scrcpy for developers

Overview

This application is composed of two parts:

- the server (`scrcpy-server`), to be executed on the device,
- the client (the `scrcpy` binary), executed on the host computer.

The client is responsible to push the server to the device and start its execution.

Once the client and the server are connected to each other, the server initially sends device information (name and initial screen dimensions), then starts to send a raw H.264 video stream of the device screen. The client decodes the video frames, and display them as soon as possible, without buffering, to minimize latency. The client is not aware of the device rotation (which is handled by the server), it just knows the dimensions of the video frames.

The client captures relevant keyboard and mouse events, that it transmits to the server, which injects them to the device.

Server

Privileges

Capturing the screen requires some privileges, which are granted to `shell` .

The server is a Java application (with a `public static void main(String... args)` method), compiled against the Android framework, and executed as `shell` on the Android device.

To run such a Java application, the classes must be *dexed* (typically, to `classes.dex`). If `my.package.MainClass` is the main class, compiled to `classes.dex` , pushed to the device in `/data/local/tmp` , then it can be run with:

```
adb shell CLASSPATH=/data/local/tmp/classes.dex \
    app_process / my.package.MainClass
```

The path `/data/local/tmp` is a good candidate to push the server, since it's readable and writable by `shell` , but not world-writable, so a malicious application may not replace the server just before the client executes it.

Instead of a raw *dex* file, `app_process` accepts a *jar* containing `classes.dex` (e.g. an APK). For simplicity, and to benefit from the gradle build system, the server is built to an (unsigned) APK (renamed to `scrcpy-server`).

Hidden methods

Although compiled against the Android framework, hidden methods and classes are not directly accessible (and they may differ from one Android version to another).

They can be called using reflection though. The communication with hidden components is provided by *wrappers* classes and aidl.

Threading

The server uses 3 threads:

- the **main** thread, encoding and streaming the video to the client;

- the **controller** thread, listening for *control messages* (typically, keyboard and mouse events) from the client;
- the **receiver** thread (managed by the controller), sending *device messages* to the clients (currently, it is only used to send the device clipboard content).

Since the video encoding is typically hardware, there would be no benefit in encoding and streaming in two different threads.

Screen video encoding

The encoding is managed by `ScreenEncoder`.

The video is encoded using the `MediaCodec` API. The codec takes its input from a surface associated to the display, and writes the resulting H.264 stream to the provided output stream (the socket connected to the client).

On device rotation, the codec, surface and display are reinitialized, and a new video stream is produced.

New frames are produced only when changes occur on the surface. This is good because it avoids to send unnecessary frames, but there are drawbacks:

- it does not send any frame on start if the device screen does not change,
- after fast motion changes, the last frame may have poor quality.

Both problems are solved by the flag `KEY_REPEAT_PREVIOUS_FRAME_AFTER`.

Input events injection

Control messages are received from the client by the `Controller` (run in a separate thread). There are several types of input events:

- keycode (cf `KeyEvent`),
- text (special characters may not be handled by keycodes directly),
- mouse motion/click,
- mouse scroll,
- other commands (e.g. to switch the screen on or to copy the clipboard).

Some of them need to inject input events to the system. To do so, they use the *hidden* method `InputManager.injectInputEvent` (exposed by our `InputManager` wrapper).

Client

The client relies on SDL, which provides cross-platform API for UI, input events, threading, etc.

The video stream is decoded by libav (FFmpeg).

Initialization

On startup, in addition to *libav* and *SDL* initialization, the client must push and start the server on the device, and open two sockets (one for the video stream, one for control) so that they may communicate.

Note that the client-server roles are expressed at the application level:

- the server *serves* video stream and handle requests from the client,
- the client *controls* the device through the server.

However, the roles are reversed at the network level:

- the client opens a server socket and listen on a port before starting the server,

- the server connects to the client.

This role inversion guarantees that the connection will not fail due to race conditions, and avoids polling.

(Note that over TCP/IP, the roles are not reversed, due to a bug in `adb reverse`*. See commit 1038bad and issue #5.)*

Once the server is connected, it sends the device information (name and initial screen dimensions). Thus, the client may init the window and renderer, before the first frame is available.

To minimize startup time, SDL initialization is performed while listening for the connection from the server (see commit 90a46b4).

Threading

The client uses 4 threads:

- the **main** thread, executing the SDL event loop,
- the **stream** thread, receiving the video and used for decoding and recording,
- the **controller** thread, sending *control messages* to the server,
- the **receiver** thread (managed by the controller), receiving *device messages* from the server.

In addition, another thread can be started if necessary to handle APK installation or file push requests (via drag&drop on the main window) or to print the framerate regularly in the console.

Stream

The video stream is received from the socket (connected to the server on the device) in a separate thread.

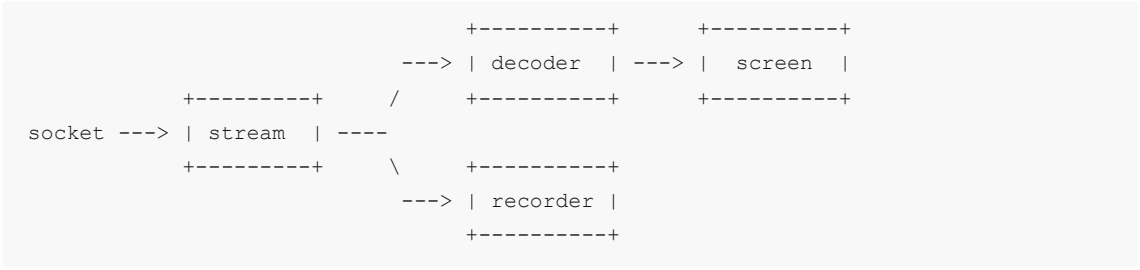
If a decoder is present (i.e. `--no-display` is not set), then it uses *libav* to decode the H.264 stream from the socket, and notifies the main thread when a new frame is available.

There are two frames simultaneously in memory:

- the **decoding** frame, written by the decoder from the decoder thread,
- the **rendering** frame, rendered in a texture from the main thread.

When a new decoded frame is available, the decoder *swaps* the decoding and rendering frame (with proper synchronization). Thus, it immediately starts to decode a new frame while the main thread renders the last one.

If a recorder is present (i.e. `--record` is enabled), then it muxes the raw H.264 packet to the output video file.



Controller

The controller is responsible to send *control messages* to the device. It runs in a separate thread, to avoid I/O on the main thread.

On SDL event, received on the main thread, the input manager creates appropriate *control messages*. It is responsible to convert SDL events to Android events (using convert). It pushes the *control messages* to a queue hold by the controller. On its own thread, the controller takes messages from the queue, that it serializes and sends to the client.

UI and event loop

Initialization, input events and rendering are all managed in the main thread.

Events are handled in the event loop, which either updates the screen or delegates to the input manager.

Hack

For more details, go read the code!

If you find a bug, or have an awesome idea to implement, please discuss and contribute ;-)

Debug the server

The server is pushed to the device by the client on startup.

To debug it, enable the server debugger during configuration:

```
meson x -Dserver_debugger=true
# or, if x is already configured
meson configure x -Dserver_debugger=true
```

If your device runs Android 8 or below, set the `server_debugger_method` to `old` in addition:

```
meson x -Dserver_debugger=true -Dserver_debugger_method=old
# or, if x is already configured
meson configure x -Dserver_debugger=true -Dserver_debugger_method=old
```

Then recompile.

When you start scrcpy, it will start a debugger on port 5005 on the device. Redirect that port to the computer:

```
adb forward tcp:5005 tcp:5005
```

In Android Studio, *Run* > *Debug* > *Edit configurations*... On the left, click on `+`, *Remote*, and fill the form:

- Host: `localhost`
- Port: `5005`

Then click on *Debug*.