2015 IEEE 29th International Parallel and Distributed Processing Symposium

Making BFT Protocols Really Adaptive

Jean-Paul Bahsoun
IRIT, UPS
Toulouse, France
bahsoun@irit.fr

Rachid Guerraoui
EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Ali Shoker
IRIT, UPS
Toulouse, France
ali.shoker@irit.fr

***Abstract*—Many state-machine Byzantine Fault Tolerant (BFT) protocols have been introduced so far. Each protocol addressed a different subset of conditions and use-cases. However, if the underlying conditions of a service span different subsets, choosing a single protocol will likely not be a best fit. This yields robustness and performance issues which may be even worse in services that exhibit fluctuating conditions and workloads.**

In this paper, we reconcile existing state-machine BFT protocols in a single adaptive BFT system, called ADAPT, aiming at covering a larger set of conditions and use-cases, probably the union of individual subsets of these protocols. At anytime, a launched protocol in ADAPT can be aborted and replaced by another protocol according to a potential change (an *event*) in the underlying system conditions. The launched protocol is chosen according to an "evaluation process" that takes into consideration both: protocol characteristics and its performance. This is achieved by applying some mathematical formulas that match the profiles of protocols to given user (e.g., service owner) preferences. ADAPT can assess the profiles of protocols (e.g., throughput) at run-time using *Machine Learning* prediction mechanisms to get accurate evaluations. We compare ADAPT with well known BFT protocols showing that it outperforms others as system conditions change and under dynamic workloads.

***Keywords*-Byzantine fault tolerance; Adaptive BFT; dynamic switching**

I. Introduction

Fault tolerance is becoming more challenging due to the leap of on-line services, the reliance on clouds, and the software bugs introduced with new software technologies and programming languages due to human imperfection. This can cause unpredictable arbitrary service problems that are sometimes dreadful and can affect a large population; the outages of Amazon AWS [1] and Google Mail [2] are recent examples. *Byzantine fault tolerance* [3], [4] (BFT) is a replication-based approach used to improve the resilience of systems to *Byzantine* (arbitrary) faults. A typical state-machine [5] BFT protocol ensures consistency among system replicas if at most a fraction of replicas (e.g., one third) can be Byzantine [3], [4]. Although BFT trades the cost of replication and agreement for fault tolerance, this cost is nowadays acceptable as commodity hardware are becoming cheap; and as the cost of existing non-BFT fault tolerance mechanisms, e.g., the three-way replication of storage in Google file system, is comparable to BFT [6].

A fairly high number of BFT protocols ([4], [7], [8], [9], [10], [11], etc.) have been introduced in literature. Due to the complexity of the Byzantine generals problem [3] and the variations in system conditions, it is almost impossible to establish one-size-fits-all BFT protocol. For instance, PBFT [4] operates in the presence of Byzantine nodes; however, it suffers from low performance as compared to *speculative* protocols (e.g., [8], [9]). Q/U [7] and Quorum [9] exhibit the lowest latency and fault scalability, however, only in contention-free cases. Zyzzyva [8] and Chain [9] achieve a high throughput, but they rather suffer from expensive recovery, etc. Unfortunately, this suggests that a service can only benefit from some properties (those provided by the chosen protocol) and give up other interesting ones. Although this can be acceptable in some systems, it can have significant drawbacks on systems that encounter different conditions and workloads. We show in this paper that it is possible to get closer to one-size-fits-all protocol through combining existing protocols together, and using a smart *dynamic switching* mechanism between them (without the burden of again introducing a new protocol).

The *abortability* approach of *Aliph* [9] proposed a modular way to use some existing BFT protocols and *switch* from one to another when failures occur. In spite of its performance improvements in some cases, our experiments show that Aliph's performance can be close to existing protocols, e.g., Zyzzyva, or even worse under fluctuating conditions and workloads. The reason is that Aliph runs a (1) predefined static order of a (2) specific set of protocols, with PBFT as *backup* under failures, and (3) uses a *backoff scheme* to switch back from PBFT to a faster protocols to gain some performance when failures heal. These very reasons cause Aliph to fall short as an efficient adaptive protocol for systems that are prone to variable conditions and dynamic workloads. We explain how our *dynamic switching* approach resolves these drawbacks to improve reliability and performance.

In this paper, we propose an *adaptive abortable* BFT system, called ADAPT. ADAPT launches a BFT protocol from a set of candidate ones (any existing protocol). Once the system conditions change (i.e., an event), ADAPT *aborts* the running protocol, and launches another one that is "more adequate" to the new conditions. To decide *when* to launch *which* protocol, ADAPT launches an *evaluation process* to make run-time evaluations of protocols and matches their properties and performance against user preferences [1]. The evaluation process executes some mathematical formulas we introduce, powered using Machine Learning techniques to make accurate run-time

[1]We refer to the "user" as the service owner that is using a BFT protocol.

performance assessment.

To the best of our knowledge, this is the first adaptive BFT approach that orchestrates multiple BFT protocols in a dynamic way using Machine Learning techniques, namely SVR [12]. (Theoretical analysis and simulation-based methods like those in [13] and [14] only give a general inaccurate theoretical results that are not effective at run-time). In addition, conducting run-time evaluations of protocols by considering their characteristics, performance, and user preferences is also novel to BFT.

The abortability in our system, i.e., ADAPT, is similar to Aliph [9] by the fact that it runs a set of protocols and switches between them; on the contrary, any existing BFT protocol can be used in ADAPT; whereas, Aliph could not use many protocols since it is not easy to define how and when to efficiently switch from one to another. In addition, ADAPT is adaptive since switching occurs using run-time evaluations when "something happens", thus relaxing the conditions of Aliph that switches only when "something wrong happens". This brings three additional benefits over Aliph: (1) no order or number of protocols has to be defined a priori, (2) switching occurs not only upon failures but also when performance can be gained, and (3) no backoff scheme is required since switching immediately occurs once a change in system conditions is detected.

We implemented ADAPT in C/C++ code, and experimented it on Emulab [15]) using Redis key-value store [16] as an application. Our experiments convey that ADAPT outperforms six well-known existing BFT protocols, including Aliph, under any condition, and especially under dynamic workloads.

In the rest of the paper, we present a background about the BFT protocols we consider in Section II. Section III presents the architecture of ADAPT, and Section IV explains its *evaluation process*. Experimentation results are then presented in Section V. Finally, the paper discusses related works in Section VI and concludes in Section VII.

II. Background of BFT Protocols

This section recalls the BFT fault model and selected well-known BFT protocols that make our presentation easier, following the criteria: these protocols are clearly different in at least one important feature. In principle, any state-based BFT protocol can be added to ADAPT, but we believe that the protocols addressed here are enough to explain our idea. In addition, we explain the BFT *abortability* approach [9] that is also used in ADAPT.

A. Fault Model

BFT fault model [4] assumes a message-passing distributed system using a fully connected network among nodes: clients and servers. The network may (not infinitely) fail to deliver, corrupt, delay, or reorder messages. Faulty replicas and clients may either behave arbitrarily, i.e., in a different way to their designed purposes, or they just crash (*benign* faults). A strong adversary coordinates faulty replicas to compromise the replicated service. However, we assume that the adversary cannot break cryptographic techniques like: collision-resistant hashes, encryption, and signatures. Liveness, however, is only guaranteed when the system is *eventually* synchronous, i.e., during intervals in which messages reach their correct destinations within some fixed worst case delay. ADAPT complies with this BFT model.

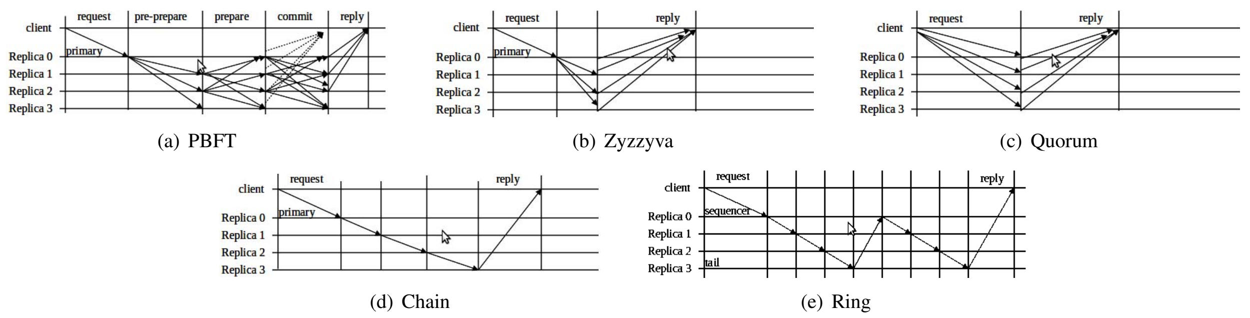
B. BFT Protocols

BFT protocols maintain system resilience against Byzantine failures using replication. A protocol ensures safety and progress if up to a fraction (often $1/3$) of the replicas is faulty. (We refer to "faults" as Byzantine faults in this paper). **PBFT** [4] is the first practical BFT protocol. Messages are exchanged in three phases (see Fig. 1(a)): *pre-prepare, prepare, and commit*. Since most replicas contribute in message exchange, in each phase, consensus can be achieved even when f replicas are Byzantine. This extensive all-to-all messaging makes PBFT robust, but causes significant performance drawbacks. **Zyzzyva** [8] is a speculative BFT protocol where a client sends a request to a *primary* replica that assigns it a sequence number and forwards it to other replicas. These replicas speculatively execute the request and send their replies (or digests) back to the client (Fig. 1(b)). This makes Zyzzyva fast as long as the client receives matching replies from all replicas. Otherwise, complex recovery phases are launched when a replica or the primary is faulty; and consequently, its performance drops sharply.

C. BFT Abortability Approach

Abortability [9] was introduced to reduce the complexity of BFT protocols and improve their performance using modularity: any BFT protocol is first launched on a set of replicas. When "something wrong happens", the current protocol is *aborted* and another one is launched on the same set of replicas, starting a new phase where replicas are initialized with an *abort history*: a log of recently applied operations (starting from the last checkpoint). The authors used *abortability* to build Aliph [9]. In Aliph, "something wrong happens" practically means a fault is detected. Aliph used three abortable protocols: Quorum, Chain, and *backup* (an "abortable" version of PBFT). Aliph initially runs Quorum. Upon failures, caused by contention, it aborts to Chain. Again, once failures occur, due to any reason, Chain aborts to *backup* (simply PBFT in the rest of the paper) in this specific order, then it continuously tries to switch back to Quorum using a *backoff scheme*, either rigorous (one try per request) or exponential (one try each 2^n requests, n being the number of tries). Next, we recall three important abortable protocols: Quorum [9], Chain [9], and Ring [17].

Quorum [9] has the theoretical minimum latency among BFT protocols in contention-free systems due to its simple one-phase message pattern: a non-faulty client broadcasts a request to all replicas, and the replicas reply back directly to the client (see Fig. 1(c)). Since there is no central replica for sequence number assignment in Quorum, problems can arise under contention or Byzantine behaviors, and thus it aborts



(a) PBFT (b) Zyzzyva (c) Quorum

(d) Chain (e) Ring

Fig. 1. Message patterns of the state-of-the-art BFT protocols; for $f = 1$.

to another protocol. **Chain** [9] is another abortable protocol that has the highest theoretical throughput. All replicas are ordered in a chain fashion. The head of the chain receives a request from a client. Each replica forwards the request to its successor in the chain until the tail sends the reply back to the client (Fig. 1;(c)). Although this technique increases the end-to-end delay, the throughput improves as the number of MAC operations by each replica is close to one, i.e. the theoretical lower bound. With large message payloads, Chain loses its charm due to the network bottlenecks formed on the *head* and *tail* replicas. **Ring** [17] is an abortable BFT protocol (Fig. 1(e)) where replicas are organized in a ring fashion, and each replica has a *predecessor* and a *successor*. A client can send a request to any replica and receive the reply from the predecessor of that replica. Any request is forwarded in two rounds around the ring to complete. In the first, it gets assigned a sequence number by a specific replica, called "sequencer"; whereas the second round is needed to execute it on all replicas (Fig. 1(e)). This long trip causes large delays in Ring responses which makes Chain better in throughput, in normal conditions (though both require one MAC operation per replica). However, Ring has a high throughput when the network becomes a bottleneck as all replicas can receive and send requests from/to clients.

III. ADAPT ARCHITECTURE

ADAPT is composed of three sub-systems: BFT System (BFTS), Event System (ES), and Quality Control System (QCS). BFTS is composed of the libraries of BFT protocols and *abortability* [9]. We adjusted existing BFT protocols to fit the modular logic of BFTS. BFTS operates in a similar manner to the abortability approach described in Section II-C above. The Event System (ES), on the other hand, monitors the whole system and collects the defined *Impact Factors*: chosen metrics that have a significant impact on the performance and reliability of the system (e.g., number of client, request size, etc). ES sends periodic event notifications to the Quality Control System (explained next) informing it with any changes in the impact factors. In ADAPT, we used a simple version of ES described in Section V-B; (a more sophisticated ES is a future plan). Since this work focuses on *dynamic switching*, and to gain some space, we skip any further discussions about BFTS and ES, and we focus on the Quality Control System (QCS) in the rest of the paper.

IV. QUALITY CONTROL SYSTEM (QCS)

A. Overview

QCS is the control unit of ADAPT that is in charge of taking the decisions: whether switching is needed due to possible changes in system state, and which protocol is the best to be launched in the next phase. QCS works as follows: consider the set of n BFT protocols in BFTS, and assume that BFTS is initially running $p_i,\ i \in [1, n]$. Once QCS receives an event from ES indicating some change in the system *state*, i.e., the considered impact factors (e.g., request size), this event triggers an *evaluation process*. If this process resulted in a new protocol p_j with a sufficient improvement, i.e., a predefined *threshold*, over the current performance of the system, then QCS orders the BFTS to abort the current protocol and switch to p_j; otherwise, p_i is kept running.

B. The QCS Evaluation Process

The evaluation process operates in three modes: *static, dynamic,* and *heuristic*. In the static mode, the evaluation process selects the best protocol according to its (predefined) characteristics, and is done before the system starts. In the other two modes, however, the evaluation process selects the best protocols in a dynamic way at run-time. Evaluation includes the characteristics of the protocols and their performance. In addition, the heuristics mode (as explained later) uses some heuristic rules to adjust the system behavior in exceptional cases. In this paper, we focus on the dynamic and heuristic modes whereas details about the static mode can be found in [18], [19]. We make the evaluation process easy to understand by providing a simple example in Fig. 2.

1) Evaluation metrics: Evaluations are conducted considering two types of metrics: *Key Characteristic Indicators* (KCIs), and *Key Performance Indicators* (KPIs). The KCIs represent the fixed (or static) characteristics of the protocol like: whether it tolerates client faults, the number of replicas needed to tolerate f faults, etc. The KPIs are the dynamically computed metrics that evaluate the performance of a protocol, e.g., throughput and latency. KCI values can be defined by analyzing the BFT protocols, whereas, KPI values are computed at run-time using prediction mechanisms.

Example: *A simple example that describes the computations of the evaluation process.*

1. KCI Symbols: *spec=**Spec**ulative, byz=tolerates**Byz**antineClients, ip=No**IP**Multicast*
2. KPI Symbols: *thr=**Thr**oughput, lat=**Lat**ency, and cap=**Cap**acity.*
3. Considered protocols: *PBFT, Zyzzyva, and Quorum.*
4. Given user preferences (metrics weights): *U means a protocol must be tolerant to Byz. clients, with no IP multicast, and not necessarily speculative. V means throughput is given higher priority over latency and capacity. W means no heuristics are used.*

$$U = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{array}{l} \leftarrow spec \\ \leftarrow byz \\ \leftarrow ip \end{array} ; V = \begin{pmatrix} 5 \\ 2 \\ 3 \end{pmatrix} \begin{array}{l} \leftarrow thr \\ \leftarrow lat \\ \leftarrow cap \end{array} ; \text{and } W = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \begin{array}{l} \leftarrow thr \\ \leftarrow lat \\ \leftarrow cap \end{array} ;$$

5. KCI and KPI values: *The predefined KCI values are presented in matrix A. The KPI values in matrix B are theoretically estimated using message pattern and MAC operations. (In Section V we use accurate experimental results). $B^{\pm}$ is derived from B using Eq. 3.*

$$A = \begin{array}{c} \begin{array}{ccc} spec & byz & ip \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} \leftarrow PBFT \\ \leftarrow Zyzzyva \\ \leftarrow Quorum \end{array} ; B = \left. \begin{array}{c} \begin{array}{ccc} thr & lat & cap \end{array} \\ \begin{pmatrix} 0.36 & 0.4 & 0.8 \\ 0.43 & 0.3 & 0.7 \\ 0.5 & 0.2 & 0.3 \end{pmatrix} \end{array} \begin{array}{l} \leftarrow PBFT \\ \leftarrow Zyzzyva \\ \leftarrow Quorum \end{array} \right\} \Longrightarrow B^{\pm} = \begin{array}{c} \begin{array}{ccc} thr & lat & cap \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0.5 & 0.5 & 0.8 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

6. Computing C: *C indicates that Quorum does not satisfy the KCI user requirements (in matrix U).*

$$C = \left[\frac{1}{a}.\left(A \dot{\vee} \left(e_a - U\right)\right)\right] = \left[\frac{1}{3}.\left(\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \dot{\vee} \left(\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right)\right)\right] = \left[\begin{pmatrix} 1 \\ 1 \\ 2/3 \end{pmatrix}\right] = \begin{pmatrix} \mathbf{1} \\ \mathbf{1} \\ 0 \end{pmatrix} \begin{array}{l} \leftarrow PBFT \\ \leftarrow Zyzzyva \\ \leftarrow Quorum \end{array}$$

7. Computing P: *P indicates that Quorum, theoretically, achieves the best performance (without considering matrix C yet).*

$$P = B^{\pm}.(V \circ W) = \begin{pmatrix} 0 & 0 & 1 \\ 0.5 & 0.5 & 0.8 \\ 1 & 1 & 0 \end{pmatrix} . \begin{pmatrix} 5 \\ 2 \\ 3 \end{pmatrix} \circ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0.5 & 0.5 & 0.8 \\ 1 & 1 & 0 \end{pmatrix} . \begin{pmatrix} 5 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 3 \\ 5.9 \\ \mathbf{7} \end{pmatrix} \begin{array}{l} \leftarrow PBFT \\ \leftarrow Zyzzyva \\ \leftarrow Quorum \end{array}$$

8. Computing E: *E indicates that Zyzzyva is chosen as the best protocol (theoretically) since Quorum is now ruled out.*

$$E \;=\; C \circ P = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \circ \begin{pmatrix} 3 \\ 5.9 \\ 7 \end{pmatrix} = \begin{pmatrix} 3 \\ \mathbf{5.9} \\ 0 \end{pmatrix} \begin{array}{l} \leftarrow PBFT \\ \leftarrow \textbf{Zyzzyva} \\ \leftarrow Quorum \end{array}$$

Fig. 2. A simple example about using the BFT evaluation process.

2) Best protocol selection: The evaluation process ends up by selecting the preferred BFT protocol among a set of candidate ones in BFTS under the new conditions; this is achieved through computing the *evaluation scores* of the competing protocols, and then selecting the protocol that corresponds to the maximum score. More formally, for any state s, and protocol $p_i \in$ BFTS that has an evaluation score $E_{i,s}$; the best protocol p_{pref} is chosen according to Eq. 1:

$$p_{pref} = p_i, \;\; s.t. \; E_{i,s} = \underset{1 \leq j \leq n}{max} \, E_{j,s}. \tag{1}$$

Any evaluation score $E_{i,s}$ is calculated according to the formulas introduced in Eq. 2 which is explained next.

$$\begin{cases} E = C \circ P \\ where \; C = \left[\dfrac{1}{a}.\left(A \,\dot{\vee}\, \left(e_n - U\right)\right)\right] \\ and \; P = B^{\pm}.(V \circ W). \end{cases} \tag{2}$$

*a) **Matrix E**:* The evaluation matrix E is the *Schur* product [2] of the KCI matrix C and the KPI matrix P. C represents the part of the evaluation that deals with the KCIs of the protocols; whereas, P represents the evaluation part that deals with the KPIs. E is calculated after computing the values of C and P. The combination of C and P in computing evaluations is important as it eliminates the protocol that does not match the characteristics required by the user and, at the same time, recommends the protocol that has the best performance. If the mode of the system is dynamic or heuristic, then E may change at run-time as P also changes.

*b) **Matrix C**:* The matrix $C = \left\lfloor \frac{1}{a}.\left(A \,\dot{\vee}\, \left(e_n - U\right)\right)\right\rfloor$ matches given the user preferences against the characteristics of different protocols. Matrix A represents the profiles (i.e., the KCIs) of the protocols; each row represents a vector of a different KCIs. U is a vector matrix that represents the given user preferences (i.e., the weights). The operator $\dot{\vee}$ is similar to the product operator with using '$\vee$' boolean operator instead of the dot '.'. This is used to calculate the total number of KCIs that do match user preferences in U, per each protocol. The column matrix e_n is a unit matrix that is only used to invert the values of the matrix U to $-U$. The use of $1/a$ within the integer value operator [ ] rules out the protocol that does not match all user preferences in matrix U (see Fig. 2).

*c) **Matrix P**:* Matrix P is defined in the formula: $P = B^{\pm}.(V \circ W)$. B represents the matrix of considered KPIs of the protocols, one protocol per row. Since all KPIs are represented in the same matrix and the same formula, a special care must be taken to avoid "comparing oranges to apples". For instance, consider the two KPIs: throughput and latency. Throughput can have very high numbers (e.g., in

[2]A *Schur*, a.k.a., *Hadamard*, product of two matrices with entries a_{ij} and b_{ij}, resp., returns a new matrix where an entry $c_{ij} = b_{ij} \times a_{ij} \; \forall i, j$.

thousands) whereas latency can have small numbers (likely less than 1 sec). Simply adding or multiplying their values would give high significance for throughput. In addition, a higher throughput is better, while a higher latency is worse, which gives a wrong evaluation.

To handle these issues, we say that a KPI has the property Tendency='high' if a higher value means better evaluation score E, e.g., throughput; this KPI is denoted by β^+. On the contrary, a KPI of type β^- has the property Tendency='low', e.g., latency, where a higher KPI value means a worse evaluation score E. Now, suppose the number of β-KPIs is b, then the matrix B can be divided into b column matrices (i.e., vectors): $B_1, B_2, B_i, ...,$ and B_b. Let the maximum (resp., minimum) entry value of each vector B_i be max_i (resp., min_i), Then, we compute a new normalized matrix $B^\pm$ whose entries can be calculated according to Eq. 3:

$$\begin{cases} \beta_{ji}^+ = 1 - \dfrac{max_i - \beta_{ji}}{max_i - min_i}; \\ \beta_{ji}^- = 1 - \dfrac{\beta_{ji} - min_i}{max_i - min_i}; \\ where\ i \leq b\ and\ j \leq n. \end{cases} \quad (3)$$

where the entries of the matrices $B^\pm$ and B are denoted by $\beta^\pm$ and β, respectively. This brings three benefits: (1) all the KPI values are now bounded by the same interval $[0, 1]$ which does not give any significance for a KPI over the others; (2) a higher value in $B^\pm$ means a better performance whatever is the KPI; (3) values are now small floats which are easier to compute by the processor.

V is a column matrix that represents the KPI user-defined weights for evaluations. The matrix follows two constraints: (1) its entries are in $[0, 10]$, and (2) their sum $\sum_{i=1}^{b} v_{i1} = 10$. Matrix W is a column matrix only used in the heuristic mode. W is important to adjust the user preferences given in V (using the Schur product $V \circ W$) according to predefined heuristic rules (discussed next). Finally, the computation of P becomes straightforward (see Fig 2).

C. Heuristics

The weights in matrix V are offered by the user telling which KPI is given a higher priority. However, our experience shows that there are practical considerations that make these weights less effective and thus require special intervention, which we enforce using heuristic rules. To clarify our idea, consider the following two heuristic rules:

(H1) when concurrent clients are few, latency becomes more important than throughput and capacity and

(H2) under high contention, capacity and throughput become significantly more critical than latency.

The first heuristic rule is valid as no bottlenecks are present and thus the priority is to give faster replies to current users. Thus regardless of the weights provided by the user in matrix V, latency must be given high priority under this condition, this is achieved by giving a large weight for latency in the matrix W. The second rule follows the opposite logic and thus latency is given low priority regardless of the user choice. The values of matrix W have the same constraints as matrix V explained in the previous section. Using these rules improves the evaluations that would have resulted using V alone (notice that V and W are *Schur* multiplied). (Defining a more complete set of heuristic rules can be a future study).

D. Worthy Switching

The evaluation process described above runs the protocol that is roughly better than others under the current state (Eq. 1). This is theoretically sound; however, in practice, it can impose a high cost due to the switching overhead, discussed in Section V. This makes it costly and useless to switch from one protocol to another if the improvement induced by the new one is not significant. Therefore, it makes sense to only switch when the new protocol brings sufficient benefits, i.e., exceeding a predefined *switching threshold*: S_{thr}. Consequently, Eq. 1 can be confined with the following constraint:

$$\frac{p_{max}}{p_{curr}} \geq S_{thr}$$

where p_{max} and p_{curr} are the evaluation scores corresponding to the best protocol (under the current state) and the currently running one, respectively. S_{thr} can be defined by the service administration (e.g., 10%).

It can happen that the current switching phase was triggered by a malicious behavior event (induced by ES) and that the performance of other protocols is not sufficiently better than the current one. This is however safe, since the evaluation process considers KCIs too, which will exclude the protocols that are not robust to the current malicious state. In Section V, we show that ADAPT switches to PBFT under faults, being the most robust protocol in BFTS.

E. Prediction of KPIs

The evaluation process strictly depends on the values of KCIs and KPIs. As mentioned above, KCIs are fixed values defined before the system starts. However, KPI values must be computed at run-time in order to perform a correct evaluation since a protocol's performance can change as the system conditions vary. In ADAPT, we assess the KPI values, experimentally, as it gives run-time accurate numbers; to the contrary of theoretical analysis and simulation-based methods in [13] and [14] that give a general inaccurate theoretical results. This is done using prediction mechanisms, like Support Vector Machines for Regression [12] (SVR).

We briefly describe the prediction process of a single KPI in the following. First, each protocol is run a period of time while tuning the *impact factors* and getting the KPI values under each state. After a sufficient set of records is collected, i.e, a *training set*, a *prediction function* is designated and trained on it. The prediction function takes the impact factors as input and outputs a KPI value. The purpose is to find the parametric values of the prediction function that give the most accurate predictions. Once training is done, at any instant while the system is running, when ES sends an event with new values for the impact factors, the prediction function is executed on these values and returns a new predicted value for each KPI (that is used in matrix B above). To get accurate predictions, the training set must always be updated by ES, while the system is running, such that the prediction function periodically "improves itself".

F. A Robust QCS

The role of QCS is essential in ADAPT as it controls the entire system and thus must also be Byzantine resilient. To achieve this, a shadow control channel runs QCS using PBFT protocol on the same set of replicas, in parallel with the other running protocol. The parallel protocols are completely separated and not allowed to interfere. This means that QCS is run on all replicas and the switching decisions are taken by the primary replica in the current PBFT *view*. QCS runs PBFT since it is a robust protocol suitable for critical services, like QCS, where performance is not a requirement, being not exposed to clients (the clients in this case are the replicas themselves). Since the evaluation process often occurs silently, while the current protocol is running, using PBFT has no effect on the performance of this process. The only affected case is when the system switches due to some fault; this forces the current protocol to abort and wait until evaluation finishes. In this case, the delay to achieve consensus in PBFT imposes additional time on the evaluation process that we discuss in Section V-G.

V. Evaluation

A. Evaluation Methodology

To evaluate our approach, we first briefly describe ADAPT code, and then we move to the experimental settings including the code setup, applications, and prediction methods. Second, we present the scores inferred by the evaluation process showing that different protocols are chosen in different conditions. These scores are also important to help explain the performance graphs of ADAPT that dominates those of other protocols. Then, we show how ADAPT improves the switching dynamics of Aliph, and that the switching cost in ADAPT is also not significant. At the end, we present an experiment with dynamic workload showing that Zyzzyva sometimes perform better than Aliph due to the classical static switching used; and that this issue is absent in ADAPT. In our experiments, we considered:

- Five BFT protocols: PBFT, Zyzzyva, Quorum, Ring, and Chain; we have chosen these protocols as they have clear improvement over others which helps explaining our idea.

- Three KPIs: throughput, latency, and capacity; being the most important metrics for performance. Capacity in our context means the maximum number of clients that a protocol tolerates.

- Four impact factors: number of clients, request size, response size, and faulty replicas; as we noticed through our experiments that these are the most significant factors on most BFT protocols.

For the evaluation process (introduced in Section III), we used the client preferences represented by the following matrices: $U = e_n$ meaning that no constraints are made on the KCIs of the protocols, i.e., all protocols are accepted to show the strengths of the dynamic switching using KPIs. $V = (3, 3, 4)$ corresponding to (throughput, latency, capacity), meaning that the three metrics are given similar weights. In addition, we translate the heuristic rules, H1 and H2, defined in Subsection IV-C to using $W = (1, 8, 1)$ and $W = (2, 1, 7)$, respectively.

B. The Implementation in Brief

The code of ADAPT is divided into several modules. The QCS is implemented in 1274 lines of C/C++ code comprising the evaluation process and the mathematical equations introduced above. We used one thread per KPI such that evaluations of different KPIs take place in parallel. A plug-in of eight Bash scripts was also implemented to interface the control module with the prediction library used, i.e., the Java LIBSVM [20]. Another module implements the switching logic of abortability which is a modified version of the one used in Aliph [9]. This module also collects the BFT libraries of the considered protocols in our experiments. Finally, we implemented a simple module in C++ to act as an Event System in order to conduct our experiment, e.g., we use message buffers and sockets to get the number of concurrent clients and to check the message sizes, etc.

C. Experimental Settings

1) The Setup: We experimented ADAPT on a cluster of 25 64-*bit* $Xeon$ machines with 2 GB of memory, running *Ubuntu* OS, and deployed on Emulab [15] test-bed. All modules were installed on all replicas, whereas, the QCS was only run on one machine. The number of replicas is four (i.e., f=1). Each replica runs on a separate machine and the client processes share 20 machines. The maximum bandwidth of the network is set to 100Mb to be able to easily saturate the network and observe the behavior of the protocols. To compare our results, we use the standard *a/b* benchmark [3] with different payload sizes [4]: 0/0, 4/0 and 0/1. (We experimented other payload sizes too but we don't present them since they are similar to the presented ones, and because we believe these are enough to clarify our idea.)

2) About Applications: We considered three different applications in our experiments: (1) a simple integer-increment, (2) the key-value store Redis [16], and (3) OpenLDAP [21]. The purpose was to cover a wide range of applications where BFT protocols may behave differently. It turned out that as the execution time of the considered application increases, the different protocols converge in performance. For instance, our results, confirm the results published in literature, e.g., [4], [7], [8], [9], [10], [11], while negligible execution time is considered, as in case (1). On the contrary, with OpenLDAP (having more than 1ms execution time), the performance of the protocols was very close ([22] explains this observation in more detail). Consequently, ADAPT is not very effective

[3]In a/b benchmarks, *a*, and *b* correspond to request size, and response size in KB, respectively.

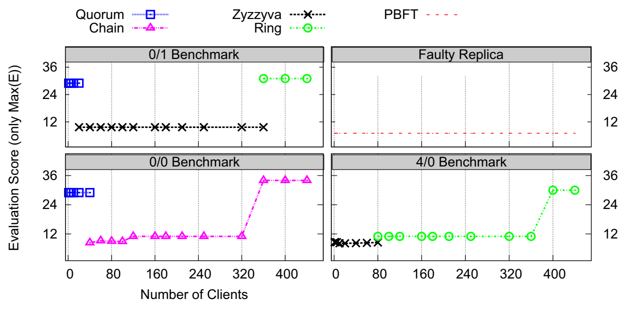


Fig. 3. The protocols with the highest evaluation scores (E) under different conditions.

with OpenLDAP since no switching will occur. On the other hand, using a naive application like integer incrementing is not too realistic, though it can show the strengths of ADAPT more clearly. This encourages us to only consider Redis (due to space limitations) as a trade-off, and being a cutting-edge application that spans a wide market. Redis server was installed on all system replicas, and accessed locally by each BFT library through its API.

3) Computing Predictions: KPI predictions were done using the Machine Learning SVR method [12] with the Radial Basis Kernel (RBK) supported by the LIBSVM library [20]. The parameters of the SVR prediction function (C and γ) were chosen using the *five-fold* cross-validation [12] over a *training set* and *test set*. In total, the size of the data instances was around 500 records of the form (number of clients, request size, response size, throughput, latency, capacity). We divided the data instances into: a training set (85%), and a test set (15%). The former was used to train the prediction function, and the latter to evaluate its accuracy. We have run the five protocols for 10 continuous days with varying the values of the four impact factors introduced above. In essence, we used nine request size in the range [0B, 4KB]; and used three response size: 0B, 64B, and 1KB. The number of simultaneous clients ranged between 1 to 450 clients, with running more frequent experiments with fewer clients. The achieved prediction accuracy using RBK was more than 95% which was sufficient to get accurate evaluation scores for ADAPT. (We omit prediction details for space limits). The prediction database (raw data files) of the QCS was hosted by the same replica where QCS is located.

D. Evaluation Scores

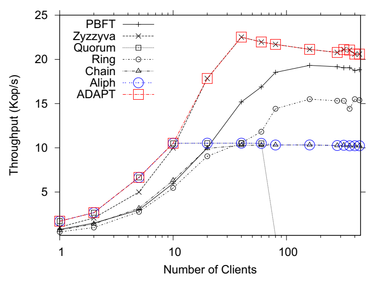
ADAPT launches the protocol that achieves the best evaluation score as described in Section III. Fig. 3 conveys the results of the evaluation scores (y-axis) returned by the evaluation process with different payloads 0/0, 4/0, and 0/1 as the number of concurrent clients change (x-axis). (We present 0/1 instead of 0/4 since we had problems with Zyzzyva code). The figure only depicts the protocols with the highest scores, i.e., those will be launched by ADAPT in the corresponding conditions. The figure shows that different protocols are better under different conditions. For instance, Quorum leads other protocols with small message sizes and few clients, e.g., up to 30 clients in the 0/0 payload experiment. We expected this since Quorum has the lowest latency among other protocols, and due to using the heuristic rule H1 in this case. Beyond 30 clients, Chain is chosen as best protocol since it theoretically requires almost one MAC operation per replica. With larger message sizes, e.g., 4/0 and 0/1 experiments, Chain saturates the network (since a request visits all replicas); consequently, Zyzzyva is chosen in this case up to 80 clients. This is referred to the short messaging pattern in Zyzzyva and having replicas to respond with response digests directly to the clients. Under high contention, e.g., beyond 80 clients in the 4/0 experiment, Ring achieves the highest evaluation score since it allows all replicas to receive requests from clients. Finally, PBFT protocol is the only protocol that operates in presence of faults, and thus it gets the highest evaluation score under these conditions. Next, we show how these evaluation scores impact the performance of ADAPT.

E. Performance Comparison

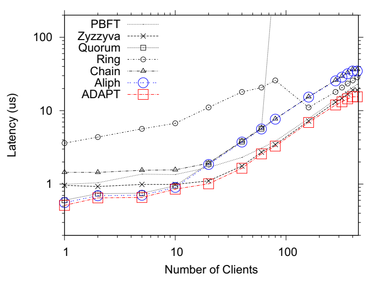
In general, a chosen protocol in Fig. 3 would mean a high throughput and low latency in the performance graphs presented next in Fig. 4. For brevity, we only present the results for 4/0 and 0/1 payloads (other payloads are similar).

a) 0/1 benchmark.: As shown in Fig. 4(a) and 4(b), ADAPT outperforms other protocols with large responses, i.e., 0/1 payloads, in throughput and latency. ADAPT launches Quorum up to 10 clients (using heuristic rule H1) and then switches to Zyzzyva as noticed before in Fig. 3. Using Quorum with few clients makes the throughput of ADAPT slightly better than Zyzzyva, however, significantly better than Zyzzyva in latency (since H1 says that latency is more important with few clients). The throughput of ADAPT, as well as latency, is close to Aliph that also runs Quorum with few clients. Then, the throughput and latency of ADAPT becomes close to Zyzzyva since ADAPT switches to Zyzzyva that leads other protocols. This result is interesting since as it shows the strengths of the dynamic switching of ADAPT in comparison to static switching in Aliph. In fact, Aliph switches from Quorum to Chain with 80 clients once Quorum crashes. Between 10 and 80 clients, ADAPT runs Zyzzyva that significantly dominates Quorum (almost double its throughput). Then, Aliph has the only choice to switch to Chain; whereas, ADAPT switches in a smart way to Zyzzyva, thus achieving a higher throughput than Aliph; this limitation is due to the pre-defined order of protocols in Aliph (i.e., Chain, Quorum, then PBFT).

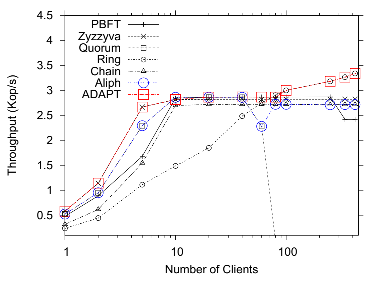
b) 4/0 benchmark.: With 4/0 payloads, ADAPT leads the other protocols too, as depicted in Fig. 4(c) and 4(d). ADAPT launches Zyzzyva with up to 80 clients, after which Ring leads Zyzzyva, and thus, ADAPT switches to Ring to benefit from its performance under high load (see Fig. 3); consequently, ADAPT outperforms both Zyzzyva and Ring. ADAPT outperforms Quorum and Chain too. We refer this to using message digests with MAC authentication in Zyzzyva instead of Quorum that uses complete messages and RSA authentication. On the other hand, sending message digests



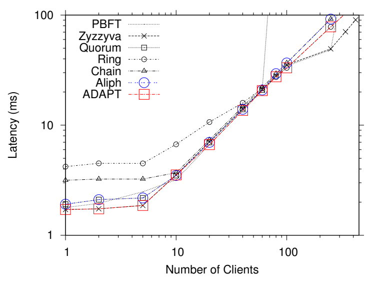
(a) Throughput (0/1).



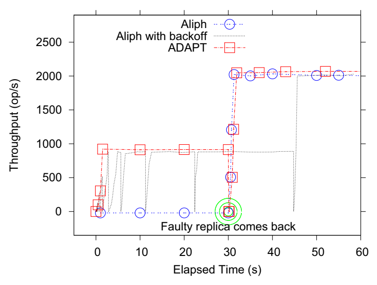
(b) Latency (0/1).



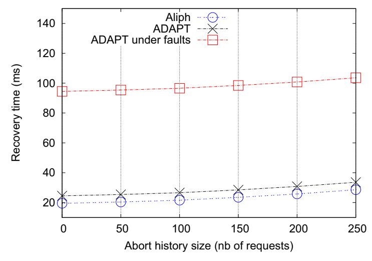
(c) Throughput (4/0).



(d) Latency (4/0).



(e) Throughput (0/0) under faults.



(f) Switching time.

Fig. 4. Performance comparison of BFT protocols.

directly to the client in Zyzzyva avoids the network bottlenecks found in Chain. Moreover, ADAPT dominates Aliph since the latter uses Quorum with few clients an switches to Chain with 80 clients (when Quorum crashes). Aliph could not benefit from the performance of Ring and Zyzzyva, as ADAPT does, due for two reasons: (1) its predefined switching order: Quorum, Chain, and then PBFT; and (2) since it lacks the sense of intelligent dynamic switching like ADAPT (e.g., switch when you can gain performance). As for latency, Fig. 4(d) shows that the latencies of ADAPT, Aliph, and Zyzzyva are close. The reason is that Quorum and Zyzzyva exhibit similar latencies under these conditions, and since ADAPT and Aliph run Zyzzyva and Quorum, respectively. Notice that with more than 400 clients, ADAPT launches Ring as it has a higher capacity to clients (2000 clients), though the latency of Zyzzyva is lower. This is explained by the heuristic rule H2 that gives more weight to throughput and capacity as the number of clients gets very high (to avoid the risky running of Zyzzyva).

F. The Case of Faulty Replicas

Fig. 4(e) depicts a 0/0 payload comparison of throughput between ADAPT and Aliph with one faulty replica, where both protocols switch to PBFT. We do not plot other protocols since Chain and Quorum could not run under failures; whereas Zyzzyva achieves 15% lower throughput than PBFT under failures as the authors mention in [8]. Ring is worst than PBFT under faults too [17] due to its long *two-chain-rounds* messaging pattern.

For Aliph, Fig. 4(e) plots the curves for the default rigorous switching and for the exponential backoff switching. To simulate faulty replicas, we disconnect one replica for 30 seconds. During this period, ADAPT switches to PBFT (as already seen in Figure 3). Once the faulty replica comes back, ADAPT switches back to Quorum that achieves the best evaluation score with 0/0 payload. On the contrary, Aliph with the rigorous scheme repeats the process of "executing one request using PBFT, switching back to Quorum, it fails, an then switches to PBFT again" until the faulty replica comes back, where Aliph finally runs Quorum. The throughput during these 30 seconds period is close to zero. With exponential backoff scheme, Aliph runs PBFT for 2 requests and attempts to switch to Quorum, it fails, and runs back PBFT each time with 2^i requests (where i is the switching attempt number). Aliph with backoff scheme switches back to Quorum where $i = 14$; thus, executing around $36K$ operations during 45 seconds. Notice that, in this scheme Aliph could not immediately switch back to Quorum after the faulty replica recovers, at 30 seconds. Therefore, during 45 seconds, ADAPT executes around 66K operations by the time Aliph switches back to Quorum with 36K operations in the backoff scheme, and 30K operations in the rigorous scheme. This is expected since ADAPT switches either upon failures or when performance changes, whereas Aliph only switches upon failures.

G. Switching Cost

We only compare switching cost in ADAPT with Aliph since it is the only protocol among others that uses switching. The results in Fig. 4(f) show that the switching overhead of 1KB requests is negligible (up to 30 ms), even with an *abort history (AH)* of 250 requests. In fact, ADAPT uses a similar fine-grained checkpoint algorithm as in Aliph [9], which prevents AH from getting too large. Although applications can have long request execution times (which is $\approx 20\mu s$ in

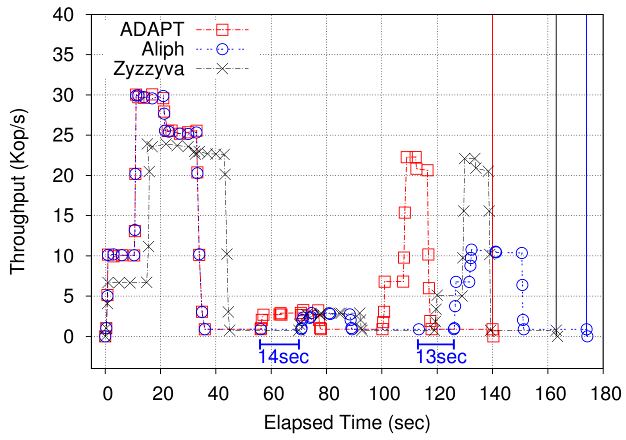


Fig. 5. Throughput under dynamic workloads.

Redis [16]), this does not affect ADAPT and Aliph since they do not execute operations that are previously executed. The switching time of ADAPT is close to Aliph with no faults since ADAPT runs the evaluation process before switching, and only switches if it is *worthy* (as explained in Subsection IV-D). Upon failures, however, ADAPT runs the evaluation process (including predictions) after aborting; this causes a higher switching overhead that goes up to 100 ms with 250 AH size. This cost is considered tolerable and has no big impact on the performance as explained in Subsection V-E above.

H. Performance Under Dynamic Workloads

We compare ADAPT, Aliph, and Zyzzyva under dynamic workloads. We discard the other protocols as they exhibit poor performance in this case. We injected to the system 1M operations in three bursts of different message payloads: 7×10^5 0/0, 6×10^4 4/0, and 25×10^4 0/1. We used fewer requests with larger payloads since it is very time consuming (though ADAPT can perform better than others under these conditions since it runs Ring). For each burst, we changed the number of clients from 5, to 30, and then to 250. To consider workload under faults, after each burst, we reduced the number of clients to 5 and we disconnected one replica (not the primary), then we injected 20K 0/0 operations.

Fig. 5 shows that Aliph and Zyzzyva require 24% and 16.5% additional time to finish the 1M requests, respectively. We expected this result as ADAPT uses a combination of Aliph (that uses Quorum, Chain, and PBFT), Zyzzyva, and Ring, choosing the best protocol under different conditions. The surprising result, however, is that Zyzzyva finished before Aliph. We did not expect this at first; however, after deeper observation, we referred this to two reasons:

- The first reason is that, under faults, and although the throughput of Aliph (i.e., 871KB/s) is better than Zyzzyva [4] (i.e., 760KB/s), Aliph is delayed 27 seconds after two faulty periods, labeled in blue on Figure 5, due to the exponential backoff scheme. In the first faulty period (at 38sec), for example, we noticed that Aliph was forced to switch back and forth from Quorum to PBFT 15 times; and thus executing 2^{15}=32K requests instead of 20K, meaning that it executed 32K-20K=12K requests while running PBFT (of throughput 871KB/s) instead of Quorum (of throughput 2290KB/s), yielding a backoff overhead of 14 seconds. Zyzzyva, however, finished these 12K requests in 4.5 seconds as it has a 2820KB/s throughput. Notice that ADAPT does not pay this handoff cost as it immediately switches to Zyzzyva once the faulty replica recovers.

- The second reason is that Zyzzyva performs better than Chain (that is used by Aliph) with large responses (1KB and more). This is expected since Zyzzyva replicas (except one replica) send response digests directly to the client, whereas a response in Chain visits all replicas that overloads the network. Simply using Zyzzyva in Aliph (in addition to Quorum, Chain, and PBFT) is not trivial since it is not clear when to abort from Chain to Zyzzyva or vice versa; unless similar methods to those used in ADAPT are used.

[4]Since our Zyzzyva code does not work properly under faults, we estimated Zyzzyva's throughput under faults according to the Zyzzyva paper [8] stating that it performs 15% worst than PBFT under failures.

VI. Related Work

BFT Protocols. The *Byzantine generals problem* was introduced by Leslie Lamport in [3]. Then, the first practical BFT protocol (PBFT) was introduced in [4] by Castro et al. PBFT has proven to be robust under faults, though it exhibits a low performance in fault-free conditions. Later works ([7], [8], [9], [11], [10], etc.) then appeared to enhance the performance of PBFT, often using *speculation* in fault-free periods, and an expensive recovery phase upon failures. Nevertheless, no single protocol dominated the others under all conditions. The notion of *abortability*, proposed in [9], enabled the use of multiple BFT protocols and switching between them in a static pre-defined order. This paper shows that abortability could not achieve the expected results under dynamic payloads using the simple static switching with *backoff scheme*, as in [9], and proposes an alternative smart dynamic switching policy.

Performance Assessment. Analytical models like [13] and [14] give a general idea about the performance of BFT protocols; however, these models are not very accurate in systems that have highly dynamic conditions, and they could not be used efficiently at run-time. Our approach requires accurate predictions in order to achieve meaningful decision making. In this paper, we adopt a Machine Learning (ML) prediction method (i.e., SVR [12]) based on real experimentation, leading more than 95% prediction accuracy. To the best of our knowledge, the idea of using ML is new to fault tolerant protocols either *Byzantine* or *benign*.

Adaptive Fault Tolerance. To the best of our knowledge, this work represents the first dynamically adaptive BFT approach. The abortable BFT protocol introduced in [9] is statically adaptive, meaning that the protocols and their order must be defined before deployment. Other adaptive fault tolerant approaches existed in literature, e.g., in databases [23] and clouds [24], but they are adaptive in the sense that they try to change the strategy of distributed objects across replicas or by using a variant number of nodes; however, ours is adaptive by changing the running protocol itself without touching the data objects. For instance, the fault tolerant database idea in [23] changes the replication scheme of an object (number of reads/writes); in [24] the authors present a fault tolerant cloud system where add/remove operations are done according to the reliability level of an image. Other multi-agent systems [25] and sensor networks [26] tackled similar problems by changing the strategy played among agents, or by changing paths in sensor networks, respectively.

VII. Concluding Remarks

The fault tolerance area was flooded by dozens of BFT protocols trying to improve their robustness and performance. But it remained hard to cope with systems that exhibit variable conditions and workloads. A promising idea, i.e., *abortability*, was introduced in [9] to combine existing protocols in a single system that can run one of them at a time. In this paper, we have shown that this approach does not achieve the anticipated results without a *dynamic switching* mechanism to move from one protocol to another, as the system conditions change.

We introduced ADAPT, an adaptive *abortable* BFT system with a dynamic switching method that evaluates each protocol at run-time, using Machine Learning predictions, and switches to the protocol having the highest evaluation score. This is triggered by an event sent by a system module that monitors the system state (number of clients, faults, message sizes, etc.). To compute the evaluation scores, we devised some mathematical equations that match the characteristics of a protocol and its performance metrics against the BFT user (i.e., service owner) preferences. We conducted some experiments showing that ADAPT outperforms other protocols under most conditions, and especially dynamic workloads.

Our approach is useful for systems having variable conditions and workloads, and such that the execution time of operations is small (e.g., up to 100ms). Longer execution times mask the communication overhead of protocols and makes their performance close. In this case, switching is a waste of time. Furthermore, systems that experience stable conditions and constant message payloads are advised to use a single BFT protocol to avoid the switching overhead and the added complexity induced by our approach. Our experience shows that, in many cases, Zyzzyva (or another variant, e.g., Aardvark [10]) is known to perform very well, whereas PBFT is recommended when the performance is not a major concern.

A future work can be to involve more BFT protocols in ADAPT, and to define a set of heuristic rules to improve its performance further. Introducing an Event System (ES) for ADAPT is another possible future direction.

References

[1] "Amazon aws outage," http://aws.amazon.com/message/656481/, Dec. 2013, accessed: 2014-10-10.

[2] "Google mail outage," http://googleblog.blogspot.pt/2014/01/todays-outage-for-several-google.html, Jan. 2014, accessed: 2014-10-10.

[3] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, 1982.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[5] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[6] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 253–267, Oct. 2003.

[7] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, 2005.

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 45–58, 2007.

[9] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 363–376.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–168.

[11] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 4, pp. 564–577, Jul. 2011.

[12] Smola, Alex J. and Schölkopf, Bernhard, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, pp. 199–222, 2004.

[13] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX, 2008, pp. 189–204.

[14] Raluca Halalai, Thomas Henzinger, and Vasu Singh, "Quantitative evaluation of bft protocols," *Quantitative Evaluation of Systems, International Conference on*, pp. 255–264, 2011.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *SIGOPS Oper. Syst. Rev.*, pp. 255–270, 2002.

[16] Redis, "Redis." [Online]. Available: http://redis.io/

[17] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, Marco Vukolic, "Stretching bft," EPFL, Tech. Rep. EPFL-REPORT-149105, 2011.

[18] Ali Shoker and Jean-Paul Bahsoun, "BFT Selection," in *Proceedings of the International Conference of Networked Systems*, ser. NETYS'13. Springer, May 2013.

[19] ——, "Bft selection," IRIT, France., Tech. Rep., 2013.

[20] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[21] K. Zeilenga, H. Chu, P. Masarati, and Others, "OpenLDAP," Computer Software, 1998. [Online]. Available: http://www.openldap.org/

[22] Ali Shoker and Jean-Paul Bahsoun, "Towards Byzantine Resilient Directories," in *The 11th IEEE International Symposium on Network Computing and Applications*, ser. NCA'12. IEEE Computer Society, August 2012.

[23] O. Wolfson, S. Jajodia, and Y. Huang, "An adaptive data replication algorithm," *ACM Trans. Database Syst.*, vol. 22, pp. 255–314, Jun. 1997.

[24] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *Proceedings of the 2011 IEEE World Congress on Services*, ser. SERVICES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 280–287.

[25] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, pp. 560–579, Jun. 1999.

[26] I.-R. Chen, A. P. Speer, and M. Eltoweissy, "Adaptive fault-tolerant qos control algorithms for maximizing system lifetime of query-based wireless sensor networks," *IEEE Transactions on Dependable and Secure Computing*, pp. 161–176, 2011.