QUANTAS
Quantitative User-friendly Adaptable Networked Things Abstract Simulator

Joseph Oglio, Kendric Hood, Mikhail Nesterenko
joglio@kent.edu,khood5@kent.edu,mikhail@cs.kent.edu
Kent State University
Kent, USA

Sébastien Tixeuil
Sebastien.Tixeuil@lip6.fr
Sorbonne Université, CNRS, LIP6
FR-75005, France

ABSTRACT

We present QUANTAS: a simulator that enables quantitative performance analysis of distributed algorithms. It has a number of attractive features. QUANTAS is an abstract simulator, therefore, the obtained results are not affected by the specifics of a particular network or operating system architecture. QUANTAS allows distributed algorithms researchers to quickly investigate a potential solution and collect data about its performance. QUANTAS programming is relatively straightforward and is accessible to theoretical researchers working in this area. To demonstrate QUANTAS capabilities, we implement and compare the behavior of two representative examples from four major classes of distributed algorithms: blockchains, distributed hash tables, consensus, and reliable data link message transmission.

ACM Reference Format:
Joseph Oglio, Kendric Hood, Mikhail Nesterenko and Sébastien Tixeuil. 2022. QUANTAS Quantitative User-friendly Adaptable Networked Things Abstract Simulator . In *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED '22), July 25, 2022, Salerno, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3524053.3542744

1 INTRODUCTION

Theoretical work in distributed algorithms often involves establishing a possibility of solution existence, proving an algorithm correct or determining its message or computation complexity. If the new algorithm improves on the existing ones, this improvement is quantified in terms of these complexity metrics. These approaches may be lacking as they do not provide sufficient insight into the realistic behavior of the algorithm. Indeed, hidden constants and system parameters, such as message delay or relative computation power, influence the performance of most algorithms.

Alternatively, the algorithm is implemented in a real distributed architecture such as a computer cluster, a collection of virtual machines, a cloud computing service [13], or using a general purpose network simulator such as ns-3 [25] or OMNET++ [28]. Although such efforts demonstrate practical algorithm implementation and enable its immediate application, the obtained results make it difficult to separate the operation of the algorithm from the influence of the particular network and operating system used in performance evaluation. For example, it is unclear how the interaction between virtual machines and network switches at the server farm affects the results obtained in real network performance evaluation or whether the selection of a particular physical layer network protocol made a difference in a network simulator.

Another obstacle for these approaches is difficulty of performing large-scale performance evaluation. Large scale physical systems are expensive to procure for experimentation. Moreover, in a physical system, there is difficulty instrumenting and then ascertaining conditions of interest for experimenter such as specific network delay. Network simulators, due to extensive simulation detail, also have limited scalability.

To demonstrate the behavior of a distributed algorithm and compare it with the alternatives, abstract simulation may be used. Abstract simulation closely follows the communication and computation model used in distributed algorithms research. The algorithm is represented as a collection of nodes and communication channels or shared variables. The computation is modeled as series of simulation rounds where nodes perform concurrent processing and exchange messages. Such modelling of algorithms make abstract simulation attractive to distributed algorithms researchers.

However, we believe there is a lack of general purpose tools for such abstract simulation. Most papers use ad hoc one-off implementations built for one paper [3, 7, 12, 16] or, at best, a domain specific abstract simulation that is reused for a limited number of papers [4, 5, 22]. The simulation code and obtained data are seldom made publicly available. This duplicates effort and makes it difficult to verify obtained results, compare them across several publications, or make further improvements.

The existing general purpose abstract simulators, that we are aware of, tend to be used for education rather than research. However, we think that the focus of an educational simulator differs from that of a research simulator. Indeed, the major concern of an educational simulator is to give novices an exposure to the distributed algorithm operation, and a visual representation of the algorithm as it executes [2, 10, 19]. Therefore, simplicity and ease of use are of primary importance. While simplicity certainly does not harm a distributed algorithm simulator, other important characteristics such as scalability, simulation speed, versatility, and ability to obtain quantitative measures for metrics of interest come to the fore. The closest simulation framework we could find is Neko [27]. However, each simulated node implemented as a protocol stack on

ApPLIED '22, July 25, 2022, Salerno, Italy

ACM ISBN 978-1-4503-9280-8/22/07…$15.00
https://doi.org/10.1145/3524053.3542744

top of a Java Virtual Machine. The scalability of this approach is questionable.

In this paper, we present QUANTAS: an abstract simulator specifically designed for distributed algorithms research. Our primary research area is distributed algorithms. The development of QUANTAS arose out of our own need to do performance evaluation. We, therefore, built QUANTAS to satisfy the needs of researchers similar to our own. We used QUANTAS prototype in several studies [16, 24]. The QUANTAS software is publicly available [1] for other researchers to download and use.

2 SIMULATOR DESIGN PRINCIPLES, ARCHITECTURE AND SETUP

Design principles. Let us outline our major design principles and how QUANTAS achieves them.

Simplicity, ease-of-use: The foremost principle is simplicity, ease of use, and ability to obtain quantitative results quickly. That is, a newcomer should be able to implement algorithms, and get simulation data in a relatively straightforward manner. To demonstrate the simulator capabilities, with QUANTAS distribution, we provide a set of representative examples. The researchers can use these examples as a starting point for their own algorithms. The simulator core is coded in C++. No further compilers, libraries, or specialized languages for distributed algorithm specification are used. Parameter customization and experiment set up is done through simple text-based configuration files. QUANTAS has a relatively compact codebase: it contains about 4, 000 lines of C++ code.

Flexibility: The major simulation goal is to obtain data for analysis and presentation. QUANTAS is configured to output simulation data in JSON format for ease of further processing. One can then use various available interactive or automated tools to analyze and plot the data.

Scalability: Once the basic behavior of a distributed algorithm is ascertained, the researchers usually want to observe its behavior at scale: large system size, extensive simulated time or resource usage. To support this, we implemented QUANTAS in C++ with minimum overhead. C++ threading is used to implement concurrent simulation of multiple nodes. Potentially, the simulated network size is limited by the host computer processor and memory resources.

Terms and operation. A simulated *distributed algorithm* operates on a list of nodes connected via unicast channels. Each channel connects a single sender and a single receiver. Every node has a unique identifier. Each *computation* of the distributed algorithm is a sequence of receive-compute-send *rounds*. Each round has three *phases*: receive messages, perform local computation, and send newly formed messages. A computation *length* is its number of rounds. A message takes at least one round to pass between nodes that are directly connected through a channel. A message can be delayed. Delay length is configured. The delay is also configured to be either deterministic, uniformly random, or following a Poisson distribution. Communication channels are FIFO by default. Other message propagation delay disciplines may be added by the user. A transmitted message may be configured to be lost with a certain probability. A message may be sent to an individual node or broadcast to the entire network. A single *run* of the QUANTAS simulator executes several algorithm computations with the same parameters. This allows QUANTAS to execute multiple individual experiments for a single data point.

Architecture. QUANTAS architecture is shown in Figure 1. The components represent the larger C++ templates and classes. The components are in two categories: user-provided and the simulator proper. The user-provided components encode the algorithm to be simulated. The simulator proper components carry out the simulation. The run-time operation of the simulator is controlled by configuration files.

The *Simulation Component* configures and initializes the simulation run. It then carries out the receive-compute-send computation rounds of individual computations of the run. The Simulation Component uses the *Configuration Component* for processing user-supplied configuration file containing network topology and size, parameters of the run, message delay discipline and parameters, computation length, etc. The network topology is specified as adjacency list and can be generated by hand or by a separate tool.

Since the execution of the same round in the separate simulated nodes is not casually related, this execution is done concurrently by separate threads. To carry this out, the Simulation Component maintains a thread pool. By default, the number of threads in this thread pool is the maximum that the host computer can concurrently run.

The *Network Component* configures distributed algorithm topology, sets up communication channels and executes receive- compute- and send- phases of the round. The *Abstract Node Component* is a C++ abstract class that lists the interfaces to be implemented by a user-provided *Concrete Node* component. The main part of this interface is the code to be executed in local computation phase of the round.

The *Node Network Interface Component* executes receive and send phases of the round. In the receive phase, The Node Network Interface Component examines all the channels, and determines if any of the messages currently in transit are ready to be received. The ready-to-receive messages are made available for the computation phase. If the computation phase generates messages to be transmitted, the Node Network Interface Component collects them and puts them in the appropriate destination channels.

A message is enclosed in a packet. The packet contains the source, destination, and the delay for this particular message. The *Packet Component* provides this header and the Node Network Interface Component uses this header for message routing. The actual message format and its payload are provided by the *User-Defined Message Component*.

Let us discuss QUANTAS data output capabilities. QUANTAS provides a global logging facility, so that each component may output to the log file. All simulator components may output data about their particular operation. For example, the Node Network Interface Component may output sender and receiver identifiers for each individual message. The user-provided components may output arbitrary data, which enables user-specific metrics to be easily implemented. User-provided components have access to the computation round number maintained by the simulator. This round

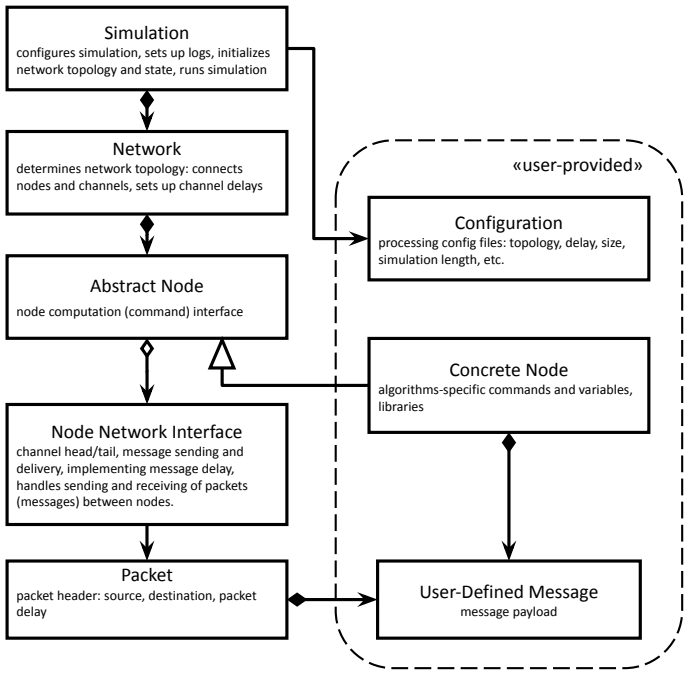

Figure 1: QUANTAS architecture.

```
struct HelloMessage {
        std::string messageText;
};
...
void performComputation() {
    HelloMessage msg;
    msg.messageText = "Hello From " + std::to_string(id()); // send "hello" to all nodes
    broadcast(msg);

    // service all received messages
    while (!inStreamEmpty()) {
        Packet<HelloMessage> newMsg = popInStream();

        // logger is a Singleton, "Greetings" is a tag
        // getRound() returns simulated computation round
        LogWriter::instance()->
            data["Greetings"].push_back(newMsg.getMessage().messageText + "at round: " + getRound())
    }
}
```

Figure 2: Example QUANTAS code for a local computation phase: each node broadcasts a single message and receives messages from its neighbors.

number can be included in the output for analysis. To simplify later processing, logger allows to attach an arbitrary tag to output lines. QUANTAS example code is shown in Figure 2.

3 EXAMPLES AND TESTS

In this section, we demonstrate how QUANTAS may be adapted to fit diverse experimentation needs of distributed algorithms researchers. We chose four domains and a pair of previously published well-known algorithms in each. We then implemented the algorithms in QUANTAS and compared their performance. While the results themselves are not surprising, they demonstrate how

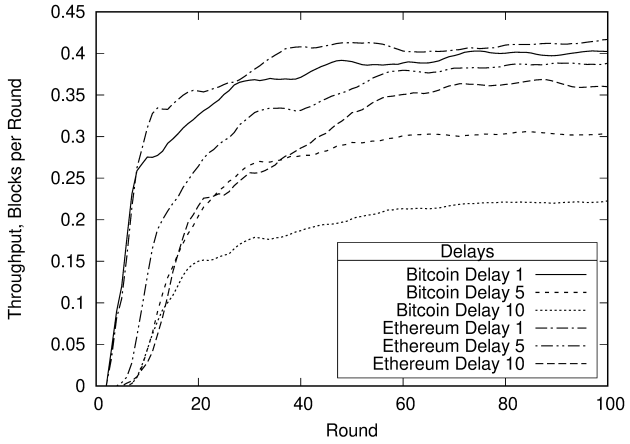

Figure 3: Blockchain throughput during a computation.

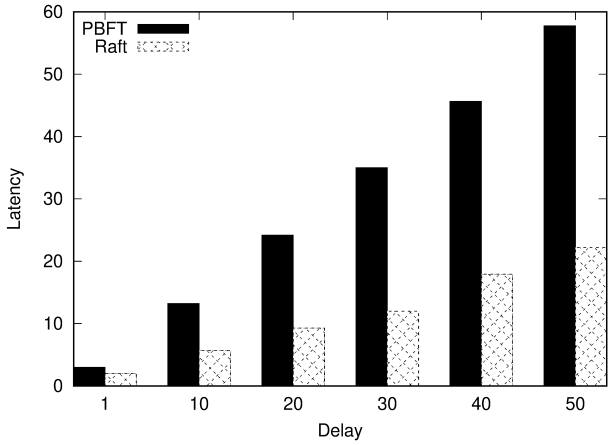

Figure 4: Consensus. Latency of achieving a single decision vs message delay.

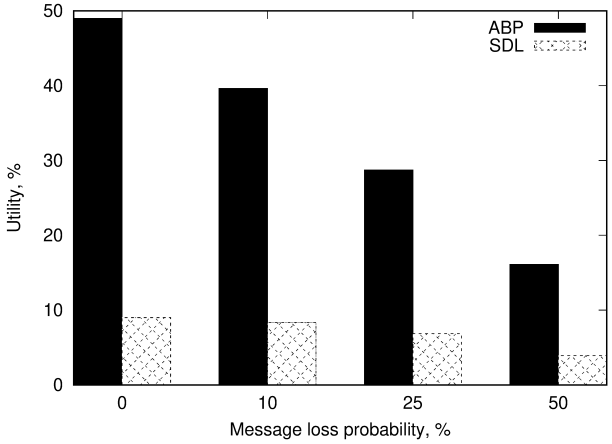

Figure 5: Reliable Data Link. Message utility depending on message loss.

QUANTAS may be used for performance evaluation of a variety of algorithms.

We also evaluated the speed of QUANTAS simulation execution. Specifically, we measured the speedup achieved by QUANTAS as threads are added to the concurrent thread pool allowing greater concurrency and faster execution.

Blockchains. Blockchain is a secure distributed ledger maintained by a network of peers that compete to add blocks of transactions to the tail of the chain. We simulate simplified versions of the two most widely used blockchain algorithms: Bitcoin [23] and Ethereum [29]. The peer-to-peer system has 20 peers. Each peer maintains its own copy of the blockchain. A transaction is submitted to a random peer with probability 5% per peer per round, i.e. on average, it is one transaction per round. The peer receiving the transaction, broadcasts it to the rest of the network. In Bitcoin, each peer mines one of the received transactions attempting to link it to the longest chain. The mining probability for each peer is 2.5%. In Ethereum, each block links to all previously unlinked blocks. The single computation was executed for 100 rounds. Each simulator run had 10 experiments.

The results are shown in Figure 3. We estimate the number of confirmed blocks by considering the longest chain for each peer and determining the shortest among those. For each blockchain algorithm, we executed a computation for 100 rounds and calculated the average number of blocks per round. Figure 3 shows a moving average of this value with a window of 5 rounds. The results shown for message delays 1 through 10. The results indicate that our implementation of Ethereum has better throughput than Bitcoin since Ethereum, unlike Bitcoin, may confirm multiple competing blocks concurrently.

Robust Consensus. In robust consensus, a network of nodes attempts to agree on a single input value. We simulated two resilient consensus algorithms: PBFT [11] and Raft [17]. Both algorithms process a sequence of consensus requests. PBFT is resilient to Byzantine faults [20]. In our implementation of PBFT, there is a fixed leader node l. The leader l has a sequence of values to commit. For the confirmation, l consults the rest of the nodes. For each individual value, l executes PBFT. Specifically, l broadcasts *pre-prepare* message to all nodes with the value to be committed. Once a node receives *pre-prepare*, it broadcasts *prepare* message. When a node receives sufficiently many *prepare*s with the same value, it commits the value and broadcasts *commit* message informing everyone of this. After receiving sufficiently many *commit*s, l considers this PBFT instance terminated and moves on to committing the next value. The leader change is not implemented.

Raft [17] is resilient to crashes and churn but not to Byzantine faults. In Raft, the leader l broadcasts requests to all nodes in the system and waits for majority of responses. After receiving this majority, l moves to the next value to be committed.

The commitment *latency* is the number of rounds it takes the algorithm from the moment the initial message is transmitted by the leader until the last required commit message is received by the leader. We used 20 nodes, we executed a computation for $1,000$

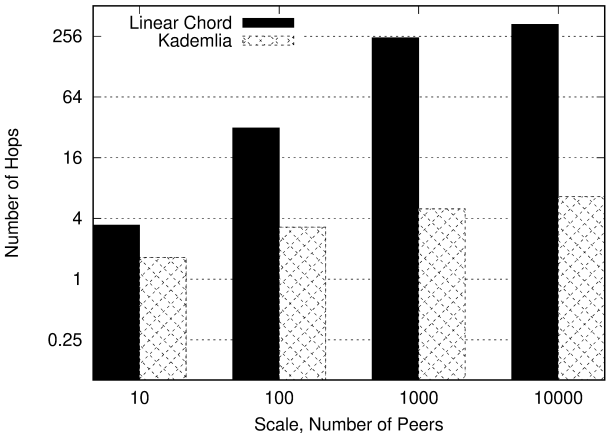

Figure 6: Distributed Hash Table query speed.

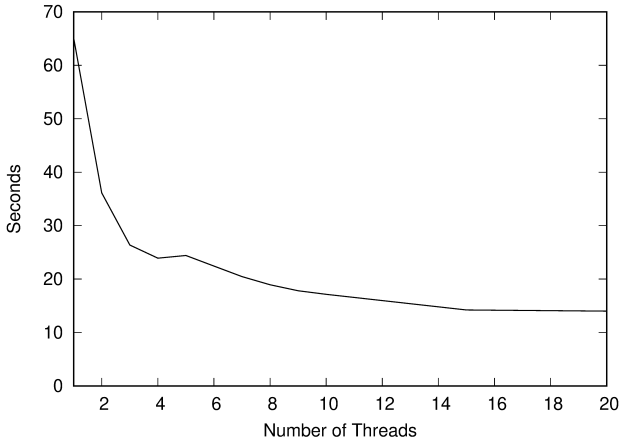

Figure 7: Distributed Hash Table simulation speedup.

rounds. Each simulator run had 10 computations. We average commitment latency across the run. We varied message delay and recorded the latency of RAFT and PBFT. The results are shown in Figure 4. RAFT has significantly lower commitment latency than PBFT as there are fewer rounds of message exchanges. This speed is obtained at the expense of resiliency to Byzantine faults.

Reliable Data Link. In a data link algorithm, the sender node attempts to transmit data to the receiver node despite message loss in the communication channel. A self-stabilizing algorithm [14] is resilient to global state corruption. We implemented two self-stabilizing data link algorithms: alternating-bit protocol (ABP) [18] and stabilizing-data link algorithm (SDL) [15]. ABP requires FIFO channels. SDL operates correctly even in non-FIFO channels. In ABP, the sender transmits a single data message and waits for acknowledgement from the receiver. If either the data message or the acknowledgement is lost, the sender times out and retransmits the message. In SDL, to enable the receiver to get messages in correct order in a non-FIFO channel, the sender transmits the same message multiple times. The number of transmissions is determined by maximum channel size.

To compare the two algorithms, we used channels of size one. For SDL, this channel size means that the sender sends the same message 5 times. In our simulation, we used 2 nodes: the sender and the receiver. We executed the computation for 100 rounds. Each simulator run was 10 computations. We computed *message utility* — the ratio of successfully received message over transmitted. We varied message loss rate and recorded the utility of the two algorithms.

The results are shown in Figure 5. In our simulation, as message loss increased, both algorithms had to submit more messages to get the data across. This means that the utility decreased for both algorithms. However, SDL effectively submitted about five times as many messages as ABP. This is the overhead needed by the SDL to enforce sequential message delivery in a non-FIFO channel.

Distributed Hash Tables. In a distributed hash table (DHT), a peer-to-peer system provides query service for key to data items spread throughout the network. The algorithm is optimized to minimize the number of lookups per query. Some of the most widely used DHTs are Chord [26] and Kademlia [21].

In our Chord implementation, the peer identifiers form a ring. Shortcut links are not implemented. A query for an identifier chosen uniformly at random is generated by another random node. The query is routed to the destination node in the shortest direction.

In Kademlia, on top of this basic Chord implementation, we also build shortcut links as follows. Peer identifiers are treated as bit fields. A prefix peer group for a particular peer p is a set of peers whose identifiers share a prefix of particular length l with p and differ from p at length $l+1$. For example, if the prefix is one bit less than the complete id length, then, there is a single member in this peer group. A peer group for a prefix that is two bits shorter than id length, contains two members. A peer group three bit shorter than id length contains 4 members and so on. For each group, a peer selects a random member and creates a shortcut link to it. The query routing is as follows. The peer selects a member with the closest prefix to the destination and routes the query there.

The results are shown in Figure 6. The results indicate that our implementation of Kademlia outperforms Chord because Kademlia query routes are logarithmic with respect to the network size.

Parallellization speedup. Debugging runs for small network sizes in QUANTAS are relatively fast: it takes one to two minutes to complete the experiment of 10 runs of 100-round computations on a network on 100 simulated notes on a laptop with 8 cores and 16 GB of RAM. This makes the QUANTAS compile-test cycle short.

To test QUANTAS parallel performance at scale, we varied the number of threads in the simulator thread pool and measured the runtime speedup of the simulation. We used Kademlia implementation. We simulated 500 peers. Each computation was 100 rounds. We run 10 computations per data point. The simulator used approximately 40 GB of RAM and ran on a virtual machine with a host machine having 2 Intel Xeon Gold 6132 CPUs running at 2.60 GHz. The virtual machine had 12 cores. The results are shown in Figure 7. The results indicate that the simulation speed increases as more threads are added to the simulator thread pool. This speed

increase ends as all available host processor cores are used for the concurrent simulation.

4 FUTURE WORK AND CONCLUSIONS

We anticipate further enhancements of QUANTAS capabilities. Besides already implemented message loss, we would like to add more sophisticated fault injection. In particular, we plan to add support for self-stabilizing algorithms evaluation. Even though a self-stabilizing algorithm is proven to recover from an arbitrary global state, evaluating the algorithm's performance starting from a state generated uniformly at random is not realistic as not all such states are equally likely to appear. A more sophisticated approach was developed by Adamek et al [6]: an achievable state of a self-stabilizing algorithm is selectively perturbed. We would like to implement this kind of fault-injection in QUANTAS. Adding crash faults would be a simple and useful addition. A more challenging task is adding Byzantine faults since Byzantine nodes are expected to behave so as to inflict the most damage to the algorithm. Hence, despite extensive studies of Byzantine fault tolerance, few of them have performance evaluation. We believe adding Byzantine fault injection [8] would be helpful to the research community.

We would like to add random topology generation that QUANTAS so that the nodes and channels are configured randomly according to the topology graph parameters provided in the configuration file, for example a random graph with the specified node number and edge probability. As a further enhancement, we would like to add the ability to change the network parameters and even network topology during a single computation. This would allow researchers to model mobile or dynamic networks.

Another feature we find useful is facilitation of application level separation. This would allow the simulator to evaluate levels of multi-level algorithms separately, for example, evaluate the same consensus algorithm over different broadcast algorithms.

Another important issue is scalability increase, be it related to the number of simulated nodes [9] or the number of simulations per data point to obtain quantitative results. To improve the performance of QUANTAS at scale, we plan to pursue multithreading inside the simulator more aggressively. For example, by using parallel algorithms in the Standard Template Library of C++. In the future, we would like to explore distributed multi-computer simulation.

In this paper, we presented QUANTAS, a general abstract simulator dedicated to distributed algorithms quantitative evaluation. While we provided a number of case studies, we welcome contributions from the Distributed Computing community, to build a library of ready-to-use templates for most algorithmic paradigms, that enables fair comparison with previous work when designing new solutions. We believe that QUANTAS fulfils the need for an abstract simulator among researchers of distributed algorithms and we hope it proves to be useful and convenient.

ACKNOWLEDGMENTS

We are thankful to Shishir Rai of Kent State University for providing code for an example as the simulator was being developed. This work is supported in part by ANR Project ESTATE (ANR-16-CE25-0009-03), and ANR Project SAPPORO (2019-CE25-0005-1).

REFERENCES

[1] Quantas: Quantitative user-friendly adaptable networked things abstract simulato. https://github.com/QuantasSupport/Quantas.
[2] Sinalgo: simulator for network algorithms. https://sinalgo.github.io/.
[3] Jordan Adamek, Giovanni Farina, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. *J. Parallel Distributed Comput.*, 109:63–74, 2017.
[4] Jordan Adamek, Mikhail Nesterenko, James Scott Robinson, and Sébastien Tixeuil. Stateless reliable geocasting. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*, pages 44–53. IEEE Computer Society, 2017.
[5] Jordan Adamek, Mikhail Nesterenko, James Scott Robinson, and Sébastien Tixeuil. Concurrent geometric multicasting. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 9:1–9:10. ACM, 2018.
[6] Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In Andréa W. Richa and Christian Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, volume 7596 of *Lecture Notes in Computer Science*, pages 126–132. Springer, 2012.
[7] Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 233–244. IEEE, 2015.
[8] Ali Asim and Sébastien Tixeuil. Advanced faults patterns for WSN dependability benchmarking. In Violet R. Syrotiuk, Fatih Alagöz, Brahim Bensaou, and Özgür B. Akan, editors, *Proceedings of the 13th International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems, MSWiM 2010, Bodrum, Turkey, October 17-21, 2010*, pages 39–48. ACM, 2010.
[9] Ali Asim and Sébastien Tixeuil. Xs-wsnet: Extreme scale wireless sensor network simulation. In *11th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WOWMOM 2010, Montreal, QC, Canada, 14-17 June, 2010*, pages 1–9. IEEE Computer Society, 2010.
[10] Arnaud Casteigts. Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks. In Georgios Theodoropoulos, editor, *Proceedings of the 8th International Conference on Simulation Tools and Techniques, Athens, Greece, August 24-26, 2015*, pages 290–292. ICST/ACM, 2015.
[11] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.
[12] Thomas Clouser, Mark Miyashita, and Mikhail Nesterenko. Fast geometric routing with concurrent face traversal. In *International Conference On Principles Of Distributed Systems*, pages 346–362. Springer, 2008.
[13] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. Microsoft azure. *New York, NY, USA:: Apress*, 2015.
[14] Edsger W Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
[15] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Information Processing Letters*, 111(18):912–920, 2011.
[16] Kendric Hood, Joseph Oglio, Mikhail Nesterenko, and Gokarna Sharma. Partitionable asynchronous cryptocurrency blockchain. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2021.
[17] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.
[18] Rodney R Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *Journal of Parallel and Distributed Computing*, 62(5):792–817, 2002.
[19] Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. LYDIAN: an extensible educational animation environment for distributed algorithms. *ACM J. Educ. Resour. Comput.*, 6(2):1:1–1:21, 2006.
[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.
[21] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
[22] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In Henning Schulzrinne, Karl Aberer, and Anwitaman Datta, editors, *Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing, 9-11 September 2009, Seattle, Washington, USA*, pages 99–100. IEEE, 2009.
[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[24] Shishir Rai, Kendric Hood, Mikhail Nesterenko, and Gokarna Sharma. Blockguard: Adaptive blockchain security. In *2nd International Conference on Blockchain Economics, Security and Protocols*, page 9, 2021.

[25] George F. Riley and Thomas R. Henderson. The *ns-3* network simulator. In Klaus Wehrle, Mesut Günes, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

[26] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[27] Peter Urban, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking*, pages 503–511. IEEE, 2001.

[28] Andras Varga. *OMNeT++*, pages 35–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[29] Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.