Introducing scrcpy

08 Mar 2018

I developed an application to display and control Android devices connected on USB. It does not require any root access. It works on GNU/Linux, Windows and Mac OS.

It focuses on:

- **lightness** (native, displays only the device screen)
- **performances** (30~60fps)

- **quality** (1920×1080 or above)
- **low latency** (70~100ms)
- **low startup time** (~1 second to display the first image)
- **non-intrusiveness** (nothing is left installed on the device)

Like my previous project, *gnirehtet*, Genymobile accepted to open source it: scrcpy.

You can build, install and run it.

How does scrcpy work?

The application executes a server on the device. The client and the server communicate via a socket over an *adb tunnel*.

The server streams an H.264 video of the device screen. The client decodes the video frames and displays them.

The client captures input (keyboard and mouse) events, sends them to the server, which injects them to the device.

The documentation gives more details.

Here, I will detail several technical aspects of the application likely to interest developers.

Minimize latency

No buffering

It takes time to encode, transmit and decode the video stream. To minimize latency, we must avoid any additional delay.

For example, let's stream the screen with `screenrecord` and play it with VLC:

```
adb exec-out screenrecord --output-format=h264 - | vlc - --(
```

Initially, it works, but quickly the latency increases and frames are broken. The reason is that VLC associates a PTS to frames, and buffers the stream to play frames at some target time.

As a consequence, it sometimes prints such errors on *stderr*:

```
ES_OUT_SET_(GROUP_)PCR  is called too late (pts_delay increa
```

Just before I started the project, Philippe, a colleague who played with WebRTC, advised me to "manually" decode (using *FFmpeg*) and render frames, to avoid any additional latency. This saved me from wasting time, it was the right solution.

Decoding the video stream to retrieve individual frames with FFmpeg is rather straightforward.

Skip frames

If, for any reason, the rendering is delayed, decoded frames are dropped so that *scrcpy* always displays the last decoded frame.

Note that this behavior may be changed with a configuration flag:

```
mesonconf x -Dskip_frames=false
```

Run a Java main on Android

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

It is possible to execute Java code as `shell` on Android, by invoking `app_process` from `adb shell`.

Hello, world!

Here is a simple Java application:

```
public class HelloWorld {
    public static void main(String... args) {
        System.out.println("Hello, world!");
    }
}
```

Let's compile and *dex* it:

```
javac -source 1.7 -target 1.7 HelloWorld.java
"$ANDROID_HOME"/build-tools/27.0.2/dx \
    --dex --output classes.dex HelloWorld.class
```

Then, we push `classes.dex` to an Android device:

```
adb push classes.dex /data/local/tmp/
```

And execute it:

```
$ adb shell CLASSPATH=/data/local/tmp/classes.dex app_proces
Hello, world!
```

Access the Android framework

The application can access the Android framework at runtime.

For example, let's use `android.os.SystemClock`:

```
import android.os.SystemClock;

public class HelloWorld {
    public static void main(String... args) {
        System.out.print("Hello,");
        SystemClock.sleep(1000);
        System.out.println(" world!");
    }
}
```

We link our class against `android.jar`:

```
javac -source 1.7 -target 1.7 \
    -cp "$ANDROID_HOME"/platforms/android-27/android.jar
    HelloWorld.java
```

Then run it as before.

Note that scrcpy also needs to access hidden methods from the framework. In that case, linking against `android.jar` is not sufficient, so it uses reflection.

Like an APK

The execution also works if `classes.dex` is embedded in a zip/jar:

```
jar cvf hello.jar classes.dex
adb push hello.jar /data/local/tmp/
adb shell CLASSPATH=/data/local/tmp/hello.jar app_process /
```

You know an example of a zip containing `classes.dex` ? An APK!

Therefore, it works for any installed APK containing a class with a main method:

```
$ adb install myapp.apk
…
$ adb shell pm path my.app.package
package:/data/app/my.app.package-1/base.apk
$ adb shell CLASSPATH=/data/app/my.app.package-1/base.apk \
    app_process / HelloWorld
```

In scrcpy

To simplify the build system, I decided to build the server as an APK using gradle, even if it's not a real Android application: *gradle* provides tasks for running tests, checking style, etc.

Invoked that way, the server is authorized to capture the device screen.

Improve startup time

Quick installation

Nothing is required to be installed on the device by the user: at startup, the client is responsible for executing the server on the device.

We saw that we can execute the main method of the server from an APK either:

- installed, or
- pushed to `/data/local/tmp`.

Which one to choose?

```
$ time adb install server.apk
…
real    0m0,963s
…

$ time adb push server.apk /data/local/tmp/
…
real    0m0,022s
…
```

So I decided to push.

Note that `/data/local/tmp` is readable and writable by `shell`, but not world-writable, so a malicious application may not replace the server just before the client executes it.

Parallelization

If you executed the *Hello, world!* in the previous section, you may have noticed that running `app_process` takes some time: `Hello, World!` is not printed before some delay (between 0.5 and 1 second).

In the client, initializing SDL also takes some time.

Therefore, these initialization steps have been parallelized.

Clean up the device

After usage, we want to remove the server (`/data/local/tmp/scrcpy-server.jar`) from the device.

We could remove it on exit, but then, it would be left on device disconnection.

Instead, once the server is opened by `app_process`, *scrcpy* unlinks (`rm`) it. Thus, the file is present only for less than

1 second (it is removed even before the screen is displayed).

The file itself (not its name) is actually removed when the last associated open file descriptor is closed (at the latest, when `app_process` dies).

Handle text input

Handling input received from a keyboard is more complicated than I thought.

Events

There are 2 kinds of "keyboard" events:

- key events,
- text input events.

Key events provide both the *scancode* (the physical location of a key on the keyboard) and the *keycode* (which depends on the keyboard layout). Only *keycodes* are used by *scrcpy* (it doesn't need the location of physical keys).

However, key events are not sufficient to handle text input:

> Sometimes it can take multiple key presses to produce a character. Sometimes a single key press can produce multiple characters.

Even simple characters may not be handled easily with key events, since they depend on the layout. For example, on a French keyboard, typing `.` *(dot)* generates `Shift + ; .`

Therefore, *scrcpy* forwards key events to the device only for a limited set of keys. The remaining are handled by *text input* events.

Injecting text

On the Android side, we may not inject text directly (injecting a `KeyEvent` created by the relevant constructor

does not work). Instead, we can retrieve a list of `KeyEvent` s to generate for a `char[]` , using `getEvents(char[])` .

For example:

```
char[] chars = {'?'};
KeyEvent[] events = charMap.getEvents(chars);
```

Here, `events` is initialized with an array of 4 events:

1. press `KEYCODE_SHIFT_LEFT`
2. press `KEYCODE_SLASH`
3. release `KEYCODE_SLASH`
4. release `KEYCODE_SHIFT_LEFT`

Injecting those events correctly generates the char `'?'` .

Handling accented characters

Unfortunately, the previous method only works for ASCII characters:

```
char[] chars = {'é'};
KeyEvent[] events = charMap.getEvents(chars);
// events is null!!!
```

I first thought there was no way to inject such events from there, until I discussed with Philippe (yes, the same as earlier), who knew the solution: it works when we decompose the characters using combining diacritical dead key characters.

Concretely, instead of injecting `"é"` , we inject `"\u0301e"` :

```
char[] chars = {'\u0301', 'e'};
KeyEvent[] events = charMap.getEvents(chars);
// now, there are events
```

Therefore, to support accented characters, *scrcpy* attempts to decompose the characters using `KeyComposition` .

EDIT: Accented characters do not work with the virtual keyboard Gboard (the default Google keyboard), but work with the default (AOSP) keyboard and SwiftKey.

Set a window icon

The application window may have an icon, used in the title bar (for some desktop environments) and/or in the desktop taskbar.

The window icon must be set from an `SDL_Surface` by `SDL_SetWindowIcon`. Creating the surface with the icon content is up to the developer. For exemple, we could decide to load the icon from a PNG file, or directly from its raw pixels in memory.

Instead, another colleague, Aurélien, suggested I use the XPM image format, which is also a valid C source code: `icon.xpm`.

Note that the image is not the content of the variable `icon_xpm` declared in `icon.xpm`: it's the whole file! Thus, `icon.xpm` may be both directly opened in Gimp and included in C source code:

```
#include "icon.xpm"
```

As a benefit, we directly "recognize" the icon from the source code, and we can patch it easily: in debug mode, the icon color is changed.

Conclusion

Developing this project was an awesome and motivating experience. I've learned a lot (I never used *SDL* or *libav*/*FFmpeg* before).

The resulting application works better than I initially expected, and I'm happy to have been able to open source it.

Discuss on reddit and Hacker News.

Comments

Cascador

8 March 2018, 14:00

Salute,

Tu t'en sers pour faire quoi ? Quels sont les cas d'usage ?

Merci, Tcho !

®om

8 March 2018, 22:00

@Cascador :

I cross-post my answer (in English):

> It has been developed for a specific use case (it is included in a B2B application where the user may need to display and control devices connected on USB).
>
> But you might use it for other reasons, like typing text messages from your computer, or showing your device in a conference (e.g. to demo an app).

https://news.ycombinator.com/item?id=16546025

Tcho ! ;-)

Miglen

8 March 2018, 22:01

Hola,

This is such great piece of work, thanks for sharing!

Cascador

9 March 2018, 08:51

Salute,

Tu comptes publier un article en Français ? Je l'aurai bien remonté sur le Journal du hacker. Pour envoyer des SMS c'est très simple/pratique ou il faut avoir envie et faire des efforts selon toi ?

Thanks ha ha, Tcho !

®om

9 March 2018, 09:06

J'avais hésité, mais finalement je ne vais pas publier en français: ça demande trop de travail d'écrire les articles en double.

Je ne suis pas sûr d'avoir compris ta question sur les SMS. Moi, je trouve plus pratique de les écrire avec le clavier du PC.

onioncoder

9 March 2018, 09:41

This is awesome. Love that it is a cross platform native app :)

dro

9 March 2018, 11:52

Could you make it run on older android versions like 4.3? Also can you please share your app on f-droid too?

DaScritch

9 March 2018, 11:57

@cascador : utilises plutôt kconnect pour récupérer les sms vers ton ordi. Tu peux même faire du cross ^C^V !

rayworks

9 March 2018, 13:43

A great project！ About one year ago, I just made a demo app to take a screenshot dynamically. However,the result you achieved is what I actually want.

Terpe

9 March 2018, 19:07

Thanks! Learned new thinks

j-gatsby

9 March 2018, 19:13

Awesome project. Excellent write-up.

And thank you for sharing it with us.

Great work! Very well done.

Trevor Parsons

9 March 2018, 21:15

Thanks for writing and publishing scrcpy. It was easy to install via AUR on my Arch-based desktop and it works perfectly. Much appreciated!

jg

9 March 2018, 23:27

This looks amazing! I tried it on Windows with my V20 and the image displays immediately and updates quick but it does not respond to my clicks. :(

®om

9 March 2018, 23:31

@jg, it seems that clicks do not work with LG devices on this version.

Louie

10 March 2018, 02:50

What version of Windows is compatible? It does not work on Windows 10.

Hoang

10 March 2018, 18:01

Thanks for the app. Looks good and fast, but I can’t get edge-swiping to work, which limits usability quite a bit. Likewise no pinch-zoom. Hope to see these in a future iteration.

Seb

10 March 2018, 22:21

@Louie, I don't know if it's specific to Windows 10 but the first time I started the app it appeared not to work either. In my case the debugging mode trust dialog appeared on my phone but nothing happened after allowing the connection. Executing the app again worked but it spawned a duplicate process on my computer (CTRL+ALT+DEL -> Task Manager – didn't know which one was the good one so I killed both). It appears to be working fine now.

@Romain, nice work! Thanks for opening the source! I haven't checked the code yet but maybe there's an issue when connecting the phone in debugging mode for the first time on an untrusted computer?

®om

10 March 2018, 22:26

@Seb, Thank you for reporting the problem. I didn't check yet. Please open an issue on GitHub.

just_found

10 March 2018, 23:29

Works great! Ran error free.

@Louie…using win10.

I have been looking for something like this for a while.

Thank you for contributing to the open source community.

Seb

11 March 2018, 00:45

@Romain,

It seems like the issue (or at least a very similar one) has already been submitted (Issue #9). I've replicated the problem on another computer, so I've added this info as a comment to the existing issue.

Thanks!

claude

11 March 2018, 07:45

On Windows 7 32 bit. it doesn't work.

Jakob

11 March 2018, 11:00

Thanks, that's great! Any chance this would work in 1280x720 screens ?

Post a comment

Name : *(required)*

E-mail : *(optional, non published)*

Website : *(optional)*

What is the 3rd letter of the word `blog` ? *(antispam)*

Comments are formatted in markdown.

Send