Move Fast and Break Everything: Testing major changes to a core component of GNOME
Sam Thursfield

>> Hello, hello, everyone. Welcome back. Our next presenter today is Sam Thursfield.

Sam Thursfield: Hi, everybody. Welcome back. So, we're going to --

>> And he will be talking about move fast and break everything. testing major components to the core component of GNOME. Good luck, Sam.

Sam Thursfield: Thank you. Hopefully everyone can hear me. I'm sure you'll tell me if you can't. My talk today, I realized it has the wrong title. I should have called it, move fast and don't break anything. Because that's the point. It's about how to work on some of the lower level parts of GNOME. And I'm hoping interesting both to relative newcomers who have maybe done a little bit of development in GNOME but haven't yet dived into the core. And also, interesting for people who want to improve the developer experience. It's always important to reduce friction in how we develop the project. And I don't have any ideas, but at least I know the problems. So, that's a start.

So, the talk is gonna be in two parts. The first part is about a bit of background on system services which we also call daemons for those who aren't familiar. And the second part is things I've learned while I have been working on one specific daemon called Tracker. Which is the indexer. So, GNOME has a newcomers workflow and it's very good. This is taken from the newcomer's page on the Wiki. And if we could show hands, then I would ask a show of hands who came into the project using this? I don't know if we can show hands. I think we can't. But you can write in the chat if you have followed the newcomer's workflow and it's worked for you. We're very interested in how well this is working.

So, the newcomer's page lets you choose a project and you can choose one of 11 different apps. So, that's good. But apps. Is GNOME just apps? What is GNOME? GNOME is apps. It's a shell. So, I drew this diagram. We have apps. We have the shell. But what's underneath? Sea monsters. A mystery. Well, the core, as we tend to call it, is mostly two things. We have libraries, things like GTK and GStreamer. Those are generally linked into the apps. And they usually -- well, a library, as you probably know, is a way to share code functions. So, you can link it into the app and this has some advantages. But we also provide quite a lot of services. Now, maybe you would be surprised how many services GNOME actually provides. I've listed five examples -- six examples here. So, let's have a look. dconf, evolution-data-server and Tracker. These are about configuration storage. dconf serves configuration data. It's a daemon, you want to store in one place. And evolution-data-server and Tracker are similar. Centralize data, which is more efficient. It reduces duplication. We have NetworkManager and hardware. You need a system for that. Because if five different apps try to talk to the hardware at once, it generally doesn't work because there's only one piece of hardware and it usually only has one communication channel. Another reason we have system services is to talk to hardware. And then I also listed GNOME keyring, which is to partly to centralize data. But also for security reasons. That we want to make sure there's a barrier between the thing that stores all your passwords and the apps that should only get to -- well, they shouldn't get to know any passwords.

So, there are a lot of system services supplied by GNOME. How many? Well... you can find out by trying this command. If you run pstree, you might need to install it. If you run pstree, you can see a list of all of the processes running on your system. This is just a small number from my system. And we can see there's ModemManager and NetworkManager, which I think are part of GNOME. Maybe not. Accounts daemon is I think maintained by geometry. Colord is part of GNOME. And there are many more if you go down the list. Which are things provided and maintained within the GNOME project. So it's not just about apps.

System services, some important things about system services. They usually community over DBus. So, this is different to libraries which are linked into the application, the system service communicates over DBus. Sometimes an app will access them through a library. But that library will actually just be calling the DBus interface. So, it's a helper. But there's no magic there. And in terms of sandboxing, life is difficult for system services because they can't go in the sandbox normally. So, they are part of the surface between a sandbox application and the host operating system. Which means since we've started sandboxing apps and bundling apps, system services have quite strict requirements now that if they break API, they may break bundled apps. Which is exactly what we don't want to happen. So, maintaining system services is quite difficult and it's very important to test them.

Right. We've done this. So, that's a brief introduction to daemons, or system services. Now, let's talk about how to hack them. This is -- this is actually the FreeBSD logo. But it's a picture of a demon because it's a play on words, right? Demons are not inherently evil. But we do represent them sometimes with a demon icon. So, how would you hack on a project that's a systems service? I'm gonna use as an example Tracker because that's the one I have been working on. But to be clear, the same thing should be -- the same thing should apply to anything, really. If you want to make a change to colord or you want to change a change to NetworkManager or to dconf, you should be able to follow the same kind of processes that I'm talking about here.

So, what do we do? The newcomer's workforce says we use GNOME builder for apps. I'm going to use it for everything else as well. I'm not going to do a live demo of this. But I've taken screenshots. There's me opening GNOME Builder, and I subtract it. And the project opens. Great, I can click the play button and it builds. And then it runs something. But it's not run something particularly useful. What it's done is it's run the commandline tool that you can use to control the Tracker daemon. So, this commandline tool will be talking to the Tracker daemon installed on your system.

So, you've just built one. And this isn't gonna talk to it. You've built it and it sat there in the source tree. But this is going to talk to the one running on your operating system if it's there. Or it's gonna break if it's not there. So, we get so far, we can build it. But we need to do a bit more work to actually test it. Like all good projects, Tracker ships some automated tests. And gnome-builder has a way to run these. Look on the left-hand side, there's a unit tests entry. And if you click on one of the unit tests, it will run and you will see the results down in the console box. So, that is actually very good. And you can do a lot of development this way.

The unit tests, if they're implemented with, then they will have to work in the source tree. Some projects which are older maybe have unit tests which they can be a bit strange. Sometimes you have to install something and they test something on your host. But they run from the source tree

or whatever. But mess and Moore made a policy that they will run from the source tree. Hopefully that's what all tests will do in the future. We click it, we run it, there's a test.

In fact, if we've got automated tests, that's everything, right? It's quite difficult to write an automated test for a daemon. So, here are some tips. This daemon is gonna talk of a DBus. And your host DBus daemon isn't going to look in the source tree that you're working in. It's gonna look only in slash user, slash share, and a couple of other places. So, how can you run the one that you just built? One useful tool it called dbus-run-session.

So, dbus-run-session is a simple script that will start a new DBus daemon. And it works just like the one running in the session. So, run dbus-run-session and then start the daemon process and then it will run in a totally separate session bus. So, you can run from the source tree. And you can even, if you have more complicated requirements, specify a configuration file and start to point it to service files in the source tree, for example. Or if you need some custom config, you can do that. So, that's what we do in Tracker to have these automated tests that run from the source tree. And that's what I imagine a lot of other projects are doing.

Another very useful tool is called umockdev. umockdev is basically a tool for making mock hardware. So, real hardware should not be used in automated tests because maybe a CI server or another developer doesn't have the hardware. So, although there's a value to testing on real hardware. It's not the same as an automated test that we can use for everything. So, you can use umockdev. And it has built-in capabilities to mock things like batteries and you can also implement your own devices. You can plug in a device, record the USB traffic, and then play it back in your test. So, that's very useful.

And my tip, you don't have to use Python. You don't have to write your tests in Python. My point is you don't have to use the same language as your project. For example, Tracker is written in C. Some of the tests are written in C. But some of them are written in Python because it's much quicker to write them. They work the same. And also, we're testing the Python bindings. So, actually, it's turned up quite a lot of bugs in the Python bindings having started writing tests in Python. So, you can consider writing some unit tests in different language to the daemon. Especially if it's in C. Because C can be quite a lot of work to write.

And if you're interested, look in the Tracker source tree. And you can see a working example of how all this works. So, like I said, automated testing is everything, right? No.

So, automated testing, even in a project with loads of test engineers and 100% test coverage, you should still really do some manual testing if you made a big change. Because you can never be sure exactly what's gonna happen. And in GNOME, it's even more important because writing tests is very boring at times. And a lot of us are volunteers. And we want to spend our weekends doing the things which are fun like adding new features. Not the things which are boring, such as adding tests for existing features. So, our test coverage in some projects is not the best. And ideally, we will one day be able to fund -- or someone can fund work for test engineers to increase the test coverage. But in the meantime, I imagine most projects have a best coverage of maybe 70 or 80%. Just because of the developer resources we have.

And, of course, every automated test introduces a cost. It makes it harder to change the project because you have to change all the tests. So, let's look at how to manually test the thing that you're

working on. What's the secret for manually testing a daemon? Well, there is no secret. There are just lots of different approaches. All of which have upsides and downsides. So, the last thing I want to talk about is some of these different approaches. I've listed five. All of which I've done. And tried. I'm gonna go through them. And then compare what are the advantages and disadvantages of each?

So, you can run it from the source tree. Well, you probably can't, actually. Because most daemons would require some kind of configuration or data file. And they won't look for it in the source tree. Sometimes you can hack that into working. But this has the problem of you're now adding extra code. Which isn't in use for anything except testing. So, that's often a bad idea. Because then you still haven't actually tested the real deployed application. You've tested something slightly different.

In Tracker, we've experimented with providing a helper script which does let you run everything from the source tree. And that works okay. But at the same time, there's a lot of code there which is used only for testing. So, I'm still not sure if I like the approach. And I still don't think it replaces fully testing everything. Next option. Just install it into /usr. Why not? Well, there's a lot of good things about this approach. It's fast. It's easy. It's simple. The only problem is you will probably break the operating system.

So, you can do it if it's a throw away development machine. You can do it if you're working in a VM. But if you're working on the same computer you need for everything else in your life. Don't do this because you will probably break it. You'll think, no, this can't possibly break it. I'll just try this. And it will break. I promise you. So, what else can we do?

We can install into /opt. This is a path where we often install software that doesn't come from the distribution. It's also fast. It's safe because it's separated from the distribution. What's the problem? Had the problem is just like running from the source tree, a lot of system things don't look in /opt. So, a good example is DBus auto-activation. You install a daemon and you expect it to be started automatically by an application. The DBus daemon on your host will look in/usr, a couple other places. It won't look in /opt. Another common case is G settings. You install in /opt, it is not looking in /opt for the settings schema. There's a work around, set export XDG_DATA_DIRS to your custom path. That will fix like G settings not being found. And then install the dbus-run-session tool, sets a new daemon. And because we have this, it's going to look in this directory for service files to activate. And this works quite well for testing. I have been using it a lot.

The problem is, again, you're now using code paths which are different to what will actually be deployed by distributions. There's not much different, but there can still be bugs. Any difference means something might break, right? If you've heard of JHBuild, it's a build tool sometimes useful for building GNOME. These days it's not so useful. -- makes is easier. But it will install into you're you configure it. And the JHBuild can do some of this, I think. Not sure exactly what it does.

What's next? You can package the project for your distribution. That's not as difficult as you might think. You can clone the existing packaging, modify it a bit. Use some tools. Popular distros have tools to make this easy, Copper and Fedora, PPA. This is great, you're testing the paths that distros will be using. And it's slow because every time you need to rebuild the package and then need to install a package. You lose some value minutes or seconds of your life.

The last approach is to use BuildStream. So, this has some advantages. Unlike some of the previous approaches, it's reproducible. So, if I give instructions and say, run these commands. I do them on my computer, you do them on your computer. We'll get the same thing. That's not necessarily true if we're running commands on different distros. The downside at the moment is because BuildStream's reproducible by default, a lot of shortcuts aren't possible.

So, for example, you need to create -- if you're testing a daemon, you need to create a virtual machine. There's no magic that will do these dbus-run-session session shortcuts I talked about before. You can do those in the bst shell. But that's not the idea. The idea is to build something like exactly what we are going to build. Build a VM. Look at Valentin's talk from yesterday to see the progress. You can run it on a laptop now if you don't care about security at all. But building them is quite slow. Downloading the files to build them can take hours and a lot of disk space.

Deploying an update is faster because you can build an OSTree update and put that in the machine. But it still took around 20 minutes for me with a laptop on an SSD. There's more work to be done. One thing we talked about yesterday was installing podman and toolbox on the -- inside the VM images. Which would make it more easy to develop inside the VM.

So, you can combine that with the first approach. Just install stuff into /usr like a crazy monkey and nothing breaks because it's a PM. And then you can build stuff the correct, slow way and do some final testing. So, here's my summary of the approaches.

Maybe you can take a screenshot. I've put a tick by each approach which is fast. Each one which is reproducible. Each one which is exactly what will be running on real distributions. And each one where you can do it without needing a VM or an extra computer. As you can see, there's not yet a perfect answer. Hopefully in the future the last one will become faster. And it will become three ticks. Which is a pretty good approach.

In the meantime, try what works for you. I need to wrap up. So, in conclusion, don't be afraid to contribute to projects just because they are system services. If you test them sufficiently, you can be sure not to break anything. Not 100% sure. But 95% sure. If you maintain a daemon at the moment, I think some of us have outdated READMEs. So, until recently the Tracker README was just a standard GNU how to use this package. Configure, install. So, in the README, we were telling installers to install into /usr and break their operating system as the first step. We don't do that anymore. Thankfully. I've tried to update the README to be welcoming and to reduce friction in the hope that more people will start to contribute to the project and there will be less friction to new developers.

Automated testing, of course, is great. And if you want help adding automated testing to an existing daemon, then I'm happy to help. Having done it for Tracker, I've made all of the mistakes and I can tell you what mistakes not to make.

But we also will always need to manually test things and we need to make sure we have a good -- a good way to do that. So, that's everything. I don't know if we have time for questions. But thank you for watching my talk.

>> Hi, Sam. Thank you a lot for the talk. We have -- I think that we have one question. And I think that we can go through it. Would you like me to read it for you?

Sam Thursfield: Yes. I can read it. The question, dbus-launch versus dbus-run-session. I would say use dbus-run-session because it's simpler and it's also -- it's for debugging. So, dbus-run-session puts the standard input and standard error of all the processes it starts on to console. Or on to standard error. You use dbus-run-session and you can see everything that's going on and all the daemons that have started in your sandbox. So, I think that's the easier approach. dbus-launch works as well. I think it's lower level. So, you can make it work. But it's maybe more difficult than you need for testing.

Okay. Are there any more questions?

>> I think that we can answer the next one as well. The last one.

Sam Thursfield: Yeah. Do you know where those 20 minutes went -- where the 20 minutes were spent when creating the VM with BuildStream. More or less, yes. So, there were two reasons it's slow. One is that the VM image is 5 gigabytes. So, you simply can't create an OSTree delta over a 5 gigabyte image without it taking some time. And I think that's not -- well, what we need is faster hard drives or a smaller VM. Ultimately. There is also some inefficiency. I think at the moment, BuildStream is creating the new image and then copying it somewhere. And then creating the delta. So, that can be optimized. It could be done in a single step. And I imagine that would double the speed. But I think as long as we're working with a 5 gigabyte VM image, we're gonna struggle to make it updated in a matter of seconds. Because the scale is just too big.

So, that's why I think toolbox is a good approach. All right. Sorry. Let's finish. Thank you again.

>> Thank you. Thank you, Sam, for joining us.