Appendix A
Artifact Appendix

This appendix outlines the evaluation methodology for our artifacts. In Section VI, we present experimental results by deploying Ipotane on *Amazon Web Service* (AWS) with replicas distributed across five geographically dispersed regions: N.Virginia, Stockholm, Tokyo, Sydney, and N.California. Since configuring AWS involves a relatively complex process, we additionally provide local experimental instructions that can be executed on a single machine, thereby enabling easy validation of the code's functionality.

A. Description & Requirements

1) How to access: Source codes of Ipotane are available on Github[1]. Detailed configuration and step-by-step execution instructions are available in the README.md file in the repository.

2) Hardware dependencies: No special hardware requirements are required.

3) Software dependencies: Ubuntu 22.04 LTS is recommended as the operating system. Other Linux distributions can technically support deployment, as long as the operator can complete the configuration of the environment dependencies. The runtime environment mandates the installation of Python 3.9 or later, Rust compiler version 1.50.0 or newer (with full toolchain support), Clang compiler (with standard library headers), and tmux terminal multiplexer for session management.

4) Benchmarks: We select two-chain VABA, Ditto, Abraxas, and ParBFT as our benchmarks. Among these, two-chain VABA[2] represents a purely asynchronous protocol, Ditto[2] exemplifies a serial dual-path asynchronous protocol, while both Abraxas[3] and ParBFT[4] serve as representatives of parallel dual-path asynchronous protocols.

B. Artifact Installation & Configuration

Our repository can be downloaded using the `git clone https://github.com/CGCL-codes/ipotane` command. We provide two ways to run our code: one is for local testing, and the other is for running on AWS.

1) Local testing: We offer two options for installing dependencies. The first is a 'dockerfile' that automatically generates a Docker image with all dependencies installed. The second is a manual installation method. To facilitate manual installation, we also provide a `build.sh` script. We recommend using the Docker approach for installation and execution. Detailed installation and configuration instructions will be provided in the *[Preparation]* part of Section A-D1.

2) Testing on AWS: After setting up AWS credentials and SSH keys, you can configure the environment by running commands such as `fab create` and `fab install`. For specific details, please refer to the *[Preparation]* part of Section A-D2.

[1]https://github.com/CGCL-codes/ipotane
[2]https://github.com/danielxiangzl/Ditto
[3]https://github.com/sochsenreither/abraxas
[4]https://github.com/ac-dcz/parbft-parbft1-rust

C. Major Claims

- (C1): Ipotane is a novel dual-path asynchronous BFT consensus protocol that matches the performance of partially-synchronous protocols under favorable conditions while maintaining throughput and latency comparable to purely asynchronous protocols in unfavorable conditions.
- (C2): Under varying replica failure rates (ρ), Ipotane consistently achieves near-optimal performance. Specifically, when ρ=0, Ipotane performs on par with partially synchronous protocols, while at ρ=100%, its performance remains comparable to purely asynchronous protocols. These are supported by experiments, whose results are demonstrated in Section VI.B and VI.E, as well as Figures 6 and 9.
- (C3): In favorable conditions, Ipotane can continuously commit blocks by leveraging the two-chain HotStuff protocol. In unfavorable conditions, it still achieves block commitment through consecutive DBA instances. This ensures stable throughput and latency across varying conditions. Experimental results, presented in Sections VI.C and VI.D as well as Figures 7 and 8, validate this performance.

D. Evaluation

In this section, we present the workflows for running Ipotane locally and on AWS.

1) Local experiment process: Local deployment of Ipotane is relatively simple to implement.

[Preparation] There are two options to set up the testing environment.

Option 1: With Docker (Recommended). After changing to the project directory, execute the command below to build the Docker image, which has installed all dependencies required to run the experiment.

```
docker build -t ipotane
```

Then, execute the following command to launch a Docker container instance and enter its shell.

```
docker run -it ipotane /bin/bash
```

Option 2: Without Docker. You may choose to manually install the required dependencies, including:

- Rust 1.50.0+
- Python 3.9+
- tmux (for running processes in the background)
- Clang (dependency for RocksDB compilation)

For convenience, we include a `build.sh` script that automates the installation of all required dependencies.

[Execution] After successfully launching the Docker container or completing the manual environment setup, you can now perform the following operations:

```
git clone https://github.com/CGCL-codes/
ipotane
```

```
cd ipotane && cargo build
cd benchmark
pip install -r requirements.txt
```

These commands serve to clone the repository and install the required Python libraries. Note that the initial `cargo build` execution may take considerable time, as our implementation utilizes RocksDB—which requires compilation during this step.

To run the system, execute the `fab local` command within the ipotane/benchmark directory. The benchmark parameters can be customized in `fabfile.py`. Key configuration categories include:

[Benchmark parameters (bench_params)]

- `nodes`: number of replicas to run (default: 4)
- `duration`: test duration in seconds (default: 30)

[Node parameters (node_params)]

- `random_ddos`: whether to mount random DDos attacks on the leaders (default: False)
- `random_ddos_chance`: the probability of mounting random DDos attacks (default: 0)

[Results] When the `fab local` command completes, it displays an execution summary in the console and automatically saves detailed logs to the `logs` directory. You can use `fab logs` to parse these logs again, generating formatted results that match the console output and are saved to the `results` directory.

Using the default parameters described above, the Ipotane system will run locally with four replicas deployed on a single machine. These replicas benefit from an optimized network environment, resulting in significantly reduced latency measurements. This differs from the results reported in Section VI of our paper, which were obtained under *Wide Area Network* (WAN) conditions.

2) AWS-Based experiment process: The key difference between AWS and local deployments of Ipotane is in the preparation phase.

[Preparation] To deploy Ipotane on AWS, the following configuration steps must first be completed to set up the experimental environment.

- **Configure AWS credentials.** Enable programmatic access to your AWS account from your local machine. These credentials will authorize your system to programmatically create, modify, and delete EC2 instances.
- **Add SSH public key.** Manually add your SSH public key to each AWS region you intend to use.
- **Testbed Configuration** The file `settings.json` located in `ipotane/benchmark` contains all the configuration parameters of the testbed to deploy.
- **Testbed configuration.** Modify the `settings.json` file located in ipotane/benchmark to configure your testbed parameters.
- **Testbed deployment.** Execute `fab create` to provision new AWS instances. The creation logic is defined in `fabfile.py` under the 'create' task.
- **Dependency installation.** Run `fab install` to: (1) clone the repository on remote instances, and (2) install Rust language prerequisites.

For routine maintenance:

- Use `fab stop` to gracefully shut down the testbed.
- Use `fab start` to restart the testbed without recreating instances.

[Execution] After setting up the testbed, execute the protocol on AWS instances by running `fab remote`.

[Results] The `fab remote` command automatically collects logs from all replicas, enabling the result aggregation and log analysis similar to the local experiment workflow.

E. Customization

In addition to the parameters mentioned in Section D (`nodes`, `duration`, `ddoS`, `random_ddos`), you can also modify other parameters in `fabfile.py`, including:

- `tx_size`: transaction size in bytes (default: 512)
- `rate`: transactions input per second (default: 10,000)
- `faults`: Byzantine replicas to simulate (default: 0)