This project is still in a very early stage of development, and the following definitions might change at any time. That said, I think it's far enough along that we should share it with the world to start gathering and iterating on user feedback.

motivation

Execution-layer client developers have been asking for a reliable way to test their nodes under realistic conditions. There are a lot of ways to go about this, and lots of questions to consider:

- Is the test a realistic model of real-world user behavior?
- Does the testing scenario induce (or even consider) instances of state contention?
- Is the base state on which the test is executed realistic, and (if not perfectly realistic) how would the results differ in a production environment?
- How long does it take to change the test?

Most EL nodes test performance in production. reth

logs gas/second benchmarks for every block while it's running. While testing in production is technically safe (you're not likely to cause any consensus errors), it may require some devops expertise (staging deployments, A/B testing), and it's relatively time-consuming and costly overall.

Developers love testing locally (at least I do, anyways). However, in the EVM execution context, testing locally isn't really a true indicator of real-world performance. This is because the chain state on mainnet (or whatever you're running) is much more complex. Executing transactions against this state, compared to a relatively empty local state, will lead to different state trie access patterns, which will change how performance is affected.

We need something that can reliably & repeatably simulate transactions & bundles on real-world chain state. But sometimes you don't even need identical block-per-block results, you just need a convenient way to send a bunch of traffic to your live node to see how it behaves under load.

Contender aims to cover all these use cases.

previous work

Some work has already been done to make local testing more realistic:

mev-flood

mev-flood was my first attempt at something like this. It deploys the Uniswap V2 protocol along with some tokens to trade, and allows users to run scripts wherein some agents make simple token trades, and others execute arbitrage on those transactions. All this is sent to the target RPC node via eth_sendRawTransaction

or eth_sendBundle

(or mev_sendBundle

if you're targeting mev-share).

It does a good job at generating lots of juicy transactions/bundles for builders to play with, but it has some glaring drawbacks:

- it only supports Uniswap trades
- it's not designed to extend to other protocols/scenarios
- it doesn't provide any tooling to replicate a realistic chain state before running scenarios

gas-benchmarks

Nethermind's gas-benchmarks project provides a nice way to repeatably test a node's performance. It includes scripts which build a series of blocks to recreate some chain state. The test scripts (example) are defined as newline-delimited text files which specify a hardcoded array of instructions (primarily consisting of calls to engine_newPayloadV3

and engine_forkchoiceUpdatedV3

, which builds blocks).

These do work well, but the tests are very specific and static by nature. They aren't simple to change or extend.

This file will set up your node for a repeatable benchmark, but what if you want to modify it? With this design, you need a lot of different files that look like this. They're quite large, and not easy to read (for humans). Additionally, you don't get much information about what's actually happening onchain just by looking at the file; you just see that you're building a bunch of blocks.

While an inflexible test definition does provide a reliable, repeatable benchmark, the cost of adding new tests doesn't scale well. Each new testing scenario needs a whole new set of blocks built and hardcoded into a new text file. However, I wasn't able to find how they're generated in the first place — it could be simpler than I think.

more?

It was surprisingly hard to find any projects that benchmark ethereum nodes in a realistic way. If you know of any projects that do this kind of thing, please mention them in the comments!

enter contender

The core idea behind contender is simple: deterministically spam any execution-layer client to produce realistic, repeatable benchmarks. So far, we've built out the following features:

- deploy & configure onchain protocols and test scenarios with TOML-based config file
- requests are split into three categories: create

, setup

, and spam

, allowing you to execute various stages of your testing setup independently

- requests are split into three categories: create

, setup

, and spam

, allowing you to execute various stages of your testing setup independently

- write deterministic test scenarios (in the EVM context)
- TxRequest fields can be "fuzzed" — "fuzzed fields" are replaced at runtime
- fuzzed values can be manually seeded for deterministic, repeatable, random-looking numbers
- TxRequest fields can be "fuzzed" — "fuzzed fields" are replaced at runtime
- fuzzed values can be manually seeded for deterministic, repeatable, random-looking numbers
- spam target RPC with eth_sendRawTransaction

and eth_sendBundle

- collect data on time-to-inclusion

The North Star for contender is to generate the most realistic, most repeatable benchmarks possible, and to provide a universal benchmark for execution-layer clients. But we've still got work to do before we can accomplish all of that.

technical context

To provide context for our goals, this section provides a high-level technical outline of contender in its current state.

Here's a quick introduction video that shows off the current feature set:

https://youtu.be/uPkLTxFZXgY

concept: scenarios

A "scenario file" in contender-speak refers to a TOML file that specifies all the necessary information about a test scenario; contracts to deploy, setup steps to run before spamming, and transaction call templates used to generate transactions to spam your target node.

Scenarios define transaction request templates, in which you can specify function arguments to fuzz, accounts to send from,

and how much ether to send with the tx. Deployed contracts can be referenced by giving them a name and mentioning the name in a {placeholder}. Placeholders can also be used to inject custom variables which are specified in the file.

You can learn more about how these are defined by checking out the examples:
https://github.com/flashbots/contender/tree/main/scenarios

concepts: generators & spammers

tl;dr:

- Generators

implement an API that produces transaction requests

, which are wrapped with metadata to be used in later stages. Here, we only specify calldata which triggers function calls to specific smart contracts which are determined by the scenario configuration.

- Spammers

are responsible for taking transaction requests

from Generators and turning them into executable transactions

(fetching & setting gas price/limit, nonce, etc, then signing the tx if needed), and then sending those to the appropriate JSON-RPC endpoint.

more on generators

Generators address your onchain test targets; they get the contract addresses you plan to interact with, and encode function calls to them. The Generator

trait actually implements most of the functionality generically (see load_txs

); you just need to give it a DB and a Templater (for both of which we provide built-in default implementations).

Templaters

simply replace placeholder values; they're used by Generators to build transaction requests out of templates (e.g. replacing the to

address with a newly-deployed contract address). Templater is built as a trait, so you can easily write your own in case you want to customize your own templates.

Generators also make use of the Seeder

trait, which is responsible for generating values to use in transactions at runtime. The implementation used in the contender cli uses RandSeed

to repeatably generate random-looking values from a user-provided seed.

more on Spammers

Spammers are responsible for getting transactions from Generators, and sending them to the RPC node of your choice, each implementing a specific "spamming strategy." Contender has two built-in spammers: TimedSpammer

and BlockwiseSpammer

. Currently, spammers are able to send transactions (eth_sendRawTransaction

) and bundles (eth_sendBundle

).

TimedSpammer

signs & sends X requests per second (which can be used to extrapolate gas/second). It's also useful in cases where your node doesn't support eth_getFilterChanges

, which is used to listen for new blocks in the next spammer.

BlockwiseSpammer

signs and sends a batch of requests as soon as a new block is published, using eth_getFilterChanges

.

Both spammers are available to use via the CLI in the spam

command, using --tps

for the timed spammer, or --tpb

for blockwise.

contender long-term goals

The overarching goal of contender is to provide the most accurate, reliable, and portable benchmarks for EVM execution-layer clients. We've made some progress in this direction with the current build, but it still needs a few things.

Immediate next key goals are outlined below.

test against live-chain-equivalent state

Every tx causes the node to change state, and repeating the test without resetting the chain state may produce slightly different results. Deploying a test protocol is one thing, but an actively-used production deployment may look substantially different, and can be very hard to model.

Task

: provide tooling to ensure consistent base chain state against which we can run our tests.

make test scenarios easier to use & model

The TOML-based scenario configurations we use now are quite verbose, and while they do the job well, they're still a bit hard on the eyes. Additionally, a scenario config may require several changes based on the target node, which can be tedious and lead to mistakes.

Task 1

: provide an interactive interface for users to configure/edit custom test scenarios.

Task 2

: provide sensible defaults that require zero configuration ("plug-n-play benchmarks").

moving forward

Contender is a tool for client developers by client developers. Development is currently in the pre-alpha stage, so there's still a lot to do.

https://github.com/flashbots/contender

If you're interested in contributing to the project, here are some more key areas that need eyes:

- universal solution to replicate mainnet state on a target node
- relying on standard JSON-RPC methods
- relying on standard JSON-RPC methods
- default testing scenarios
- scenarios that emphasize state-trie access; benchmark writes/rewrites of specific state trie sections
- one small branch's leaves (a narrow trie)
- every leaf (a broad trie)
- one small branch's leaves (a narrow trie)
- every leaf (a broad trie)
- scenario to fill a block with transactions

- common transfers
- simple smart contract calls (write a contract that stores some data when a function is called, then call that function)
- common transfers
- simple smart contract calls (write a contract that stores some data when a function is called, then call that function)
- broad-spectrum opcode usage
- use every opcode possible, to provide a robust and complex test
- use every opcode possible, to provide a robust and complex test
- … more ideas welcome!
- scenarios that emphasize state-trie access; benchmark writes/rewrites of specific state trie sections
- one small branch's leaves (a narrow trie)
- every leaf (a broad trie)
- one small branch's leaves (a narrow trie)
- every leaf (a broad trie)
- scenario to fill a block with transactions
- common transfers
- simple smart contract calls (write a contract that stores some data when a function is called, then call that function)
- common transfers
- simple smart contract calls (write a contract that stores some data when a function is called, then call that function)
- broad-spectrum opcode usage
- use every opcode possible, to provide a robust and complex test
- use every opcode possible, to provide a robust and complex test
- … more ideas welcome!
- support more RPC methods
- support simulations with eth_call

and eth_callBundle

- support block building with engine_newPayload[V3]

& engine_forkchoiceUpdated[V3]

- support simulations with eth_call

and eth_callBundle

- support block building with engine_newPayload[V3]

& engine_forkchoiceUpdated[V3]

We want to hear your thoughts and feedback throughout the entire development process. No issue or PR is too small, so we encourage you to fork the repo, try it yourself, and let us know what you think!

p.s.

Also check out builder-playground and rbuilder.

builder-playground runs a fully-featured PBS stack locally, to which you can send bundles using rbuilder!

in builder-playground/

go run main.go

in rbuilder/

cargo run --bin rbuilder run config-playground.toml

note: you may need to edit the config file here to make bundles work alongside mempool txs

try enabling all builder algos

in contender/

git checkout send-bundles cd cli cargo run -- setup ./scenarios/spamMe.toml http://localhost:8545 cargo run -- spam ./scenarios/spamMe.toml http://localhost:8545 -b http://localhost:8645 --tpb 1 -d 2