BUCK basics and Building gRPC with BUCK

What is BUCK?

- BUCK is a highly efficient build system that is optimized for large projects where parallel build processes are possible
- BUCK is built and maintained by Facebook, it is based off of the design of Google's Bazel (and Blaze) build system
- It is designed to optimize both full and to a greater degree smaller builds

How does BUCK work?

- When a BUCK build is called DAG is created representing the dependencies of various modules
- The build is the performed in a bottom up fashion, building any node with has no unbuilt child nodes
- Already completed build work is cached and unmodified nodes in the graph are **not** rebuilt

How does BUCK handle caching?

- BUCK generates hashes for each rule to determine if the rule needs to be rebuilt
- An optimization exists that tracks the file structure and keeps an updated list of dependencies to be rebuilt
- The output of a build rule is cached, caches can be non-local to allow for greater improvements in aggregate performance

ONOS transition from Maven to BUCK

- Benefits:
  - Use of BUCK has given us order of magnitude performance improvements for incremental builds
  - BUCK caching should enable caching of common builds on a shared server further reducing the build cost on developers
  - BUCK provides finer control of dependencies as it has no concept of transitive dependencies
- Challenges
  - Significantly fewer developed plugins make porting to BUCK harder
  - Pulling in and determining moduledependencies is non-trivial

Dependency management

- In order to include a dependency in BUCK you must specify it in the deps set of the rule requiring it as the output of a rule
- For the common case of inter-module dependencies this means specifying the rule that built the jar you wish to depend upon (e.g. ‘//core/api:onos-api’)
- The more complex case of remote dependencies requires a separate rule to download and export the remote artifact so it can be used

ONOS adaptation of BUCK

- Standard BUCK provides a significant set of build rules designed for packaging java libraries, producing android artifacts and similar tasks
- For ONOS we needed BUCK to create OSGi related files (Manifests etc.)
- In order to do this we have had to develop a number of additions to BUCK including simplified fetching of maven artifacts

Using BUCK in ONOS

- External dependencies:
  - External maven dependencies are pulled in by being listed **//lib/deps.json** using their maven coordinates
  - Maven dependencies are then pulled by a script (’onos-lib-gen’) which uses **deps.json** to generate **//lib/BUCK** which is responsible for creating the remote_jar rules which fetch the dependencies at compile time
  - Once pulled any artifact can be referenced in another BUCK files dependencies as **//lib:***artifact-chosen-name**

Extensibility

- Because not every need can be anticipated BUCK contains a very generic rule called **genrule()**
  - **genrule()** allows the execution of arbitrary shell commands or scripts to generate an output file
  - Using **genrule()** has enabled many of the ONOS rules that have enabled us to transition towards BUCK from the legacy Maven build
  - Generation of OSGi jars, ONOS apps and many other ONOS specific build tasks are achieved using these generic rules

Extensibility

- In addition to **genrule()** we have used bucklets to extend the default behavior of BUCK
  - Bucklets python files that define a number of rules that are used to generate artifacts in ONOS, they can string together one or more existing BUCK rules including other bucklets to generate zero or more desired artifacts and can be references in the same way any other rule is references in buck

Compiling gRPC code with BUCK

- How the grpc_jar rule works:
  - First the rule must have access to binaries of the protobuf compiler as well as the gRPC plugin
  - In order to pull in and mark these binaries as executable two rules are provided fetch_protoc_binary and fetch_grpc_plugin_binary, these rules download the appropriately versioned binaries and mark them as executable for use by the grpc_jar rule
  - Currently these rules must be called with the desired binary versions already pulled into //incubator/grpc-dependencies/BUCK and //incubator/protobuf-dependecies/BUCK

Compiling gRPC code with BUCK

- The **grpc_jar** rule is nearly as configurable as calling the protoc compiler, it provides the ability to specify the following parameters:
  - Standard arguments like *name & deps*
  - *proto_paths* allows the user to specify the set of paths where referenced definitions should be searched and in what order they should be resolved (similar to a classpath)
  - *proto_match_patterns* allows the user to specify a set of patterns that will be run against all files in the domain of this BUCK file, if multiple patterns are specified the sources to be compiled will be the union of all matched file

Compiling gRPC code with BUCK

- *protoc_version* and *grpc_version* specify the version strings of the the protoc compiler and grpc plugin respectively
- ***kwargs* should be used to pass arguments that the user would like passed through to the osgi_jar rule that is used to package the artifacts generated by the protoc compiler

- Notes:
  - Due to how the binaries are pulled the correct sha1 checksum must be added to a dictionary that maps version strings to checksums before it will be possible to pull that binary version

Example time!