Towards Low-Latency Byzantine Agreement Protocols Using RDMA

Signe Rüsch
TU Braunschweig, Germany
ruesch@ibr.cs.tu-bs.de

Ines Messadi
TU Braunschweig, Germany
messadi@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

***Abstract*—Byzantine fault tolerance (BFT) protocols can mitigate attacks and errors and are increasingly investigated as consensus protocols in blockchains. However, they are traditionally considered costly in terms of message complexity and latency due to the required multiple rounds of message exchanges. With the availability of Remote Direct Memory Access (RDMA) in data centers, message exchange latency can be reduced compared to TCP, as RDMA enables kernel bypassing and thereby avoids intermediate data copying. Retaining the performance benefits for RDMA during its integration, however, is non-trivial and error-prone. While the use of RDMA has previously been explored for key/value stores, databases and distributed file systems, agreement protocols especially for BFT have so far been neglected. We investigate the usage of RDMA in the Reptor BFT protocol for low-latency agreement and show first steps towards an RDMA-enabled consensus protocol. For this, we present RUBIN, a framework offering similar functionality to the Java NIO selector, which can handle multiple network connections efficiently with a single thread and is employed in several BFT protocol implementations such as BFT-SMART and UpRight.**

***Index Terms*—Byzantine Fault Tolerance, Remote Direct Memory Access, Blockchain Consensus**

I. INTRODUCTION

In recent years, the popularity of cryptocurrencies has increased immensely, the most prominent examples being Bitcoin and Ethereum. They operate on a blockchain, i. e. a chain of blocks containing ordered transactions. These blocks are linked together as each block includes the cryptographic hash of the previous one. This prevents manipulation as any changes of the hash would be immediately noticed. However, most blockchain platforms are still severely limited with regard to throughput, i. e. the number of processed transactions per second (tps): in Bitcoin, the maximum throughput is 7tps and the latency, i. e. the time until a transaction is processed (also called "confirmation time"), is one hour. This is influenced by the employed consensus protocol: Bitcoin uses Proof-of-Work (PoW), in which miners try to create a block where the block's hash is below a certain threshold. As all other miners race to solve this cryptographic puzzle while only one block is finally accepted, this leads to a large amount of wasted computational power. Additionally, the energy consumed during this computation is estimated to be higher than that of the Republic of Ireland [1].

Acknowledgments: The authors thank the anonymous reviewers for their valuable feedback. This research was supported by the German Research Council (DFG) under grant no. KA 3171/1-2.

As a countermeasure, some blockchains employ different consensus protocols. One class are Proof-of-Stake (PoS) protocols, where the user's economic stake in the network influences the decision whether she can propose the next block. The other prominent class of consensus protocols are Byzantine agreement protocols, in which a group of replicas tries to reach a consensus on the execution order and result of client requests although a subset of these replicas may behave arbitrarily faulty or maliciously. However, in a group of $3f + 1$ replicas where a majority of nodes behaves correctly, the agreement scheme can tolerate up to f faulty nodes. BFT protocols are especially well-suited for *permissioned* blockchains, where all participants are known, their number is relatively stable, and access to the blockchain is regulated. They are often used e. g. for Supply Chain Management (SCM). Tendermint [2] is one example of a permissioned blockchain that employs a BFT protocol. Some recent approaches also employ BFT protocols in permissionless blockchains, e. g. Algorand [3] and HoneyBadgerBFT [4]. BFT protocols offer several advantages compared to PoW: they guarantee consensus finality, i. e. a block that has been appended to the chain cannot be invalidated due to forks, and offer higher throughput and lower latency [5].

Generally, protocols assuming a fail-stop approach are often simpler to integrate, as BFT protocols still introduce a higher complexity compared to their crash-tolerant counterparts. BFT also requires more messages to be exchanged, which limits scalability especially if more than the minimal number of faults are to be tolerated. Reducing BFT's high latency therefore motivates the increased deployment of these protocols in blockchains. One cause of high latency in network communication is the behavior of TCP on which most distributed systems still rely. The CPU load distribution of a standard TCP/IP connection shows that more than 50 % of all CPU cycles are spent on intermediate data copying in the local host [6]. With the recent advances in computer networks, RDMA becomes an available solution with a comparable cost to Ethernet. In RDMA communication, data is placed directly in the remote memory of the communication partner in a zero-copy manner, i. e. without intermediate copying as in TCP. It is already often employed in data centers, where large delays are induced by data copying during traditional TCP communication. This makes RDMA-enabled interconnects a compelling solution to alleviate the cost of message exchanges in BFT systems. For permissioned blockchain settings, the BFT replicas responsible for consensus can be placed inside a data center without compromising the concept of the blockchain. Few researchers have explored RDMA technology with consensus algorithms so far [7]–[9]; however, all assume a fail-stop model. To our knowledge, no contribution assumed a Byzantine fault model.

In this paper, we present RUBIN, an RDMA-based communication framework modeled after the Java NIO selector, and show first steps to integrate it into Reptor, a scalable BFT framework [10]. RUBIN aims to allow Java-based BFT frameworks to take advantage of RDMA counterparts without the need to rewrite the communication stack, thereby providing direct asynchronous communication and kernel bypassing. It offers an abstraction of the Java NIO socket channel and selector, which is used to efficiently handle multiple network connections with a single thread. The Java NIO channel and selector are employed in recent BFT protocol implementations such as BFT-SMaRt for client communication and UpRight as well as Reptor for replica communication [10]–[12]. RUBIN can therefore be integrated into other protocol implementations with only limited effort. Due to the higher level of abstraction compared to the RDMA Verbs, it is possible to profit from the performance gain of RDMA without completely changing the communication stack of the application. However, it is not trivial to achieve this performance gain as RDMA performance can easily decrease to that of TCP with ill-advised configuration. Lastly, we have to consider security aspects specific to BFT, e. g. it should not be possible for a faulty replica to compromise the safety and liveness properties of the protocol.

The paper is structured as follows: Section II gives an overview over BFT and RDMA communication; we present the design (Section III) and implementation (Section IV) of RUBIN as well as optimizations to the RDMA communication; Section V presents the evaluation results; Section VI gives an overview over related work, and Section VII concludes this paper.

II. BACKGROUND

In this section, we give an overview over RDMA technology (Section II-A) and Byzantine fault tolerance (Section II-B) as well as the Reptor protocol (Section II-C).

A. Remote Direct Memory Access

It has been shown that TCP/IP network stack processing consumes a significant amount of system resources when data is transmitted [13]. During TCP communication, two copy operations are performed: first, the CPU copies data from the user space buffer into a temporary socket buffer, then the data is placed into a TCP segment and pushed to the network controller through a DMA copy. To avoid this overhead of intermediate data copies and OS context switches, RDMA has been proposed to overcome the limitations of traditional networks. RDMA is a hardware-based protocol offloading technology enabling direct data movement between the memory of remote machines without the involvement of the operating system, and thereby manages to achieve high throughput and low latency for high messaging rates. RDMA uses the operating system only to establish a channel between two hosts, then allows applications to exchange messages without any kernel support in a *zero-copy* manner, i. e. with direct data transfer from virtual memory. RDMA supports asynchronous operations and is message oriented.

RDMA operates on queue pairs (QPs): when communication is initiated, each side must create a queue pair of send and receive queues for holding data transfer requests, so-called work requests (WRs). These WRs provide information about the data to be sent (send request) or received (receive requests). Upon the completion of an RDMA operation, an event is added to a completion queue (CQ) to notify the application about this. An RDMA application is required to register memory regions with the RDMA-enabled NIC (RNIC) prior to any networking operation to specify access to these regions.

There are two main modes of RDMA communication that we investigate for their suitability: one-sided and two-sided operations. In one-sided operations, the *RDMA Read/Write*, an application can access the remote memory directly without any involvement of the remote CPU. Thus, only one side is actively engaged in the communication process, while the remote side is not aware that any access operation is performed on its memory. This mode is well-suited for smaller messages, but generally requires more communication rounds than the two-sided operations. There, in the so-called *RDMA Send/Receive*, both partners must actively participate in the transfer and do not know the remote virtual memory location directly. This mode behaves similarly to TCP and offers reliable communication. However, unlike in one-sided primitives, each send request must have a matching receive request specifying where to receive data on the remote side before the operation can be initiated. Therefore, it is important to allocate enough receive requests to handle all incoming send requests. Each communication partner has only partial knowledge of this information required to complete the communication.

B. Byzantine Fault Tolerance

Byzantine agreement schemes, where a group of possibly faulty replicas tries to reach a consensus on the execution order and result of client requests, are considered a promising alternative to PoW mining in blockchains. Typical BFT systems consist of two stages: agreement and execution. In the agreement stage, replicas exchange asynchronous messages in order to reach agreement on a specific value, e. g. the order of client requests. This stage typically starts with the leader proposing a request sequence number to other replicas. Next, the replicas coordinate through broadcasting messages in order to validate that the leader has provided them with the same number, and reach a consensus on the total order of requests. For BFT systems, reaching consensus typically requires a majority quorum of $3f+1$ nodes. This takes multiple rounds of communication, which is especially costly for a large number of replicas as most protocols require broadcasting steps [3], [14], [15]. Thus, reaching consensus entails a large network traffic and performance overhead. In the execution stage, the replicated service uses the ordered requests provided by the agreement stage as input, executes the client operations, and finally sends a reply to the clients.

The agreement stage is often considered the bottleneck of a BFT system due to the cost of the message exchanges, especially when employing the traditional TCP/IP programming model. To counteract this, requests in BFT protocols are often batched, or hashes are transmitted instead of full messages. We propose to leverage the advantages of RDMA as a novel networking technology which is already deployed in data centers and therefore well-suited for utilization in BFT protocols.

C. *Reptor*

Behl et al. [10] presented a new parallelization scheme allowing BFT systems to fully exploit improving hardware trends such as modern multi-core processors. Splitting BFT protocols into several functional modules for multi-threaded execution still limits performance by the slowest of these modules. In the proposed Consensus-Oriented Parallelization (COP) scheme, the protocol is not divided according to specific tasks such as agreement and message authentication, but instead multiple protocol instances are parallelized while the total order of requests is still maintained. This enables their prototype *Reptor*, which uses PBFT [14], to scale with the number of available cores and reach unprecedented throughputs.

III. Design

Our work aims to enable the Reptor framework to take advantage of RDMA without fundamentally changing the design of the framework's communication stack. RDMA enables low-latency communication for applications by reducing the number of intermediate data copies, thereby alleviating the network overhead as well as taking advantage of the offered bandwidth. Also, by modeling the components of the RDMA communication after traditional socket connections, this eases the cost of redesigning the Reptor framework as well as other, suitable frameworks. We develop the Rubin framework which recreates the behavior of the non-blocking Java NIO with a selector tailored for the RDMA communication model. This provides an abstraction of the RDMA queue-based programming model, thus enabling direct asynchronous communication between replicas. The Java NIO selector enables efficient handling of multiple network connections using only a single thread, and is used in several BFT protocol implementations either for replica [10], [12] or client communication [11]. As the majority of BFT messages are typically exchanged between replicas, this is what we aim to enhance using RDMA.

In this section, we first explain our decision to use two-sided RDMA Send/Receive semantics (Section III-A) and describe the components of our implementation as well as their interaction (Section III-B), before evaluating the security of our framework (Section III-C).

A. *The Choice of RDMA Semantics*

The RDMA Send/Receive semantics are better suited for replica communication, as it (i) ensures that both sides can operate independently, and (ii) does not require participants to know the address of the remote memory to exchange data. This means that an application can post, i. e. send, a request without specifying the remote address as the receiver decides in which buffer to place the data. This is similar to the behavior of Java NIO sockets. Read/Write semantics, however, often entail a read/write race resulting in corrupted data [16] or failed retries because of the required coordination with increasing number of hosts [17]. They are therefore not suitable for our needs, as we assume a high number of replicas to participate in BFT protocols employed in blockchains.

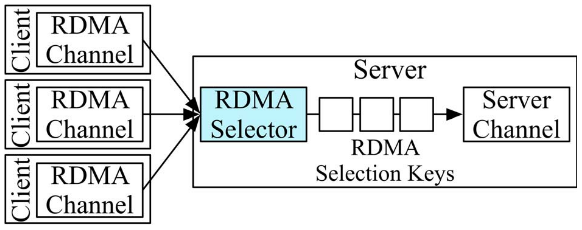


Fig. 1: Components of Rubin based on Java NIO

B. *Components of* Rubin

The Rubin framework consists of a set of components shown in Figure 1 inspired by Java NIO that were adapted for the RDMA queue pair model. These components are the RDMA channels, the RDMA selector, and the RDMA selection keys. An *RDMA channel* represents an RDMA connection. The abstraction behaves similar to a non-blocking NIO socket channel, which offers *read()* and *write()* methods, and includes all necessary RDMA resources such as QPs and WRs. When an RDMA channel is created, the list of buffers that the application will use for send and receive operations is also allocated and registered for RDMA communication. This abstraction is flexible because the number of WRs as well as the size of buffers can be independently specified, thereby allowing for the versatility needed by BFT protocols. Note that every created channel is associated with a unique connection identifier.

The *RDMA selector* is the key component in Rubin. It checks without blocking if an RDMA channel is ready for retrieving an I/O event. For example, if we are interested to know whether there is an incoming connection, the selector will receive a notification when an event of type OP_CONNECT has been added to the event channel. Afterwards, the selector will check if this event belongs to the channel through comparing the event ID with the channel ID and eventually returns the number of ready channels. This enables processing numerous RDMA channels in a single thread, similar to the Java NIO selector.

The *RDMA selection key* is the result of an RDMA channel registration with the selector and has a unique ID characterizing the connection. When an RDMA channel is registered, the type of events in which the channel is interested is specified. A selection key has four possible interests for a channel based on RDMA connection and completion notifications: an interest in incoming connections (OP_CONNECT), in con-

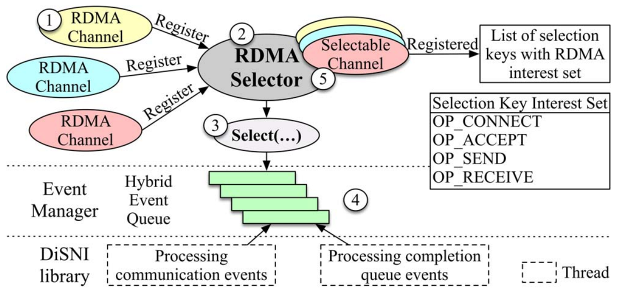


Fig. 2: An overview of the RDMA selector

nection establishments (OP_ACCEPT), in received messages (OP_RECEIVE), and in sending messages (OP_SEND). The selection key is then added to the list of connections that the selector checks when the *select()* function is invoked. In addition to the interest tag, a selection key has a ready tag that is updated when an I/O event occurred in the related channel.

Figure 2 shows the process of how these components interact in order to provide asynchronous communication.

1) The same blocking call for transmission and connection events

The Java NIO selector checks the readiness of both transmission and connection I/O on the same blocking call. RUBIN therefore includes a *hybrid event queue* containing copies of both the event channel elements and the completion queue elements. When an event is added to these channels, a copy of it will be added to the hybrid event queue of the RUBIN selector, notifying it about this new I/O operation.

2) An event-based mechanism replacing `epoll`

The Java NIO selector internally relies on `epoll` to check the readiness of the channels [18]. In RUBIN, an event manager is associated with the selector to keep track of the events added to the queue and to notify the selector. As shown in Figure 2, four phases are performed for a send or receive operation:

(1) Accepted RDMA channels start by registering to the selector and specifying which event they are interested in.

(2) The result is a set of selection keys, defining the relationship between the selector and the RDMA channel and holding the interest set. A selection key is a way to track the interest of the user and can be updated. Note that a registered channel is referred to as *selectable channel*.

(3) An invocation of a *select()* will start an indefinitely blocking call while there is no incoming I/O event.

(4) When an event occurred, a copy of it is added to the hybrid event queue. Afterwards, the event manager notifies the selector about this new incoming event.

(5) The selector checks whether the corresponding RDMA channel is interested in it. This is done by comparing the IDs and the type of the event, i. e. connection or transmission type. When the correct channel is found, its selection key's ready set is updated.

C. Security Analysis

Even though RDMA has its own protection mechanisms such as Protection Domains and access permissions on the memory area, the appealing protocol flexibilities have been proven to bring security issues [19]. However, most of these issues are design specific and only relevant for one-sided communication. In a remote Read/Write design, a client might try to read data while a second host is writing into the same buffer which results in corrupted data for the host attempting to read. A second security concern is related to the buffer identifier, the Steering Tag (STag), which will be sent to a host aiming to directly access a remote buffer. An adversary might get access to a buffer with STag enabled access, which allows her to conduct a Man-in-the-Middle attack. She can now read or modify the contents of this buffer or even invalidate the STag which prevents access of legitimate applications. Prior work [20]–[22] suggests solutions such as extended memory protection mechanisms or devising a mechanism according to the chosen design limitations.

As we use two-sided operations, this alleviates most security issues. An application does not need to expose its buffers to the connected remote nodes, but instead decides independently where the data will be placed. Now, if an attacker has compromised the memory keys, the affected BFT replica cannot operate reliably as it might not possess consistent data and will therefore be considered faulty, which can be tolerated by the protocol. Additional integrity protection mechanisms such as HMACs are employed in Reptor to detect invalid messages.

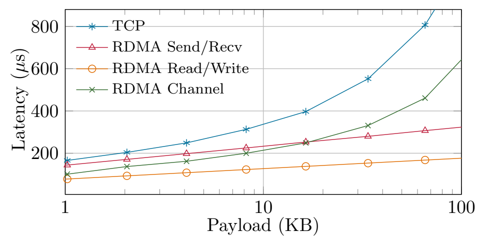
IV. IMPLEMENTATION

RUBIN is based on the OpenFabrics Enterprise Distribution (OFED) 4.0-2 by Mellanox, which offers an implementation of the RDMA Verbs interface allowing user-space processes to leverage the RNIC functionalities.

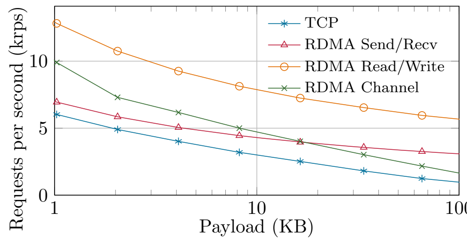
We use the jVerbs [23] library DiSNI[1] developed by IBM for support of RDMA communication. DiSNI offers two interfaces for RDMA programming: the low-level Verbs interface and an endpoints interface, which is an abstraction of the native Verbs functions similar to the regular socket functions. In our implementation, the endpoints interface is used as it resembles the non-blocking socket API and is therefore more suitable for recreating the behavior of the Java NIO channel. DiSNI was extended to support the non-blocking Java NIO features. The Reptor prototype is written in Java and implements the PBFT algorithm. We integrated RUBIN into Reptor, where it replaces the Java NIO selector and socket channel, and implemented several optimizations to the RDMA communication proposed by prior work [6].

These optimizations concern the requests as well as the buffer and completion event handling. A *pool of buffers* for send and receive requests are pre-registered and can be reused as needed. To reduce the overhead of posting, the requests are posted in *batches* of the maximum number of requests supported by the device. With *selective signaling*, no notification about the completion has to be created. Such a notification is only necessary after a certain number of messages, thus reducing the overhead for the RUBIN selector. Copying data from the application buffer to the buffer of the send request or

[1]DiSNI library. https://github.com/zrlio/disni

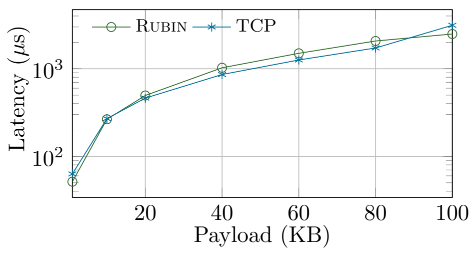


(a) Latency

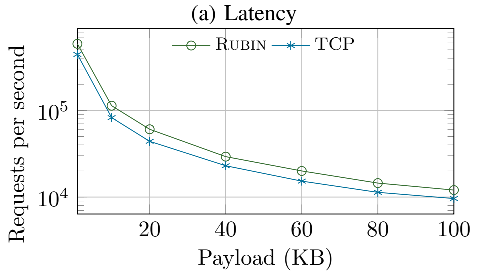


(b) Throughput

Fig. 3: Measurements for the RDMA Channel



(a) Latency



(b) Throughput

Fig. 4: Measurements for the Rubin and Java NIO selector

vice versa adds significant overhead for large messages, where buffer registration would be more performant. We therefore register the application's send buffer directly for RDMA communication, while data is still copied into a separate buffer on the receiver side. Here, we are limited by the type of buffers used in the DiSNI library and the Reptor prototype, as they are incompatible. We plan to adopt several optimizations in future versions of this work: Depending on the size of the messages, it is best to either copy the data into the request buffers (for messages ≤256Bytes) or register the application buffer directly for RDMA communication for larger messages. We therefore intend to remove any buffer copy from the RDMA communication except for small messages. The *select()* call of Rubin is less performant than that of the highly optimized Java NIO selector. We intend to improve this by implementing this functionality in native code. Sending messages as *inline* provides better latency, as the RDMA device does not need to perform additional read operations to get the payload. This is especially beneficial for small messages.

V. Evaluation

In this section, we present the results of our performance evaluation. We conducted the measurements on two machines with 4-core Xeon v2 CPUs and 16GB of memory running Ubuntu 16.04. Each machine is equipped with an RDMA-capable Mellanox Connect MT27520 network card working with RDMA over Converged Ethernet (RoCE), which enables RDMA communication over Ethernet. The machines are connected with a 10Gbps, full-duplex link and use the OFED 4.0-2 RNIC drivers.

First, we present a micro-benchmark implementing a simple client-server echo application between two machines in Figure 3. We compare the throughput (Figure 3b) and the latency (Figure 3a) of TCP, RDMA Read/Write, and RDMA Send/Receive with our implementation of an RDMA channel including the optimizations as presented in Section IV. All measurements show the average of five runs where client and server each exchange 1000 messages. We consider message sizes between 1KB and 100KB: BFT protocols exchange mostly small messages of several kilobytes, only rarely are larger messages necessary, e. g. for HTTP and IMAP use cases as presented in [24].

The measurements show that RDMA Read/Write entails the lowest latency with ≈46 % less compared to RDMA Send/Receive and 53–79 % compared to TCP. This is due to RDMA Read/Write having one-sided operations, meaning that only the client writes messages to the server without waiting for a response. This semantic, however, does not offer the required security as RDMA Send/Receive does. The RDMA channel, however, enables both client and server to coordinate the message exchanges while still maintaining a latency 33–43 % lower than that of TCP. The positive effect of the selective signaling is especially noticeable for messages smaller than 16KB where the latency decreases by up to 30 % compared to Send/Receive; for larger messages, the performance degradation due to the buffer copy for receiving messages becomes obvious. The throughput behaves accordingly: RDMA Read/Write achieves the highest throughput by 53–79 % more compared to TCP, while the throughput of the RDMA channel is up to 30 % higher than RDMA Send/Receive for messages smaller than 16KB. The RDMA channel is 33–43 % more performant than TCP.

We also evaluate the performance of the Rubin selector compared to the Java NIO selector with an echo server using the Reptor communication stack running locally on one of the machines. For both protocols, the window size and batching was set to 30 and 10 messages, respectively. The results can be seen in Figure 4. It shows comparable throughput and latency for TCP and RDMA. The latency of Rubin for messages of 1KB is 19 % lower than that of TCP; for large messages of 100KB, it is 20 % lower. For message sizes from 20KB to

80KB, RUBIN's latency increases up to 20 %. The throughput of RDMA is higher than that of TCP with values between 25 % for message sizes of 100KB and 38 % for 20KB. The performance degradation of RDMA due to the buffer copy on the receiver is again noticeable. We plan several additional optimizations to further exploit RDMA capabilities especially focusing on the performance for large message sizes and optimizing the throughput of RUBIN by removing any buffer copy steps during communication.

VI. Related Work

RDMA has gained recognition because of the high performance it can achieve. It has already been explored for key/value stores [25], databases [26], and distributed file systems [27], [28], but has received limited attention so far in connection with consensus protocols. Three approaches for RDMA-enabled crash-tolerant consensus protocols have been proposed. DARE [7], an RDMA-tailored replicated state machine protocol, aims to optimize for low latency in replica communication. The protocol uses one-sided primitives and replicates state machine updates through RDMA Read/Write operations. APUS [8] combines RDMA with Paxos and focuses on scalability regarding concurrent connections. Derecho [9] is a C++ library offering replicated crash fault-tolerant services, also aiming for RDMA communication in data centers. However, these protocols consider only fail-stop failures; to our knowledge, there is no previous work that assumed Byzantine faults. We present first steps towards RDMA communication in BFT protocols. JSOR [29] also models its RDMA endpoints after the Java socket interface, but offers a higher level of abstraction. However, this comparability layer limits the performance gain of using RDMA as it still includes intermediate buffer copies. Instead, we propose a level of abstraction between that offered by JSOR and that of RDMA's native endpoints, which currently allows us to partially profit from RDMA's zero-copy features while simultaneously allowing easy integration into existing frameworks.

VII. Conclusion

Current BFT protocols still induce a high latency due to the required multiple rounds of message exchanges, even when the replicas are placed inside a data center, as TCP/IP includes several intermediate data copy steps. This latency hinders their adoption as consensus protocols in blockchain platforms. In the BFT protocols that are deployed in blockchains, the number of participants will presumably be higher than in traditional deployment scenarios, thereby leading to a further increase in latency for inter-replica communication. This can be avoided by using RDMA, which offers kernel bypassing and zero-copy operations. We presented RUBIN, a framework to leverage RDMA interconnect features for BFT protocols such as Reptor. It does not require a redesign of the existing BFT communication stack as it is modeled after the behavior of the Java NIO selector.

In our future work, we plan to investigate how several optimizations using additional RDMA features impact the performance of RUBIN, remove any additional buffer copy steps, and to extensively evaluate the fully replicated system.

References

[1] A. Hern. (2018) Bitcoin's energy usage is huge – we can't afford to ignore it. [Online]. Available: https://goo.gl/z8MNSM

[2] J. Kwon. (2014) Tendermint: Consensus without mining. [Online]. Available: https://tendermint.com/static/docs/tendermint.pdf

[3] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *SOSP '17*, 2017.

[4] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The Honey Badger of BFT Protocols," in *CCS '16*, 2016.

[5] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication," in *IFIP WG 11.4 International Workshop, iNetSec 2015*, 2015.

[6] P. W. Frey and G. Alonso, "Minimizing the Hidden Cost of RDMA," in *ICDCS '09*, 2009.

[7] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in *HPDC '15*, 2015.

[8] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: Fast and Scalable Paxos on RDMA," in *SoCC '17*, 2017.

[9] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, S. Zink, K. Birman, and R. van Renesse, "Building Smart Memories and Cloud Services with Derecho," 2017.

[10] J. Behl, T. Distler, and R. Kapitza, "Consensus-Oriented Parallelization: How to Earn Your First Million," in *Middleware '15*, 2015.

[11] A. Bessani, J. a. Sousa, and E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," Tech. Rep., 2013. [Online]. Available: http://repositorio.ul.pt/bitstream/10451/14170/1/TR-2013-07.pdf

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight Cluster Services," ser. SOSP '09, 2009.

[13] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt, "Performance Analysis of System Overheads in TCP/IP Workloads," in *PACT '05*, 2005.

[14] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI '99*, 1999.

[15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, 2010.

[16] C. Mitchell, Y. Geng, and J. Li, "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store," in *USENIX ATC'13*, 2013.

[17] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "RFP: When RPC is Faster Than Server-Bypass with RDMA," in *EuroSys '17*, 2017.

[18] Oracle. Enhancements in Java I/O. [Online]. Available: https://goo.gl/sfZyQ7

[19] J. Pinkerton and E. Deleganes, "Direct Data Placement Protocol (DDP)/Remote Direct Memory Access Protocol (RDMAP) Security," Internet Requests for Comments, RFC 5042, 2007.

[20] B. Li, P. Zhang, Z. Huo, and D. Meng, "Early Experiences with Write-Write Design of NFS over RDMA," in *NAS '09*, 2009.

[21] M. Lee, E. J. Kim, and M. Yousif, "Security Enhancement in InfiniBand Architecture," in *IPDPS '05*, 2005.

[22] R. Noronha, L. Chai, T. Talpey, and D. K. Panda, "Designing NFS with RDMA for Security, Performance and Scalability," in *ICPP '07*, 2007.

[23] P. Stuedi, B. Metzler, and A. Trivedi, "jVerbs: Ultra-low Latency for Data Center Applications," in *SOCC '13*, 2013.

[24] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent access to byzantine fault-tolerant systems," in *DSN '18*, 2018, accepted for Publication.

[25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *NSDI '14*, 2014.

[26] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The End of Slow Networks: It's Time for a Redesign," *Proceedings of the VLDB Endowment*, 2016.

[27] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High Performance RDMA-based Design of HDFS over InfiniBand," in *SC '12*, 2012.

[28] M. Tatineni, X. Lu, D. Choi, A. Majumdar, and D. K. D. Panda, "Experiences and Benefits of Running RDMA Hadoop and Spark on SDSC Comet," in *XSEDE '16*, 2016.

[29] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. (2014) Transparent network acceleration for Java-based workloads in the cloud. [Online]. Available: https://goo.gl/P5Gtj3