.2.2) of Android, but these changes should be compatible with the vast majority of Android versions with only minor modifications. The rest of the paper assumes a that we operate on the modified runtime.

8.2 Links to source code, available online

8.2.1 *Sigma Engine and patched dependencies.* The main Sigma Engine that can be compiled into an Android package (or APK) can be found online.
The root Sigma project with dependencies: `https://github.com/kastur/SigmaRoot`.
Or, without the dependencies at: `https://github.com/kastur/Sigma`.
The XMPP library is patched slightly, and is online: `https://github.com/kastur/ASmack`

The code has been tested on a modified Android Runtime with a Android base version 4.2.2_r1 on the Android Emulator and Nexus 4 device. Modifications to the Android Runtime. Of course, the Sigma Android package will not function on a regular Android device. We have modified the Android 4.2.2_r1 version source tree to support some features.

The modified runtime can be compiled and flashed on a real device or run on the emulator by following the directions at `https://source.android.com/`. The following patches have to be applied to 2 modified sub-projects. The modified source tree for 2 sub-projects is available online:

```
https://github.com/6b72/platform_frameworks_base/tree/sigma
https://github.com/6b72/platform_frameworks_native/tree/sigma
```

The same modifications can be viewed as a patch to the baseline 4.2.2_r1 version.

```
https://github.com/6b72/platform_frameworks_base/compare/
763ef60466ac752a3031719fb86b08486c9946b1...3ae311d29691b3060e67bae1c891fb8fbbc1be0f

https://github.com/6b72/platform_frameworks_native/compare/
529cb9ed9c5d62d5b270cdd650380ae116382143...994ae7fa16165b3e85553d154111df0a2f5a5af3
```

8.3 The Binder Kernel Module's binder_call and handler function

```
binder_call(binder_state *bs, binder_io *msg, binder_io *reply, void *target, int code)

bs -     each binder transaction and service gets its own state.
         Essentially contains a file descriptor to the binder kernel module.

msg -    essentially contains a data array with the following:
         * "android.os.IServiceManager" - name of the interface published in
                                          target binder service.
          * "com.example.service"       - name of the service to be  published
          * binder_service              - reference to the local binder
                                          service

reply -   contains the returned data array after the transaction. Can be null
          if there is no return value or we don't care about it.

target -  is a reference to a remote binder service already published. Usually
          references are obtained by a binder_call to the ServiceManager to retrieve
          published services. However, we can always make a binder call to the
          ServiceManager itself, which is always at the address (void*)0.

code -    is a non-specific value passed to the handler of the remote binder object.
          In this case, we pass an integer representing the action to be performed,
          where SVC_MGR_ADD_SERVICE is an predefined int.
```

Snippet 25

All operations of a binder_service are implemented inside of a handler function; the function signature resembles that of binder_call:

```
service_handler(binder_state *bs, binder_txn *txn,  binder_io *msg, binder_io *reply)
bs -      The initiating transaction.
txn -     essentially contains code (from binder_call), and the calling uid and pid.
msg -     The incoming message from binder_call.
reply -    The return value should be written out here, if any
```

Snippet 26

8.4 The flat_binder_object struct

This struct holds either a local pointer to a binder service or a kernel-module provided handle. This same struct is used for storing file descriptors as well.

```
struct flat_binder_object {
  unsigned long type, flags  /* header */
  union {
    void *binder;        /* local object */
    signed long handle;  /* remote object */
  };
  void *cookie;          /* extra data */
};
```

Snippet 27

8.5 The binder_io struct

This struct holds the data for binder transactions as byte arrays. Objects and file descriptors are flattened as flat_binder_object structs and written into the data array, with the offset for each object written stored in the offs array.

The Java and C++ binder transactions operate on Parcel objects which encapsulate a binder_io data structure.

```
struct binder_io {
  char *data;            /* start of data buffer */
  uint32_t *offs;        /* array of offsets */
  uint32_t data_avail;   /* bytes available in data buffer */
  uint32_t offs_avail;   /* entries available in offsets array */
};
```

Snippet 28

8.6 IServiceManager’s object-oriented service interface

```
class IServiceManager : public IInterface {
 public:
    // Retrieve an existing service, blocking for a few seconds if it doesn’t yet exist.
  virtual sp<IBinder> getService(const String16& name) const = 0;
    // Retrieve an existing service, non-blocking.
  virtual sp<IBinder> checkService(const String16& name) const = 0;
    // Register a service.
  virtual status_t addService(const String16& name, const sp<IBinder>& service, ..) = 0;
    // Return list of all existing services.
  virtual Vector<String16> listServices() = 0;
}
```

Snippet 29

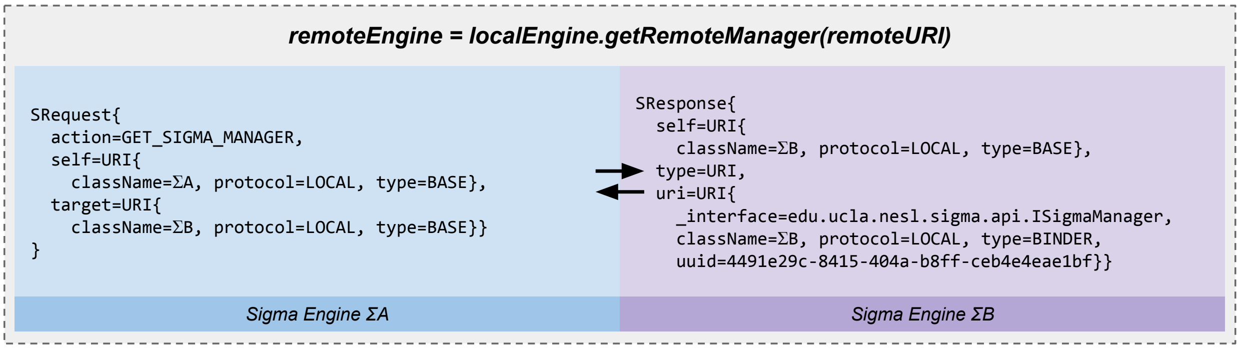
Note that what is not captured by the interface definition alone is that arbitrary process do not have permission to add new services. Inside the invocation of addService is a call to getCallingUid(), which is used to enforce permissions, limiting it so that only system processes may add services.

getCallingUid() and getCallingPid() are features of the Binder Kernel Module to get information about the caller process.

8.7 Examples of Sigma (Wire) messages

The following set of figures show in detail the Wire messages exchanged between two Sigma Engines (implemented via a LOCAL protocol, with engines running on same device and communicating to each other via native binder). The messages correspond to the example from the beginning of "Design of Sigma" section, where a remote Sigma Engine reference is first retrieved via RPC to a local Sigma Engine, and then a reference to remote service is obtained.

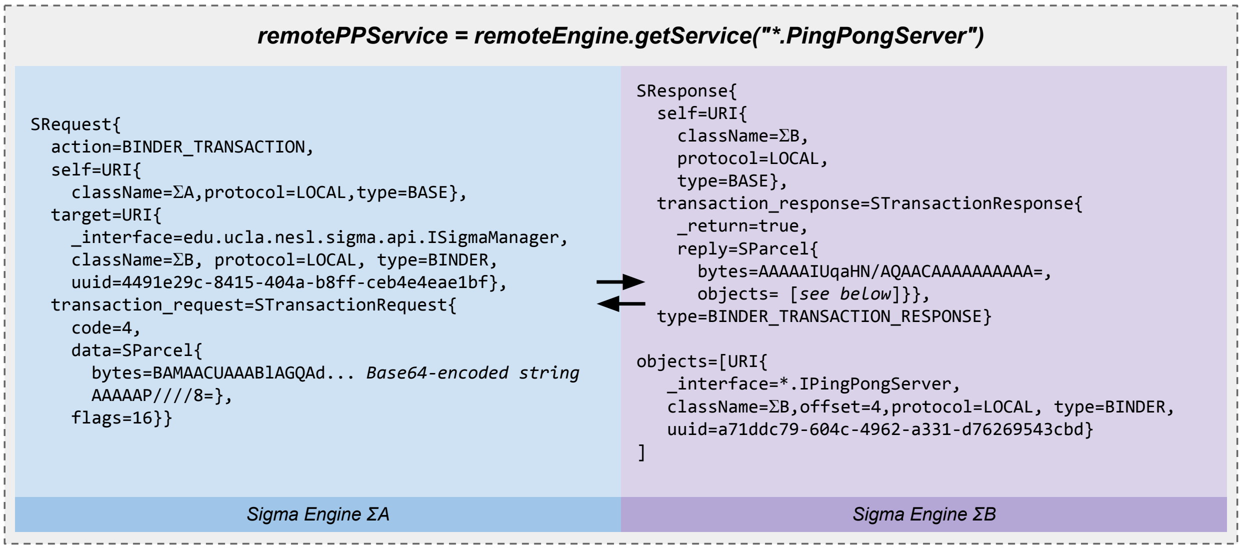
Here, since we are operating under the LOCAL protocol, all transactions proxy binder objects are transit through the local Sigma Engine and then to the remote Sigma Engine (which also runs on the same device as a different process), and then from there to the native binder service. Naturally this the transit from local to remote Sigma Engine will take over the network (through HTTP or XMPP) for those implementations.



remoteEngine = localEngine.getRemoteManager(remoteURI)

```
SRequest{
  action=GET_SIGMA_MANAGER,
  self=URI{
    className=ΣA, protocol=LOCAL, type=BASE},
  target=URI{
    className=ΣB, protocol=LOCAL, type=BASE}}
}
```

```
SResponse{
  self=URI{
    className=ΣB, protocol=LOCAL, type=BASE},
  type=URI,
  uri=URI{
    _interface=edu.ucla.nesl.sigma.api.ISigmaManager,
    className=ΣB, protocol=LOCAL, type=BINDER,
    uuid=4491e29c-8415-404a-b8ff-ceb4e4eae1bf}}
```



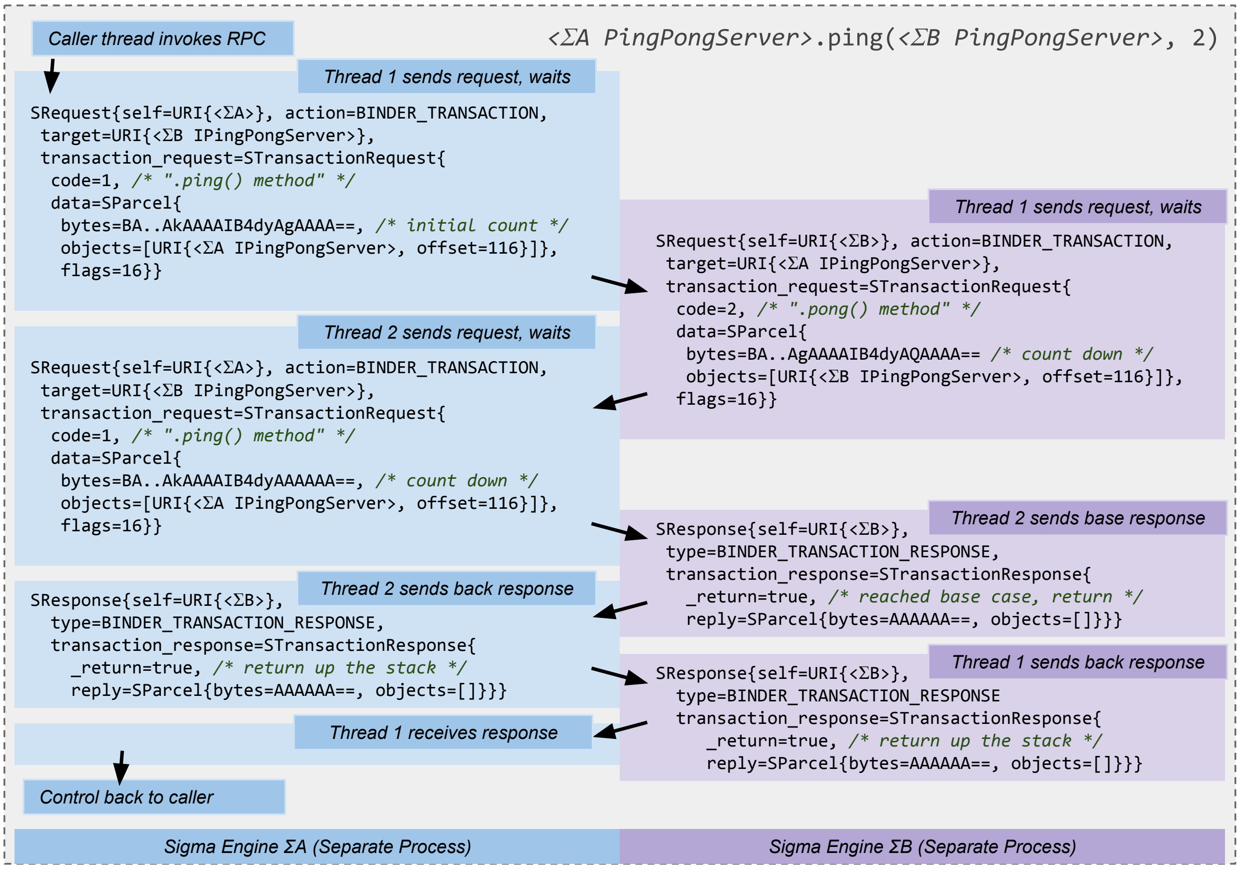
remotePPService = remoteEngine.getService("*.PingPongServer")

```
SRequest{
  action=BINDER_TRANSACTION,
  self=URI{
    className=ΣA,protocol=LOCAL,type=BASE},
  target=URI{
    _interface=edu.ucla.nesl.sigma.api.ISigmaManager,
    className=ΣB, protocol=LOCAL, type=BINDER,
    uuid=4491e29c-8415-404a-b8ff-ceb4e4eae1bf},
  transaction_request=STransactionRequest{
    code=4,
    data=SParcel{
      bytes=BAMAACUAAABlAGQAd... Base64-encoded string
      AAAAAP////8=},
    flags=16}}
```

```
SResponse{
  self=URI{
    className=ΣB,
    protocol=LOCAL,
    type=BASE},
  transaction_response=STransactionResponse{
    _return=true,
    reply=SParcel{
      bytes=AAAAAIUqaHN/AQAACAAAAAAAAAA=,
      objects= [see below]}},
  type=BINDER_TRANSACTION_RESPONSE}

objects=[URI{
   _interface=*.IPingPongServer,
   className=ΣB,offset=4,protocol=LOCAL, type=BINDER,
   uuid=a71ddc79-604c-4962-a331-d76269543cbd}
]
```

8.8 Wire messages exchanged during a recursive binder RPC

An exchange of Wire messages detailing a recursive binder that take place between two Sigma Engine instances from the example of the PingPong service.


Caller thread invokes RPC
Thread 1 sends request, waits
Thread 1 sends request, waits
Thread 2 sends request, waits
Thread 2 sends base response
Thread 2 sends back response
Thread 1 sends back response
Thread 1 receives response
Control back to caller
Sigma Engine ΣA (Separate Process)
Sigma Engine ΣB (Separate Process)

```
<ΣA PingPongServer>.ping(<ΣB PingPongServer>, 2)
```

```
SRequest{self=URI{<ΣA>}, action=BINDER_TRANSACTION,
 target=URI{<ΣB IPingPongServer>},
 transaction_request=STransactionRequest{
  code=1, /* ".ping() method" */
  data=SParcel{
   bytes=BA..AkAAAAIB4dyAgAAAA==, /* initial count */
   objects=[URI{<ΣA IPingPongServer>, offset=116}]},
   flags=16}}
```

```
SRequest{self=URI{<ΣB>}, action=BINDER_TRANSACTION,
 target=URI{<ΣA IPingPongServer>},
 transaction_request=STransactionRequest{
  code=2, /* ".pong() method" */
  data=SParcel{
   bytes=BA..AgAAAAIB4dyAQAAAA== /* count down */
   objects=[URI{<ΣB IPingPongServer>, offset=116}]},
  flags=16}}
```

```
SRequest{self=URI{<ΣA>}, action=BINDER_TRANSACTION,
 target=URI{<ΣB IPingPongServer>},
 transaction_request=STransactionRequest{
  code=1, /* ".ping() method" */
  data=SParcel{
   bytes=BA..AkAAAAIB4dyAAAAAA==, /* count down */
   objects=[URI{<ΣA IPingPongServer>, offset=116}]},
   flags=16}}
```

```
SResponse{self=URI{<ΣB>},
 type=BINDER_TRANSACTION_RESPONSE,
 transaction_response=STransactionResponse{
   _return=true, /* reached base case, return */
   reply=SParcel{bytes=AAAAAA==, objects=[]}}}
```

```
SResponse{self=URI{<ΣB>},
  type=BINDER_TRANSACTION_RESPONSE,
  transaction_response=STransactionResponse{
    _return=true, /* return up the stack */
    reply=SParcel{bytes=AAAAAA==, objects=[]}}}
```

```
SResponse{self=URI{<ΣB>},
  type=BINDER_TRANSACTION_RESPONSE
  transaction_response=STransactionResponse{
     _return=true, /* return up the stack */
     reply=SParcel{bytes=AAAAAA==, objects=[]}}}
```