Distributed software builds using the REv2 protocol

Ed Schouten <ed@nuxi.nl>

040coders.nl meetup, June 18th, 2020

Timeline (1/4)

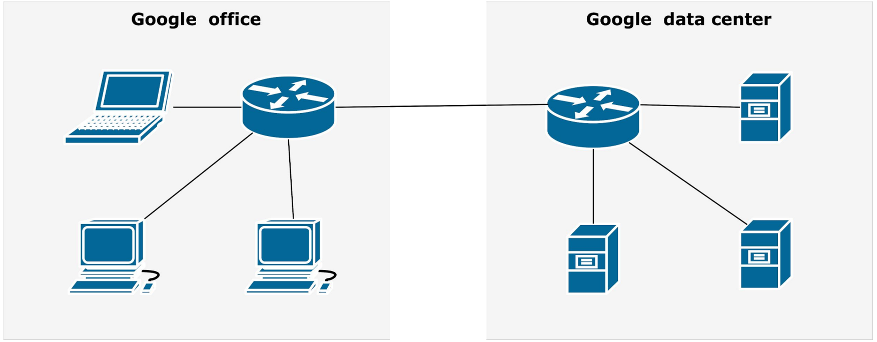
- ~2000: Google has a monorepo with shell script/Makefile build scripts.
  - It turns out that becomes unmaintainable relatively quickly.
- ~2005: Makefiles are replaced with build tool written in Python.
  - Every ‘package’ (directory) contains a `BUILD` file that is `eval()`ed by Python.
  - Directives are Python function calls that are implemented by the build tool.

```
cc_library(
    name = “stringformatter”,
    srcs = [“stringformatter.c”],
    hdrs = [“stringformatter.h”],
)
```

```
cc_binary(
    name = “hello”,
    srcs = [“hello.c”],
    deps = [“:stringformatter”],
)
```

Timeline (2/4)

- ~2010: Blaze: rewrite of Python build tool in Java.
  - Contains a very basic Python interpreter to parse `BUILD` files.
  - `java_*()`, `cc_*()`, `py_*()`, etc. rules are all implemented inside Blaze in Java.
  - Sandboxing: actions only ‘see’ files that are part of their `deps = [...]`.
  - Remote caching/execution: ‘bazel -j 1000’ from behind your desk.



Timeline (3/4)

- 2015: Bazel: tidied up Open Source version of Blaze.
  - Not extensible: mainly just `java_*()`, `cc_*()` and `py_*()` rules.
  - No remote execution: existing version was too Google specific.
- 2015-2020: Many new features appear.
  - Support for platforms other than Linux/x86, and a good notion of cross compilation.
  - Starlark: use a Python-like language to design your own build rules.
  - Support for fetching and source code and build rules remotely (HTTP, Git, etc.).

```
rust_library = rule(
    _rust_library_impl,
    attrs = {
        “srcs”: attr.label_list(),
        “deps”: attr.label_list(),
    })
```

```
def _rust_library_impl(ctx):
    ctx.actions.run(“rustc”, ...)
    return [DefaultInfo(...)]
```

Timeline (4/4)

- Bazel gains support for remote caching and execution.
  - 2017: Initial ‘RE’ protocol was designed by Google.
  - 2018: Community efforts later on led to the release of ‘REv2’.
- Open Source servers that implement RE/REv2 start to appear:
  - 2017: Uber releases Bazel Buildfarm server, written in Java.
  - 2018: Bloomberg/CodeThink release BuildGrid, written in Python.
  - 2018: I started working on Buildbarn, written in Go.
- Other clients that use REv2 start to appear: Recc, Goma, BuildStream, etc.

Goal of this talk: to explain how REv2 works.
Approach: start simplified (and incorrect) and extend onwards.

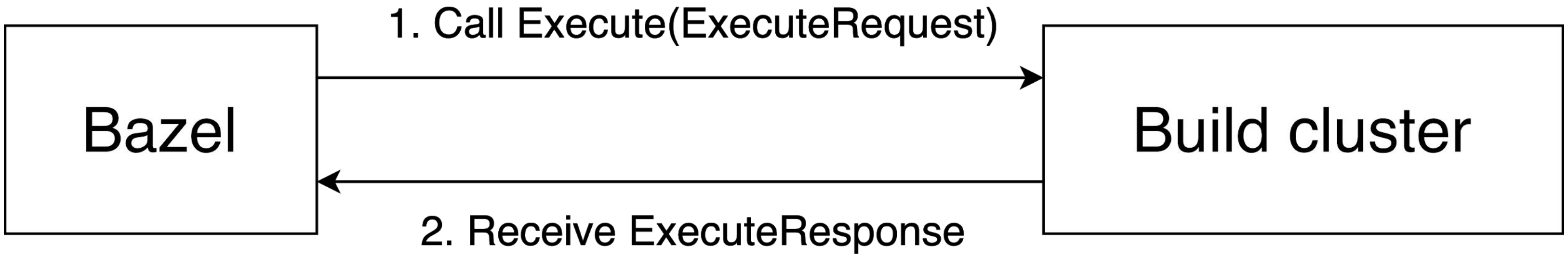
‘distcc/ccache/… did this two decades ago’

- … except that it only works for C/C++ compilation.
    - REv2 supports remote execution of arbitrary UNIX commands.
- … except that it requires that workers have toolchains/SDKs preloaded.
    - REv2 allows clients to upload full SDKs to workers.
    - Workers can be vanilla OS installations.
    - Result: easier to achieve reproducibility of work.
- … except that it only speeds up builds.
    - REv2 can also be used to run unit/integration tests remotely and cache results.

REv2 is not a fad!
It is becoming the de facto standard for remote builds.

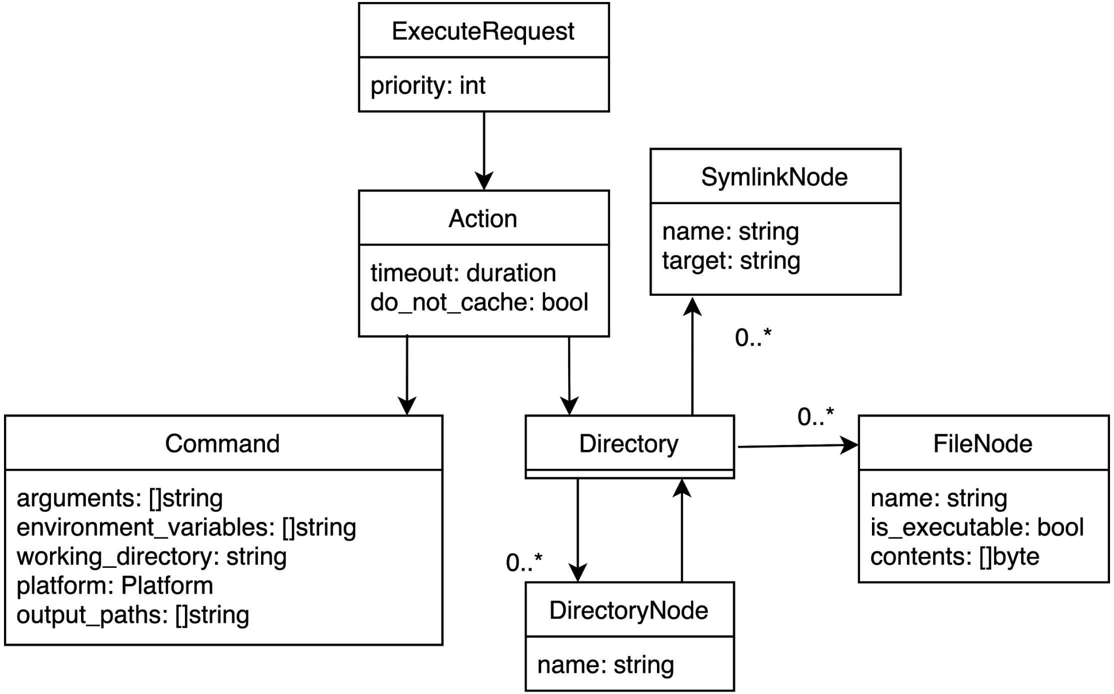
REv2

Remote Execution… simplified

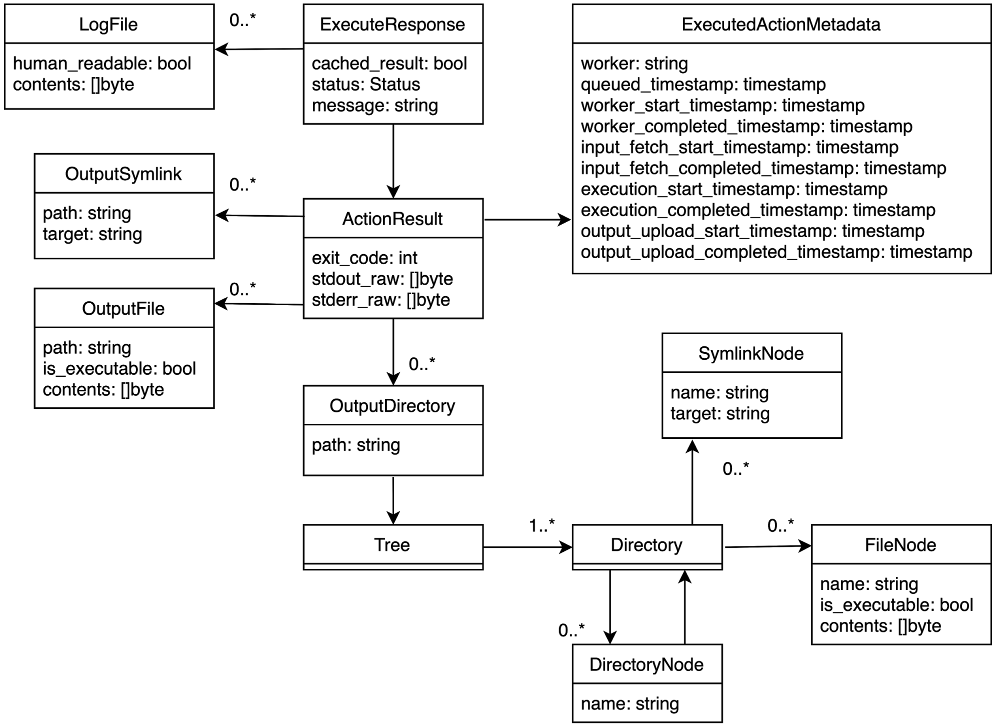


Building a project consists of hundreds/thousands of these calls.

ExecuteRequest



ExecuteResponse



LogFile
human_readable: bool
contents: []byte

ExecuteResponse
cached_result: bool
status: Status
message: string

ExecutedActionMetadata
worker: string
queued_timestamp: timestamp
worker_start_timestamp: timestamp
worker_completed_timestamp: timestamp
input_fetch_start_timestamp: timestamp
input_fetch_completed_timestamp: timestamp
execution_start_timestamp: timestamp
execution_completed_timestamp: timestamp
output_upload_start_timestamp: timestamp
output_upload_completed_timestamp: timestamp

OutputSymlink
path: string
target: string

ActionResult
exit_code: int
stdout_raw: []byte
stderr_raw: []byte

OutputFile
path: string
is_executable: bool
contents: []byte

OutputDirectory
path: string

SymlinkNode
name: string
target: string

Tree

Directory

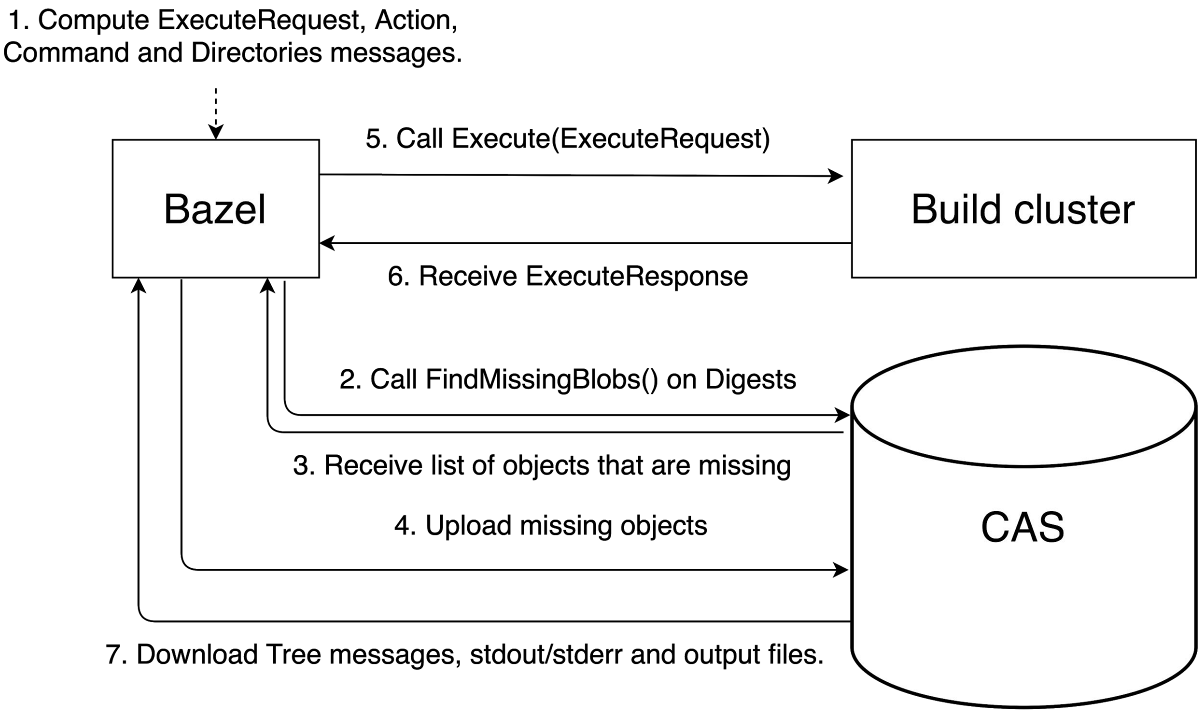
FileNode
name: string
is_executable: bool
contents: []byte

DirectoryNode
name: string

Content Addressable Storage (CAS)

- Problem: ExecuteRequest and ExecuteResponse get big and repetitive.
  - Input roots with SDKs can be hundreds of MBs in size.
  - Build-edit-build cycles create nearly identical ExecuteRequests.
- Solution: place repetitive parts in shared storage.
  - ExecuteRequest: Action, Command, Directory messages and file contents stored externally.
  - ExecuteResponse: Tree messages and (log)file contents stored externally.
  - Use content addressing: objects are identified by a Digest (i.e., SHA-256 + size).
    - Automatic deduplication of identical data.
    - Tamper proof Merkle tree: contents can be validated when loaded.
    - Immutability of data makes caching trivial.
    - (Impossible to maliciously craft cyclic directory layouts.)

Remote Execution with the CAS



1. Compute ExecuteRequest, Action, Command and Directories messages.

5. Call Execute(ExecuteRequest)

6. Receive ExecuteResponse

2. Call FindMissingBlobs() on Digests

3. Receive list of objects that are missing

4. Upload missing objects

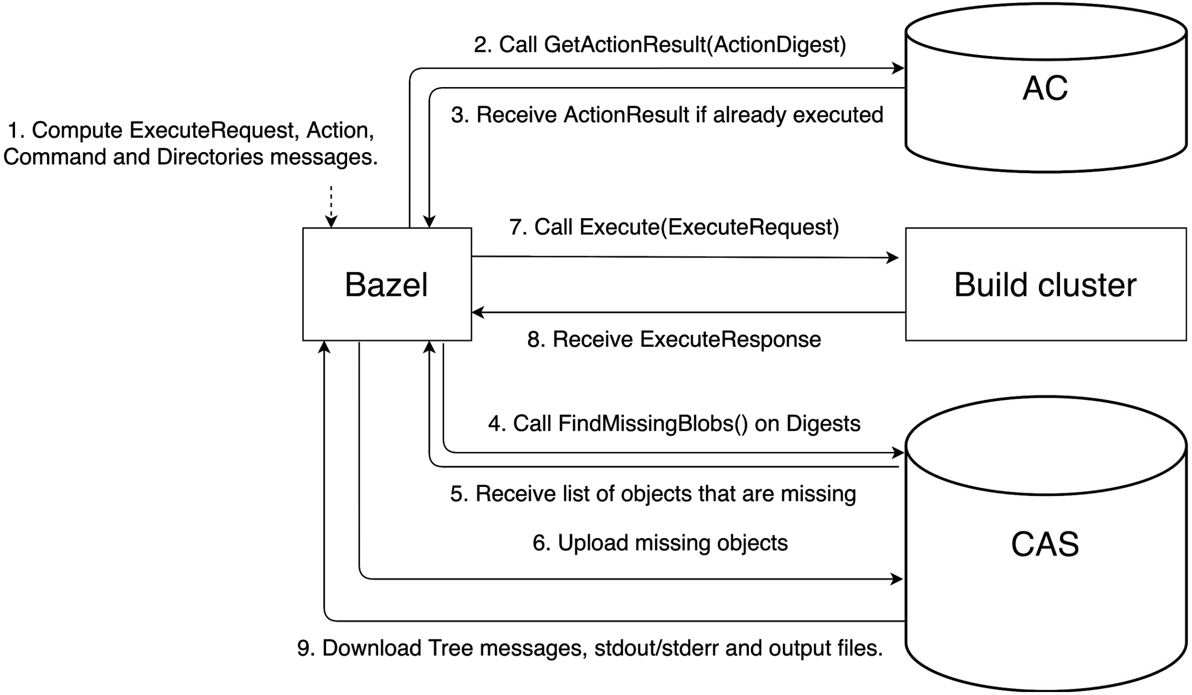
7. Download Tree messages, stdout/stderr and output files.

Note: only communication involving Bazel is shown.

Action Cache (AC)

- Problem: protocol is still expensive when actions are already cached.
  - At least two round-trips: FindMissingBlobs(), zero CAS uploads, Execute().
  - FindMissingBlobs() size grows linear w.r.t. input root file count.
- Solution: let the client first query the Action Cache directly.
  - Key: Digest of the Action.
  - Value: ActionResult.
  - AC is the only part of REv2 storage that can become poisoned.
  - AC size is minuscule compared to the CAS: about 1/1000th the size.

Remote Execution with the CAS & AC


2. Call GetActionResult(ActionDigest)
AC
3. Receive ActionResult if already executed
1. Compute ExecuteRequest, Action, Command and Directories messages.
7. Call Execute(ExecuteRequest)
Bazel
Build cluster
8. Receive ExecuteResponse
4. Call FindMissingBlobs() on Digests
5. Receive list of objects that are missing
6. Upload missing objects
CAS
9. Download Tree messages, stdout/stderr and output files.

Note: steps 4 to 8 are skipped in case step 3 returns success.

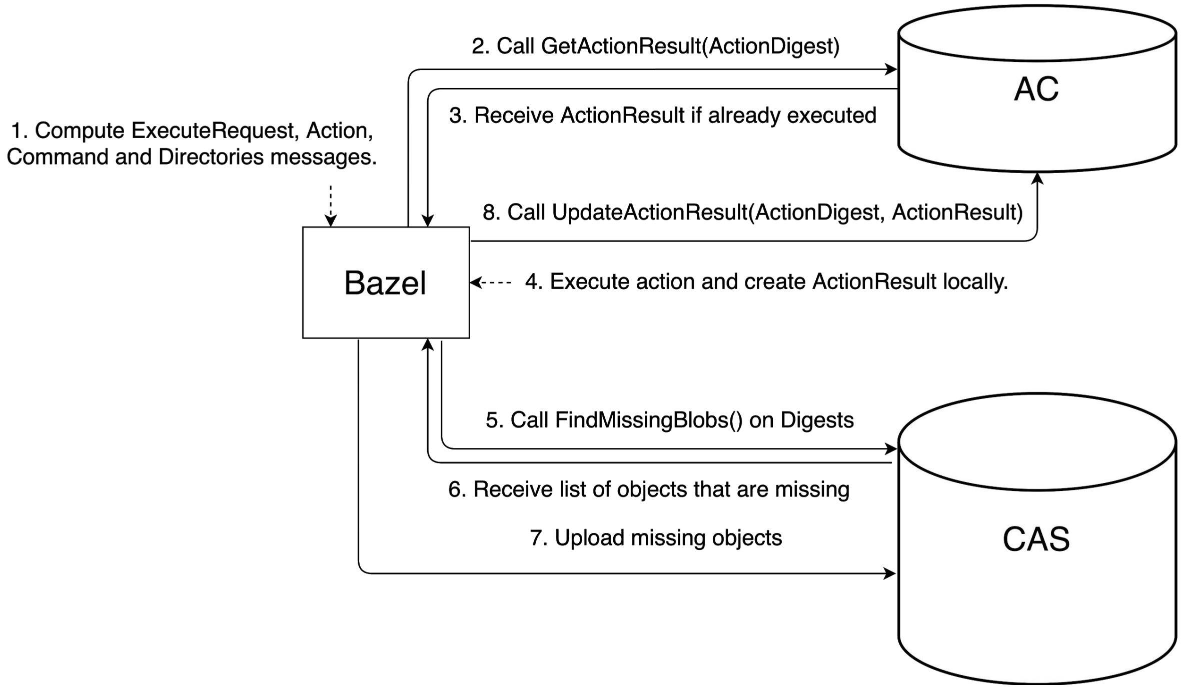
Bazel’s ‘Builds without the Bytes’

- Problem: Step 9 (i.e., downloading outputs) generates lots of network I/O.
    - Bazel downloads all intermediate artifacts (e.g., object files) from the CAS.
    - Can account for 98% of network I/O for certain workloads.
- Solution: Enable ‘Builds without the Bytes’ command line flags.
    - Only download outputs for top-level targets (e.g., binaries), or simply not at all.
    - Intermediate artifacts are handed to successive build actions by reference.
    - GetActionResult() and Execute() cannot return references to objects that disappear.
    - Requires CAS to be rock solid and big enough: losing data during builds causes them to fail.
- User experience of ‘Builds without the Bytes’ can still be improved.
    - No way to get on-demand access to CAS objects afterwards.

Remote Caching: still execute actions locally

- Problem: Full Remote Execution may not always be feasible.
  - If a small number of build actions don’t work with Remote Execution yet.
  - If setting up Remote Execution infrastructure requires too much maintenance.
- Solution: Let Bazel execute locally and seed the AC directly.
  - Can be set up in `BUILD` files on a per-target basis if needed.
  - Does potentially allow users to poison the AC.
  - Hint: Only allow CI systems to write into the AC. Users can still read the AC.

Remote Caching: still execute actions locally


2. Call GetActionResult(ActionDigest)
AC
3. Receive ActionResult if already executed
1. Compute ExecuteRequest, Action, Command and Directories messages.
8. Call UpdateActionResult(ActionDigest, ActionResult)
Bazel
4. Execute action and create ActionResult locally.
5. Call FindMissingBlobs() on Digests
6. Receive list of objects that are missing
7. Upload missing objects
CAS

Note: steps 4 to 8 are skipped in case step 3 returns success.

Topics that were presented

- History of Bazel and the REv2 protocol.
- The idea behind Remote Execution.
- How the CAS reduces request size and pass outputs to successive actions.
- How direct AC access reduces the number of round-trips.
- Bazel’s ‘Builds without the Bytes’.
- Using a subset of REv2 to do plain remote caching.

Q&A