Distributed Hash Table With Extensible Remote Procedure Calls

Lars Creutz*, Jens Schneider* and Guido Dartmann
Institute for Software Systems
Trier University of Applied Sciences
Birkenfeld, Germany
Email: {l.creutz, j.schneider, g.dartmann}@umwelt-campus.de
*These authors contributed equally to this work

***Abstract*—In this paper we present Grond, a Python implementation of a distributed hash table based on the Chord protocol that was used inside the cyber-physical contract framework Fides. We implemented several extensions and improvements over the original Chord specification. Some of these enhancements are specific for the use with Fides, but most are general enhancements to improve the stability and security of the framework. The implementation can be transferred to other domains due to its extensibility with custom remote procedure calls and is also useable on lightweight hardware, which we show in reproducable measurements.**

I. Introduction

In [1], we presented a novel concept for cyber-physical contracts, which are an alternative to regular smart or ricardian contracts [2]. This concept was implemented within its reference implementation *Fides* and published in [3]. Decentralized systems are particularly suitable for the processing of such contracts and also underline the core idea of the concept, namely self-organization. Since there are only a few stable projects in the Python programming language that provide distributed hash table (DHT) functions in the way they are used for these cyber-physical contracts, we developed our own system *Grond*[1] as part of *Fides*. It combines established DHT technologies with modern remote procedure call (RPC) techniques. We use an extended version of the Chord protocol as DHT and implemented additional important extensions like private networks. Our results can be directly integrated into other domains to provide services via RPC methods, which are processed in the background within a distributed network. We begin by explaining related works and highlighting our contribution. After that we explain our concept and implementation in Section II and also describe our use-case scenario and possible attacks along with mitigations. After that, we demonstrate the performance of the system within different measurements in Section III. In this context, we will discuss the simulation of users, the selected hardware setup, as well as the evaluation of the measured data and the insights gained for further improvements of our implementation. In Section IV, we summarize the work and describe future improvements to the implementation.

[1]*Grond* is released as module of *Fides* and is open source available under the MIT license: https://gitlab.rlp.net/l.creutz/fides
The replication package for the evaluation of this paper can be found here: https://gitlab.rlp.net/landleuchten/grond-paper

A. Related Works

In this section we give a short description of the Chord protocol which serves, in our own implemented version including some extensions, as the basis for our cyber-physical contract system *Fides* [3] and cover further DHT implementations.

Chord [4] is a distributed peer to peer lookup protocol and accomplishes this by storing hashed keys in a distributed hash table. The individual nodes are sorted by an unique identifier, which is their hashed IP address, in ascending order in a ring-shaped structure in which each node has a pointer to its direct successor. Consistent hashing, meaning the property that each key is associated with exactly one node, is achieved by always storing keys and data tuples at exactly the first node whose ID is greater than or equal to the hashed key. Efficient searches in logarithmic time are realized by so called finger tables. These are routing tables in which each node stores information about other nodes in the ring.
Thereby, the ith entry of the finger table contains the next node in the ring that follows the current node with ID n with a distance of at least $(n + 2^{i-1}) \mod 2^m$, where m is the amount of bits of the used hash function. When searching for a certain node in the ring, the finger tables can thus be used to jump from node to node in first large and then ever smaller steps until the searched node is reached. This is achieved by always searching for the closest predecessor of the searched node in the finger tables of the nodes on the path of the search. The stability of the ring structure is ensured by the two methods **stabilize** and **fix_fingers** which are executed by each node in regular intervals. Stability in this case means the ability of the individual nodes of the ring to maintain or restore their correct successor pointers when other nodes fail or are added. The method **stabilize** works by asking the direct successor for its direct predecessor. If this node is between the requesting node and its direct successor, it is adopted as its new successor. In addition, the new successor is notified about its new predecessor so that it can also update its information accordingly. [4]
The method **fix_fingers** picks a random entry of the finger table and searches the direct successor node of its index. In

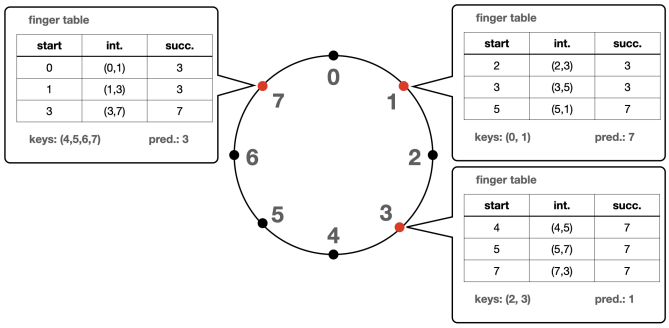

Fig. 1. Example of a Chord ring in the value range 2^3 consisting of the nodes 1, 3 and 7 with their finger tables and their assigned keys. Here int. and start are the finger table entry's interval and its respective start, while succ. is the successor node associated with the entry.

case a new successor is found, the table entry is updated accordingly. In order to increase the stability in case of node failures, not only the direct successor is held available, but a list containing several direct successors. [4]

Fig. 1 shows an example of a Chord ring with $m = 3$ containing three nodes and their corresponding finger tables.

However, there do exist several different versions of the specification and implementations of the Chord protocol. In the work of Pamela Zave [5] in which the author follows a lightweight modeling approach to verify the correctness of Chord these different versions are cited.

Apart from Chord, there are of course other peer-to-peer distributed hash tables. One such would be Kademlia [6], for example. Here, asynchronous messages are exchanged within a binary tree structure. The use of a symmetric XOR metric for routing allows to contact arbitrary nodes within given intervals. And asynchronous communication also allows parallel routing requests to be sent to appropriate nodes to prevent timeouts. As presented in [6], these features make Kademlia both efficient and robust against node failures. Another very popular protocol is Pastry [7], which is based on network locality. Using a scalar proximity metric, each node knows the route to a certain number of its network topographically nearest nodes. When routing, it is heuristically assumed that each node contains in its set of nearest nodes the route to a node that is closer to the originating node of a message.

Thus, Pastry provides an error-resistant and well-scaling basis for numerous higher level applications such as PAST [8] or SCRIBE [9]. There is also Tapestry [10], an API that uses adaptive algorithms with soft states to implement a peer to peer overlay network that provides good scalability, performance and fault tolerance for changes in the network due to node failures, for example. Efficiency in terms of latency and throughput is thereby optimized by using the locality of mobile endpoints for routing.

B. Contribution

Our contributions are novel in the sense of using extensible RPC methods in combination with a DHT in Python. The implementation is open source and can be embedded in further projects. We show reproducible measurements of a real-world application using the implementations presented here. Furthermore, we describe general approaches to improve performance in the context of our experiment, which can also be applied to other domains utilizing the same technology. We decided to extend the Chord protocol for our application in order to use it in the *Fides* framework. Since we could not find an implementation of Chord in Python that met our requirements, we have implemented our own version, which we will discuss in more detail in Section II.

II. Concept and Implementation

Below we describe the overall concept of the work and important aspects of the implementation. For the code, we refer to the official *Fides* repository, which includes the *Grond* library as an external module.

A. Fides network DHT

Our implementation was developed specifically for the *Fides* framework. Within *Fides*, cyber-physical contracts are derived from templates and then processed step by step. These templates and contracts have to be documented in a (distributed) network. For this purpose *Fides* uses a distributed index, which is updated via RPC methods within the network. The assignment of which network node, which is called a "full node" inside *Fides*, manages which template/contract is performed directly via *Grond*. The hash values of the template/contract are assigned to the respective ID of the node and the RPC requests are then distributed across the network.

B. Extensible remote procedure calls

A fundamental goal of our Chord implementation was ease of use with generic RPC methods within a wide variety of application domains. The basic approach is that the DHT functions are available in the background and the application-specific RPC methods only have to check whether the addressed node is responsible for the request. We use gRPC[2], the RPC framework from Google, for this purpose. Fig. 2 shows the relationship between the class *Grond.Node*, which implements the DHT functions, and the derived class *Application.Node*, which is extended by the RPC services (here: *gRPC.applicationServicer*). In order to offer such a service via a DHT, the RPC methods only have to be defined, implemented and added to *Grond*. The distribution of the requests is usually done via a hash, whose distance is then determined to the IDs of the nodes in the network. Accordingly, a node only has to check whether it is responsible for the current request or whether the request is forwarded. The correct location for the request is automatically determined by *Grond*.

C. Implementation

We have implemented Chord largely analogous to the specifications given in [4]. In addition, we have made some adjustments which we will explain in the following.

[2]https://grpc.io/ - Accessed 15.09.2022

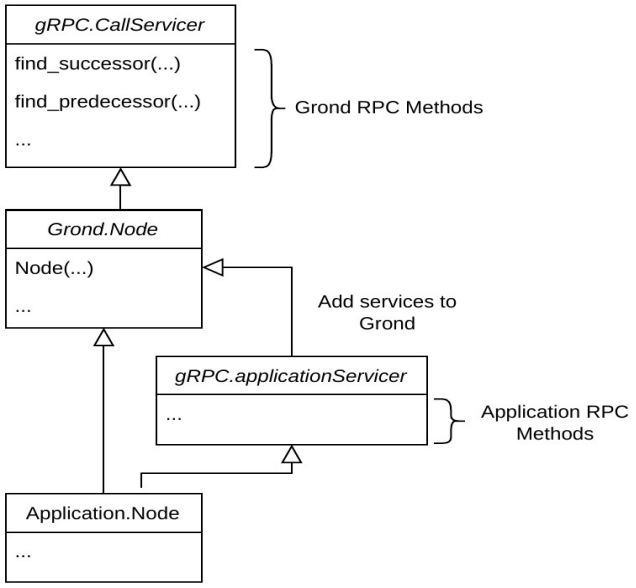

Fig. 2. Relationship between *Grond* and the application-specific RPC methods.

We have optimized connecting to other nodes by keeping stubs of once established connections for later reuse. This is advantageous because in our application it can be assumed that connections are reused multiple times. Whenever a connection should be established, we check whether a stub object already exists for the node ID to be connected and use it if so, or create a new one and save it for later reuse.

Additionally, in **fix_fingers** we do not update a random entry, instead we incrementally update one entry of the table per iteration, so that each entry is guaranteed to be kept up to date after a certain time.

We have also implemented two mechanisms to increase the security of the network. The first security enhancement we introduced is the **check** method and a network name string. Every node states to be inside a network that is defined by an arbitrary string. We do not communicate that network parameter inside the ID but use it to check the ID of other nodes. We calculate an ID of a node as follows:

$$ID = \textbf{SHA256}(network \oplus ip \oplus port) \quad (1)$$

$\oplus$ is the concatenation of the elements. Inside RPC calls nodes report their id, ip, port and in private networks a certificate. Therefore it is more difficult to join a network: the network name must also be present and part of the local ID in order to join as a full node.

The second is the use of certificates which allow the creation of private networks that can only be joined if the appropriate certificate is available. The certificate is checked in the higher level **check** method of *Fides* and can be applied in the same way to other application scenarios. This shows how easily our implementation can be extended to be specific context. The certificate string is optional and is submitted with the node information along with the ID, IP, and port on RPCs. The **check** method is called on every incoming RPC from another node to verify that the node has a correct ID, is reachable on the IP and port number it provided via RPC and is part of the same network. For this reason, we check the ID of the node as described before and additionally make sure that the node is reachable on the specified IP:port combination. For use with *Fides* we have also implemented a responsible method to be able to assign the RPC call to the correct node. This method returns for a searched key the node in the network that is associated with this key. The **find_successor** method is used for this purpose. In our implementation, we run **stabilize** and **fix_fingers** as threads. When choosing a thread interval, we have to balance between performance and stability. For both threads, we currently use 1 second as a wait period.

D. Attack vectors and mitigations

Below we present some known attacks on DHT and describe possible countermeasures. Generally speaking, most attacks involve costs/resources that are disproportionate to the benefits. This depends heavily on the specific application and the implementation (the use of the DHT), but fundamentally it is difficult for an attacking party to predict where in the network data of a particular user will be stored.

1) Sybil attack: The disruption of a network by creating many false identities is known as a Sybil attack [11]. Since it is not possible in a completely distributed system (without a central instance or other restrictions) to ensure that an identity within the network belongs to exactly one physical party an attacker might be able to create multiple identities [12].

Within the *Fides* protocol, a Sybil attack could be implemented using the *port* parameter of the ID of network nodes, since it can be arbitrary selectable within the TCP port range. To counter this, an application can specify the particular port, for example inside a configuration file, and extend the **check** method of the implementation accordingly to ensure that port is used. This significantly reduces the attack vector of a Sybil attack, since real IP addresses are now required to create multiple identities on the network, rendering the attack more expensive. An extended **check** method is easy to implement and for example used inside *Fides* to establish private *full node* networks.

2) Eclipse attack: An Eclipse attack [12] attempts to isolate individual parts of the network (possibly individual nodes). Accordingly, the attacking party places itself in the network in such a way that the node to be attacked is fully disrupted in its routing and its queries are directed to nodes managed by the attacking party. Therefore, the exact course and effort of an Eclipse attack depends on the routing protocol used. Eclipse attacks have for example already been demonstrated for Bitcoin [13] and Ethereum [14]. For the used Chord protocol, Eclipse attacks are immediately more difficult to perform due to the calculation of the ID of network nodes, but not impossible [12]. An attacking party thus has to use more resources to generate IDs with real IP addresses, in order to isolate the node to be attacked. The vulnerability to Sybil

attacks through the *port* element of the node ID does simplify such an attack.

3) Routing attack: An attack on storage/routing, involves incorrect behavior of nodes within the network [12]. Due to the openness of the software and (in public networks) the unrestricted participation in a network, such attacks cannot be prevented. An attacking party could therefore deliberately return false information to clients. If one limits the attack possibilities of Sybil or Eclipse attacks, a routing attack becomes equivalently more difficult to carry out, since it is not controllable to which parts of the network the attacking party can make false statements. A countermeasure would be to challenge nodes and observe whether the challenged node behaves correctly. If it loses the challenge, the node can be excluded accordingly.

4) Application-dependent attacks: In addition to the known attack vectors on distributed hash tables, application-specific attack vectors may occur when using our implementation. Using *Fides* [3] as an example, if a node fails, it would be possible to republish an incorrect state of a contract. Accordingly, such a failure must be detected by the other party to the contract and not cause errors there. In general, this type of attack must be evaluated and prevented individually depending on the use case.

III. Experiment and Simulation

In the following we introduce our experiment. We explain how we simulate users and describe our tests and their results. Furthermore, we evaluate the results and describe possible changes of the implementation to improve the system in general.

A. Client simulation

The system can be simulated with a configurable Python script, which is part of the replication package of the paper. The following parameters can be adjusted:

- num_clients: Number of clients to simulate in parallel. A separate *Fides* instance is created for each client.
- test_num_iterations: Iterations of the measurement.
- test_num_of_contracts: Number of contracts to be created per iteration.
- test_iteration_sleep: Waiting time between iterations. However, the contracts may not have been finished yet, as those are automated in the background.

The simulation was designed in such a way that the results should provide "real" values and are not designed to overload the node to be measured. Accordingly, background updates of the respective instances should take place periodically, as well as the processing of contracts. The simulation script performs the following actions:

1) Create callback Python scripts which will be used by all instances to automate templates and contracts.
2) Create a separate *Fides* instance per client. Here the node to be measured is added as endpoint and the respective instance is configured. For the background update interval, a random value in the range of 1-60 seconds is chosen per instance, so that the respective background updates are performed at different times. Furthermore, an account is created for the respective instance and the network is started.
3) Create a template with the same contents for all clients.
4) Store the elements of the instance for easy access (ID, path to the instance, private key of the account, hash of the template).
5) Generate the automation for the created template using the *hooks* module of *Fides* and store it in the instance of the client.
6) Start the *hook* for the template.
7) Perform a single test inside a nested loop (iterations/number of contracts):
   a) Determine two different clients c1, c2. Here c1 creates the contract, c2 manages the template.
   b) Retrieve the template from the network.
   c) Load the account.
   d) Create and finalize the contract.
   e) Create and launch the *hook* for the created contract.
   f) Publish the contract.

Both template and contract *hook* check every two seconds if the state has changed locally and react to this state change if necessary. When confirming tasks in the contract, different amounts of data are randomly generated by the two parties to create different loads on the full nodes.

TABLE I
Simulation results of 2 clients, 5 iterations with a waiting period of 120s

# Contracts	CPU user %		CPU wait %		NET Rx KB		NET Tx KB	
	avg	max	avg	max	avg	max	avg	max
5	3.7	18	1.1	15	9.6	339	38.8	504
10	3.9	21	1.1	14	12.9	294	38.2	521
20	4.9	26	1.9	18	19.6	414	61.1	485
40	5.4	22	2.5	20	27.3	373	87.6	613
80	5.4	24	2.8	20	31.6	461	85.5	595
160	6.8	22	4.1	21	38.6	453	112.9	750

TABLE II
Simulation results of 80 contracts with 2, 4 and 8 clients, 5 iterations with a waiting period of 120s

# Clients	CPU user %		CPU wait %		NET Rx KB		NET Tx KB	
	avg	max	avg	max	avg	max	avg	max
2	5.4	24	2.8	20	31.6	461	85.5	595
4	6.4	26	3.4	19	33.3	552	100.8	666
8	4.3	20	1.5	14	15.9	260	50.6	623

B. Experimental setup

To measure the resource requirements of our implementation, we built a hardware setup specifically for the experiment. For this purpose, we connected four Raspberry Pi Model 3 B Rev 1.2 and a measuring laptop via an Ethernet switch. Each Raspberry Pi operates a full node of a *Fides* test network. We use Ubuntu 22.04.1 LTS[3], Python in version 3.10.4 and *Fides* in version 0.5.1. On the measuring computer, the script

[3]https://ubuntu.com - Accessed 19.09.2022

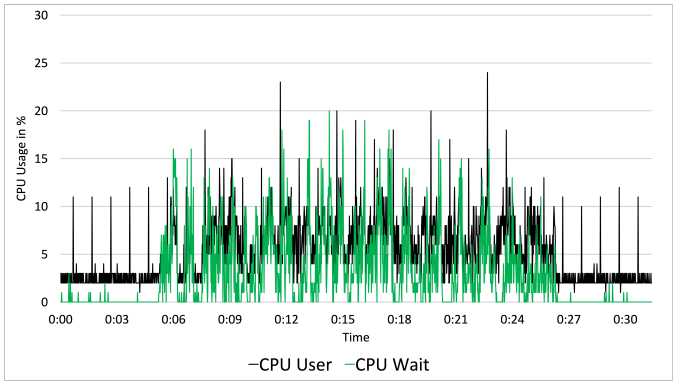

Fig. 3. CPU usage for 80 Contract on 2 clients

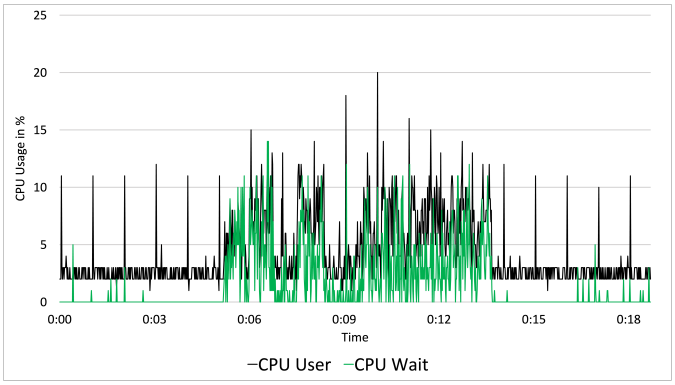

Fig. 5. CPU usage for 80 Contract on 8 clients

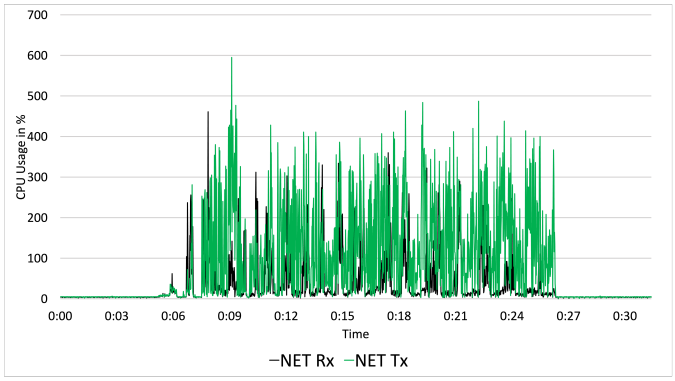

Fig. 4. Network usage for 80 Contract on 2 clients

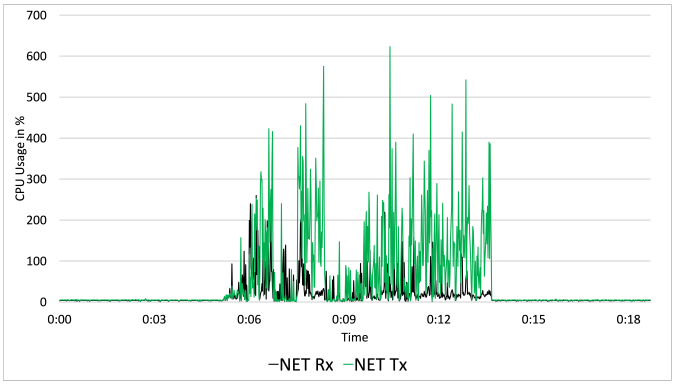

Fig. 6. Network usage for 80 Contract on 8 clients

described in III-A is executed to simulate *Fides* clients and thus generate load for the network. The switch is a Netgear Fast Ethernet Switch Model FS108 with a throughput of 10/100Mbps.

When measuring resource consumption, we analyzed the user's CPU usage, CPU wait for I/O operations, incoming and outgoing network traffic, and RAM usage. We measured different scenarios by varying the number of clients and contracts in order to be able to make well-founded statements about the resource consumption of our implementation. For two clients, we started with 5 contracts and doubled the number for each measurement up to 160 contracts. We repeated the same measurements for 4 clients. Furthermore, we measured 80 contracts with 8 clients to compare the effects of different numbers of clients and to show a situation where errors occur.

C. Results

The results of the measurements with 2 clients can be found in Table I. The RAM usage is not shown, since it remained constant at 5.4% for all measurements. The tests with the 2 simulated users show that the CPU load at maximum is very similar in all tests. In addition, an increase in the average load can be seen with the increase in contracts, which is still very low at 6.8% for the hardware used with 160 contracts. The increase in CPU wait indicates write operations in the management of the distributed index, which becomes correspondingly more as the number of objects increases. The incoming network data highlights the increase in the amount of data per test, while the outgoing network traffic shows the forwarding of requests and their responses in the network.

Table II contains the results showing the resource consumption at 80 contracts for 2, 4 and 8 clients in comparison. In addition, we have identified a load with 8 clients and 80 contracts at which errors start to occur in the *Fides* network. To show the impact of errors on performance, we have compared this measurement to a measurement with 2 clients and 80 contracts without errors. The CPU usage plot for 2 clients and 80 contracts can be found in Fig. 3 and the network traffic plot in Fig. 4. For 8 clients and 80 contracts CPU usage is shown in Fig. 5 and network traffic in Fig. 6. Looking at the CPU utilization in Fig 5, we see obvious drops and no even utilization as in Fig 3. The reduction in CPU utilization is due to piling up RPC requests that cannot be processed fast enough. This is even more noticeable on the network, where less data is received and sent over the test period than in the previous tests.

D. Evaluation

The measurements showed that our implementations can be used to easily build distributed systems even on lightweight hardware. The performance of such a network is strongly

related to the hardware used. Accordingly, restrictions may have to be imposed on the application layer so that the network cannot be overwhelmed. Since these restrictions would have to be forced by the network nodes and generally do not stop an attacking party, failures can be observed in some of our measurements. In our experiment with 80 contracts on 8 clients, the number and frequency of requests was simply too much for the low-resource hardware. The gRPC server of the DHT uses the default value for the number of threads (within a pool) to process the requests. In the Python version used, this results in ({number of CPUs} + 4)[4], which means that 8 threads are available for processing the requests with the hardware used. Furthermore, any number of RPC requests can be directed to the server (configuration element *maximum_concurrent_rpcs*) without the server reporting back that its resources are exhausted[5]. This means that the number of incoming connections by users was already equal to the maximum of threads to process the RPC requests. Furthermore, the communication between the network nodes must be considered, which is additionally sent and received by the measured node. If too many requests accumulate, this can lead to delays. Especially the selected timeout of 1 second can not be met during the communication between network nodes, which removes the overloaded node from the ring from the point of view of the other nodes. One effective approach to preventing excessive load is to operate the network nodes behind load balancers, which ensure that the number and frequency of requests can be handled by the hardware. Furthermore, prioritization between users and other network nodes is easier to perform provided that this information is known in advance. One solution within the application would be to use a variable timeout for RPC requests between network nodes. In case of a high number of requests by users, this timeout can be increased accordingly, briefly worsening the routing within the DHT (in case of fluctuations), but improving the general stability of the DHT. In the future, we plan to test different approaches and configurations to make our implementation more robust to the limitations addressed.

IV. Conclusion and future work

The paper presented *Grond* an implementation of *Chord* that is extendable with application-specific RPC methods. We then demonstrated how it is used in a real world application context inside the software *Fides*. Furthermore we measured the performance of our implementation of the DHT by simulating different scenarios on a hardware setup consisting of Raspberry Pis which we used as nodes in our testing network. Thereby we demonstrated that *Grond* works stable and reliable even on lightweight hardware. In the future, we want to continue to improve the implementation and continue to observe it in real-world applications like *Fides* to gain insights on how the software is being used. In particular, we want to investigate the points raised in the evaluation of the measurements, for example, to find a more efficient balance between robust routing and fewer failures during overloads through variable timeouts.

[4]https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor - Accessed 19.09.2022

[5]https://grpc.github.io/grpc/python/grpc.html#grpc.server - Accessed 19.09.2022

Acknowledgment

This work has been funded by the Federal Ministry for Digital and Transport (BMDV) within the research initiative "mFUND" under the grant number 19F2102F.

References

[1] L. Creutz and G. Dartmann, "Cypher social contracts a novel protocol specification for cyber physical smart contracts," in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 2020, pp. 440–447.

[2] I. Grigg, "On the intersection of Ricardian and Smart Contracts," https://iang.org/papers/intersection_ricardian_smart.html (Accessed September 2022).

[3] L. Creutz, J. Schneider, and G. Dartmann, "Fides: Distributed cyber-physical contracts," in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2021, pp. 51–60.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM computer communication review*, vol. 31, no. 4, pp. 149–160, 2001.

[5] P. Zave, "Using lightweight modeling to understand chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[6] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[8] P. Druschel and A. Rowstron, "Past: A large-scale, persistent peer-to-peer storage utility," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss elmau, Germany, IEEECompSoc*. Citeseer, 2001.

[9] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *International workshop on networked group communication*. Springer, 2001, pp. 30–43.

[10] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on selected areas in communications*, vol. 22, no. 1, pp. 41–53, 2004.

[11] J. R. Douceur, "The sybil attack," in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 251–260.

[12] G. Urdaneta, G. Pierre, and M. V. Steen, "A survey of dht security techniques," *ACM Comput. Surv.*, vol. 43, no. 2, feb 2011. [Online]. Available: https://doi.org/10.1145/1883612.1883615

[13] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on Bitcoin's Peer-to-Peer network," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 129–144. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman

[14] Y. Marcus, E. Heilman, and S. Goldberg, "Low-resource eclipse attacks on ethereum's peer-to-peer network," Cryptology ePrint Archive, Paper 2018/236, 2018, https://eprint.iacr.org/2018/236. [Online]. Available: https://eprint.iacr.org/2018/236