INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

OPERATING SYSTEMS COURSE PROJECT

Anbox source code analysis Report

Nischith Shadagopan M N - CS18B102
Arihant Samar - CS18B052

December 9, 2020

Contents

Introduction **2**

Features used by Anbox **3**

Code entry **4**

Input Output **7**

Inter process communication **13**

Graphics processing **22**

Conclusion and Further Work **24**

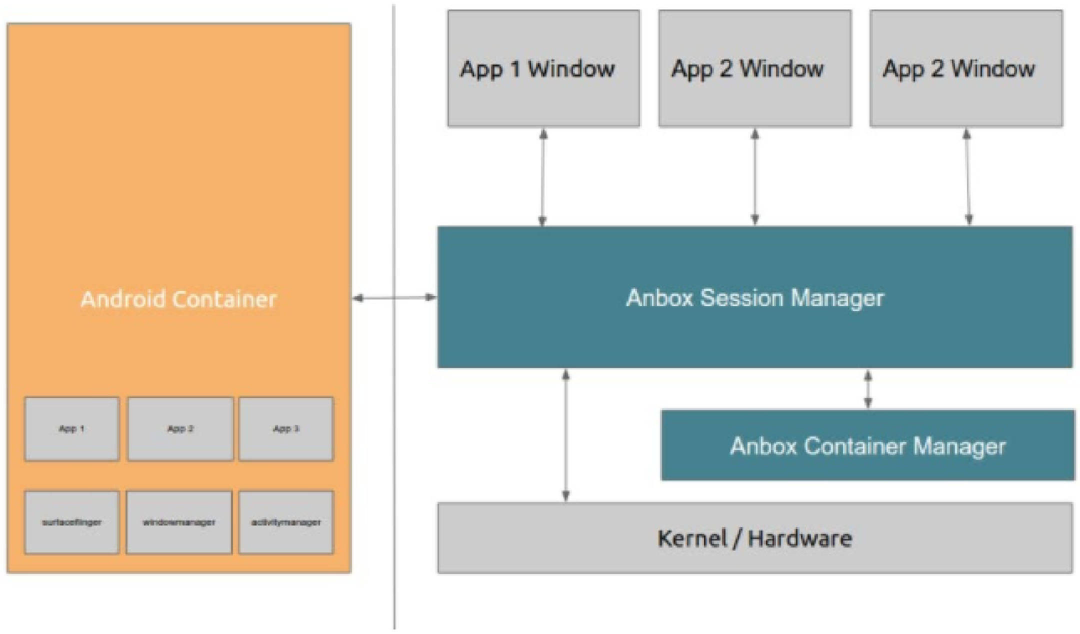
Introduction

Anbox is a free and open-source compatibility layer that aims to allow mobile applications and mobile games developed for Android to run on GNU/Linux distributions.

It executes the Android runtime environment by using LXC (Linux Containers), recreating the directory structure of Android as a mountable loop image, whilst using the native Linux kernel to execute applications. It makes use of Linux namespaces through LXC for isolation. Applications do not have any direct hardware access, all accesses are sent through the Anbox daemon.

The main purpose of Anbox is to be able to provide users Android applications onto the Linux platform in a lightweight manner. Anbox forwards hardware connection from Android container to QEMU pipe through GLES emulation. Although all of the mentioned works are related to Android container, their designs are not suitable for becoming Android sandboxes. Cells, Condroid, and Ubuntu Touch focused more on building an enhancing environment for a mobile user. Anbox aimed at building an Android app supported-environment, with support by some of the QEMU components.

Figure 1: Anbox architecture



Features used by Anbox

Shared Pointers

Dynamic memory is often problematic because we may forget to free the memory—leading to a memory leak—or we free the memory when there are still pointers referring to that memory leading to dangling pointers.To make using dynamic memory easier (and safer), C++ provides smart pointers. A smart pointer acts like a regular pointer with the important exception that it automatically deletes the object to which it points. One of the types of smart pointers are shared pointers which allows multiple pointers to refer to the same object.
The object is destroyed and its memory deallocated when either the last remaining shared_ptr pointing to the object is destroyed or assigned to another object.

This is implemented using reference counts to the object, such that when the reference count is greater than 1 it just gets decremented when a pointer is destroyed and only when there is a single reference count to the object, the memory gets freed much like the reference counts we used for implementing the Copy on Write fork.

LXC

LXC is an operating-system-level virtualization method for running multiple isolated Linux systems on a control host using a single Linux kernel.
LXC's containers provide an environment as similar as possible to a VM but without the extra computation of running a separate kernel and simulating all the hardware.
This is achieved through a combination of kernel security features such as namespaces, mandatory access control and control groups.

RPC

A remote procedure call (RPC) is when a program causes a code to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. This is a form of client–server interaction (caller is client, executor is server), typically implemented via a request–response message-passing system.Remote calls are usually orders of magnitude slower and less reliable than local calls, so distinguishing them is important.

Unix Domain Socket

A Unix domain socket or IPC socket (inter-process communication socket) is a data communications endpoint for exchanging data between processes executing on the same host operating system. The Unix domain socket facility is a standard component of POSIX operating systems.
All communication between the sockets occurs entirely within the operating system kernel. Unix domain sockets may use the file system as their address name space. (Some operating systems, like Linux (/dev/poll), offer additional namespaces.) Processes reference Unix domain sockets as file system inodes, so two processes can communicate by opening the same socket.

DBUS

D-Bus standing for Desktop Bus is an inter-process communication (IPC) that allows communication between multiple applications running concurrently.It is a message bus for processes to talk to one another.
D-Bus helps coordinate process lifecycle; it makes it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed.

Code entry

Anbox runtime is mainly composed of two separate instances, the container manager and the session manager.

```cpp
int main(int argc, char **argv) {
  anbox::Daemon daemon;
  return daemon.Run(anbox::utils::collect_arguments(argc, argv));
}
```

anbox/src/main.cpp

The execution of Anbox starts here. A daemon object is created and all the command line arguments are given to it to transfer the control to the `Run()` function.

`anbox::utils::collect_arguments()` converts the C-style command-line parameters of array of character pointers into a vector of type `std::string`.

Daemon class has only one object cmd of class `cli::CommandWithSubcommands`. This is used to organize all the commands supported by Anbox. We will see more about class `cli::CommandWithSubcommands` later.

```cpp
int Daemon::Run(const std::vector<std::string> &arguments) try {
  auto argv = arguments;
  if (arguments.size() == 0) argv = {"run"};
  return cmd.run({std::cin, std::cout, argv});
} catch (std::exception &err) {
  ERROR("%s", err.what());

  return EXIT_FAILURE;
}
}
```

anbox/src/anbox/daemon.cpp

In the `Run()` function, we simply call the run function giving it the appropriate context which consists of the input stream , the output stream and the arguments passed to Anbox.

```cpp
private:
  Name name_;
  Usage usage_;
  Description description_;
  bool hidden_;
```

anbox/src/anbox/cli.h

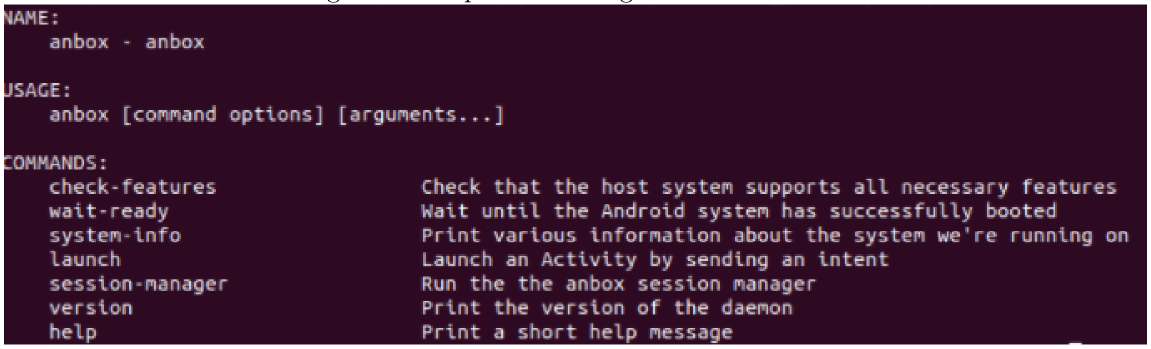
`Name, Usage and Description` are encapsulations of strings with limited length. `anbox::cli::Command` class also has a constructor for initialization and some getter and setter functions. It also has some virtual functions which are implemented by the specific command.

```cpp
/// @brief CommandWithSubcommands implements Command, selecting one of a set of
/// actions.
class CommandWithSubcommands : public Command {
 public:
  typedef std::shared_ptr<CommandWithSubcommands> Ptr;
  typedef std::function<int(const Context&)> Action;

  /// @brief CommandWithSubcommands initializes a new instance with the given
  /// name, usage and description.
  CommandWithSubcommands(const Name& name, const Usage& usage,
                         const Description& description);

  /// @brief command adds the given command to the set of known commands.
  CommandWithSubcommands& command(const Command::Ptr& command);

  /// @brief flag adds the given flag to the set of known flags.
  CommandWithSubcommands& flag(const Flag::Ptr& flag);

  // From Command
  int run(const Context& context) override;
  void help(std::ostream& out) override;

 private:
  std::unordered_map<std::string, Command::Ptr> commands_;
  std::set<Flag::Ptr> flags_;
};
```

anbox/src/anbox/cli.h

anbox::cli::CommandWithSubcommands has a function command() which is mainly used to add Command element and a flag() function which is used to add Flag element. help() function is used to output help information as shown below.

Figure 2: Output of running anbox in terminal



```
NAME:
    anbox - anbox

USAGE:
    anbox [command options] [arguments...]

COMMANDS:
    check-features                Check that the host system supports all necessary features
    wait-ready                    Wait until the Android system has successfully booted
    system-info                   Print various information about the system we're running on
    launch                        Launch an Activity by sending an intent
    session-manager               Run the the anbox session manager
    version                       Print the version of the daemon
    help                          Print a short help message
```

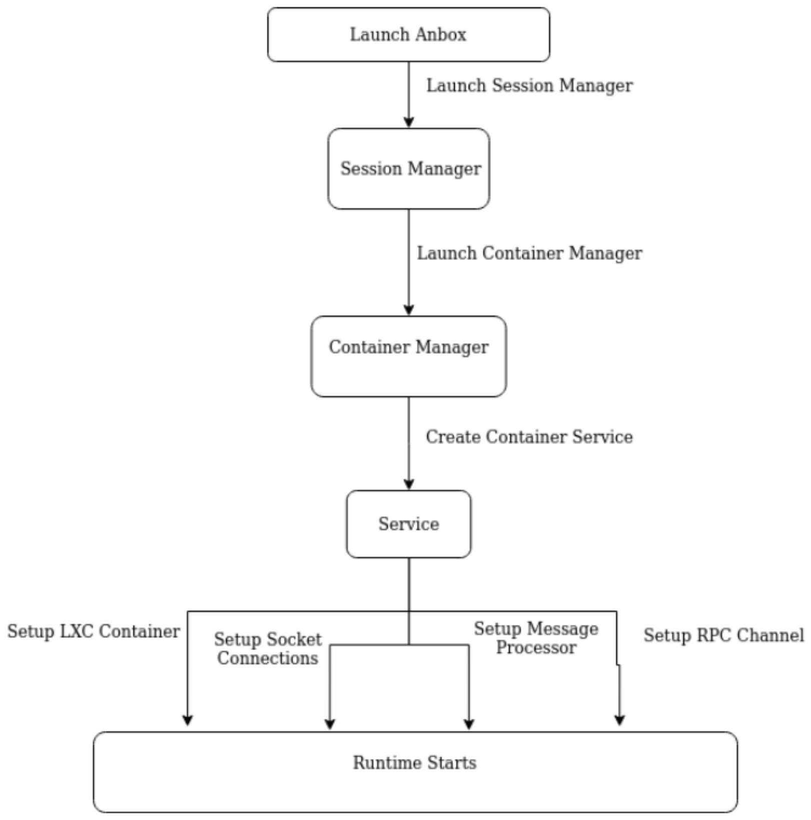
The run() function parses the command line parameters and selects the appropriate command to execute with the help of commands_ which is just a map from the string name of the command to the object of the class corresponding to the command.

```
/// @brief CommandWithFlagsAction implements Command, executing an Action after
/// handling
class CommandWithFlagsAndAction : public Command {
 public:
  typedef std::shared_ptr<CommandWithFlagsAndAction> Ptr;
  typedef std::function<int(const Context&)> Action;

  /// @brief CommandWithFlagsAndAction initializes a new instance with the given
  /// name, usage and description. Optionally the command can be marked as hidden.
  CommandWithFlagsAndAction(const Name& name, const Usage& usage,
                            const Description& description, bool hidden = false);

  /// @brief flag adds the given flag to the set of known flags.
  CommandWithFlagsAndAction& flag(const Flag::Ptr& flag);

  /// @brief action installs the given action.
  CommandWithFlagsAndAction& action(const Action& action);

  // From Command
  int run(const Context& context) override;
  void help(std::ostream& out) override;

 private:
  std::set<Flag::Ptr> flags_;
  Action action_;
};
```

anbox/src/anbox/cli.h

Here override is used to override the inherited definition of the function run() and help(). We use the run() function here to call the action function with the corresponding context. action is a function pointer that corresponds to the different actions performed by different commands of Anbox such as launch, container manager etc.

Figure 3: Anbox Runtime Setup



Input Output

The Anbox runtime consists largely of two distinct instances, namely the container manager and the session manager. But the most important thing is to handle network I/O cases, whether it's the ContainerManager or the SessionManager. The I/O model refers to the overall system design for handling I/O events for an application. For Anbox, it is mainly the overall framework design for handling various network I/O transactions in ContainerManager and SessionManager.

Anbox's I/O model is based on Boost.asio. Boost.Asio is a cross-platform C++ library for network and low-level I/O programming that provides a consistent asynchronous model using a modern C++ approach.

```cpp
class Runtime : public DoNotCopyOrMove,
                public std::enable_shared_from_this<Runtime> {
 public:
  // Our default concurrency setup.
  static constexpr const std::uint32_t worker_threads = 8;

  // create returns a Runtime instance with pool_size worker threads
  // executing the underlying service.
  static std::shared_ptr<Runtime> create(
      std::uint32_t pool_size = worker_threads);

  // Tears down the runtime, stopping all worker threads.
  ~Runtime() noexcept(true);

  // start executes the underlying io_service on a thread pool with
  // the size configured at creation time.
  void start();

  // stop cleanly shuts down a Runtime instance.
  void stop();

  // to_dispatcher_functional returns a function for integration
  // with components that expect a dispatcher for operation.
  std::function<void(std::function<void()>)> to_dispatcher_functional();

  // service returns the underlying boost::asio::io_service that is executed
  // by the Runtime.
  boost::asio::io_service& service();

 private:
  // Runtime constructs a new instance, firing up pool_size
  // worker threads.
  Runtime(std::uint32_t pool_size);

  std::uint32_t pool_size_;
  boost::asio::io_service service_;
  boost::asio::io_service::strand strand_;
  boost::asio::io_service::work keep_alive_;
  std::vector<std::thread> workers_;
};
```

anbox/src/anbox/runtime.h

This is the `Runtime Class` definition which uses the `Boost.asio.io_service` and manages the worker threads in the pool for performing various tasks.
`worker_threads` decides the number of threads to be used.
`io_service` is the facilitator for operating on asynchronous functions.It helps in running a group of processes and each process runs only when the thread calls the run method.
`strand` helps in serialising the operations for the `io_service` if required.

```cpp
void exception_safe_run(boost::asio::io_service& service) {
  while (true) {
    try {
      service.run();
      break;
    } catch (const std::exception& e) {
      ERROR("%s", e.what());
    } catch (...) {
      ERROR("Unknown exception caught while executing boost::asio::io_service");
    }
  }
}
```

anbox/src/anbox/runtime.cpp

`exception_safe_run` runs an infinite loop calling `service.run()` in each iteration. Only when the service calls `stop`, `run` returns cleanly.

`start()` creates and runs the multiple threads to execute `the exception_safe_run` function with the `io_service` variable `service_` as the argument to the `exception_service_run()`

`stop()` closes the service and waits for all the threads to finish executing that are running.

Anbox needs to deal with the following network I/O processes:

- The Session Manager communicates with the ContainerManager and with the applications in the Android container started by the ContainerManager through the Unix domain Socket. The session manager requests the ContainerManager to start the Android container, and maps these Unix domain Sockets. After the Android container is started, some processes connect to these Unix domain Sockets, and communicate with the SessionManager through these Unix domain Sockets, and then operate the host's hardware devices.

- Handle the Unix domain Socket received from the monitored Unix domain Socket.It needs to submit the newly created Unix domain Socket to the underlying I/O multiplexer, and provide the Socket with read and write I/O event processing callbacks to complete Anbox Application logic.

- Anbox's SessionManager serves as a communication bridge between Android running in the container and ADB. While communicating with Android running in the container through Unix domain Socket, it also needs to communicate with ADB on the host. SessionManager communicates with the ADB daemon on the host through TCP.

- For processing the TCP Socket,like the Unix Domain Socket,it needs to submit the newly created TCP Socket to the underlying I/O multiplexer, and provide the Socket with I/O event processing callbacks such as reading and writing to complete the Anbox application logic.

```
namespace anbox {
namespace network {
class PublishedSocketConnector : public DoNotCopyOrMove, public Connector {
 public:
  explicit PublishedSocketConnector(
      const std::string& socket_file, const std::shared_ptr<Runtime>& rt,
      const std::shared_ptr<ConnectionCreator<
          boost::asio::local::stream_protocol>>& connection_creator);
  ~PublishedSocketConnector() noexcept;

  std::string socket_file() const { return socket_file_; }

 private:
  void start_accept();
  void on_new_connection(
      std::shared_ptr<boost::asio::local::stream_protocol::socket> const& socket,
      boost::system::error_code const& err);

  const std::string socket_file_;
  std::shared_ptr<Runtime> runtime_;
  std::shared_ptr<ConnectionCreator<boost::asio::local::stream_protocol>>
      connection_creator_;
  boost::asio::local::stream_protocol::acceptor acceptor_;
```

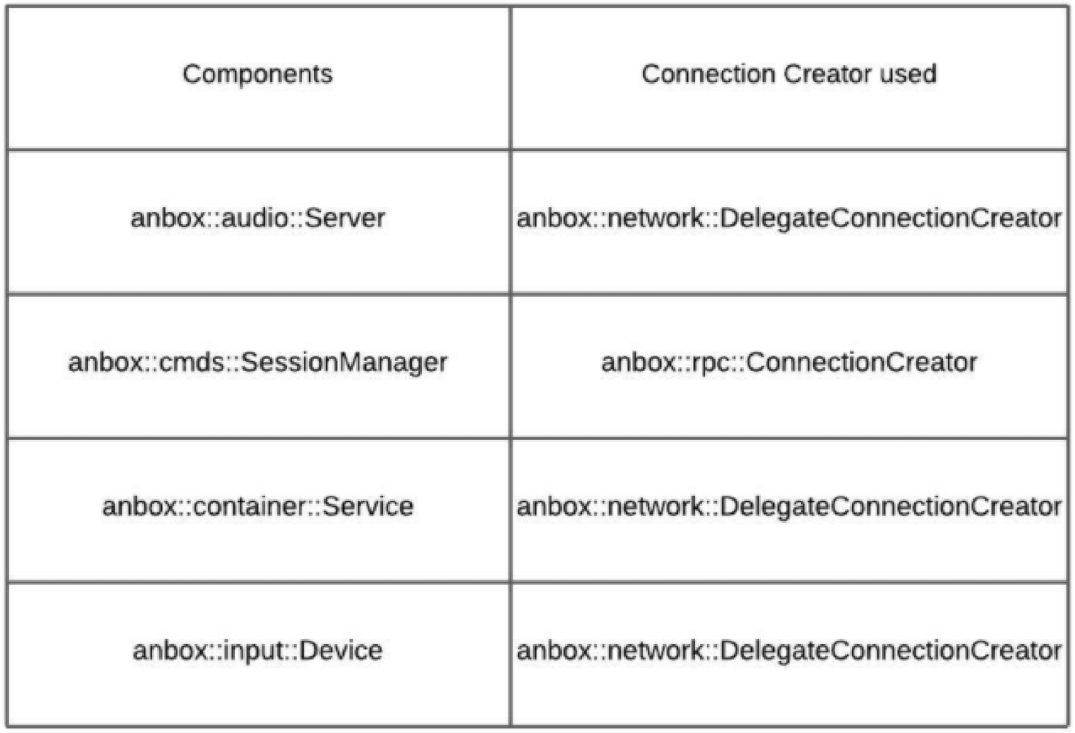
```cpp
};
}  // namespace network
}  // namespace anbox
```

anbox/src/anbox/network/published_socket_connector.h

Here `start_accept()` creates a socket and waits for a connection. `on_new_connection()` is the function which is called on receiving a connection to this socket. `on_new_connection` creates a connection and then creates a socket through a call to `start_accept()`.

Monitoring Unix Domain Socket

Figure 4: Components that monitor unix domain sockets and their corresponding connection creator



Components	Connection Creator used
anbox::audio::Server	anbox::network::DelegateConnectionCreator
anbox::cmds::SessionManager	anbox::rpc::ConnectionCreator
anbox::container::Service	anbox::network::DelegateConnectionCreator
anbox::input::Device	anbox::network::DelegateConnectionCreator

Now let's look at `anbox::rpc::ConnectionCreator`, one of the connection creators. The other connection creator is similar.

```cpp
namespace anbox {
namespace rpc {
class ConnectionCreator
    : public network::ConnectionCreator<boost::asio::local::stream_protocol> {
 public:
  typedef std::function<std::shared_ptr<network::MessageProcessor>(
      const std::shared_ptr<network::MessageSender> &)>
      MessageProcessorFactory;

  ConnectionCreator(const std::shared_ptr<Runtime> &rt,
                    const MessageProcessorFactory &factory);
  ~ConnectionCreator() noexcept;

  void create_connection_for(
      std::shared_ptr<boost::asio::basic_stream_socket<
          boost::asio::local::stream_protocol>> const &socket) override;

 private:
  int next_id();

  std::shared_ptr<Runtime> runtime_;
  std::atomic<int> next_connection_id_;
  std::shared_ptr<network::Connections<network::SocketConnection>> const
      connections_;
  MessageProcessorFactory message_processor_factory_;
};
}  // namespace rpc
}  // namespace anbox
```

anbox/src/anbox/rpc/connection_creator.h

Here create_connection_for() is used to add the newly accepted unix domain socket to the underlying I/O multiplexer. connections_ is used for message processing callback. When a message is received on the socket it is read and passed to the message processing callback function.

Now lets look at anbox::network::SocketConnection which provides the message processing callback function.

```
namespace anbox {
namespace network {
class SocketConnection {
 public:
  SocketConnection(
      std::shared_ptr<MessageReceiver> const& message_receiver,
      std::shared_ptr<MessageSender> const& message_sender, int id,
      std::shared_ptr<Connections<SocketConnection>> const& connections,
      std::shared_ptr<MessageProcessor> const& processor);

  ~SocketConnection() noexcept;

  void set_name(const std::string& name) { name_ = name; }

  int id() const { return id_; }

  void send(char const* data, size_t length);
  void read_next_message();

 private:
  void on_read_size(const boost::system::error_code& ec,
                    std::size_t bytes_read);

  std::shared_ptr<MessageReceiver> const message_receiver_;
  std::shared_ptr<MessageSender> const message_sender_;
  int id_;
  std::shared_ptr<Connections<SocketConnection>> const connections_;
  std::shared_ptr<MessageProcessor> processor_;
  std::array<std::uint8_t, 8192> buffer_;
  std::string name_;
};
}  // namespace anbox
}  // namespace network
```

anbox/src/anbox/network/socket_connection.h

set_name() is used to set the name.

connections_ is a container to store pointers to the objects of this class. This done to have some control. The objects hold a smart pointer to the container so that it can remove itself from the container when the connection is disconnected.

read_next_message submits the socket to the underlying I/O multiplexer to start receiving data.

Monitoring TCP Socket

anbox::qemu::AdbMessageProcessor monitors the tcp socket

```cpp
void AdbMessageProcessor::wait_for_host_connection() {
  if (state_ != waiting_for_guest_accept_command)
    return;

  if (!host_connector_) {
    host_connector_ = std::make_shared<network::TcpSocketConnector>(
        boost::asio::ip::address_v4::from_string(loopback_address),
        default_host_listen_port, runtime_,
        std::make_shared<
            network::DelegateConnectionCreator<boost::asio::ip::tcp>>(
            std::bind(&AdbMessageProcessor::on_host_connection, this, _1)));
  }

  try {
    auto messenger = std::make_shared<network::TcpSocketMessenger>(
        boost::asio::ip::address_v4::from_string(loopback_address),
    default_adb_client_port, runtime_);
    auto message = utils::string_format("host:emulator:%d", default_host_listen_port
    );
    auto handshake = utils::string_format("%04x%s", message.size(), message.c_str())
    ;
    messenger->send(handshake.data(), handshake.size());
  } catch (...) {
    // Server not up. No problem, it will contact us when started.
  }
}

void AdbMessageProcessor::on_host_connection(std::shared_ptr<boost::asio::
    basic_stream_socket<boost::asio::ip::tcp>> const &socket) {
  host_messenger_ = std::make_shared<network::TcpSocketMessenger>(socket);

  // set_no_delay() reduces the latency of sending data, at the cost
  // of creating more TCP packets on the connection. It's useful when
  // doing lots of small send() calls, like the ADB protocol requires.
  // And since this is on localhost, the packet increase should not be
  // noticeable.
  host_messenger_->set_no_delay();

  messenger_->send(reinterpret_cast<const char *>(ok_command.data()), ok_command.
    size());

  state_ = waiting_for_guest_start_command;
  expected_command_ = start_command;
}

void AdbMessageProcessor::read_next_host_message() {
  auto callback = std::bind(&AdbMessageProcessor::on_host_read_size, this, _1, _2);
  host_messenger_->async_receive_msg(callback, boost::asio::buffer(host_buffer_));
}

void AdbMessageProcessor::on_host_read_size(const boost::system::error_code &error,
    std::size_t bytes_read) {
  if (error) {
    // When AdbMessageProcessor is destroyed on program termination, the sockets
    // are closed and the standing operations are canceled. But, the callback is
    // still called even in that case, and the object has already been
    // deleted. We detect that condition by looking at the error code and avoid
    // touching *this in that case.
    if (error == boost::system::errc::operation_canceled)
      return;

    // For other errors, we assume the connection with the host is dropped. We
    // close the connection to the container's adbd, which will trigger the
    // deletion of this AdbMessageProcessor instance and free resources (most
    // importantly, default_host_listen_port and the lock). The standing
    // connection that adbd opened can then proceed and wait for the host to be
    // up again.
    state_ = closed_by_host;
    messenger_->close();
    return;
  }

  messenger_->send(reinterpret_cast<const char *>(host_buffer_.data()), bytes_read);
  read_next_host_message();
}
```

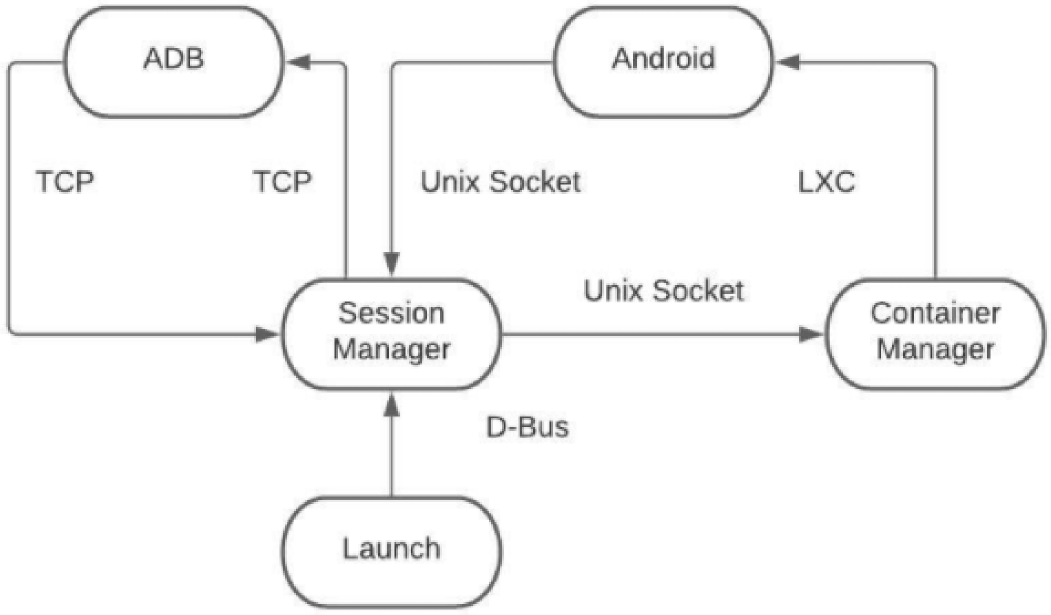
src/anbox/qemu/adb_message_processor.cpp

`anbox::qemu::AdbMessageProcessor` is inside Android in the container adbd. There is a bridge between the container and the host's ADB daemon.
`wait_for_host_connection` notifies the adb host instance with the port on which we are waiting.

`on_host_connection` dictates what happens on connecting with the host. First it creates `anbox::network::TcpSocketMessenger` which is used to submit the new Socket to the underlying I/O multiplexer, read and write data, etc. `messenger_` represents the connection with adbd in the container. It then sends an ok_command message to let adb inside the container know that we have a connection to the adb host instance. Then adbd in the container will reply with a message to start reception of messages on the new connection.

`read_next_host_message()` is used to receive data on the new connection and `on_host_read_size()` is used to forward the message to the adbd in the container.

Figure 5: Communication Architecture



- After the session manager and the container manager successfully establish a connection through the Unix domain Socket, the session manager sends a command to the container manager to request the container manager to start the Android container
- After the Android container is started, the Android process in the container establishes a connection with the session manager through the mapped Unix domain Socket
- When the Android container starts, the session manager establishes a connection with the ADB daemon
- Anbox's install and launch commands are mainly used to control the Android container. They are used to install application APKs into the Android container and start specific activities in the container. They communicate with the session manager through D-Bus.

```
namespace fs = boost::filesystem;

namespace anbox::container {
std::shared_ptr<Service> Service::create(const std::shared_ptr<Runtime> &rt, const
    Configuration &config) {
  auto sp = std::shared_ptr<Service>(new Service(rt, config));

  auto wp = std::weak_ptr<Service>(sp);
  auto delegate_connector = std::make_shared<network::DelegateConnectionCreator<
    boost::asio::local::stream_protocol>>(
      [wp](std::shared_ptr<boost::asio::local::stream_protocol::socket> const &
    socket) {
        if (auto service = wp.lock())
          service->new_client(socket);
  });

  const auto container_socket_path = SystemConfiguration::instance().
    container_socket_path();
  const auto socket_parent_path = fs::path(container_socket_path).parent_path();
  if (!fs::exists(socket_parent_path))
    fs::create_directories(socket_parent_path);

  sp->connector_ = std::make_shared<network::PublishedSocketConnector>(
    container_socket_path, rt, delegate_connector);

  // Make sure others can connect to our socket
  ::chmod(container_socket_path.c_str(), S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH);

  DEBUG("Everything setup. Waiting for incoming connections.");

  return sp;
}

Service::Service(const std::shared_ptr<Runtime> &rt, const Configuration &config)
    : dispatcher_(anbox::common::create_dispatcher_for_runtime(rt)),
      next_connection_id_(0),
      connections_(std::make_shared<network::Connections<network::SocketConnection
    >>()),
      config_(config) {
}

Service::~Service() {
  connections_->clear();
}
```

anbox/src/anbox/container/service.cpp

```
int Service::next_id() { return next_connection_id_++; }

void Service::new_client(std::shared_ptr<boost::asio::local::stream_protocol::socket
    > const
        &socket) {
  if (connections_->size() >= 1) {
    socket->close();
    return;
  }

  auto const messenger = std::make_shared<network::LocalSocketMessenger>(socket);

  DEBUG("Got connection from pid %d", messenger->creds().pid());

  auto pending_calls = std::make_shared<rpc::PendingCallCache>();
  auto rpc_channel = std::make_shared<rpc::Channel>(pending_calls, messenger);
  auto server = std::make_shared<container::ManagementApiSkeleton>(
      pending_calls, std::make_shared<LxcContainer>(privileged_, messenger->creds())
    );
  auto processor = std::make_shared<container::ManagementApiMessageProcessor>(
      messenger, pending_calls, server);

  auto const &connection = std::make_shared<network::SocketConnection>(
      messenger, messenger, next_id(), connections_, processor);
  connection->set_name("container-service");

  connections_->add(connection);
  connection->read_next_message();
}
}
```

anbox/src/anbox/container/service.cpp

The `create` function issues a new service with the current runtime and configuration and sets up a new client on the socket using the `new_client()` function.

The `client` function limits the number of connections to 1 .The message it receives is added.It set ups the messenger, pending calls , rpc_channel etc and adds the message to the map between connection id and the connection .

Anbox's container manager and session manager communicate via RPC based on Protobuf. Protobuf is a method of serializing structured data. It is useful in developing programs to communicate with each other over a wire or for storing data.

In the RPC communication between the session manager and the container manager is the `MessageProcessor` class. It is a component used for both the initiator and receiver of RPC calls. The container manager is the receiver of the RPC call, and the type received from the session manager is `MessageType::invocation`. The message format of the RPC communication between the session manager and the container manager is defined in the Protobuf Definition file shown below.

```
syntax = "proto2";
option optimize_for = LITE_RUNTIME;

package anbox.protobuf.rpc;

message Invocation {
    required uint32 id = 1;
    required string method_name = 2;
    required bytes parameters = 3;
    required uint32 protocol_version = 4;
}

message Result {
    optional uint32 id = 1;
    optional bytes response = 2;
    repeated bytes events = 3;
}

message StructuredError {
  optional uint32 domain = 1;
  optional uint32 code = 2;
}

message Void {
  optional string error = 127;
  optional StructuredError structured_error = 128;
}
```

src/anbox/protobuf/anbox_rpc.proto

```
bool MessageProcessor::process_data(const std::vector<std::uint8_t> &data) {
  for (const auto &byte : data) buffer_.push_back(byte);

  while (buffer_.size() > 0) {
    const auto high = buffer_[0];
    const auto low = buffer_[1];
    size_t const message_size = (high << 8) + low;
    const auto message_type = buffer_[2];

    // If we don't have yet all bytes for a new message return and wait
    // until we have all.
    if (buffer_.size() - header_size < message_size) break;

    if (message_type == MessageType::invocation) {
      anbox::protobuf::rpc::Invocation raw_invocation;
      raw_invocation.ParseFromArray(buffer_.data() + header_size, message_size);

      dispatch(Invocation(raw_invocation));
    } else if (message_type == MessageType::response) {
      auto result = make_protobuf_object<protobuf::rpc::Result>();
      result->ParseFromArray(buffer_.data() + header_size, message_size);

      if (result->has_id()) {
        pending_calls_->populate_message_for_result(*result,
                                                    [&](google::protobuf::
    MessageLite *result_message) {
                                                      result_message->
    ParseFromString(result->response());
                                                    });
        pending_calls_->complete_response(*result);
      }

      for (int n = 0; n < result->events_size(); n++)
        process_event_sequence(result->events(n));
    }

    buffer_.erase(buffer_.begin(),
                  buffer_.begin() + header_size + message_size);
  }

  return true;
}
```

anbox/src/anbox/rpc/message_processor.cpp

In `process_data()`, we first extract the message header, get the length and type of the message body, then extract the message body.If the message is an invocation then we parse the message to get the raw invocation and use the dispatch function.

The `dispatch` function is a virtual function in the `MessageProcessor` class and its implementation is present in the child class `ManagementApiMessageProcessor`

```
namespace anbox {
namespace container {
ManagementApiMessageProcessor::ManagementApiMessageProcessor(
    const std::shared_ptr<network::MessageSender> &sender,
    const std::shared_ptr<rpc::PendingCallCache> &pending_calls,
    const std::shared_ptr<ManagementApiSkeleton> &server)
    : rpc::MessageProcessor(sender, pending_calls), server_(server) {}

ManagementApiMessageProcessor::~ManagementApiMessageProcessor() {}

void ManagementApiMessageProcessor::dispatch(rpc::Invocation const &invocation) {
  if (invocation.method_name() == "start_container")
    invoke(this, server_.get(), &ManagementApiSkeleton::start_container, invocation)
    ;
  else if (invocation.method_name() == "stop_container")
    invoke(this, server_.get(), &ManagementApiSkeleton::stop_container, invocation);
}

void ManagementApiMessageProcessor::process_event_sequence(
    const std::string &) {}
}  // namespace container
}  // namespace anbox
```

anbox/src/anbox/container/management_api_message_processor.cpp

The implementation is very simple, and only supports two RPC calls, namely, start the Android container and stop the Android container. In its `dispatch()` the corresponding container is invoked using the method name.

```
template <class Self, class Bridge, class BridgeX, class ParameterMessage,
          class ResultMessage>
void invoke(Self* self, Bridge* rpc,
            void (BridgeX::*function)(ParameterMessage const* request,
                                      ResultMessage* response,
                                      ::google::protobuf::Closure* done),
            Invocation const& invocation) {
  ParameterMessage parameter_message;
  if (!parameter_message.ParseFromString(invocation.parameters()))
    throw std::runtime_error("Failed to parse message parameters!");
  ResultMessage result_message;

  try {
    std::unique_ptr<google::protobuf::Closure> callback(
        google::protobuf::NewPermanentCallback<
            Self, ::google::protobuf::uint32,
            typename result_ptr_t<ResultMessage>::type>(
            self, &Self::send_response, invocation.id(), &result_message));

    (rpc->*function)(&parameter_message, &result_message, callback.get());
  } catch (std::exception const& x) {
    result_message.set_error(std::string("Error processing request: ") +
                             x.what());
    self->send_response(invocation.id(), &result_message);
  }
}
}
```

anbox/src/anbox/rpc/template_message_processor.h

The `invoke` function first parses the parameters from the invocation message and then calls a function pointer which can either be referring to `ManagementApiSkeleton::start_container` or `ManagementApiSkeleton::stop_container`.

The implementation of these functions are as follows:

```
namespace anbox {
namespace container {
ManagementApiSkeleton::ManagementApiSkeleton(
    const std::shared_ptr<rpc::PendingCallCache> &pending_calls,
    const std::shared_ptr<Container> &container)
    : pending_calls_(pending_calls), container_(container) {}

ManagementApiSkeleton::~ManagementApiSkeleton() {}

void ManagementApiSkeleton::start_container(
    anbox::protobuf::container::StartContainer const *request,
    anbox::protobuf::rpc::Void *response, google::protobuf::Closure *done) {
  if (container_->state() == Container::State::running) {
    response->set_error("Container is already running");
    done->Run();
    return;
  }

  Configuration container_configuration;

  const auto configuration = request->configuration();
  for (int n = 0; n < configuration.bind_mounts_size(); n++) {
    const auto bind_mount = configuration.bind_mounts(n);
    container_configuration.bind_mounts.insert(
        {bind_mount.source(), bind_mount.target()});
  }

  try {
    container_->start(container_configuration);
  } catch (std::exception &err) {
    response->set_error(utils::string_format("Failed to start container: %s", err.
    what()));
  }

  done->Run();
}

void ManagementApiSkeleton::stop_container(
    anbox::protobuf::container::StopContainer const *request,
    anbox::protobuf::rpc::Void *response, google::protobuf::Closure *done) {

  (void)request;

  if (container_->state() != Container::State::running) {
    response->set_error("Container is not running");
    done->Run();
    return;
  }

  try {
    container_->stop();
  } catch (std::exception &err) {
    response->set_error(utils::string_format("Failed to stop container: %s", err.
    what()));
  }

  done->Run();
}
}  // namespace container
}  // namespace anbox
```

anbox/src/anbox/container/management_api_skeleton.cpp

The start_container function first assures that the container is not already running and sets the various configuration properties and then uses the start function to start the client.
The stop_container function simply stops the container using the stop function after asserting that the container is still running.
These start and stop functions belong to the Client class which simply passes these calls to the management_api class.

The start and stop functions in the client calls are shown below:

```
void Client::start(const Configuration &configuration) {
  try {
    management_api_->start_container(configuration);
  } catch (const std::exception &e) {
    ERROR("Failed to start container: %s", e.what());
    if (terminate_callback_)
      terminate_callback_();
  }
}

void Client::stop() {
  management_api_->stop_container();
}
```

anbox/src/anbox/container/client.cpp

The management_api_ is a pointer to the object of type ManagementApiStub .
It defines the interface for starting and stopping the container, and defines the callback after the container is started and stopped. It also defines Request Class,which is used to encapsulate the request response, and a WaitHandle .WaitHandle is called by the RPC and is used to wait for the end of the request.

Although the actual RPC call is asynchronous,anbox::container::ManagementApiStub class provides the illusion of synchronous execution to its caller through condition variables. The 2 functions start_container() and stop_container() are as shown below.

```
ManagementApiStub::ManagementApiStub(
    const std::shared_ptr<rpc::Channel> &channel)
    : channel_(channel) {}

ManagementApiStub::~ManagementApiStub() {}

void ManagementApiStub::start_container(const Configuration &configuration) {
  auto c = std::make_shared<Request<protobuf::rpc::Void>>();

  protobuf::container::StartContainer message;
  auto message_configuration = new protobuf::container::Configuration;

  for (const auto &item : configuration.bind_mounts) {
    auto bind_mount_message = message_configuration->add_bind_mounts();
    bind_mount_message->set_source(item.first);
    bind_mount_message->set_target(item.second);
  }

  message.set_allocated_configuration(message_configuration);

  {
    std::lock_guard<decltype(mutex_)> lock(mutex_);
```

```cpp
    c->wh.expect_result();
  }

  channel_->call_method("start_container", &message, c->response.get(),
      google::protobuf::NewCallback(this, &ManagementApiStub::container_started, c.
    get()));

  c->wh.wait_for_all();

  if (c->response->has_error()) throw std::runtime_error(c->response->error());
}

void ManagementApiStub::container_started(Request<protobuf::rpc::Void> *request) {
  request->wh.result_received();
}

void ManagementApiStub::stop_container() {
  auto c = std::make_shared<Request<protobuf::rpc::Void>>();

  protobuf::container::StopContainer message;
  message.set_force(false);

  {
    std::lock_guard<decltype(mutex_)> lock(mutex_);
    c->wh.expect_result();
  }

  channel_->call_method("stop_container", &message, c->response.get(),
      google::protobuf::NewCallback(this, &ManagementApiStub::container_stopped, c.
    get()));

  c->wh.wait_for_all();

  if (c->response->has_error()) throw std::runtime_error(c->response->error());
}

void ManagementApiStub::container_stopped(Request<protobuf::rpc::Void> *request) {
  request->wh.result_received();
}
```

anbox/src/anbox/container/management_api_stub.cpp

In ManagementApiStub::start_container() we encapsulate the parameters into the corresponding Protobuf message, and then we use the expect_result of the wait handle class to indicate we are expecting a response.
After the RPC calls to start and stop the container are completed, the corresponding callbacks are called, and they pass the corresponding request

```cpp
namespace anbox {
namespace common {
WaitHandle::WaitHandle()
    : guard(), wait_condition(), expecting(0), received(0) {}

WaitHandle::~WaitHandle() {}

void WaitHandle::expect_result() {
  std::lock_guard<std::mutex> lock(guard);

  expecting++;
}

void WaitHandle::result_received() {
  std::lock_guard<std::mutex> lock(guard);

  received++;
  wait_condition.notify_all();
}

void WaitHandle::wait_for_all()  // wait for all results you expect
{
  std::unique_lock<std::mutex> lock(guard);

  wait_condition.wait(lock, [&] { return received == expecting; });

  received = 0;
  expecting = 0;
}

void WaitHandle::wait_for_pending(std::chrono::milliseconds limit) {
  std::unique_lock<std::mutex> lock(guard);

  wait_condition.wait_for(lock, limit, [&] { return received == expecting; });
}

void WaitHandle::wait_for_one()  // wait for any single result
{
  std::unique_lock<std::mutex> lock(guard);

  wait_condition.wait(lock, [&] { return received != 0; });

  --received;
  --expecting;
}

bool WaitHandle::has_result() {
  std::lock_guard<std::mutex> lock(guard);

  return received > 0;
}

bool WaitHandle::is_pending() {
  std::unique_lock<std::mutex> lock(guard);
  return expecting > 0 && received != expecting;
}
}  // namespace common
}  // namespace anbox
```

anbox/src/anbox/common/wait_handle.cpp

WaitHandle has various functions related to waiting and locking of mutexes.
`wait_for_all()` function blocks the current thread till the predicate is satisfied.This happens when all the results expected have been received.
`wait_for_pending()` is similar to the `wait_for_all()` function except the fact that it has a limit for the amount of time it waits since it might happen that the container manager dies before the session manager and hence the session manager will not be able to exit if it waits for all.

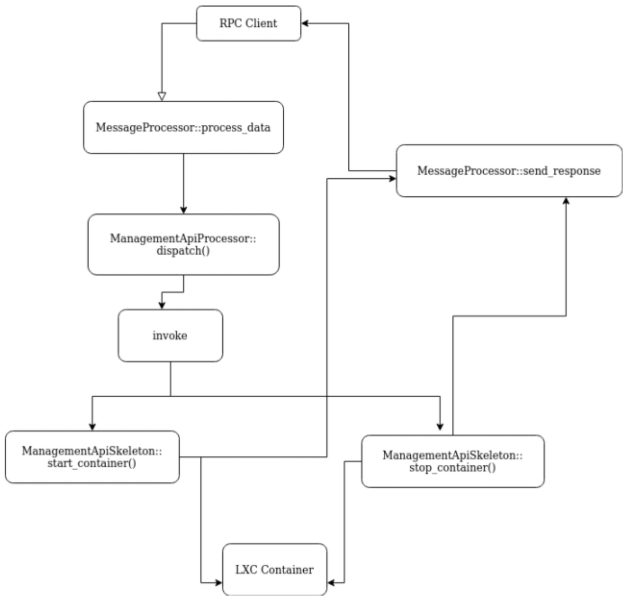
PendingCallCache is a class used to store the description and completion callback of the RPC call that has sent the request message, has been initiated but has not yet received a response.It contains a PendingCall structure used to store the response and the callback upon completion.

The `save_completion_details` functions simply maps the invocation id to the corresponding Pending Call structure created .
`populate_message_for_result()` finds the corresponding call from the result id and fills using the response.
`Complete_response()` is used to erase the call from the map once the messageprocessor receives a message of response type.It then goes to the corresponding callback function.

```
namespace anbox {
namespace rpc {
PendingCallCache::PendingCallCache() {}

void PendingCallCache::save_completion_details(
    anbox::protobuf::rpc::Invocation const& invocation,
    google::protobuf::MessageLite* response,
    google::protobuf::Closure* complete) {
  std::unique_lock<std::mutex> lock(mutex_);
  pending_calls_[invocation.id()] = PendingCall(response, complete);
}

void PendingCallCache::populate_message_for_result(
    anbox::protobuf::rpc::Result& result,
    std::function<void(google::protobuf::MessageLite*)> const& populator) {
  std::unique_lock<std::mutex> lock(mutex_);
  populator(pending_calls_.at(result.id()).response);
}

void PendingCallCache::complete_response(anbox::protobuf::rpc::Result& result) {
  PendingCall completion;

  {
    std::unique_lock<std::mutex> lock(mutex_);
    auto call = pending_calls_.find(result.id());
    if (call != pending_calls_.end()) {
      completion = call->second;
      pending_calls_.erase(call);
    }
  }

  if (completion.complete) completion.complete->Run();
}

void PendingCallCache::force_completion() {
  std::unique_lock<std::mutex> lock(mutex_);
  for (auto& call : pending_calls_) {
    auto& completion = call.second;
    completion.complete->Run();
  }

  pending_calls_.erase(pending_calls_.begin(), pending_calls_.end());
}

bool PendingCallCache::empty() const {
  std::unique_lock<std::mutex> lock(mutex_);
  return pending_calls_.empty();
}
}  // namespace rpc
}  // namespace anbox
```

anbox/src/anbox/rpc/pending_call_cache.cpp

Figure 6: RPC call acceptance flow


RPC Client
MessageProcessor::process_data
MessageProcessor::send_response
ManagementApiProcessor::dispatch()
invoke
ManagementApiSkeleton::start_container()
ManagementApiSkeleton::stop_container()
LXC Container

Graphics processing

The entry function of session manager is in anbox/src/anbox/cmds/session_manager.cpp

```
const auto should_force_software_rendering = utils::get_env_value("
    ANBOX_FORCE_SOFTWARE_RENDERING", "false");
auto gl_driver = graphics::GLRendererServer::Config::Driver::Host;
if (should_force_software_rendering == "true" || use_software_rendering_)
 gl_driver = graphics::GLRendererServer::Config::Driver::Software;

graphics::GLRendererServer::Config renderer_config {
  gl_driver,
  single_window_
};
auto gl_server = std::make_shared<graphics::GLRendererServer>(renderer_config,
    window_manager);

platform->set_window_manager(window_manager);
platform->set_renderer(gl_server->renderer());
window_manager->setup();
```

anbox/src/anbox/cmds/session_manager.cpp

Here we are deciding whether we want to use software rendering or hardware rendering and setting `gl_driver` accordingly.
The window manager and renderer are initialised and the window manager is setup.
Now let's see what is in GLRendererServer:

```
GLRendererServer::GLRendererServer(const Config &config, const std::shared_ptr<wm::
    Manager> &wm)
    : renderer_(std::make_shared<::Renderer>()) {

  std::shared_ptr<LayerComposer::Strategy> composer_strategy;
  if (config.single_window)
    composer_strategy = std::make_shared<SingleWindowComposerStrategy>(wm);
  else
    composer_strategy = std::make_shared<MultiWindowComposerStrategy>(wm);

  composer_ = std::make_shared<LayerComposer>(renderer_, composer_strategy);

  auto gl_libs = emugl::default_gl_libraries();
  if (config.driver == Config::Driver::Software) {
    auto swiftshader_path = fs::path(utils::get_env_value("SWIFTSHADER_PATH"));
    const auto snap_path = utils::get_env_value("SNAP");
    if (!snap_path.empty())
      swiftshader_path = fs::path(snap_path) / "lib" / "anbox" / "swiftshader";
    if (!fs::exists(swiftshader_path))
      throw std::runtime_error("Software rendering is enabled, but SwiftShader
    library directory is not found.");

    gl_libs = std::vector<emugl::GLLibrary>{
      {emugl::GLLibrary::Type::EGL, (swiftshader_path / "libEGL.so").string()},
      {emugl::GLLibrary::Type::GLESv1, (swiftshader_path / "libGLES_CM.so").string()
    },
      {emugl::GLLibrary::Type::GLESv2, (swiftshader_path / "libGLESv2.so").string()
    },
    };
  }
```

anbox/src/anbox/graphics/gl_renderer_server.cpp

We first choose the layer composition strategy based on whether it is a single window configuration. First we set the path of the swiftshader library using the environment variables for software rendering. If the path of the swift shader library is found then it will read libEGL.so ,libGLES_CM.so and libGLESv2.so which are the three library files corresponding to OpenGL ES.

OpenGL ES

OpenGL for Embedded Systems (OpenGL ES or GLES) is a part of the OpenGL computer graphics for rendering computer graphics used by video games, typically hardware-accelerated using a GPU. It is designed for embedded systems like smartphones, tablet computers, video game consoles .

Three dynamic libraries are generated for the translation process These dynamic libraries are installed under libEGL_emulation.so, libGLESv1_CM_emulation.so and libGLESv2_emulation.so. That is, the OpenGL ES rendering library provided by Anbox as a virtual hardware manufacturer to Android.
The rendering work of these libraries is not performed by Anbox but by the host. The OpenGL ES instructions provided by the Android application are collected by these three libraries and sent to the host via the high-speed transmission channel qemu-pipe

We also use swiftshader to get support for OpenGL ES.Swiftshader is a pure software rendering method provided by Google and provides hardware independence for advanced 3D graphics.
If the graphics card and driver in the host machine support the OpenGL ES, we can directly use the graphics card in the host machine for rendering. However, most graphics cards currently cannot directly support OpenGL ES ,thereby requiring Swiftshader.

Conclusion and Further Work

- We found it difficult to modify the source code due to its large size. So we first focused on understanding the important parts of Anbox.
- In the future we want to modify the source code to redirect input from another system to anbox. We think we need to modify source code relating to LXC to achieve this. We think we need a TCP socket to receive the input from another system and modify LXC to redirect it to Android.
- Anbox currently allows only one user. A possible improvement would be to support multiple users. Also Anbox currently supports Android 7 and another improvement would be to support the latest version of Android.
- Anbox can be modified to simulate camera. Also anbox mainly uses software rendering due to incompatibility with OpenGL ES, this can be improved to do graphics processing natively on the gpu.