Overview

Upgrading FoundationDB can be a challenging process. FDB has an internal wire protocol for communication between server processes that is not guaranteed to be stable across versions. Patch releases for the same minor version are protocol-compatible, but different minor versions are not protocol-compatible. This means that when you are doing a minor version upgrade, you need to upgrade all of the processes at once, because the old and new processes will be unable to communicate with each other. `fdbcli` uses the same wire protocol, so you will need to use a version of `fdbcli` that matches the version of FDB that is running at the time.

Additionally, clients must have a client library that is protocol-compatible with the database in order to make a connection. To avoid client outages during upgrades, you must install both the old and new client libraries, using FDB's multi-version library feature to load both library versions at the same time.

Despite these challenges, it is possible to build a safe, zero-downtime upgrade process for FoundationDB. This document will describe that process, using an upgrade from 6.1.12 to 6.2.8 as an example. This process assumes that you are running `fdbserver` through `fdbmonitor`, and that you have the capability to install new binaries and new config files into the environment where your processes are running.

Upgrade Process

The high-level upgrade process is:

1. Install the new fdbserver binaries alongside the old binaries, with each binary in a path that contains its version. For instance, you might have the old binary at `/usr/bin/fdb/6.1.12/fdbserver`, and the new binary at `/usr/bin/fdb/6.2.8/fdbserver`.
2. Update the monitor conf to change the fdbserver path to `/usr/bin/fdb/6.2.8/fdbserver`.
3. Using the CLI at version 6.1.12, run the command `kill; kill all; status`.
4. Using the CLI at version 6.2.8, connect to the database and confirm that the cluster is healthy.

Handling Client Upgrades

To ensure that clients remain connected during the upgrade, you should use the multi-version client. The recommended process for managing client libraries is:

1. Install version 6.2.8 in a special folder for multi-version clients. For instance, `/var/lib/fdb-multiversion/libfdbc_6.2.8.so`. You should include the version in the filename for the multiversion libraries to make sure you can support as many as you need to have, and to help with debugging.

2. Set the `FDB_NETWORK_OPTION_EXTERNAL_CLIENT_DIRECTORY` environment variable to `/var/lib/fdb-multiversion`.
3. Bounce the client application.
4. Use the JSON status from the database to confirm that all clients have compatible protocol versions. You can get this client information in `cluster.clients.supported_versions`. That will hold a list of every version supported by any connected client of the database. Each version entry will hold the client version, the protocol version, and the list of clients that are using that client version. You can get the protocol version for the new version of FDB by running `/usr/bin/fdb/6.2.8/fdbcli --version`. To confirm that the clients are ready for the upgrade, check that for every client address that exists for any client version, there exists an entry under a client version whose protocol version matches the new version.
5. Run the server upgrade steps above.
6. Once the database is running on the new version, you can update the clients to use `6.2.8` as the main client library version, and remove any older client libraries that you no longer need.

Steps 1 through 3 can be done at any point before the upgrade of the server. You may want to have your client applications include new versions of the FDB client library as part of there normal build and deployment process, so that you can decouple the upgrades of the clients and the servers. It is generally safe to have clients use multiple client libraries, and if you encounter any issues with that it may be easier to debug them as part of the normal process for updating the client application.

Upgrading fdbmonitor

The upgrade process above does not restart fdbmonitor, so it will continue running at the old version. This is generally not a problem, since fdbmonitor does not change with every release, but you may want to get it running on the new version for the sake of consistency in your configuration. Once you have the database running at the new version, you can upgrade fdbmonitor as a follow-on task. You should note that restarting fdbmonitor will also restart fdbserver, and depending on how you are upgrading fdbmonitor it may take longer for the processes to come back up. You may need to do a rolling bounce of your fdbmonitor processes to make sure that you maintain availability.

Other Binaries

The fdbbackup and fdbdr binaries also must be protocol-compatible with the running version of the database. The process for upgrading those binaries will depend on your infrastructure and your orchestration tooling. You should be able to run and upgrade those processes through the same process you

would use for any other application. This will create a gap between when the database is upgraded and when the backup and DR binaries are upgraded. This will produce a temporary lag in backup and DR. Once all of the components are running on the same version, the backup and DR will catch up.

Additional Notes

To ensure that fdbmonitor does not kill the old processes too soon, you should set `kill_on_configuration_change=false` in your monitor conf file.

If fdbserver processes restart for organic reasons between steps 2 and 3 in the upgrade, they will not be able to connect to the rest of the cluster. If this happens to a single process, then you should be able to kill the remaining processes through the CLI, and the process that restarted early will be able to connect. If this happens to enough processes, it can take the database unavailable, and you won't be able to kill processes through the CLI. If this happens, you can restart all of the fdbmonitor processes to bring everything up on the new version. We recommend minimizing the gap between steps 2 and 3 to help mitigate this risk.

This process of installing new binaries while the process is still running can present additional challenges in containerized environment, but it is still possible, as long as the deployment system allows making changes to running containers. While this can violate goals of container immutability, it is only necessary during the upgrade itself. Once the upgrade is complete, you can roll out the new version of the container image through a rolling bounce, through the fdbmonitor upgrade process described above. We have implemented a process like this in our Kubernetes Operator. # Requirements

Our standard development setup uses docker. This means that any Linux distribution can be used as long as docker is available.

Setting up Docker

First install docker through the package manager of your Linux distribution.

Make sure that the docker daemon is running. In most distribution this can be done through `systemctl` (refer to the documentation of your OS to find the exact commands):

```
systemctl enable docker # this will make sure that the daemon is started at boot time
systemctl start docker # start the docker daemon
```

Additionally you need to give your user permissions to run docker containers. You can do this by adding your user to the group `docker` (on certain distributions, like CentOS 7, this group has a different name. For example on CentOS 7 you need to add your user to the group `dockerroot`). You can do this by running the following command:

```
sudo usermod -aG docker $USER
```

After that you should log out and then log in again to. Then you should be able to run docker as a user. If you still need to setup a `bin` directory, you can hold off with login out and back in until you have done that.

Set up a bin Directory

We use shell scripts to wrap our build commands. These should be in your `PATH` so that you can run them from any location within your home directory.

First make sure `$HOME/bin` exists:

```
mkdir -p $HOME/bin
```

Then add the following line to your `.bashrc`, `.profile`, or `.zshrc` (depending on your login shell and configuration):

```
export PATH=$HOME/bin:$PATH
```

Make sure `$HOME/bin` is the first entry in the `PATH`. After this is done you need to log out and back in.

The Wrapper Scripts

The main way we call into docker is a quite long script. Therefore we use the following wrapper script:

```
#!/usr/bin/env bash

set -e

if [ -t 1 ] ; then
    TERMINAL_ARGS=-it `# Run in interactive mode and simulate a TTY`
else
    TERMINAL_ARGS=-i `# Run in interactive mode`
fi

docker run --rm `# delete (temporary) image after return` \
    ${TERMINAL_ARGS}  \
    --privileged=true `# Run in privileged mode ` \
    --cap-add=SYS_PTRACE \
    --security-opt seccomp=unconfined \
    -u $(id -u):$(id -g) \
    -v "${HOME}:${HOME}" `# Mount home directory` \
    -v /etc/passwd:/etc/passwd:ro \
    -v /etc/group:/etc/group:ro \
    -w="$(pwd)" \
```

```
    -e CC=clang \
    -e CXX=clang++ \
    -e BOOST_ROOT=/opt/boost_1_72_0 \
    -e USE_CCACHE=ON \
    ${ccache_args} \
   foundationdb/devel:centos7-latest scl enable devtoolset-8 rh-python36 rh-
ruby27 -- "$@"
```

If you want to set additional environment variables in your docker environment you can add them to the command using the following syntax:

```
     -e NAME=VALUE
```

Copy the above script and paste it into a new file. Save this file as `$HOME/bin/fdb-dev` and make it an executable:

```
chmod +x $HOME/bin/fdb-dev
```

Additionally we recommend using `clangd` for development.`clangd` is also installed in the docker container - but in order to use it you need an executable. This is easy to do: create a new file and paste the following content into it:

```
#!/usr/bin/env bash

set -e

fdb-dev clangd
```

save it as `$HOME/bin/clangd`, and make it an executable:

```
chmod +x $HOME/bin/clangd
```

Congratulations, your FoundationDB development machine is now setup. The first time you call into `fdb-dev`, docker will fetch the image. This will take a few minutes and you need to be online for this to succeed.

Building FoundationDB

To build fdb you don't need to start a shell in the docker container (although you can by calling `fdb-dev bash`). The following script should check out the code and build fdb

```
cd $HOME
git clone git@github.com:apple/foundationdb.git
mkdir build
cd build
fdb-dev cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DUSE_CCACHE=ON ../foundationdb
fdb-dev ninja
```

Note: If you get an error like `ld.lld: error: undefined symbol: std::__once_callable` set `FORCE_BOOST_BUILD` to `ON` in CMakeCache.txt

Note: If you get an error like `WARNING: IPv4 forwarding is disabled. Networking will not work.` restart the docker daemon.

Note: Sometimes the cmake step can potentially run into a bizarre error. Turned out it's caused because of clock skew. Fix the clock with `sudo hwclock --hctosys` if you have clock skew.

Ccache

The version of ccache currently shipping in the `foundationdb-dev` image (ccache v3.1.6) defaults to a maximum of 1GB of files stored in the cache. It is recommended to increase this size as much as possible for faster builds. Run the following to increase the cache size to 100GB:

```
$ fdb-dev ccache --max-size 100G
```

You can also view ccache stats by running `$ fdb-dev ccache -s`.

Setting up a Development Environment

We use `clangd` for code navigation and completion. `clangd` is a language server for C++ and can be used by many editors (Emacs, Vim, Atom, etc). This section describes how to set up Visual Studio Code - this is one of the easiest editors to set up.

After installing VS Code you have to install the `clangd` extension. After start VS Code click on the extension tab:

In the search box type `clangd` to search for the extension. You should get exactly one result. After clicking on `Install` you are all done:

Now you should be open to open a folder (`File→Open Folder...` or press `Ctrl+k Ctrl+O`) and open the source folder. After opening a C++ file, you should be able to navigate code and use auto completion (it might take a while for all features to work, as `clangd` will need to index the code).

Warning: The first time you open a C++ file, VS Code will ask you whether you want to install the C++ extension. Don't install those, they will break stuff!

Troubleshooting

If just doing the above doesn't work you will need to make sure that you're including a `compile_commands.json` file in the root directory of your FDB folder.

This file is generated in your `~/build` folder when you run CMAKE with the `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` option (see above).

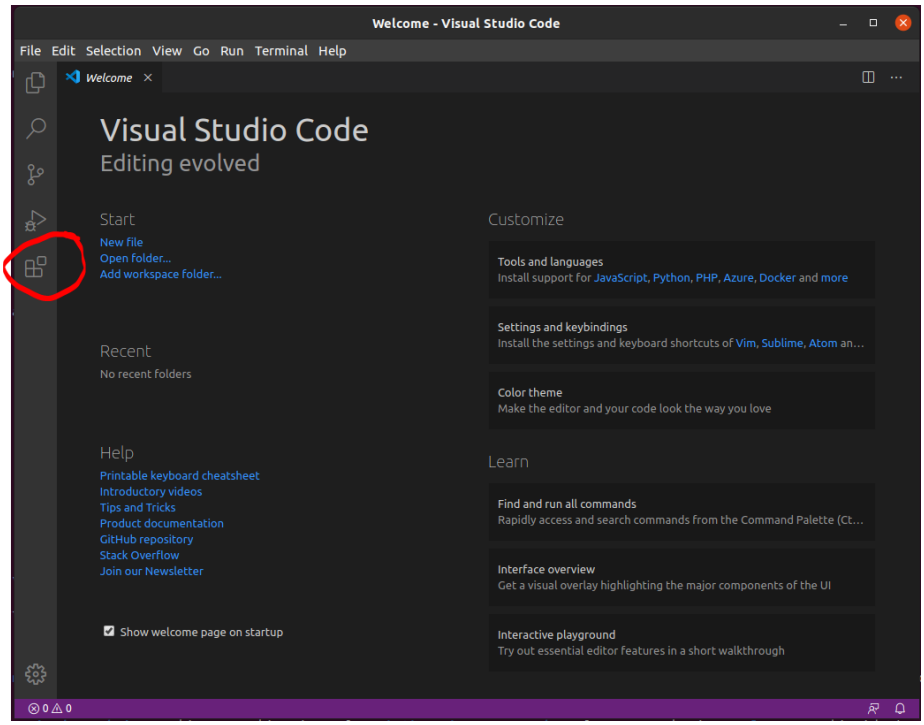

Figure 1: startup

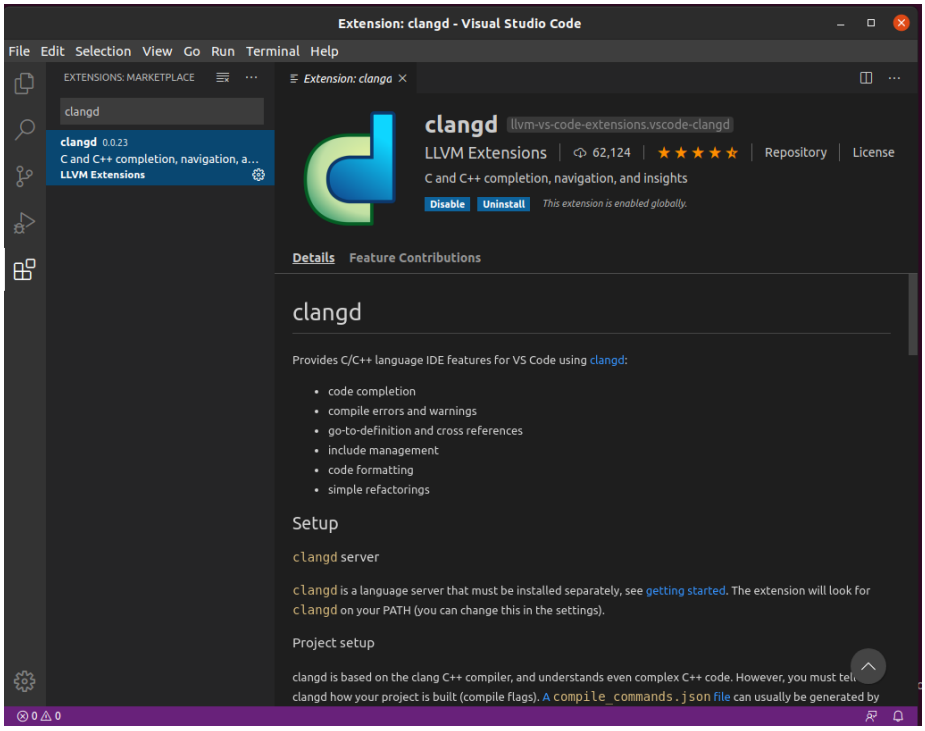

Figure 2: image

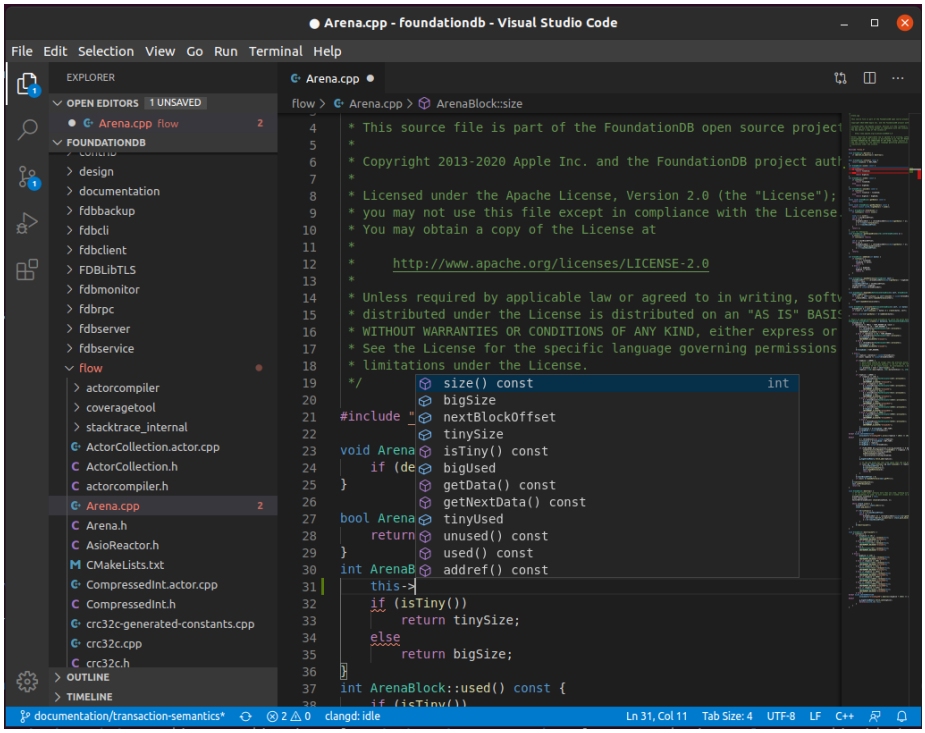

Figure 3: image

You can either run `make processed_compile_commands` (this may not work) to get this file into your local fdb project directory or create a symbolic link to it: `ln -s /home/<user>/build/compile_commands.json /home/<user>/foundationdb/`

Setting up clang-format with clangd on VSCode

Since FDB doesn't use a linter as part of it's CI (yet) its important that developers format their code locally before commiting (using clang-format).

To set up clang-format on VSCode follow these steps (assumes clangd extension is already installed):

1. Go to settings (type ⍰ + ,). In the search bar look for format
2. Set the Editor: Default Formatter to `llvm-vs-code-extensions.vscode-clangd`
3. Check the box Editor: `Format On Save`
4. Make sure Editor: `Format On Save Mode to file`
5. Now each time you save a file it should clang format any changes

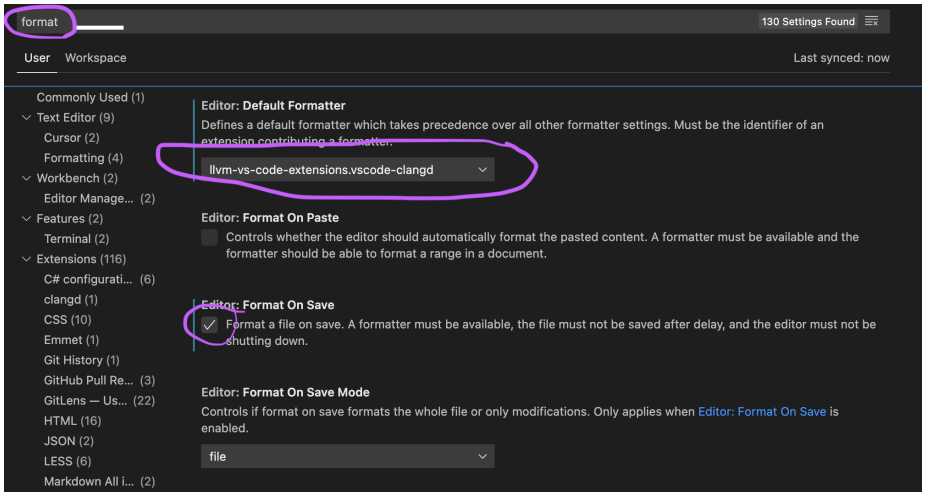

Figure 4: image