Modernizing Desktop
Linux Development

Christian Hergert
Principle Software Engineer

📧 chergert@gnome.org
🐦 @hergertme
Desktop Linux Development

• Mostly in C/C++
• Build systems from last millennium that are a mess
• Every desktop deployment is bespoke and developers throw their hands up and close -EWORKSFORME
• Apps are shipped on OS release cycle, lucky if we get security updates at all after initial release
• Debugging in the wild is nearly impossible
• We value independence which can cause fragmentation
Why Modernize?

• Potential contributors skip past us after it’s too difficult to setup workstation for development
• We need new lifeblood to keep desktop Linux going and new contributors have more expectations
• Our current level of fragmentation makes it hard to write polished software and results in more bugs
• Training contributors is complex and time consuming
• Be more efficient with our time so each of us can do more
Can we...

• Maintain independence while reducing fragmentation?
• Separate applications from the OS without losing long-term stability?
• Empower software vendors to ship better software to more users in less time?
• Be a playground for ambitious ideas without sacrificing stability of the platform?
Why is it hard to contribute?

• Lacking or missing documentation
• Dependency incompatibility for application vs platform or even between two separate applications
• Wide-scale app distribution is hard and costly, so most don’t and certainly not across every distro
• Workstation setup is a really high bar for newcomers
• Fragmentation leaves contributors confused about which and what platforms to support and to what degree
Why is it hard to contribute?

• Shifting and different priorities between projects
• Systems complexity and overlapping concerns between unrelated projects
• Lack of training materials
• Too many technologies to learn at once
What makes setup difficult?

• New contributors often ask what Linux distribution to use so that they match others setup, out of frustration

• Dependencies for development outpace stable OS

• Bespoke OS deployments result in many WFM bugs

• Getting working stack traces often puts people at odds with distribution compiler flags
Has the ecosystem changed?

• Continuous integration
• Valgrind/ASan/TSan/UBSan more ubiquitous
• Git, Gitlab, Github have by-and-large won
• Containers are everywhere
• Programming language diversity
• Device and interface diversity (laptop, desktop, mobile, IoT)
• Cross-toolchains generally work now
• GL/EGL/Vulkan can now be relied on
What changes can we expect?

•Immutable base OS with reliable OS updates (ChromeOS and Silverblue already here, albeit different designs)

•Containers for applications becomes almost necessary

•Stricter sandboxes for user privacy and security

•Even more kinds of computing devices (possibly at the expense of generalized computing devices)

•New models of privileged access (less sudo, more implied access via workflow)

•More language diversity as OS vendors are less involved
So what is this Builder thing?

• Great app with a terrible name (my fault, sorry)
• Likely the first container-native IDE
• Memory conscientious (written in C, read-only mmap() search indexes, fancy data-structures when appropriate)
• Every API of consequence is asynchronous by design
• Fast b-tree and rope-based text editor with overview maps and buttery smooth scrolling, even on HiDPI systems
• Integrated UI designer based on Glade
So what is this Builder thing?

- Integration points can be extended with plugins written in C/C++, Python, or Vala. Rust support is not far away.
- Unit test integration for build systems
- Debugger integration (currently just gdb, but language specific debugger plugins are welcomed)
- Profiler based on the perf-based Sysprof profiler
- Completion, diagnostics, fix-its
- Terminal access to host, build environment, or runtime
So what is this Builder thing?

• 10 build systems supported and counting
• Easy installation of Builder via Flatpak in two clicks
• Manage developer SDKs to simplify system setup – which can now be done in minutes rather than evenings
• Code-indexers with lightening fast fuzzy-search
• Non-opinionated on developer’s language choice
• Multi-monitor support
• Semantic indentation, code-formatting, highlighting
So what is this Builder thing?

- Code execution abstractions to allow for cross-architecture execution (qemu-user-static with flatpak supported natively)
- Device abstractions to quickly setup cross-compilation
- Deployment APIs for plugins to support execution on non-local systems
Improving Builder
Improves the Platform
Is contributing easier?

• Preserves independence: bring your own distribution, thanks to Flatpak
• Easy install from gnome-software, flathub.org, or an app market near you
• Getting a shared toolchain is automatically handled for you and your newcomers
• One-click to clone common GNOME apps. One more to build, run or debug
• Reducing time to first patch boosts contributor funnel
What challenges come with a container native IDE?
Challenges - PTY

• Sharing pseudo-terminals across pid namespaces is tricky
• FD passing PTY outside of direct fork()/exec() requires extra setup for controlling PTY (TIOCSCTTY ioctl)
• Some shells like ZSH want to coordinate between instances and that breaks with multiple pty-namespaces
• Guessing the users preferred shell requires checking passwd/getent on the host
• Users expect a shell for their host system, but also for the build container, runtime container, etc
Challenges – FD Passing

- FD passing is a convenient way to share information between processes, particularly when shared memory is not an option.
- If you cross pid/mount/network namespaces, you can exec() processes and still have access to stdin/out/err.
- IPC with container for high-bandwidth communication.
- Podman recently added FD passing support for us.
Challenges – Path Translation

- Different tooling may need to access files outside of container (symbols, debugger sources, etc)
- Paths inside the container and outside the container may differ (or even clash with host system)
- Some additional work could be required to translate based on path such as /usr vs /app in Flatpak
- Tooling that runs in the IDE may use different paths than tooling that runs inside build container
Challenges – Debugging

• Currently we use gdb, but more backends will be supported in the future. That requires some abstraction from the start.

• Passing FD for PTY to be used by inferior

• Control gdb using primary PTY (using gdbwire+MI2)

• But where does gdb binary come from

• Where do application symbols come from, and does the DWARF data (including absolute/relative paths match?)

• How about `__FILE__` and other paths to sources?
Challenges – Profiling

• Currently we use Sysprof, another project of mine based on Linux-kernel perf

• Symbol access has similar problems to gdb, we have to resolve that based on project information

• Resolving functions based on instruction-pointer+ELF still requires further resolution when symbols in external file

• Containers usually have PTRACE/perf disabled, so we need coordination from the host (sysprofd in our case)

• Someday, we want to provide sysprofd from Builder flatpak
Challenges – Execution

• User namespaces should be usable, but lots of CVE means that it’s often disabled by default on major distributions

• Suid helpers can help work around this (bubblewrap) but not after usersns capability is dropped

• Flatpak session helper allows Builder to execute programs available on the host (bash, getent, flatpak, etc)

• Can chain features to also get podman-exec, jhbuild-run, and whatever is next
Language Servers
Language Servers – The Good

• Convenient way to share code between IDEs and Editors

• Allows tooling to run in container matching the build environment which can simplify server greatly at cost of complicating build environment

• When combined with Flatpak SDK extensions, we can get automated setup of complex languages out-of-the-box. Relying on user-setup would basically guarantee nobody will use them.

• The protocol itself is reasonably good
Language Servers – The Bad

• JSON is laughably inefficient when dealing with large data sets like clang completion results, especially when client-side filtering is required

• JSON parsers can result in memory fragmentation unless you are very careful, resulting in lots of small strings on the heap or more fragmentation w/ unpredictable lifetimes

• Builder’s internal language servers optionally use GVariant for zero-parse structures and good memory slice reuse. Also can reference strings inline from the message

• Many of the language servers out there are brittle at best
SDKs
SDK Management

- Flatpak and OSTree based
- SDK extensions allow for extra build components that are useful, but specialized, thereby keeping base SDK smaller
- Containerized (using flatpak/bubblewrap)
- Toolchains, libraries, debuggers, build systems, all shared by your development team, completely eliminating complex system setup
- Incremental SDK updates using OSTree diffs/static-deltas
- GNOME Software can keep SDKs up-to-date too
OSTree

• Like git for binaries
• Incremental updates by diff of 2 versions
• “static deltas” provide optimized, pre-compiled diff between two versions for very fast downloads. Typically generated for N-1, N-2 releases
• Content addressed for automatic deduplication
• Hard-link farm design means low-overhead and works on any POSIX compliant file-system (but better w/ reflink)
• File-system powering Fedora Silverblue, Atomic
Flatpak

- Sandboxed (using bubblewrap, now shared with other tooling)
- Using lots of the new container technology such as pid, mount, network, and user namespaces
- OSTree based to gain all the benefits designed for large scale container deployments
- Runtime vs App split keeps download overhead low, helps share and reduce CVE/Security burden beyond distros
- Portals for safe escalation of access from a sandbox
- Usable on top of read-only base OS, live-cd, etc
Flatpak SDK extensions

• SDKs are really just runtimes before the includes, libraries, and build toolchains are removed

• Most if not all the objects shared with runtime will be de-duplicated automatically by OSTree

• SDK extensions allow specialized tooling to be integrated into the mount namespace (golang, java, mono, etc)

• Great place for language servers so that you don’t rely on application manifest or host system to provide them
Flatpak SDK extensions

• Builder automatically discovers and installs them from configured flatpak repositories

• Reproducible builds are closer due to shared toolchains and other bits that can differ between Linux-based OS
Plugins
Builder Extension Points

- Application (singletons)
- Workbench (per project)
- Workspace (per window)
- Editor Page (per buffer view)
- Runtimes (container support)
- Pipeline (hook into build phases)
- Build Configurations
- Run Handlers (how/where to exec)
- Completion, Diagnostics, Fix-its
- Symbol Resolvers
- Debuggers
- Buffer Addin (per-buffer)
- Search Providers
- Project Tree
- Devices
- Unit Test
- Build Targets
- Preferences
- Frame (per page stack)
- Session save/restore
- Commands
- Refactoring (renaming)
- Toolchains (sysroots)
- Hover tooltips
- Project Templates
- Semantic highlighters, formatters
- And much more…
Writing Builder plugins

• You can write in C/C++, Vala, or Python (with Rust around the corner)
• 2 files necessary
  • my_plugin.plugin — Plugin Metadata
  • my_plugin.py — Plugin implementation
• Subclass and implement plugin interfaces by overriding the virtual functions
Writing Builder plugins

```python
from gi.repository import GObject, Ide

class MyBufferAddin(GObject.Object, Ide.BufferAddin):
    def do_file_loaded(self, buffer, file):
        print("Loaded file", file.get_uri())
    def do_save_file(self, buffer, file):
        print("Saving file", file.get_uri())
# ...
```
Demo
What features can we expect?

• More language servers (likely provided via flatpak)
• Simulators API for plugins (maybe for Librem5, GNOME developer builds, etc)
• More container support (podman coming soon)
• Maintainer tools and project management
• Version control extensions (repo browser, gitlab, commits)
• Database integration
What features can we expect?

• Device emulation (built upon simulator and cross-arch build pipelines)
• Deployment (push releases to flathub, devices, etc)
• Improved documentation access
• Tighter device integration (Librem5, etc)
• More features moved out-of-process for resilience
Questions?
Come Join Us!

https://wiki.gnome.org/Apps/Builder

@BuilderGNOME
irc://irc.gnome.org/#gnome-builder