Debusine as Debian package build infrastructure

Debusine has been designed to run a network of generic “workers” that can perform various “tasks” producing “artifacts”. Interesting artifacts that we want to keep in the long term are stored in “collections”. Collections can also be used to store JSON data instead of artifacts.

While tasks can be scheduled as individual “work requests”, the power of debusine lies in its ability to combine multiple (different) tasks in “workflows” :

- Each workflow has its own logic to orchestrate multiple work requests across the available workers, it can even embed other workflows
- Output artifacts from one work request can become input artifacts of another work request
- Work requests can be run in parallel or have dependencies between them so that they run in the correct order
- Output artifacts can be stored in collections
- The structure and behavior of each workflow can be altered with parameters, but also from external database (like a manually maintained “collection” of JSON data)

A debusine instance can be multi-tenant, divided into “scopes” of users and groups. These contain “workspaces” that have their own sets of artifacts and collections. A distribution like Debian would probably host itself in a single “scope” with multiple cooperating workspaces.

The central debusine server hosts the database and API, drives the workflows forward by scheduling work requests on workers. Workers come in several types. Package builds happen on “external” workers that have no privileged access to debusine databases. These can be hosted anywhere but have to be approved by debusine administrators. There are other more specialized workers for special purposes: internal workers deal with tasks that need direct database access (e.g. mirroring a remote repository inside debusine), signing workers manage private keys, key generation and signing operations.

More info about the core concepts here:
https://freexian-team.pages.debian.net/debusine/explanation/concepts.html

Some of the important artifacts and collections

In the context of a package build, there are only a couple of artifacts and collections that you should be aware of:

- Artifact **debian:source-package**. A .dsc and its associated files.
  https://freexian-team.pages.debian.net/debusine/reference/artifacts.html#category-debian-source-package

- Artifact **debian:binary-packages** and **debian:binary-package**. The former is all .deb files produced by a specific package build. The latter are the same files but managed individually. Deduplication at the filestore level makes this a non-issue. https://freexian-team.pages.debian.net/debusine/reference/artifacts.html#category-debian-binary-package
- Artifact **debian:upload**. A .changes file and its associated files. https://freexian-team.pages.debian.net/debusine/reference/artifacts.html#category-debian-upload
- Artifact **debian:package-build-log**. The log file of the package build process. https://freexian-team.pages.debian.net/debusine/reference/artifacts.html#category-debian-package-build-log
- Artifact **debian:system-tarball**. This artifact contains a tarball of a Debian system. In the context of the package build, it is used to provide the build chroot in which sbuild will invoke the package build process. https://freexian-team.pages.debian.net/debusine/reference/artifacts.html#category-debian-system-tarball
- Collection **debian:environments**. This collection stores all the debian:system-tarballs artifacts for all combinations of suite and architectures that we care about and is used to retrieve the desired “runtime environment” for each work request. https://freexian-team.pages.debian.net/debusine/reference/collections.html#category-debian-environments
- Collection **debian:package-build-logs**. This collection stores historical package build logs.

About the sbuild task

The sbuild task is the main task used to build binary Debian packages out of a Debian source package. It takes many parameters letting you control among other things:

- The build backend to use (defaults to “unshare”, but “incus-vm”, “incus-lxc” and “qemu” are also supported)
- The chroot tarball in which the build process will be run
- Additional packages to make available (sbuild --extra-package)
- Additional repositories to enable (sbuild –extra-repository, currently not implemented)
- The host architecture
- The components to build (arch any, arch all, source package)
- The build profiles to enable
- The build options to set
- Whether the build is an binNMU and associated parameters (arbitrary suffixes are supported, not only +bX)

Thus direct usage of the sbuild task lets you experiment with rebuilds with non-standard parameters and non-standard build environments. Official builds would likely happen through a workflow that forces most parameters to their desired values for an official build and where the user would only be able to supply the source package and the target distribution.

More information about the sbuild task:
https://freexian-team.pages.debian.net/debusine/reference/tasks.html#sbuild-task

But most parameters are inherited from the generic PackageBuild task:
https://freexian-team.pages.debian.net/debusine/reference/tasks.html#task-packagebuild

About the sbuild workflow

The purpose of the sbuild workflow is only to run a package build across multiple architectures and to store the generated build logs. In most cases, it will not be used standalone but as a sub-workflow of a more complex workflow that will perform more things like running tests and uploading the built packages somewhere else.

Note: Setting up workflows is a privileged operation within a workspace, but starting workflows is available to all users who are members of the workspace containing the workflow. While setting up a workflow, you decide what workflow parameters are set in stone and which one can be set by the users.

The main duty of the sbuild workflow is to figure out the set of architectures where the package build will be scheduled. It computes it by intersecting different sources of information:

- The list of architectures provided as input parameter
- The list of architectures defined in the source package
- The list of architectures accepted by the target distribution (not implemented yet)
- The list of architectures with available workers (not implemented yet)

Then all the builds are scheduled in parallel. A temporary entry is immediately added to the debian:package-build-logs collection (if one has been provided in the parameters) to inform users that a package build is in process and that some log will show up there shortly. When a build finishes, its package build log is really added to the collection.

Note that this workflow can be used for initial package builds but also for binNMU.

More information about the sbuild workflow:
https://freexian-team.pages.debian.net/debusine/reference/workflows.html#workflow-sbuild

Signing and uploading packages

Debusine has its own signing infrastructure with dedicated (restricted) signing workers. It is able to sign .changes and .dsc, as well as UEFI binaries for secure boot. It can use keys generated and stored in hardware tokens, and keys stored in a local private database.

This means that packages are not signed by the worker that built the package, but by a separate restricted signing worker, as part of an explicit signing task that runs in a second step (in a larger “debian-pipeline” workflow combining the sbuild workflow and the package-upload workflow).

Debusine also supports “remote signature”, i.e. the user who initiated the workflow can provide the required signatures by running “debusine provide-signature” on the computer having the required GPG key.

The package-upload workflow is a server-side reimplementation of dput supporting sftp and ftp uploads, allowing Debusine to upload into an existing distribution archive.

What comes next

Everything that has been presented so far is already available (or close to it). While the features are available, we are still lacking some management tools, some glue between all those features, and proper ACL to control most sensitive operations (in particular access to signing keys) so it is too early to consider deploying debusine in production for Debian’s main workflows.

The rest of this document presents planned tasks and/or ideas that we could work on if there is interest.

Managing packages with special requirements

We have designed a mechanism where administrators can maintain a collection of build configuration with default values and overrides for all the workflow and task parameters. Those values can be set at the global level, the suite level, the package level, the package-in-suite level.

This means one should be able to configure tweaks like those:

- For jessie package builds, use the “schroot” backend.
- For the package linux, always use a big worker.
- For the package grub in jessie, never try to build on riscv64.
- For the package silo, only try to build on sparc and sparc64.
- For the package rust23 in forky, build with build_profile=stage1.
- For bullseye and bookworm, use chroot tarballs with variant=buildd. For others, use chroot tarball with variant=essential.
- Etc.

More information about the current design and plans:

- https://freexian-team.pages.debian.net/debusine/reference/devel-blueprints/task-configuration.html
- https://freexian-team.pages.debian.net/debusine/reference/devel-blueprints/build-instructions.html#control-workers-assigned-for-specific-packages-tasks

Managing trusted workers

Workers have a set of metadata defining which tasks they’ll take on. By default all workers will execute all tasks that they have the ability to, that is if they have the correct build backend

dependencies and support the host_architecture of the task. We plan to be able to provide more restrictions on which tasks can run on which worker, e.g. to avoid leaking security builds or to avoid running untrusted code on distribution workers.

This will leverage the same tag-based mechanism designed to assign work requests with special requirements to workers fulfilling those requirements (cf point above).

Another approach is to leverage the stronger isolation provided by some backends. Running builds or tests of untrusted code could be acceptable on trusted workers provided that they use a qemu backend that ensures full isolation between the worker's host environment and the actual task performed by the work request.

Dep-wait mechanism

We ran out of time to implement a proper dep-wait mechanism this year. However we implemented a mechanism to retry failed builds over configurable time periods :
https://salsa.debian.org/freexian-team/debusine/-/issues/497
https://freexian-team.pages.debian.net/debusine/reference/workflows.html#retry-with-delays

We also have the possibility to manually retry a work request that failed.

In the future, we would like to have something smarter where a failed build due to uninstallable build dependencies would only be retried when the build dependencies are satisfiable:
https://salsa.debian.org/freexian-team/debusine/-/issues/460

In general, we want workflows to complete: if the build dependency does not become satisfiable on a given architecture, the corresponding build will be marked as failed instead of waiting indefinitely for it. To compensate for this design choice, we can imagine to create a "ScheduleMissingBuilds" workflow that would run dose-builddebcheck on all (source package, architecture) where there's no build and schedule the builds if the build dependencies are currently satisfiable. That workflow could be scheduled to run on a regular basis (e.g. weekly).

Building arch all packages on a specific architecture

We want to extend the sbuild workflow to be able to indicate on which architecture arch-all package builds are expected to happen. Currently it's forced on amd64 but with a new field and the above possibility of overriding fields for each source package, we have a workable solution.

See plans in https://salsa.debian.org/freexian-team/debusine/-/issues/511

In the future, it would be trivial to improve the workflow to extract the value from a field in the source package (for instance XS-Build-Indep-Architecture that Ubuntu uses).

Possible ways to use debusine as Debian's buildd

Our ideal workflow is that Debian developers upload to debusine and trigger a workflow that builds packages, runs tests, and when the developer is satisfied with the results, then the built packages are directly integrated in unstable. This will soon be possible from debusine.debian.net (using Debusine to build and test binary packages, and then uploading the source package to Debian, without Debusine's binaries).

We would love to work towards full migration of package building to debusine, but we are fully aware that we need to support this new workflow in parallel with the current workflow to help convince people first. Hosting builds in debusine means we need to trigger some specific workflow every time that a new source package appears in Debian unstable (or other suites accepting uploads).

The easiest solution would be to implement the "ScheduleMissingBuilds" workflow introduced above and run that every 15 minutes or so. It would be smart enough to be aware of currently running package builds and not reschedule those.

But if we want something more reactive, we could also build something custom where the workflow is started remotely through the API calls. I do not know exactly how dak and wanna-build interact at this level and whether that's something possible/desirable.

Questions / feedback welcome

We would love to work with you to make experiments on debusine.debian.net and gather your feedback on what would be required for debusine to become a viable alternative to the current infrastructure.