Towards Truly Adaptive Byzantine Fault-Tolerant Consensus

Chenyuan Wu
University of Pennsylvania

Haoyun Qin
University of Pennsylvania

Mohammad Javad Amiri
Stony Brook University

Boon Thau Loo
University of Pennsylvania

Dahlia Malkhi
UC Santa Barbara

Ryan Marcus
University of Pennsylvania

Abstract

To acheive maximum performance, Byzantine fault-tolerant (BFT) systems must be manually tuned when hardware, network, or workload properties change. This paper presents our vision for a reinforcement learning (RL) based Byzantine fault-tolerant (BFT) system that adjusts effectively in real-time to changing fault scenarios and workloads. We identify several variables that can impact the performance of a BFT protocol, and show how these variables can serve as features in an RL engine in order to choose the context-dependent best-performing BFT protocol in real-time. We further outline a decentralized RL approach capable of tolerating adversarial data pollution, where nodes share local metering values and reach the same learning output by consensus.

1 Introduction

Byzantine fault-tolerant (BFT) consensus protocols are the core engines powering the state machine replication (SMR) paradigm, ensuring that non-faulty replicas execute client requests in the same order, despite the existence of f Byzantine replicas. The ability to tolerate up to f arbitrary failures makes BFT protocols a key component in various distributed systems, including permissioned blockchains [5, 29], distributed file systems [14, 17], locking services [18], firewalls [20, 21], key-value stores [19, 22], and SCADA systems [7, 27].

BFT protocols differ across several dimensions, with each protocol making different assumptions about the incoming workload, possible attacks, network configurations, and the underlying hardware. The proliferation of different BFT protocols has made it difficult to determine the best protocol for a given scenario. Worse yet, the ideal BFT protocol may change over time, since workloads, network conditions, and adversarial behaviors can frequently change. This is exacerbated in blockchain systems, which must additionally support a diverse set of applications.

While there have been multiple attempts to address these challenges (e.g., Abstract [6, 23] and ADAPT [8]), such solutions suffer from either (1) a lack of flexibility or (2) operational limitations. For example, when Abstract detects slow progress, current requests are aborted and a predefined alternative protocol is selected. While this approach can avoid long stalls, flexibility is limited since the switching order is pre-defined, and the system might fail to switch to the optimal protocol for the current context. ADAPT, on the other hand, uses supervised learning where a single replica collects data, trains the learning model, and then distributes the decision to all other replicas. Such a centralized mechanism, however, is an operational limitation in Byzantine environments.

How can we build a system with the required flexibility to achieve high performance in a wide variety of scenarios, but simultaneously avoid the operational limitations that come with centralized solutions? In this paper, we articulate our vision for a reinforcement learning (RL) based Byzantine fault-tolerant system. At a high level, given a performance metric to optimize, our proposed system will smartly adapt to changes by switching between a set of BFT protocols at run-time. Instead of manually choosing from a set of alternative BFT protocols for deployment, or running a prolonged data collection process before the deployment, we only require running one system that automatically re-configures itself to implement a top-performing protocol in real-time.

Intuitively, our approach works by building a comprehensive performance model capable of capturing different factors that impact the performance of a BFT protocol. By formulating the selection of a BFT protocol as a contextual multi-armed bandit problem [37], the RL engine strategically tests different protocols at run-time to learn which ones are well-suited to the current system conditions. Such RL process is coordinated in a decentralized manner, where nodes share local features/rewards by consensus and reach the same learning output, achieving resilience to adversarial data pollution.

2 The Case for Reinforcement Learning

Why use reinforcement learning [34] for BFT protocol selection? Prior work [4] has shown that no single BFT protocol is always "better" or "worse" than others, but rather that the performance of each protocol is context dependent. One could imagine building heuristics (or supervised learning models) that map conditions to the best-performing protocol, and switching protocols at run-time according to the current perceived conditions. However, such approaches suffer from several drawbacks.

1. **Space size.** The number of factors that can impact the performance of a BFT protocol is surprisingly large [4]: request and reply size, request arrival rate, execution overhead, transaction contention, and different failure scenarios can all have an impact. Even with coarse-grained sampling and an automated toolkit, experimentally exploring just a small subset of the condition space would take a long time. Unfortunately, building good heuristics and supervised models requires *complete* data, which are hard to obtain.

2. **Hardware and time dependence.** Even if the entire space could be mapped, the mapping from conditions to the best-performing protocol depends on the underlying hardware and system configuration. In cloud environments, these conditions can change arbitrarily [9] or even adversarially [35], rendering any pre-computed mapping on a specific network less useful.

3. **New protocols.** When new BFT protocols emerge (e.g., HotStuff-2 [30]) or changes are introduced to existing implementations (e.g., DiemBFT-v1 to v4 [1]), any precomputed mapping would need to be recomputed. In other words, one would need to re-collect data and re-craft the heuristics or retrain the supervised model virtually every time a new BFT consensus protocol is proposed.

Reinforcement learning addresses this daunting and complex task and has shown superior performance in other learned systems [15, 25, 31, 32, 36, 38]. While supervised learning assumes training data is complete, and thus requires a separate lengthy data collection process prior to deployment, reinforcement learning allows one to simply "plug-and-play:" RL systems learn from their mistakes and optimizes long-term rewards through trials in an online fashion. With reinforcement learning, an adaptive system can optimize itself to whatever client workloads, faults, hardware, system configurations and BFT protocols present, providing adaptivity and significant operational benefits.

3 Overview

A high-level overview of our proposed systems is shown in Figure 1. Our RL-based BFT system contains three key components: (1) a reinforcement learning algorithm (i.e., the core of the learning agent) that guides the choice of BFT protocols according to the perceived underlying dynamic environment,

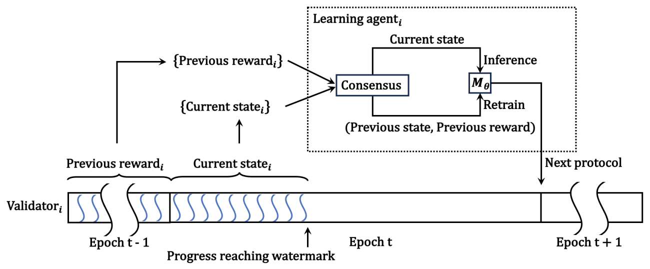

Figure 1: Overview of the proposed system. For readability, we only present the internals of one node i.

(2) a coordination protocol that collects data in a distributed fashion at run-time, and (3) a switching mechanism that allows a seamless transition from one BFT protocol to another while ensuring safety and liveness.

Our proposed system operates in *epochs*, where each epoch is marked by the completion of k blocks. Here, k is a predefined constant hyper-parameter. Within one epoch, the protocol remains unchanged. When the learning agent finds a protocol candidate, it instructs the validator to use that protocol for the next epoch.

Learning agent. The learning agent models the problem of selecting a BFT protocol as a contextual multi-armed bandit (CMAB) problem [37]: periodically, the agent examines the most recent state of the workload and faults in the system (*context*), and then selects one of many BFT protocols (*arms*) in our protocol pool. After making the selection, it observes the performance of the newly selected protocol (*reward*). To be successful, the agent must balance the *exploration* of new, untested protocols with *exploiting* past experience to maximize performance. That is, without a careful balance of exploration and exploitation, the agent risks failing to discover an optimal protocol (too much exploitation), or performing no better than random (too much exploration). We select this CMAB formulation (as opposed to full reinforcement learning) because CMABs are exceptionally well-studied, enable faster convergence, and many asymptotically-optimal algorithms exist to solve them [2, 16]. Details about the learning algorithms are provided in Section 4.1 to Section 4.3.

Since we consider a Byzantine environment, a centralized learning agent cannot be trusted. We consider each validator process to have a companion learning agent running on the same node, and accepting instructions only from its companion learning agent. The learning agents themselves also form a replicated state machine. Specifically, they start with the same *initial state*, i.e., the same random seed of machine learning models. For the same epoch t, as we will show later, different learning agents agree on the same *sequence of operations*, i.e., training data points where each data point consists of context and reward. With deterministic training, benign learning agents host the same parameters for their machine learning models. As a result, if different learning agents perceive the same context for epoch $t+1$, they will render the same decision (i.e., choice of protocol) for epoch $t+1$.

Distributed online data collection. In a Byzantine environment, no centralized entity could be trusted to collect training data. Therefore, the learning agents also participate in a protocol that coordinates distributed data collection in an online fashion. At a high level, for every epoch, each learning agent monitors its local context and reward at runtime, then exchanges them with other agents via a separate instance of BFT consensus independent of the consensus that validators are running. For each epoch, agents form agreement over an aggregation of contexts and rewards that include input from at least a quorum of two-thirds of agents. The consensus algorithm used for forming this agreement is left open for the system designer (note that it is invoked only once per epoch, hence does not need to have high throughput). Once an agreed quorum of local contexts and rewards is obtained, each learning agent can apply the same robustness filter to the quorum in order to get a global context and reward, constituting a training data point. Details about this learning-coordination protocol are provided in Section 4.4.

Switching BFT protocols. After a BFT protocol is selected by the learning agent, the switching mechanism allows each validator to make use of this protocol for the next epoch. The switching mechanism can be implemented based on Abstract [6], which aborts a BFT instance if a certain progress condition is not met. An epoch then is equivalent to a *Backup* instance in Abstract.

Figure 1 presents an overview of the system, where each node in the system follows the same workflow as depicted. In the middle of epoch t, when the number of executed blocks reaches a certain watermark, the validator on node i notifies its local learning agent. The learning agent featurizes its current local state (i.e., context) observed in epoch t, and uses it to approximate the next local state $state_i^{t+1}$ for epoch $t+1$. Each agent exchanges $state_i^{t+1}$ and its locally measured reward of previous epoch $reward_i^{t-1}$ with other agents via the learning-coordination protocol. Therefore, each agent obtains the same global state $state^{t+1}$ and reward $reward^{t-1}$. Subsequently, each agent adds the $(state^{t-1}, protocol^{t-1}, reward^{t-1})$ triplet to its experience buffer, and retrains its predictive model M_θ based on its experience buffer as well as the chosen algorithm to solve CMAB. Once retrained, the predictive model M_θ inferences the performance of each protocol candidate under $state^{t+1}$, and selects $protocol^{t+1}$ that is predicted to have the best performance. The learning agent then informs the validator to switch to $protocol^{t+1}$ for epoch $t+1$, the reward of which is then measured locally upon reaching the end of epoch $t+1$. The validator only starts epoch $t+1$ once it receives a decision for that epoch from the companion agent.

The learning agent is designed in a way such that the BFT system is not delayed due to learning. First, within one epoch, when the model undergoes retraining and inference, the parallel validator process still commits blocks simultaneously. Second, with a lightweight model design and limiting the size of the experience buffer, model training and inference can be viewed as a synchronous process. In other words, with a reasonable epoch length, the learning agent can complete protocol selection before the validator finishes its current epoch, without impeding the start of the next epoch.

4 Learning Algorithms

This section delves into learning algorithms. We first formalize the learning problem and explain the use of Thompson sampling. The state and action space design is then outlined, followed by the predictive model description and learning coordination.

4.1 Problem Formulation

We formulate the learning problem as a contextual multi-armed bandit (CMAB) problem, where an agent periodically makes decisions in a sequence of epochs. In epoch t, the agent selects an action a_t in its protocol pool based on a provided state s_t, and then receives a reward r_t. The agent's goal is to select actions in a way that minimizes *regret*, i.e., the difference between the reward sum associated with an optimal selection strategy and the reward sum associated with the chosen actions. CMABs assume that epochs are independent from each other, and that the optimal action depends only on the state s_t. Since the current choice of protocol does not affect the pattern of workloads and faults in future epochs, CMAB could be a reasonable choice for BFT protocols. Using this CMAB formulation, users will be able to specify any performance metric (e.g., throughput or latency) as the reward function to optimize.

Thompson sampling. Amongst different CMAB algorithms, we select Thompson sampling [2, 16] for its simplicity: at the start of each epoch, the learning agent trains a model based on current experience, and then selects the best action as predicted by the model. In Thompson sampling, instead of selecting the model parameters that are most likely given the training data as used by supervised learning, it *samples model parameters proportionally to their likelihood given the training data*. More formally, we can define maximum likelihood estimation as finding the model parameters θ that maximize likelihood given experience E: $\arg\max_\theta P(\theta \mid E)$ (assuming a uniform prior). Instead of maximizing likelihood, Thompson sampling simply samples from the distribution $P(\theta \mid E)$. As a result, if we have a lot of data suggesting that our model weights should be in a certain part of the parameter space, our sampled parameters are likely to be in that part of the space. Conversely, if we have only a small amount of data suggesting that our model weights should be in a certain part of the parameter space, we may or may not sample parameters in that part of the space during any given epoch.

4.2 State and Action Space

We next list factors that affect the performance of BFT protocols, broadly grouped into workloads, faults, and hardware/system configurations categories, jointly constituting the state space. Within each epoch, each learning agent leverages a window of the last w executed requests to featurize such factors, where w is a constant hyper-parameter.

State 1: Workloads (W). The first category consists of factors that are influenced by application and client dynamics.

W1: Request size. The request size is dependent on the application workload, where some requests contain little data while others are more involved and require updating files with large chunks of data. Although most protocols separate request dissemination from sequencing (i.e., only the leader proposals contain the actual requests while the remaining messages contain the hash of requests), request size is still an important factor impacting the performance of different protocols in different ways. We use the average request size to represent this feature.

W2: Reply size. Depending on the application, request and reply size can be asymmetric. Reply size also impacts different protocols in different ways, but with a distinct boundary from that of request size. We use the average reply size to represent this feature.

W3: Load on system. The load on the BFT system is dependent on the number of clients and the rate at which clients send new requests. Specifically, each honest client allows a *quota* of outstanding unacknowledged requests before issuing new ones, controlling the rate at which requests are generated relative to the system's capacity to process them.

W4: Execution overhead. Execution overhead captures the computational cost of request execution, which impacts the system in two ways. First, it directly affects the execution latency in state machine replication. Second, it indirectly affects other components of the BFT protocol that are also compute-intensive. For instance, requests with high execution overhead compete for CPU resources that are otherwise used to sign and verify messages, especially when machines have limited compute capacity or a small number of cores. Higher compute load results in excessive context switching, and potentially pushes the system towards being compute-bound instead of network-bound. We use the CPU cycles consumed by the executor thread to represent this feature.

State 2: Faults (F). The next category of factors is tied to faulty behaviors. BFT protocols make different assumptions about "steady state" and "common faults", and hence, each protocol is often optimized for specific fault scenarios. The features below help in identifying the type of fault scenarios the system is experiencing and choose the most promising protocol accordingly.

Note that these features do not aim to defend against *transient* or *broad-spectrum* faults. An example of transient fault is a crashed leader or a malicious leader which equivocates, such that no progress is made and a view-change will be triggered to replace the leader. Such transient faults are handled timely at the protocol level and require no protocol switching. For broad-spectrum faults, orthogonal and effective solutions exist. Examples of broad-spectrum faults include network flooding which can be mitigated via resource isolation [6], and malformed client requests which can be handled by enforcing client signatures (instead of MACs) [18].

F1: Absence from participation. In BFT consensus, validators can be absent from participation for various reasons: a (benignly) crashed validator is absent from all protocol phases after crashing, while an alive malicious validator could be absent from any arbitrary phases.

Measuring absence is tricky in a Byzantine environment, especially considering that a collusion of f malicious participants could taint validator participation simply by *excluding* some alive benign validators (up to f) and progressing without them. For example, a malicious leader could deliberately avoid sending leader proposals to them, while the f malicious validators work together with the remaining $f+1$ benign ones to make sure all requests are committed successfully on these $2f+1$ validators. We refer to such excluded, non-faulty validators as being placed *in-dark*. In-dark validators could further be excluded in other protocol phases, in addition to leader proposals, by f malicious validators. Since no state transition is ever triggered on in-dark validators, they remain in the initial state and are thus absent from participation. Although they will timeout and complain, since fewer than $f+1$ validators complain, view-change is not triggered to replace the malicious leader and they are in-dark continuously.

All protocols tolerate the absence of up to f validators by design, but the performance of different protocols is impacted differently. In particular, dual-path protocols (e.g., Zyzzyva [28], SBFT [24]) are adversely impacted since the more expensive slow paths are initiated, while single-path protocols (e.g., PBFT [13, 14]) could be positively impacted due to less resource consumption.

The learning agents can featurize absence by utilizing information that is already collected locally during protocol execution. First, *fast path ratio* captures the percentage of slots committed in the fast path over the total number of committed slots. For single-path protocols, all slots are committed in the slow path. Second, for each slot, the agent sums the number of (valid) distinct messages from each sender, deriving the *number of received messages per slot*. Note that this feature does not require more messages to be sent or received; it simply counts messages as they arrive and pass preliminary processing (de-serializing and sender verification) before they can be excluded from protocol steps like voting.

F2: Slowness of proposal. In leader-based BFT protocols, every slot is initiated by a leader proposal, which significantly affects the system's end-to-end performance. In the case of a faulty leader, validators use a timer to trigger view-change, which will replace the leader, hence guaranteeing liveness. However, a malicious leader can deliberately slow down its proposals without being replaced, resulting in poor latency and throughput. Slowness may not necessarily be a deliberate act by a malicious leader, it could result simply from a weak or overloaded leader, albeit to a lighter extent. In general, protocols with routine or proactive leader replacement (e.g., HotStuff-2 [30], Prime [3]) maintain good performance under such slowness, but perform sub-optimally in other normal cases. To featurize this factor, each node needs to timestamp every leader proposal received, and measures the average time interval between receiving two consecutive proposals.

State 3: Hardware and system configurations. The last category comprises hardware and system configurations. Hardware-level factors include standard data-center infrastructure network settings that affect network latency and bandwidth, and also machine-level configurations such as CPU frequency and the number of cores. System-level configurations include the number of nodes in the consensus system and the geo-distribution of the nodes. Compared to workload and faults, both hardware and system configurations are fairly static and do not change rapidly at the timescale of a consensus deployment. Thus, there is no need to explicitly featurize these factors because (1) the impact of these factors can be implicitly encoded in the predictive model trained online, and (2) CMABs will converge even without any explicit features [10], the purpose of which is to accelerate convergence so that the optimal action is reached before the world changes.

Actions. The action space consists of the set of BFT protocols. Here, we pick six representative protocols: PBFT [13, 14], Zyzzyva [28], CheapBFT [26], Prime [3], SBFT [24] and HotStuff-2 [30]. All six protocols are *leader-based*, working in the partial synchrony settings with networks of $n = 3f + 1$ nodes.

4.3 Predictive Model

Each learning agent hosts some predictive models, which follow the *value based* RL approach: given the featurized current state, predict the performance (i.e., reward) of each action (i.e., protocol). The simplest implementation would be to build a single predictive model for all protocols, but this has a major drawback. While features *W1-W4* in the workloads category are completely independent from the previous action, the featurized absence-from-participation $F1$ and slowness-of-proposal $F2$ have a "one-step dependency": the current observed $state^{t+1}_{F1,F2}$ is dependent on the previous $protocol^{t}$. When the workload and fault scenario shift, such one-step dependency might prevent convergence to the new optimal protocol. For instance, if the system has converged to protocols whose leader has lower parallelism (e.g., Prime due to message aggregation in global ordering), the measured slowness of proposals will be higher than other protocols, regardless of whether a fault is actually happening or not. In other words, the interpretation of the slowness-of-proposal feature changes based on the previous action. If the model learns that a high slowness-of-proposal is bad for Zyzzyva, then once Prime is chosen, slowness-of-proposal will always seem high, and Zyzzyva may not ever be selected again.

Luckily, unlike in the general case of Markov decision processes, the dependency observed here is limited to a single time step. That is, the prediction of the next best action is independent *given the immediately prior action*. To solve this issue rooted in one-step dependency, the learning agent trains a separate model for each possible (previous protocol, protocol) pair, and divides the experience buffer into several smaller *buckets* according to the (previous protocol, protocol) pair as well. In terms of bandit theory, assuming there are K protocols in the action space, the proposed approach is equivalent to playing K bandit games where each game has K arms. In each game, the current observed $state^{t+1}_{F1,F2}$ is independent from the previous action $protocol^{t}$.

It is worth mentioning that this transformation does not completely remove the one-step dependency. The action made at epoch t will determine which of the K bandit games is being played at epoch $t+1$. A multi-armed bandit algorithm will not be able to take advantage of the fact that some of the K bandits may have significantly better reward distributions than others. Thus, the convergence bound on regret of bandit algorithms will not apply to our scenario. However, since each of the K bandits will be played an unbounded number of times eventually (assuming the probability of any action never fully reaches 0), regret is still bounded in the limit.

Specifically, for each possible $(protocol^{t}, protocol^{t+1})$ pair, a lightweight random forest [12] can be used as the predictive model, which is trained exclusively on the corresponding experience bucket. The model takes the featurized state as input, and outputs the predicted performance for the corresponding candidate action $protocol^{t+1}$. Thus, at inference time, given a known previous protocol and the current state, the learning agent enumerates K models to get the predicted performance for each candidate protocol, and then chooses the candidate with the best predicted performance to be carried out. Once there is a tie on the best predicted performance, we break the tie randomly to avoid local maxima. When an experience bucket is empty, the system prioritizes exploring this bucket by choosing the corresponding candidate protocol to be carried out.

Integration with Thompson sampling. Integrating a predictive model with Thompson sampling requires the ability to sample model parameters from $P(\theta \mid E)$ — the distribution of model parameters given the current experience. The simplest technique (which has been shown to work well in practice [33]) is to train the model as usual, but only on a bootstrap [11] of the training data. In other words, the predictive model is trained using $|E|$ random samples drawn with replacement from experience E, inducing the desired sampling properties. This bootstrapping technique can be used on each experience bucket and predictive model for its simplicity.

Overhead of learning. First, training overhead is not supposed to be larger than the strawman of building a single predictive model, since in every epoch, only one model which corresponds to the updated bucket needs to be retrained. For such a bucket, the time complexity for training a single random forest is $O(n \log n)$, where n is the number of data points. Thus, given the same total population of data, it even incurs less training overhead than the strawman solution, since the bucket contains fewer data points than the single unified experience buffer. Second, the inference overhead is $O(K)$, where K is the number of candidate protocols. Lastly, the memory overhead would be the same as the strawman for storing training data. However, it incurs $O(K^2)$ memory overhead for storing the models. Since random forest is a very lightweight model as compared to deep neural networks, such model storage overhead is negligible.

4.4 Learning Coordination

The goal of the *learning-coordination* mechanism is to form an agreement at each epoch on a *report quorum* that includes local metrics collected from $2f+1$ nodes.

Specifically, learning coordination is performed in every epoch t. After executing w requests (a hyper-parameter) in epoch t, each node i gathers local performance indicators p_i^{t-1} measured during epoch $t-1$, featurizes the next state f_i^{t+1}, and broadcasts both metrics inside a report message. To ensure that at least $f+1$ metrics in the report quorum are honest measurements, it is important that the metrics reported by honest nodes are measured by **themselves**. That is, if a node j has been placed in-dark (defined in Section 4.2) or temporarily slows down during epoch t, it may not have executed w requests by itself. Rather, node j will have recovered the consensus state through a *state-transfer* from other nodes. In this case, j should avoid reporting the state features it has copied from others, and likewise, avoid reporting performance indicators collected from partial or no execution. Therefore, node j will not report any metrics for epoch t. Note that in addition to the f benign nodes being placed in-dark, in the meantime, the f Byzantine nodes that contributed to committing requests can refuse to report their metrics. Hence, there may not be enough $2f+1$ nodes reporting for the epoch.

In order for nodes to agree on a quorum of (valid) reports to be used as input for the learning engine, any "blackbox" validated Byzantine consensus primitive (VBC) seeded with a leader collecting reports from $2f+1$ nodes can be utilized. Specifically, for each epoch t, the leader of VBC initiates VBC-PROPOSE($(t, reportQC^t)$, P) once it receives valid report messages (p_i^{t-1}, f_i^{t+1}) where both fields are non-null from $2f+1$ nodes, or when a timer expires. Here, P is an external validity predicate that checks if $reportQC^t$ includes at least $f+1$ distinct reports.

Each node participating in VBC gates voting for a leader proposal it receives by applying the validity predicate P to it. Once a quorum of reports $reportQC^t$ is decided by VBC, if it includes sufficient $2f+1$ reports, each node takes the median value of each field in order to obtain a robust global performance measurement p^{t-1} and state feature f^{t+1}, thereby triggering the retraining and inference process. Taking the median value from an aggregated set of metrics guarantees that despite f arbitrarily manipulated values from Byzantine nodes, the global value taken is between two honest measurement values. Otherwise, if $reportQC^t$ does not include sufficient reports, each node retains the decision from the previous epoch instead of deriving any new learning decision, and complains about the leader in VBC as well as the leader in the current protocol used by the node for committing client requests. Note that since VBC is a separate consensus instance, the leader of VBC can be different from the leader of the current protocol. Either of them acting maliciously can result in insufficient reports being collected.

5 Conclusion

Existing BFT protocols lack flexibility and adaptability, leading to suboptimal performance in various scenarios. In this paper, we articulate our vision for a practical reinforcement learning-based BFT system, which dynamically selects the top-performing BFT protocols in real-time.

References

[1] The diem team. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf, 2021.

[2] Shipra Agrawal and Navin Goyal. Further optimal regret bounds for thompson sampling. In *The International Conference on Artificial Intelligence and Statistics*, AISTATS '13.

[3] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[4] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of byzantine fault tolerance: A unified platform for bft protocols analysis, implementation, and experimentation. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2024.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro,

David Enyeart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*, pages 30:1–30:15. ACM, 2018.

[6] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[7] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. Deploying intrusion-tolerant scada for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 328–335. IEEE, 2019.

[8] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making bft protocols really adaptive. In *Int. Parallel and Distributed Processing Symposium*, pages 904–913. IEEE, 2015.

[9] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 35–46. ACM. tex.acmid= 1730842 tex.numpages= 12.

[10] Omar Besbes, Yonatan Gur, and Assaf Zeevi. Stochastic multi-armed-bandit problem with non-stationary rewards. In *Advances in neural information processing systems*, NIPS '14, pages 199–207.

[11] Leo Breiman. Bagging predictors. In *Machine Learning*, Maching Learning '96.

[12] Leo Breiman. Random forests. 45(1):5–32.

[13] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association, 1999.

[14] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[15] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL @ PLDI '20. ACM.

[16] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, NIPS'11.

[17] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Symposium on Operating Systems Principles (SOSP)*, pages 277–290. ACM, 2009.

[18] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 153–168. USENIX Association, 2009.

[19] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Conf. on Computer and communications security (CCS)*, pages 285–298. ACM, 2013.

[20] Miguel Garcia, Nuno Neves, and Alysson Bessani. An intrusion-tolerant firewall design for protecting siem systems. In *Conf. on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–7. IEEE, 2013.

[21] Miguel Garcia, Nuno Neves, and Alysson Bessani. Sieveq: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing*, 15(3):511–525, 2016.

[22] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 135–144. IEEE, 2004.

[23] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *European conf. on Computer systems (EuroSys)*, pages 363–376. ACM, 2010.

[24] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE/IFIP, 2019.

[25] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. Roq: Robust query optimization based on a risk-aware learned cost model.

[26] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, pages 295–308. ACM, 2012.

[27] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, 2013.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)*, 41(6):45–58, 2007.

[29] Jae Kwon. Tendermint: Consensus without mining. 2014.

[30] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[31] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters.

[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21. Award: 'best paper award'.

[33] Ian Osband and Benjamin Van Roy. Bootstrapped thompson sampling and deep exploration.

[34] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition.

[35] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in {Multi-Tenant} public clouds. USENIX Security '15, pages 913–928.

[36] Chenyuan Wu, Bhavana Mehta, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. AdaChain: A learned adaptive blockchain. *Proc. of the VLDB Endowment*, 16(8):2033–2046, 2023.

[37] Li Zhou. A survey on contextual multi-armed bandits.

[38] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. Lero: A learning-to-rank query optimizer. 16(6):1466–1479.