Buildbarn

A distributed build cluster

Ed Schouten <ed@nuxi.nl>
https://github.com/buildbarn

History of Bazel

Timeline (1/4)

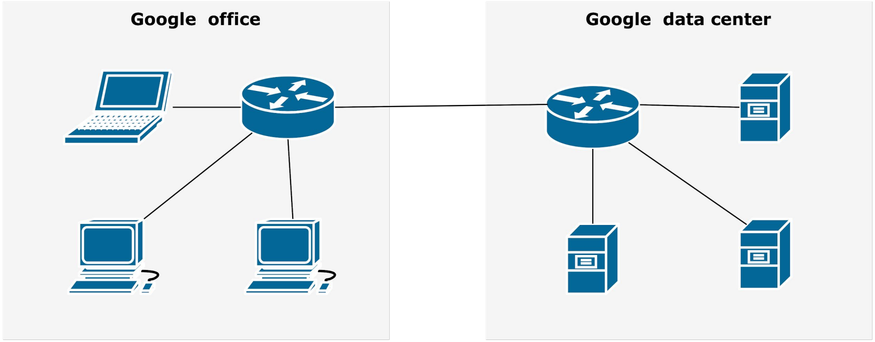
- ~2000: Google has a monorepo with shell script/Makefile build scripts.
  - It turns out that becomes unmaintainable relatively quickly.
- ~2005: Makefiles are replaced with build tool written in Python.
  - Every ‘package’ (directory) contains a `BUILD` file that is `eval()`ed by Python.
  - Directives 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 primitive Python interpreter to parse existing `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: ‘blaze -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.
- 2015-2023: 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’.
- Servers that implement REv2 start to appear:
  - 2017: Uber releases Bazel Buildfarm, written in Java.
  - 2018: Bloomberg & CodeThink release BuildGrid, written in Python.
  - 2018: I started working on Buildbarn, written in Go.
- Clients other than Bazel that use REv2 start to appear:
  - Drop-in replacements for /usr/bin/cc: recc, Goma.
  - Existing build systems add REv2 support: Pants, Please, BuildStream, Buck.

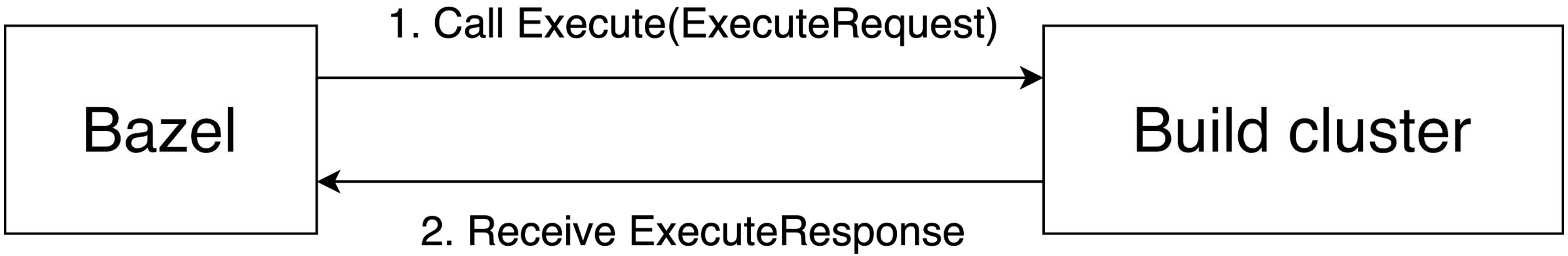
‘distcc/ccache/… did this two decades ago’

- … 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 run unit/integration tests remotely and cache results.

REv2 is not a fad!
It is the de facto standard for distributed software builds.

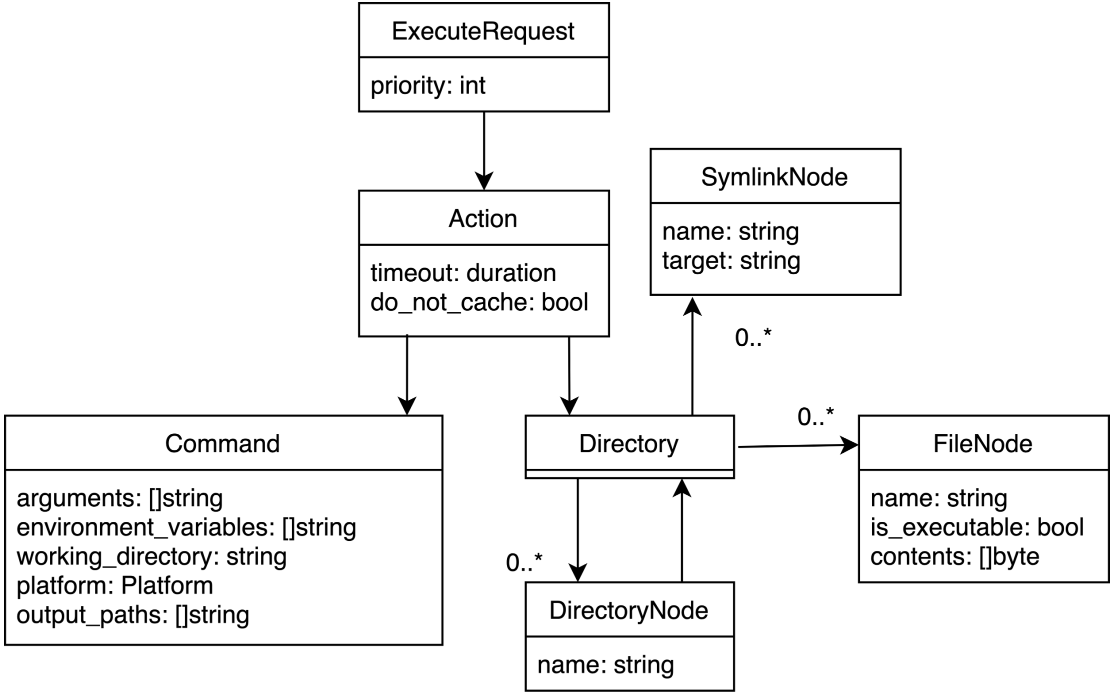
REv2

Remote Execution… simplified

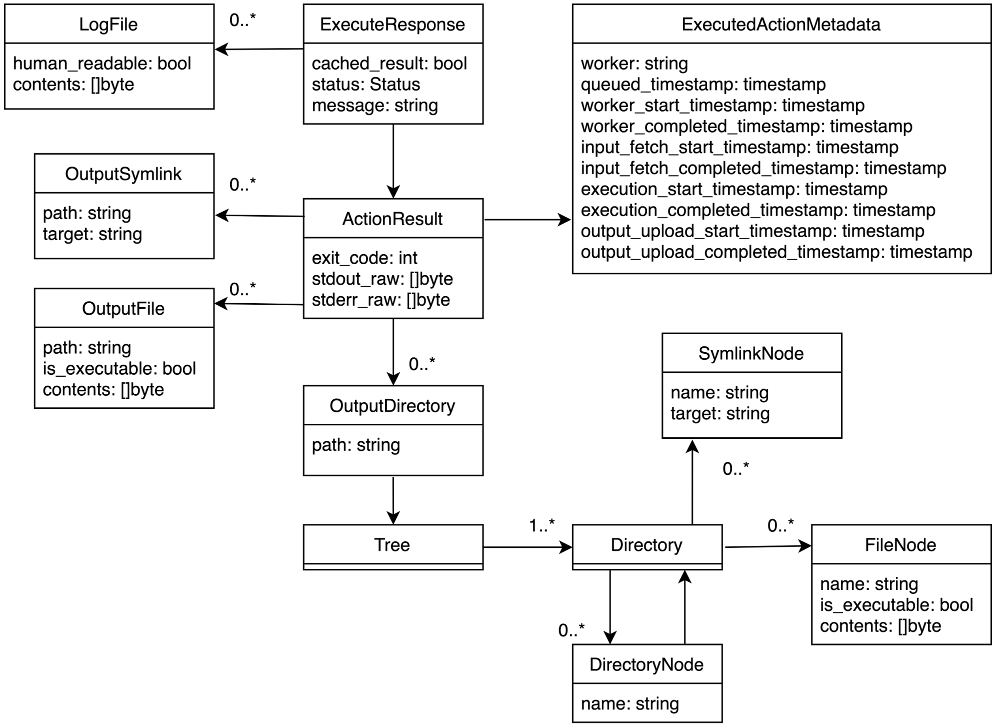


Building a project consists of hundreds/thousands of these calls.

ExecuteRequest



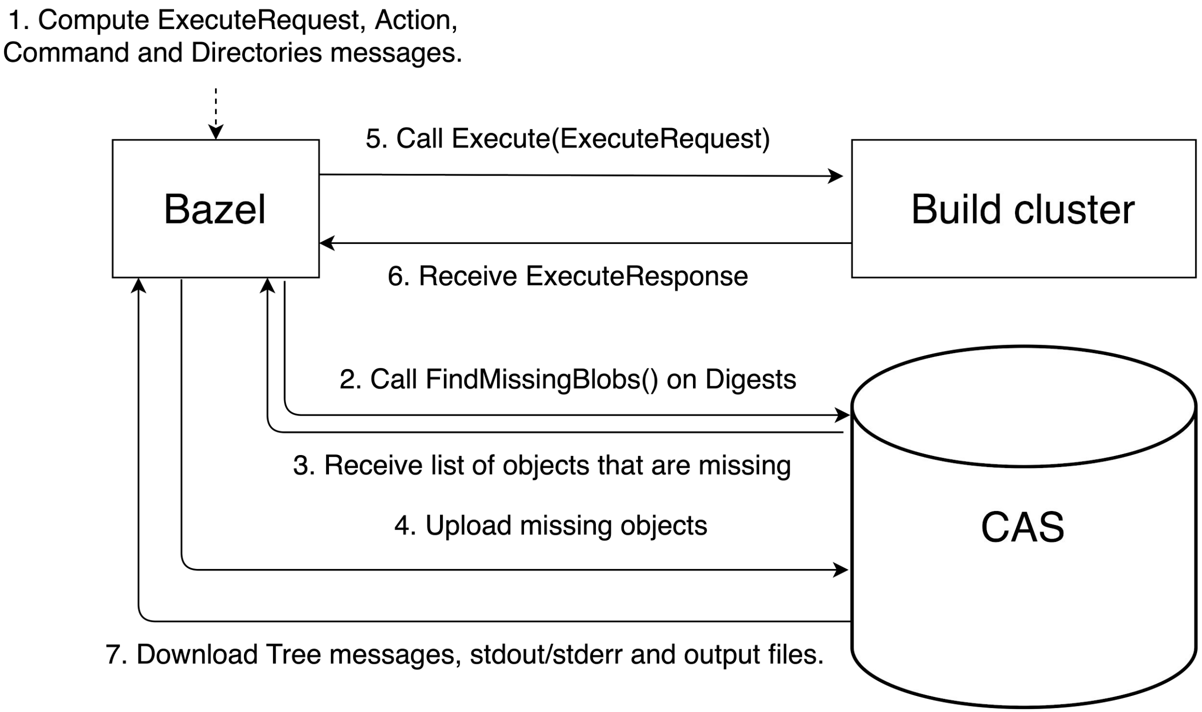
ExecuteResponse



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.

Remote Execution with the CAS

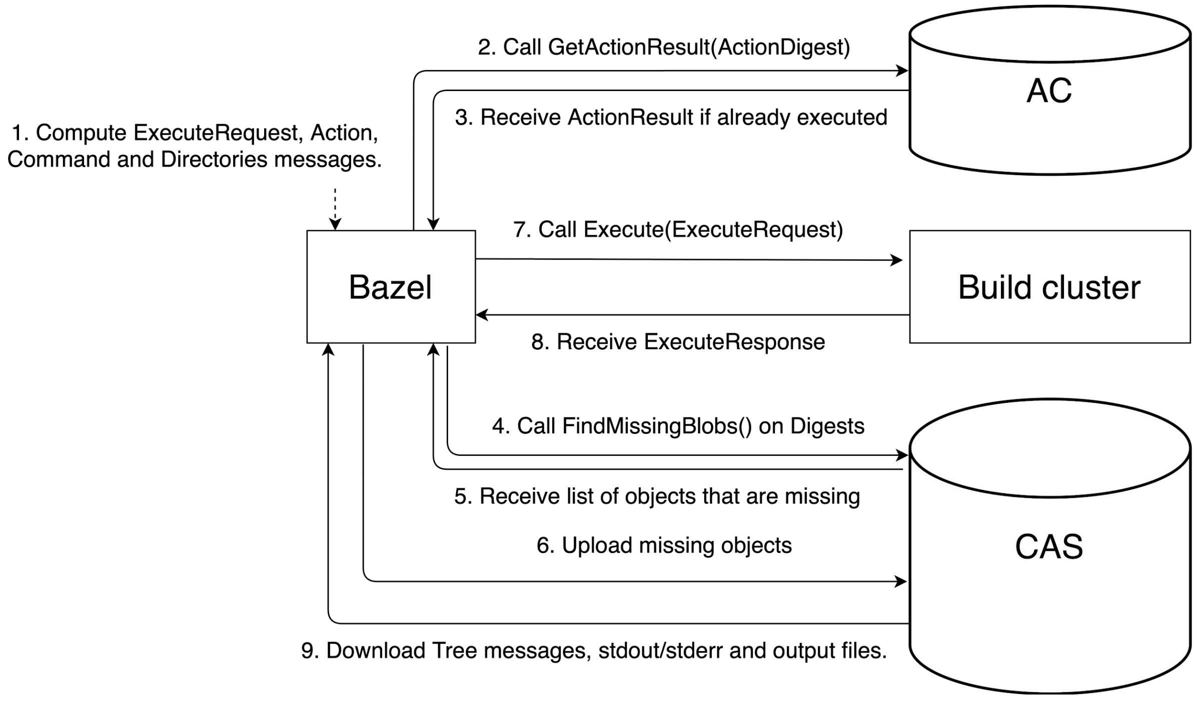

1. Compute ExecuteRequest, Action, Command and Directories messages.
Bazel
5. Call Execute(ExecuteRequest)
Build cluster
6. Receive ExecuteResponse
2. Call FindMissingBlobs() on Digests
3. Receive list of objects that are missing
4. Upload missing objects
CAS
7. Download Tree messages, stdout/stderr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

Note: only communication involving Bazel is shown.

Action Cache (AC)

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

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.

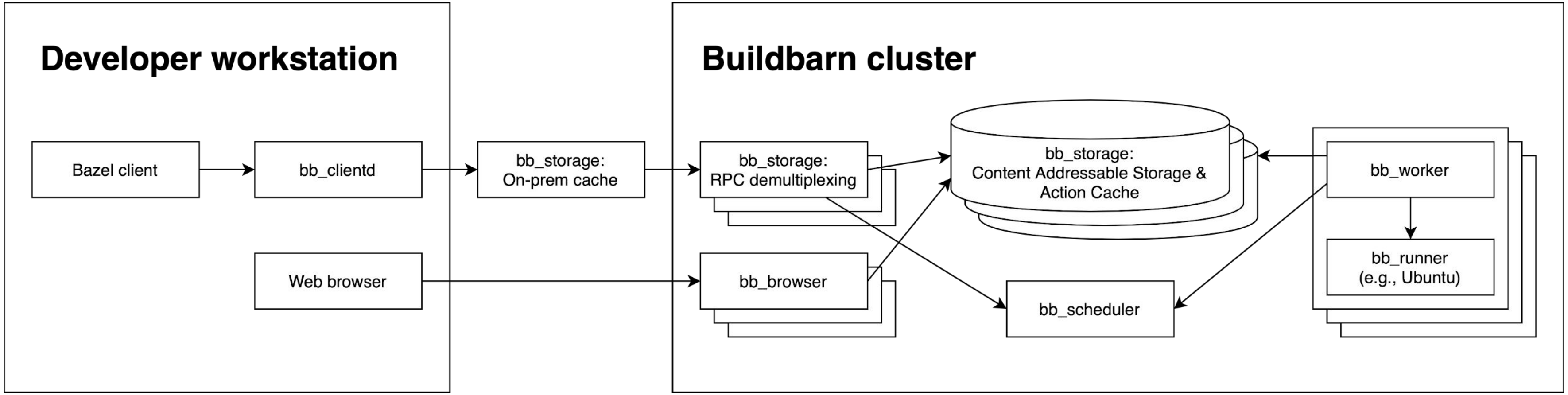
Note: steps 4 to 8 are skipped in case step 3 returns success.

REv2 Summary

- RPC to run commands remotely: **Execute()**.
  - The client sends ExecuteRequests.
    - References an Action, a Command, Directories, and individual files.
  - The cluster sends back ExecuteResponses.
    - References an ActionResult, Trees, and individual files.
- Data store #1: **Content Addressable Storage (CAS)**.
  - Stores Actions, Commands, Directories, Trees and individual files.
  - Big. Cannot be poisoned. Safe to provide write access to all workers and clients.
- Data store #2: **Action Cache (AC)**.
  - Stores mapping of Action → ActionResult. A cache of previously run actions.
  - Small. Can be poisoned. Restrict write access to trusted workers and clients!

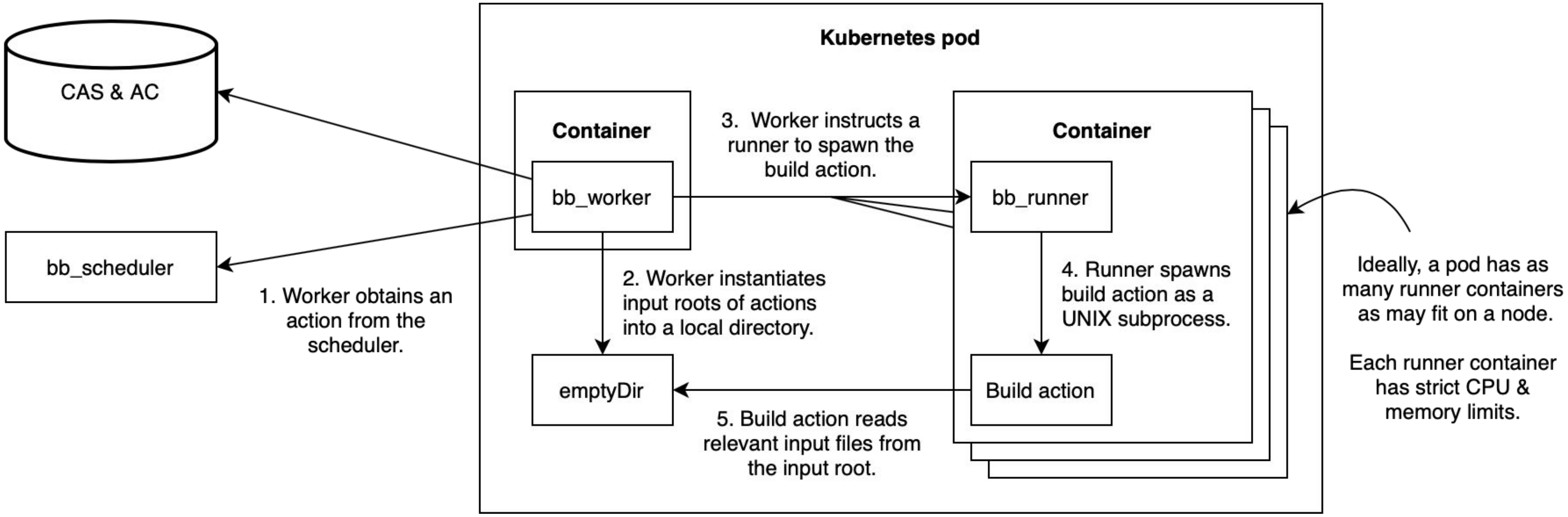
The Buildbarn project

A fairly complete Buildbarn setup



Note: not all components are required. Just cherry-pick the parts you need!

A typical worker pod running on Kubernetes



1. Worker obtains an action from the scheduler.

2. Worker instantiates input roots of actions into a local directory.

3. Worker instructs a runner to spawn the build action.

4. Runner spawns build action as a UNIX subprocess.

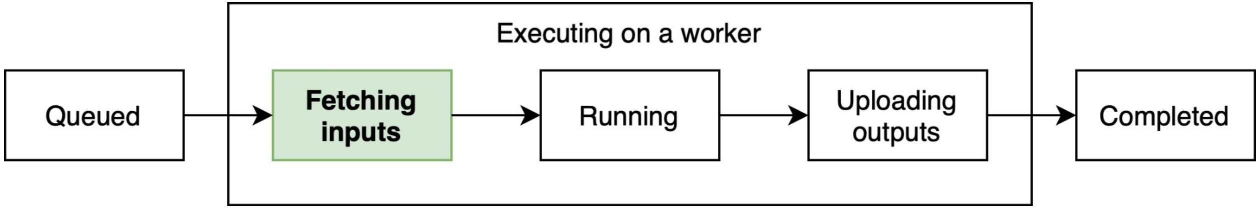
5. Build action reads relevant input files from the input root.

Ideally, a pod has as many runner containers as may fit on a node.

Each runner container has strict CPU & memory limits.

bb_worker:
Input root population

Worker-level file cache



- **Problem:** Input root population can be a costly process.
  - Download gigabytes from the CAS in case actions bring their own SDKs.
  - In case of highly concurrent workers: lots of redundant disk space usage.
- **Solution:** Add a worker-level file cache.
  - A single directory that contains a single copy of every recently used file.
  - **Cache hits:** files are hardlinked from the cache into the input root.
  - **Cache misses:** files are downloaded into the input root and hardlinked into the cache.
  - Population time may become proportional to file count; not total file size.

The contents of the worker-level file cache

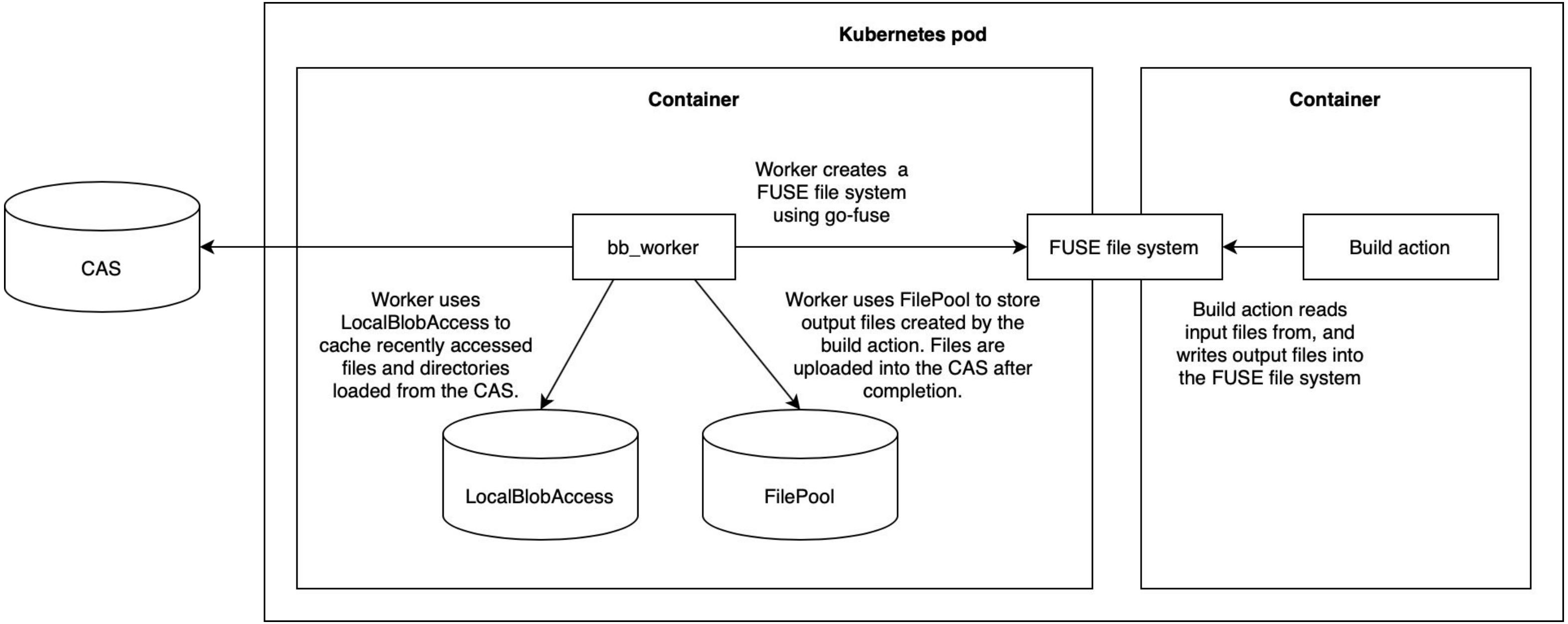
```
$ ls -l /worker/cache
-r-xr-xr-x 1 root root  3259 Jan 1 2000 000095eedb03648c4f9abaf74a248ccbf5a2fb5c0f307ddb31d04bb32cfdf370-3259+x
-r--r--r-- 1 root root  2469 Jan 1 2000 00010c0e5a6aa2c95ac6b8163176a616fb7e6ff95e93eb040ab4cdb57bdce4d0-2469-x
-r--r--r-- 1 root root   320 Jan 1 2000 000c00bad31d126b054c6ec7f3e02b27c0f9a4d579f987d3c4f879cee1bacb81-320-x
-r-xr-xr-x 1 root root   320 Jan 1 2000 000c00bad31d126b054c6ec7f3e02b27c0f9a4d579f987d3c4f879cee1bacb81-320+x
-r-xr-xr-x 1 root root   742 Jan 1 2000 00145314b959a6dfa16f7d37452f3cf358ef614bdf7b54a28ab9dce9117e31cf-742+x
-r-xr-xr-x 1 root root  1141 Jan 1 2000 001464d2ef94de500cb053cd345164d696f7f84cf38fa522c77327ab04d32982-1141+x
...
-r-xr-xr-x 1 root root  6393 Jan 1 2000 ffc12c3075ba18eaa46dd60e730e06a1ec338216151e056a9c7ecfd74d280fd9-6393+x
-r--r--r-- 1 root root  1913 Jan 1 2000 ffd70c5181898b8b8e17f9f8a76f2d30793abebed370c739bd1136b34782b09c-1913-x
-r-xr-xr-x 1 root root  3642 Jan 1 2000 ffecdbd3f9ba0d71c6e59984f8384817f0fe5b0ac69ba62e1e40a31faf596a6c-3642+x
-r--r--r-- 1 root root  1440 Jan 1 2000 ffede6eabd976f361148ae9c7e4c7c31bba51db02493f05015d2145db1a4ea44-1440-x
-r--r--r-- 1 root root  3541 Jan 1 2000 ffeeb99fa19cac2db45fa444da20f2c5aa157b87b836bee1f91e18ddf8412c94-3541-x
-r-xr-xr-x 1 root root 75928 Jan 1 2000 fffe9f6f1470c3a12c6588e26f37485b7fac3af83a6f3d52e2205d02d4a646bc-75928+x

$ sha256sum < f7fec7e9c5fec8265163391bb32716ac97deec644637020600869ae3f5cef793-10011+x
f7fec7e9c5fec8265163391bb32716ac97deec644637020600869ae3f5cef793
```

Lazy input root population

- **Problem:** Worker-level file cache may still perform poorly.
  - Input roots are downloaded up front entirely, even if only a fraction is used.
  - Requires all runners to share one file system: running out of space causes flakiness for all.
- **Solution:** Run builds inside a virtual file system offered by bb_worker.
  - Starts build actions inside a placeholder input root directory.
    - **Accessing a placeholder directory:** instantiate it, creating placeholder files inside.
    - **Accessing a placeholder file:** read it from the CAS.
    - CAS read errors causes action to be killed, returning a retriable error.
  - Build actions can create output files inside the input root directory.
    - Quotas are only applied against output files.
    - Exceeding quota causes action to be killed, returning a non-retriable error.

bb_worker with the virtual file system enabled



Kubernetes pod

Container

Container

CAS

bb_worker

Worker creates a FUSE file system using go-fuse

FUSE file system

Build action

Worker uses LocalBlobAccess to cache recently accessed files and directories loaded from the CAS.

Worker uses FilePool to store output files created by the build action. Files are uploaded into the CAS after completion.

Build action reads input files from, and writes output files into the FUSE file system

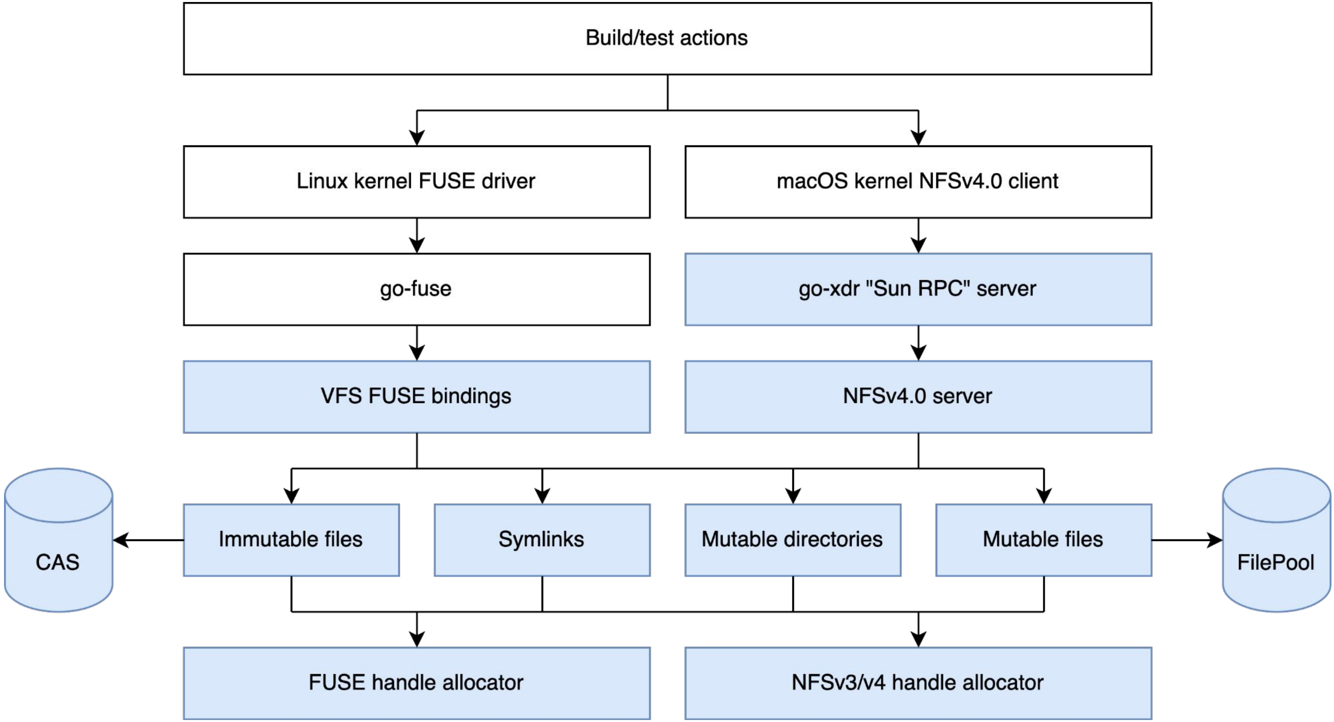
LocalBlobAccess

FilePool

Recent developments in the virtual file system

- 2020: Initial release of the FUSE file system for bb_worker.
  - Linux: use in-kernel FUSE driver, which works great.
  - macOS: use OSXFUSE, which isn’t always stable (and proprietary).
- 2022: FUSE file system refactored into a generic virtual file system.
  - Added support for NFSv4.0 in addition to FUSE.
  - macOS 13.3+ systems can ‘mount -t nfs localhost’ the integrated NFSv4.0 server.
- 2023: File system access profiling.
  - bb_worker can record which paths are accessed during build, and store those centrally.
  - Subsequent executions of similar actions reload the profile from central storage.
  - Profile is used to load expected set of accessed files in the background, in parallel.
  - Impact: Workers now spend <1% of their time fetching input files.

Layering of the virtual file system



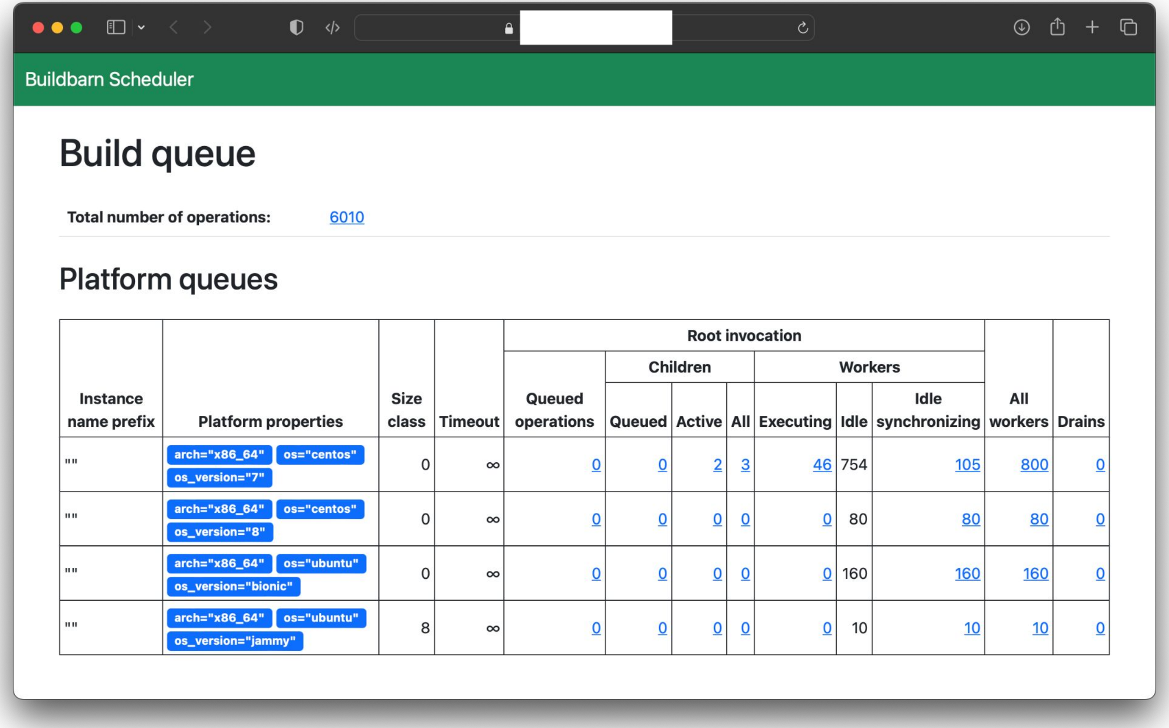
Note: components in blue are maintained as part of the Buildbarn project

bb_scheduler

bb_scheduler overview

- Multiple levels of nested queues.
  - Scheduler has one platform queue for every distinct platform (Ubuntu, CentOS, macOS, etc.)
  - Every platform queue has one or more size class queues.
  - Every size class queue has a queue of running Bazel invocations.
  - Every Bazel invocation has a queue of scheduled operations (actions).
  - **Scheduler steers towards fairness between Bazel invocations.**
- Scheduler keeps all queue state in memory; no persistency.
  - Restart means that clients need to restart all current operations.
  - Not an issue, considering that scheduler restarts don’t happen frequently.
- gRPC protocol and integrated web UI for exploring the current worker state.
- Uses a simple and robust scheduler ↔ worker protocol.
  - Specific to Buildbarn. No consensus within community yet.

Screenshot of the bb_scheduler web UI



Buildbarn Scheduler

Build queue

Total number of operations: 6010

Platform queues

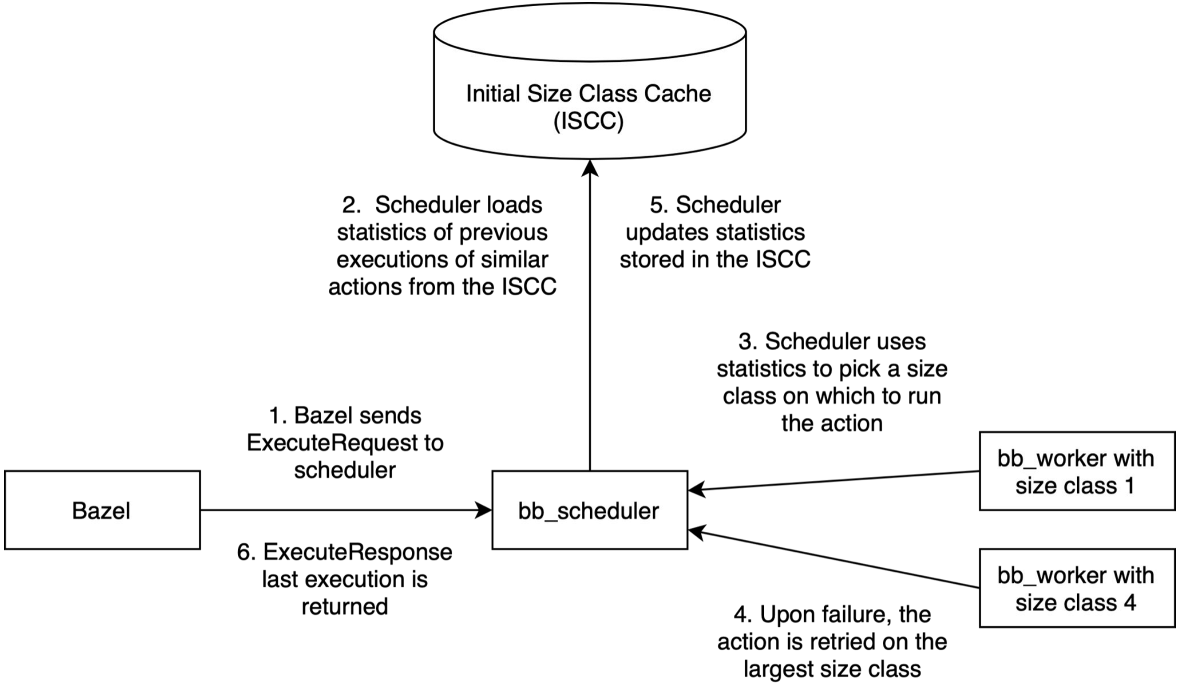
				Root invocation								
					Children			Workers				
Instance name prefix	Platform properties	Size class	Timeout	Queued operations	Queued	Active	All	Executing	Idle	Idle synchronizing	All workers	Drains
""	arch="x86_64" os="centos" os_version="7"	0	∞	0	0	2	3	46	754	105	800	0
""	arch="x86_64" os="centos" os_version="8"	0	∞	0	0	0	0	0	80	80	80	0
""	arch="x86_64" os="ubuntu" os_version="bionic"	0	∞	0	0	0	0	0	160	160	160	0
""	arch="x86_64" os="ubuntu" os_version="jammy"	8	∞	0	0	0	0	0	10	10	10	0

bb_scheduler:
Worker size classes

Problem definition

- Resource requirements of actions follow a hockey stick curve.
  - Almost all of them are single-threaded / don’t use a lot of RAM.
  - Only a handful benefit from multi-threaded workers / need more RAM.
- **Traditional solution:** Multiple worker types with different platform properties.
  - Requires annotating targets in BUILD files.
  - Race to the bottom: “I just want my test to run fast. Let me pick the largest worker!”
  - Hard to get right when multiple build configurations are used (e.g., release, debug, asan).
  - Bazel lacks granularity for doing this right (e.g., when using sharded tests).
  - Third-party libraries that ship with BUILD files cannot get it right for your cluster.
  - Annotations need to be revisited every time infrastructure is redesigned.
- **Desired solution:** Let Buildbarn sort it out itself.

Rough outline of the solution



Initial Size Class Cache (ISCC)

2. Scheduler loads statistics of previous executions of similar actions from the ISCC

5. Scheduler updates statistics stored in the ISCC

3. Scheduler uses statistics to pick a size class on which to run the action

1. Bazel sends ExecuteRequest to scheduler

6. ExecuteResponse last execution is returned

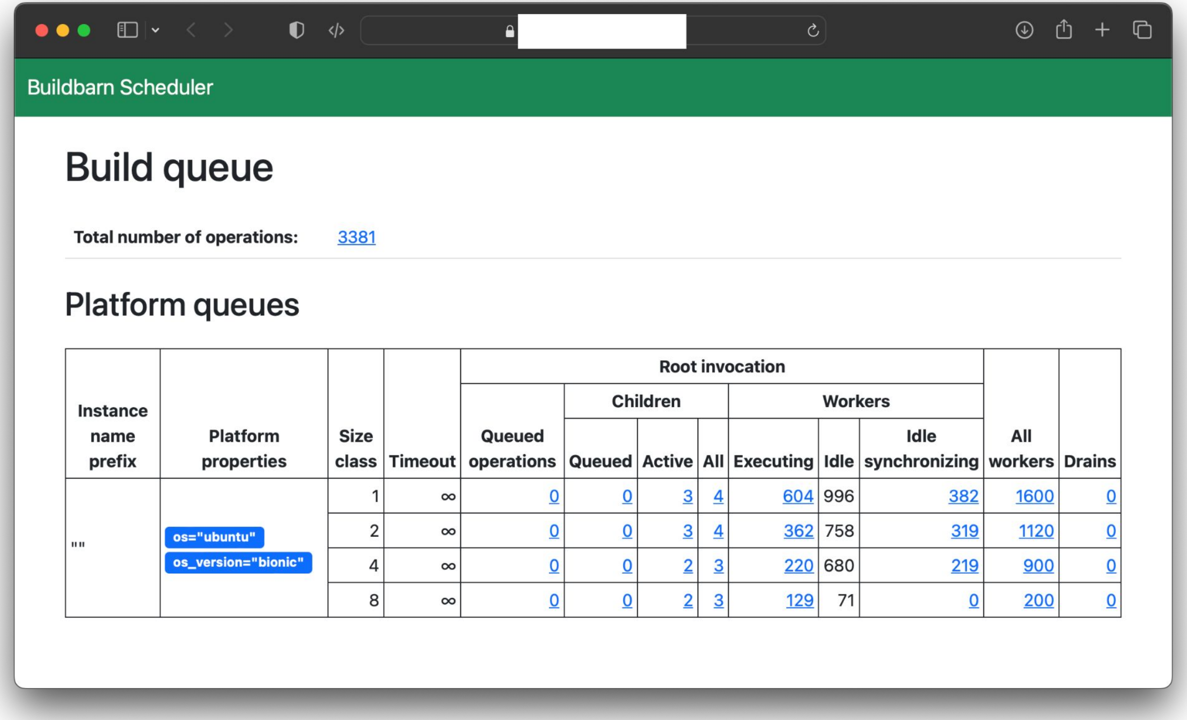
4. Upon failure, the action is retried on the largest size class

Note: arrows indicate directions of RPCs

Stats in ISCC for a single kind of action

```
message PreviousExecution {
  oneof outcome {
    google.protobuf.Empty failed = 1;
    google.protobuf.Duration timed_out = 2;
    google.protobuf.Duration succeeded = 3;
  }
}
message PerSizeClassStats {
  // List of recent outcomes sorted by date, truncated to a configurable size.
  // Only outcomes of actions that eventually succeeded are stored.
  repeated PreviousExecution previous_executions = 1;
  ...
}
message PreviousExecutionStats {
  map<uint32, PerSizeClassStats> size_classes = 1;
  ...
}
```

bb_scheduler web UI with size classes



[redacted]

Buildbarn Scheduler

Build queue

Total number of operations: 3381

Platform queues

Instance name prefix	Platform properties	Size class	Timeout	Root invocation							All workers	Drains
					Children			Workers				
				Queued operations	Queued	Active	All	Executing	Idle	Idle synchronizing		
""	os="ubuntu" os_version="bionic"	1	∞	0	0	3	4	604	996	382	1600	0
		2	∞	0	0	3	4	362	758	319	1120	0
		4	∞	0	0	2	3	220	680	219	900	0
		8	∞	0	0	2	3	129	71	0	200	0

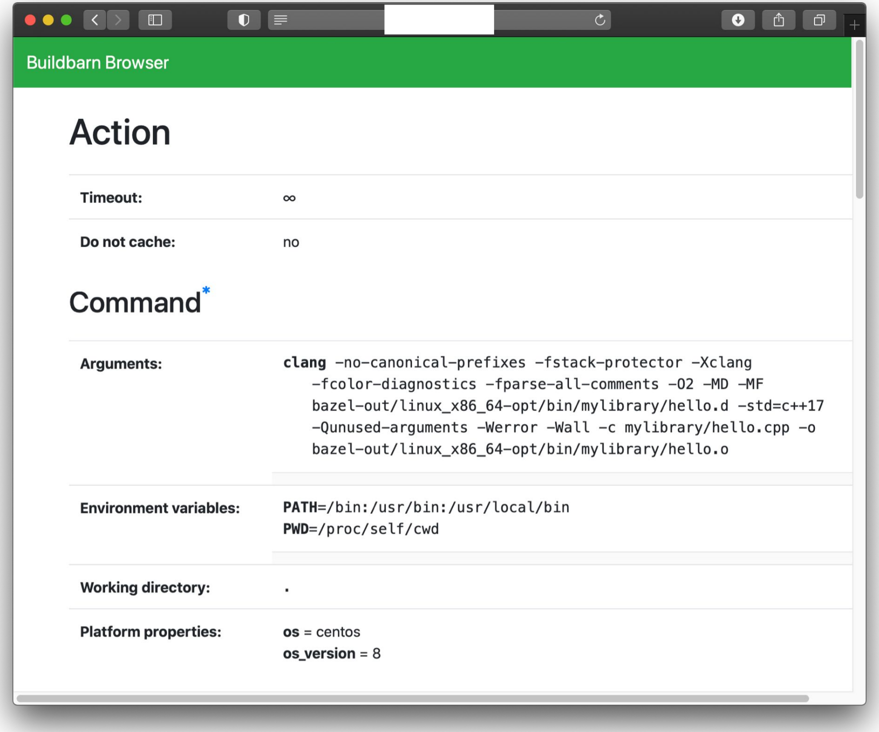
bb_browser

What is bb_browser?

Simple web UI for REv2 data stores.

- Can display REv2 Actions, Commands, Directories, ActionResults and Trees.
- Can display Buildbarn specific extensions:
  - UncachedActionResult: Stable copy of a historical ActionResult that's preserved in the CAS.
  - PreviousExecutionStats: Build action timing information that's stored in the ISCC.
  - FileSystemAccessProfiles: Which input files are expected to be used.
- Other Buildbarn tools link to bb_browser.
  - Every operation in bb_scheduler links to an Action page.
  - bb_worker can let Bazel print UncachedActionResult links upon build failures.

Screenshots of bb_browser



Buildbarn Browser

Action

Timeout:	∞
Do not cache:	no

Command*

Arguments:

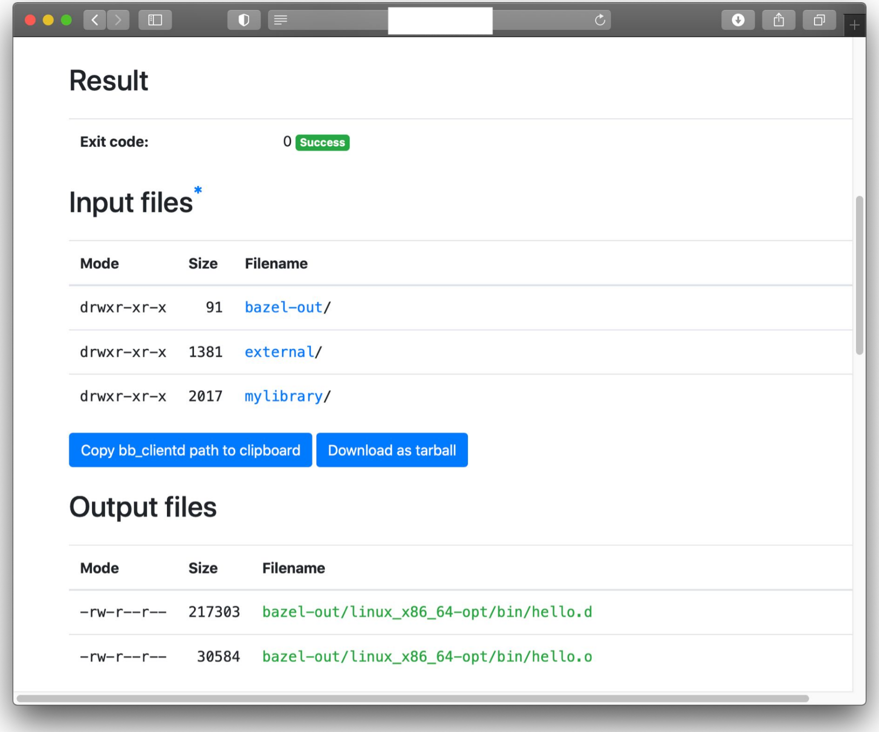
```
clang -no-canonical-prefixes -fstack-protector -Xclang
   -fcolor-diagnostics -fparse-all-comments -O2 -MD -MF
   bazel-out/linux_x86_64-opt/bin/mylibrary/hello.d -std=c++17
   -Qunused-arguments -Werror -Wall -c mylibrary/hello.cpp -o
   bazel-out/linux_x86_64-opt/bin/mylibrary/hello.o
```

Environment variables:

```
PATH=/bin:/usr/bin:/usr/local/bin
PWD=/proc/self/cwd
```

Working directory:	.
Platform properties:	**os** = centos **os_version** = 8

Screenshots of bb_browser



Result

Exit code:	0 Success

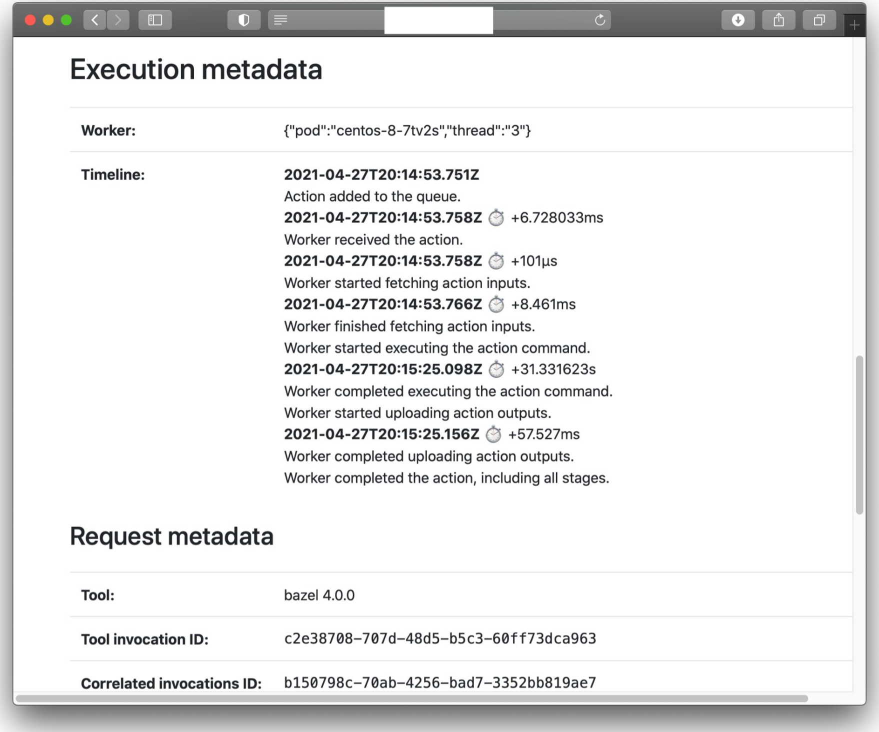
Input files*

Mode	Size	Filename
drwxr-xr-x	91	bazel-out/
drwxr-xr-x	1381	external/
drwxr-xr-x	2017	mylibrary/

Output files

Mode	Size	Filename
-rw-r--r--	217303	bazel-out/linux_x86_64-opt/bin/hello.d
-rw-r--r--	30584	bazel-out/linux_x86_64-opt/bin/hello.o

Screenshots of bb_browser



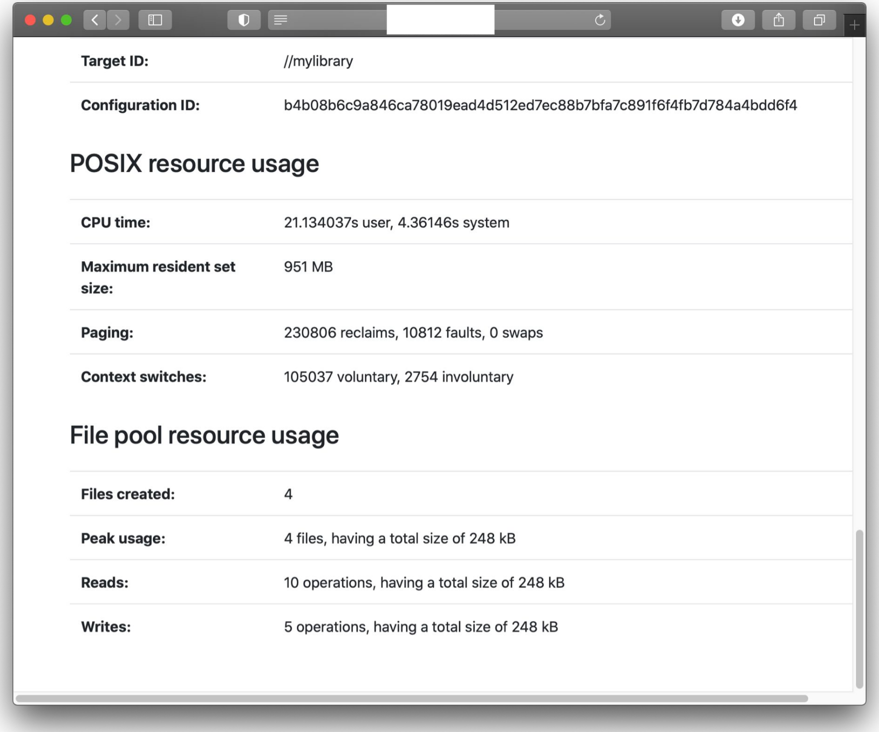
Execution metadata

Worker:	{"pod":"centos-8-7tv2s","thread":"3"}
Timeline:	**2021-04-27T20:14:53.751Z** Action added to the queue. **2021-04-27T20:14:53.758Z** ⏱ +6.728033ms Worker received the action. **2021-04-27T20:14:53.758Z** ⏱ +101µs Worker started fetching action inputs. **2021-04-27T20:14:53.766Z** ⏱ +8.461ms Worker finished fetching action inputs. Worker started executing the action command. **2021-04-27T20:15:25.098Z** ⏱ +31.331623s Worker completed executing the action command. Worker started uploading action outputs. **2021-04-27T20:15:25.156Z** ⏱ +57.527ms Worker completed uploading action outputs. Worker completed the action, including all stages.

Request metadata

Tool:	bazel 4.0.0
Tool invocation ID:	`c2e38708-707d-48d5-b5c3-60ff73dca963`
Correlated invocations ID:	`b150798c-70ab-4256-bad7-3352bb819ae7`

Screenshots of bb_browser



Target ID:	//mylibrary
Configuration ID:	b4b08b6c9a846ca78019ead4d512ed7ec88b7bfa7c891f6f4fb7d784a4bdd6f4

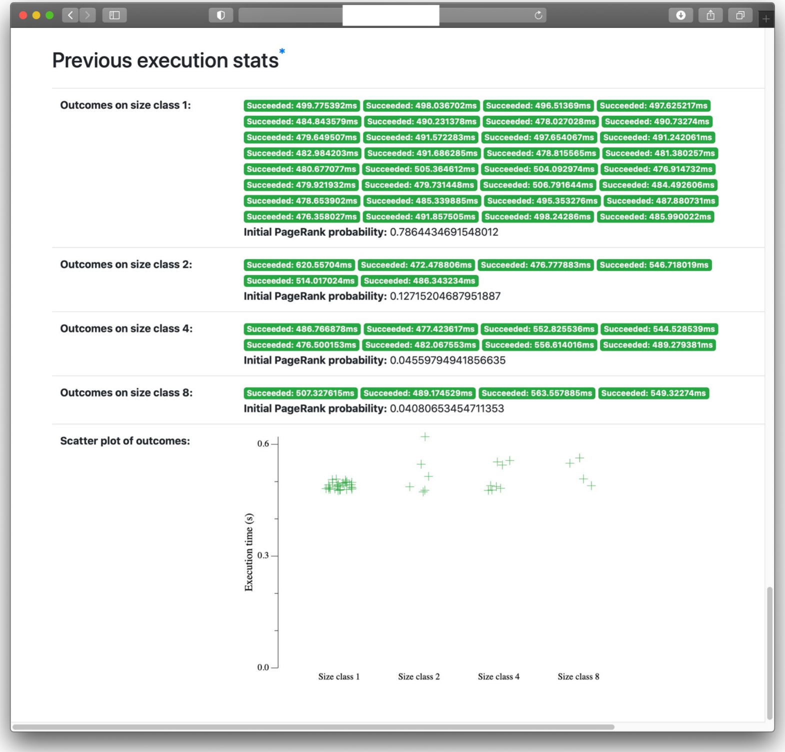
POSIX resource usage

CPU time:	21.134037s user, 4.36146s system
Maximum resident set size:	951 MB
Paging:	230806 reclaims, 10812 faults, 0 swaps
Context switches:	105037 voluntary, 2754 involuntary

File pool resource usage

Files created:	4
Peak usage:	4 files, having a total size of 248 kB
Reads:	10 operations, having a total size of 248 kB
Writes:	5 operations, having a total size of 248 kB

Screenshots of bb_browser



Previous execution stats*

Outcomes on size class 1:

Succeeded: 499.775392ms Succeeded: 498.036702ms Succeeded: 496.51369ms Succeeded: 497.625217ms
Succeeded: 484.843579ms Succeeded: 490.231378ms Succeeded: 478.027028ms Succeeded: 490.73274ms
Succeeded: 479.649507ms Succeeded: 491.572283ms Succeeded: 497.654067ms Succeeded: 491.242061ms
Succeeded: 482.984203ms Succeeded: 491.686285ms Succeeded: 478.815565ms Succeeded: 481.380257ms
Succeeded: 480.677077ms Succeeded: 505.364612ms Succeeded: 504.092974ms Succeeded: 476.914732ms
Succeeded: 479.921932ms Succeeded: 479.731448ms Succeeded: 506.791644ms Succeeded: 484.492606ms
Succeeded: 478.653902ms Succeeded: 485.339885ms Succeeded: 495.353276ms Succeeded: 487.880731ms
Succeeded: 476.358027ms Succeeded: 491.857505ms Succeeded: 498.24286ms Succeeded: 485.990022ms

Initial PageRank probability: 0.7864434691548012

Outcomes on size class 2:

Succeeded: 620.55704ms Succeeded: 472.478806ms Succeeded: 476.777883ms Succeeded: 546.718019ms
Succeeded: 514.017024ms Succeeded: 486.343234ms

Initial PageRank probability: 0.12715204687951887

Outcomes on size class 4:

Succeeded: 486.766878ms Succeeded: 477.423617ms Succeeded: 552.825536ms Succeeded: 544.528539ms
Succeeded: 476.500153ms Succeeded: 482.067553ms Succeeded: 556.614016ms Succeeded: 489.279381ms

Initial PageRank probability: 0.04559794941856635

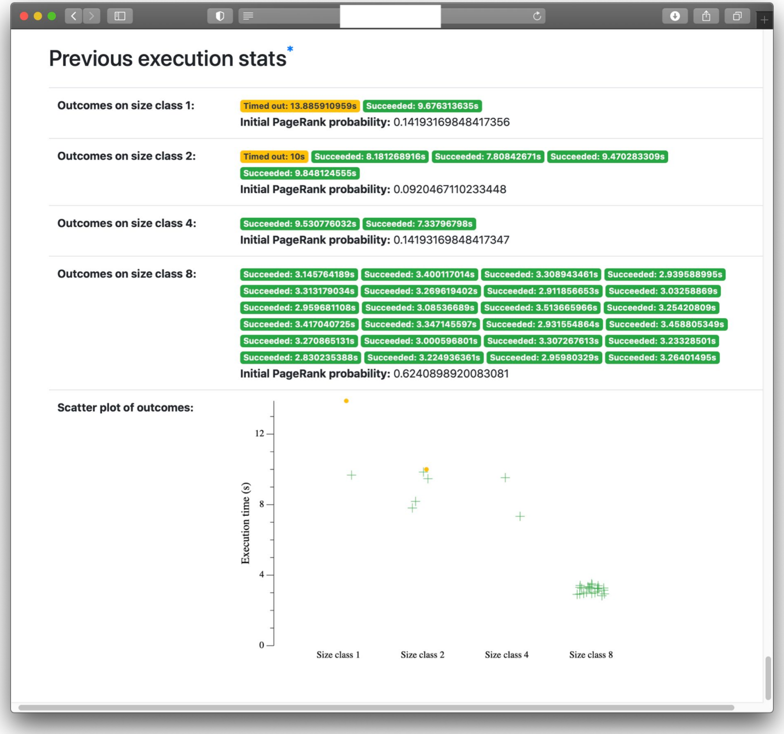
Outcomes on size class 8:

Succeeded: 507.327615ms Succeeded: 489.174529ms Succeeded: 563.557885ms Succeeded: 549.32274ms

Initial PageRank probability: 0.04080653454711353

Scatter plot of outcomes:

Screenshots of bb_browser



Previous execution stats*

Outcomes on size class 1:

Timed out: 13.885910959s Succeeded: 9.676313635s
Initial PageRank probability: 0.14193169848417356

Outcomes on size class 2:

Timed out: 10s Succeeded: 8.181268916s Succeeded: 7.80842671s Succeeded: 9.470283309s
Succeeded: 9.848124555s
Initial PageRank probability: 0.0920467110233448

Outcomes on size class 4:

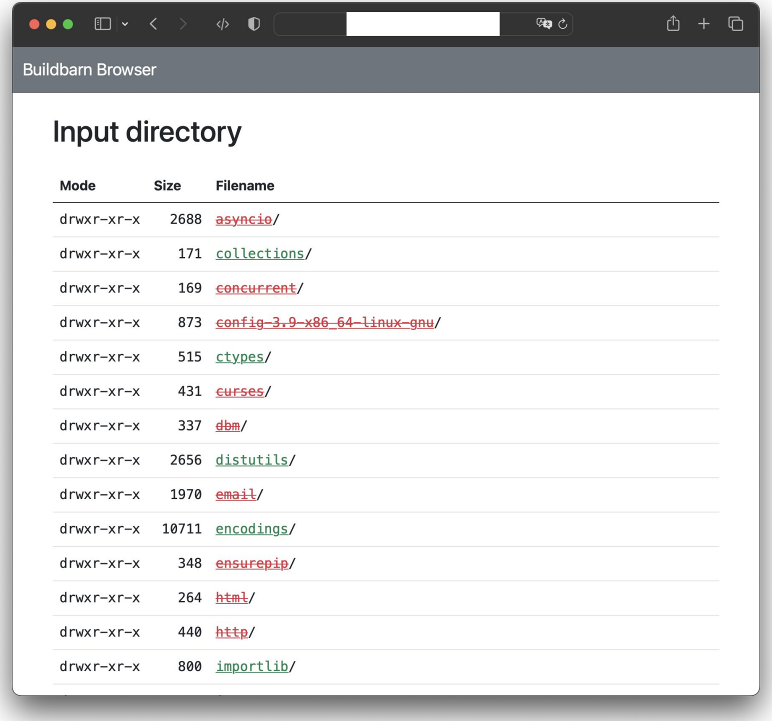
Succeeded: 9.530776032s Succeeded: 7.33796798s
Initial PageRank probability: 0.14193169848417347

Outcomes on size class 8:

Succeeded: 3.145764189s Succeeded: 3.400117014s Succeeded: 3.308943461s Succeeded: 2.939588995s
Succeeded: 3.313179034s Succeeded: 3.269619402s Succeeded: 2.911856653s Succeeded: 3.03258869s
Succeeded: 2.959681108s Succeeded: 3.08536689s Succeeded: 3.513665966s Succeeded: 3.25420809s
Succeeded: 3.417040725s Succeeded: 3.347145597s Succeeded: 2.931554864s Succeeded: 3.458805349s
Succeeded: 3.270865131s Succeeded: 3.000596801s Succeeded: 3.307267613s Succeeded: 3.23328501s
Succeeded: 2.830235388s Succeeded: 3.224936361s Succeeded: 2.95980329s Succeeded: 3.26401495s
Initial PageRank probability: 0.6240898920083081

Scatter plot of outcomes:

Screenshots of bb_browser



Buildbarn Browser

Input directory

Mode	Size	Filename
drwxr-xr-x	2688	~~asyncio~~/
drwxr-xr-x	171	collections/
drwxr-xr-x	169	~~concurrent~~/
drwxr-xr-x	873	~~config-3.9-x86_64-linux-gnu~~/
drwxr-xr-x	515	ctypes/
drwxr-xr-x	431	~~curses~~/
drwxr-xr-x	337	~~dbm~~/
drwxr-xr-x	2656	distutils/
drwxr-xr-x	1970	~~email~~/
drwxr-xr-x	10711	encodings/
drwxr-xr-x	348	~~ensurepip~~/
drwxr-xr-x	264	~~html~~/
drwxr-xr-x	440	~~http~~/
drwxr-xr-x	800	importlib/

Q&A