## horizontal line



Hyperledger Improvement Project (HIP):

busywork Exerciser for the Hyperledger Fabric v0.2

2016.05.12

**─**

Bishop Brock

IBM Research

11501 Burnet Road

Austin, Texas 78758

bcbrock@us.ibm.com

# Abstract

***busywork*** is an exerciser framework for the [Hyperledger fabric](https://github.com/hyperledger/fabric) project. As an *exerciser*, ***busywork*** is not a real blockchain application or applications, but is simply a set of scripts, programs and utilities for stressing the blockchain fabric in various, often randomized ways - hence the name. ***busywork*** applications can be used both for correctness and performance testing, as well as for simple benchmarks.

# Context

This is a new project that is proposed to be included in the [*hyperledger/fabric*](https://github.com/hyperledger/fabric) repository. The project code currently exists as an internal repository inside IBM, and is under active development and use inside of IBM for correctness testing and performance characterization. This is currently a private project within IBM Research. The IBM blockchain development team, and other members of the *hyperledger/fabric* community may be simultaneously working on similar capabilities.

# Motivation

The form of the project is motivated by techniques used in the verification and validation of large systems. Lacking formal specifications and proofs of correctness, and in some cases even formal coverage models, large hardware and hardware/firmware systems are validated by continuously executing comprehensive, highly-randomized self-checking exercisers [1, 2]. This is admittedly a brute-force approach to verification, but when approached conscientiously has been proven to be very effective at hitting “corners” of complex protocols and finding bugs.

The ***behave*** tests that are in current use for continuous integration (CI) testing of the *hyperledger/fabric* offer minimal test coverage, and thus the false sense of security that if the system is able to pass these tests, then the system is “working”. A quick study of the open issues and pull requests, however, will show that the *hyperledger/fabric* is currently quite fragile and fails under even moderate load. If the *hyperledger/fabric* is to become the transaction processing backbone of a new class of enterprise applications then we need to quickly and dramatically improve the robustness of the system, and exerciser frameworks like ***busywork*** are a proven way to do this.

In terms of performance characterization, ***busywork*** applications can serve as simple benchmarks or profiling workloads in that they are designed to be easily configurable to stress different aspects of the blockchain protocols and databases. In the end, standard benchmarks are probably best when based on real-world applications rather than artificial test applications, but ***busywork*** applications might still have a place similar to microbenchmarks used in other areas.

# Status : Incubation

# Solution

As suggested in the *Motivation* section, ***busywork*** is a framework for developing and executing comprehensive, highly-randomized self-checking exercisers. The solution is currently under development, but even in this initial state has already proven useful in *hyperledger/fabric* testing and characterization. See for example Issues [#1091](https://github.com/hyperledger/fabric/issues/1091), [#996](https://github.com/hyperledger/fabric/issues/996), [#915](https://github.com/hyperledger/fabric/issues/915) and [#647](https://github.com/hyperledger/fabric/issues/647), which were uncovered or studied with busywork tests and benchmarks. The current solution consists of several parts:

* Scripts that create *hyperledger/fabric* networks. Currently these are standalone or Docker-compose networks, but in the future could be based on OpenStack or POWER VM virtual machines. There are likely a few other community efforts in this area that could be consolidated here (and offer improvements over what busywork currently provides).
* Scripts for reporting and comparing blockchains generated on different peers
* Self-checking or easily checkable chaincodes, parameterizable to allow stressing the blockchain infrastructure along various dimensions. Currently there is one Go language chaincode with a simple semantics, however supporting
  + Unlimited instances of the chaincode to be running simultaneously
  + Configurable number of data objects to be managed by each chaincode
  + Each data object potentially of a different size
  + Self-checking that the expected data context of the chaincode is correct and consistent on each invocation
  + Support for generating non-deterministic results (error injection)
* An environment for creating test drivers. This environment is currently based on the Tcl programming language. The driver for the chaincode mentioned above currently supports the following features:
  + Configurable/randomized transaction targeting including multiple client threads, bursting, continuous targeting or closed-loop interlock (i.e., waiting for batches of transactions to commit before continuing)
  + Configurable security setup (1 or multiple users, per network or per peer)
  + Comprehensive consistency checking of the final state of all peers at the end of the run

***busywork*** comprises a philosophy for how to exercise the *hyperledger/fabric*, along with a working prototype implementation. As a philosophy, there is no reason that chaincodes and drivers written in other than the current implementation languages could not be included here. One end goal would be to develop an *omnibus* exerciser that simultaneously exercises as many chaincodes, languages, security protocols, etc. as possible.

The project code would be licensed under the Apache 2.0 license when released.

# Effort and Resources Committed

Currently one person (Bishop Brock, IBM Research) is committed part-time to developing and maintaining the project. This is not the main thrust of his research however. To realize its full potential, other developers or test engineers will need to become involved.

# How To

The user-facing parts of the project are currently well-documented (in our opinion), with README files and documented code. In many cases we find it easier to maintain consistent documentation directly in the headers of scripts and Makefiles rather than as separate files or documents. Some of the library code is several years old and admittedly not well documented, but that would not be an issue for users, only certain developers.

# Closure

The project will succeed if people involved in CI and testing find it useful and integrate it into their environments, or if researchers publish performance results based on ***busywork*** benchmarks. These results should be easy to measure. The project may also be considered to have succeeded if chaincodes or other ideas are taken from ***busywork*** and integrated into other *hyperledger/fabric* testing frameworks that might prove to be more popular, even if ***busywork*** eventually falls out of use.

# Addendum

During the discussion with the Hyperledger Technical Steering Committee today, it was suggested to add a section listing all of the things that an exerciser/benchmark like this might test about a generic blockchain. Wherever the word *test* is used below, one could equally substitute the word *specify* and use those bullets as a checklist for a specification document. Here are my thoughts:

1. Test all possible ways that a network might be set up and run. In the Hyperledger fabric this includes
   1. Single-node networks where the peer is either a process or a container;
   2. Multiple-node networks where the peers are individual processes, containers running on a single system, VMs in a cloud, or widely distributed nodes.
2. Test all possible ways that chaincodes (smart contracts) might be deployed. The Hyperledger fabric currently supports
   1. System chaincodes that are compiled into the peers;
   2. Independent chaincodes deployed as Docker containers in the node/VM;
   3. (TBD) chaincodes or sets of chaincodes deployed in secure VMs separate from the peer.
3. Test all possible ways that smart contracts might be implemented, e.g., all supported programming languages
4. Vary the number of smart contracts deployed
5. Vary the number of accounts (or other appropriate metric) managed by each smart contract
6. Test/vary the sizes and types of the data objects managed by smart contract. The current Hyperledger fabric supports
   1. Key/value store;
   2. Row store;
   3. UTXO semantics
7. Test all possible ways that smart contracts can interact with each other. Currently Hyperledger fabric chaincodes can *invoke* each other. Other interactions may be available in the future
8. Test all possible ways that the peer network can interact with the environment. I’m particularly thinking of the *event* mechanism in the Hyperledger fabric
9. Test all possible ways that the chaincodes can interact with the environment
10. Test all consensus mechanisms, under a wide variety of use cases. In the end this will require a sophisticated error injection methodology and careful consideration of the correctness specification. Cases:
    1. With a fully functional infrastructure;
    2. With failed or intermittently failing peers;
    3. With malevolent peers;
    4. With temporary network outages;
    5. With smart contracts that fail;
    6. With nondeterministic smart contracts
11. Test correctness of the blockchain
    1. All transactions accepted by the network are eventually committed on all functional peers;
    2. Correct block structure, depending on the implementation. Either
       1. The block structure is always identical on all peers;
       2. All forks are eventually resolved to yield identical block structures on all peers
    3. No unexpected or duplicate transactions appear;
    4. Global ordering constraints, if applicable
12. Test blockchain checkpointing and recovery (if appropriate)
13. Test what happens if, by extreme bad luck, hashes or other “statistically unique” identifiers are not in fact unique
14. Test the ACID properties of blockchain transactions (as appropriate)
15. Test using varying number of clients
16. Test various client configurations, from all peers and clients running on a single system to a globally distributed peer network with globally distributed clients
17. Load testing, including
    1. Continuous transaction injection;
    2. Burst injection;
    3. Clients that wait for sets of transactions to commit before issuing new transactions;
    4. Differential load on peers (e.g., send all transactions to a single peer);
    5. Transactions that by design or parameterization take variable amounts of time to execute
18. Test transaction execution time limits, timeouts and recovery (if appropriate)
19. Test security infrastructure. This is a deep area (and one I am not particularly familiar with) that could probably include as many bullets as in this list so far:
    1. Without security. Although this is not a realistic use case, removing security overhead may help provide ultimate stress to the underlying infrastructure;
    2. With various security protocols
20. Performance testing
    1. Performance benchmarks based on comprehensive test cases;
    2. Quick performance regressions for Continuous Integration;
    3. Microbenchmarks, e.g., the **busywork** suite includes a set of microbenchmarks for Go language cryptographic primitives

# References

1. Schubert, K., Ludden, J., Ayub, S., Behrend, J., Brock, B., Copty, F., German, S. M., Horbach, H., Jackson, J. R., Keuerleber, K., Koesters, J., Leitner, L. S., Meil, G. B., Meissner, C., Morad, R., Nahir, A., Paruthi, V., Peterson, R. D., Pratt R. R., Rimon, M. and Schumann, J., “Solutions to POWER8 Verification Challenges”, *IBM J. Res. & Dev.*, 59(1), January/February 2015, pp. 11:1-11:17.

2. Schubert, K.-D., Roesner, W., Ludden, J. M., Jackson, J., Buchert, J., Paruthi, V., Behm, M., Ziv, A., Schumann, J., Meissner, C., Koesters, J., Hsu, J. and Brock, B., “Functional verification of the IBM POWER7 microprocessor and POWER7 multiprocessor systems”, *IBM J. Res. & Dev.*, 55 (3) May/June 2011, pp. 10:1 – 10:17.