The Next 700 BFT Protocols

Rachid Guerraoui,
Nikola Knežević
EPFL, Switzerland
rachid.guerraoui@epfl.ch,
nikola.knezevic@epfl.ch

Vivien Quéma
CNRS, France
vivien.quema@inria.fr

Marko Vukolić
IBM Research - Zurich, Switzerland
mvu@zurich.ibm.com

Abstract

Modern Byzantine fault-tolerant state machine replication (BFT) protocols involve about 20,000 lines of challenging C++ code encompassing synchronization, networking and cryptography. They are notoriously difficult to develop, test and prove. We present a new abstraction to simplify these tasks. We treat a BFT protocol as a composition of instances of our abstraction. Each instance is developed and analyzed independently.

To illustrate our approach, we first show how our abstraction can be used to obtain the benefits of a state-of-the-art BFT protocol with much less pain. Namely, we develop *AZyzzyva*, a new protocol that mimics the behavior of Zyzzyva in best-case situations (for which Zyzzyva was optimized) using less than 24% of the actual code of Zyzzyva. To cover worst-case situations, our abstraction enables to use in *AZyzzyva* any existing BFT protocol, typically, a classical one like PBFT which has been tested and proved correct.

We then present *Aliph*, a new BFT protocol that outperforms previous BFT protocols both in terms of latency (by up to 30%) and throughput (by up to 360%). The development of *Aliph* required two new instances of our abstraction. Each instance contains less than 25% of the code needed to develop state-of-the-art BFT protocols.

Categories and Subject Descriptors D.4.7 [*Organization and Design*]: Distributed Systems; C.4 [*Performance of Systems*]: Fault tolerance

General Terms Algorithms, Design, Performance, Reliability

Keywords Byzantine failures; performance; modularity.

1. Introduction

State machine replication (SMR) is a software technique for tolerating failures using commodity hardware. The critical service to be made fault-tolerant is modeled by a state machine. Several, possibly different, copies of the state machine are then placed on different nodes. Clients of the service access the replicas through a SMR protocol which ensures that, despite contention and failures, replicas perform client requests in the same order.

Two objectives underly the design and implementation of a SMR protocol: *robustness* and *performance*. Robustness conveys the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. On the other hand, performance measures the time it takes to respond to a request (latency) and the number of requests that can be processed per time unit (throughput). The most robust protocols are those that tolerate (a) arbitrarily large periods of asynchrony, during which communication delays and process relative speeds are unbounded, and (b) arbitrary (Byzantine) failures of any client as well as up to one-third of the replicas (this is the theoretical lower bound [24]). These are called Byzantine-Fault-Tolerance SMR protocols, or simply *BFT* protocols, e.g., PBFT, HQ and Zyzzyva [7, 13, 20]. The ultimate goal of the designer of a BFT protocol is to exhibit comparable performance to a non-replicated server under "common" circumstances that are considered the most frequent in practice. The notion of "common" circumstance might depend on the application and underlying network, but it usually means network synchrony, rare failures, and sometimes also the absence of contention.

Not surprisingly, even under the same notion of "common" case, there is no "one size that fits all" BFT protocol [28]. According to our own experience, the performance differences among the protocols can be heavily impacted by the actual network, the size of the messages, the very nature of the "common" case (e.g, contention or not); the actual number of clients, the total number of replicas as well as the cost of the cryptographic libraries being used. This echoes [28] which concluded for instance that *"PBFT [7] offers more predictable performance and scales better with* *payload size compared to Zyzzyva [20]; in contrast, Zyzzyva offers greater absolute throughput in wider-area, lossy networks"*. In fact, besides all BFT protocols mentioned above, one could design new protocols outperforming all others under specific circumstances. We do indeed present an example of a such protocol in this paper.

To deploy a BFT solution, a system designer will hence certainly be tempted to adapt a state-of-the-art BFT protocol to the specific application/network setting, and possibly keep adapting it whenever the setting changes. But this can rapidly turn into a nightmare. All protocol implementations we looked at involve around 20,000 lines of (non-trivial) C++ code, e.g., PBFT and Zyzzyva. Any change to an existing protocol, although algorithmically intuitive, is very painful. The changes of the protocol needed to optimize for the "common" case have sometimes strong impacts on the part of the protocol used in other cases (e.g., "view-change" in Zyzzyva). The only complete proof of a BFT protocol we knew of is that of PBFT and it involves 35 pages (even without using any formal language).[1] This difficulty, together with the impossibility of exhaustively testing distributed protocols [8] would rather plead for never changing a protocol that was tested and proven correct, e.g., PBFT.

We propose in this paper a way to have the cake and eat a big chunk of it. We present *A*bortable Byzantine faulT-toleRant stAte maChine replicaTion (we simply write Abstract): a new abstraction to reduce the development cost of BFT protocols. Following the divide-and-conquer principle, we view BFT protocols as a composition of instances of our abstraction, each instance targeted and optimized for specific system conditions. An instance of Abstract looks like BFT state machine replication, with one exception: it may sometimes *abort* a client's request.

The progress condition under which an Abstract instance should not abort is a generic parameter.[2] An extreme instance of Abstract is one that never aborts: this is exactly BFT. Interesting instances are "weaker" ones, in which an abort is allowed, e.g., if there is asynchrony or failures (or even contention). When such an instance aborts a client request, it returns an unforgeable request history that is used by the client (proxy) to "recover" by switching to another instance of Abstract. This new instance may commit subsequent requests until it itself aborts.

This paves the path to *composability* and flexibility of BFT protocol design. Indeed, the composition of any two Abstract instances is *idempotent*, yielding yet another Abstract instance. Hence, and as we will illustrate in the paper, the development (design, test, proof and implementation) of a BFT protocol boils down to:

- Developing individual Abstract instances. This is usually much simpler than developing a full-fledged BFT protocol and allows for very effective schemes. A single Abstract instance can be crafted solely with its progress in mind, irrespective of other instances.
- Ensuring that a request is not aborted by all instances. This can be made very simple by reusing, as a black-box, an existing BFT protocol as one of the instances, without indulging into complex modifications.

To demonstrate the benefits of Abstract, we present two BFT protocols:

1. *AZyzzyva*, a protocol that illustrates the ability of Abstract to significantly ease the development of BFT protocols. *AZyzzyva* is the composition of two Abstract instances: (i) *ZLight*, which mimics Zyzzyva [20] when there are no asynchrony or failures, and (ii) *Backup*, which handles the periods with asynchrony/failures by reusing, as a black-box, a legacy BFT protocol (we leveraged PBFT). The code line count and proof size required to obtain *AZyzzyva* are, conservatively, less than $1/4$ than those of Zyzzyva. In some sense, had Abstract been identified several years ago, the designers of Zyzzyva would have had a much easier task devising a correct protocol exhibiting the performance they were seeking. Instead, they had to hack PBFT and, as a result, obtained a protocol that is way less stable than PBFT.
2. *Aliph*, a protocol that demonstrates the ability of Abstract to develop novel efficient BFT protocols. *Aliph* achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. *Aliph* uses, along with the *Backup* instance used in *AZyzzyva*, two new instances: (i) *Quorum*, targeted for system conditions that do not involve asynchrony/failures/contention, and (ii) *Chain*, targeted for high-contention conditions without failures/asynchrony. *Quorum* has a very low-latency (like e.g., [1, 6, 15]) and it makes *Aliph* the first BFT protocol to achieve a latency of only 2 message delays with as few as $3f+1$ servers. *Chain* implements a pipeline message-pattern, and relies on a novel authentication technique. This makes *Aliph* the first BFT protocol with a number of MAC operations at the bottleneck server that tends to 1 in the absence of asynchrony/failures. (This contradicts the claim that the lower bound is 2 [20][3].) Interestingly, each of *Quorum* and *Chain* could be developed indepen-

[1] It took Roberto De Prisco a PhD (MIT) to formally (using IOA) prove the correctness of a state machine protocol that does not even deal with malicious faults [14].

[2] Abstract can be viewed as a *virtual type*; each specification of the this progress condition defines a concrete type. These genericity ideas date back to the seminal paper of Landin: *The Next 700 Programming Languages* (CACM, March 1966).

[3] The erroneous bound comes probably from the fact that authors considered that at least one server had to both receive requests from client and send replies to clients. This is not the case in Chain, which explains the reduced number of MAC operations at the bottleneck server.

dently and required less than 25% of the code needed to develop state-of-the-art BFT protocols.[4]

In the context of the paper, we assume a message-passing distributed system using a fully connected network among processes: clients and servers. The links between processes are asynchronous and unreliable: messages may be delayed or dropped (we speak of link failures). Certain periods of executions can be synchronous (i.e., when there are no link failures), meaning that any message m sent between two correct processes is delivered within a bounded delay Δ (known to sender and receiver) if the sender retransmits m until it is delivered. Also, certain periods are contention-free, meaning that no two clients seek to access in parallel to the same (replicated) state machine. Processes are Byzantine fault-prone; processes that do not fail are said to be correct. Any number of clients and up to f out of $3f+1$ servers can be Byzantine. A strong adversary can coordinate faulty nodes; however, we assume that the adversary cannot violate cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), and signatures.

The rest of the paper is organized as follows. Section 2 presents Abstract. We then describe and evaluate our BFT protocols: *AZyzzyva* in Section 3 and *Aliph* in Section 4. We discuss various aspects of our approach in Section 5. Section 6 discusses the related work and concludes the paper.[5]

2. Abstract

We propose a new approach for the development of BFT protocols. We view a BFT protocol as a composition of instances of Abstract. Each instance is itself a protocol that commits clients' requests, like any state machine replication (SMR) scheme, except if certain conditions are not satisfied, in which case it can abort requests. These conditions, determined by the developer of the particular instance, capture the progress semantics of that instance. They might depend on the design goals and the environment in which a particular instance is to be deployed. Each instance can be developed, proved and tested independently, and this modularity comes from two crucial properties of Abstract:

1. *Switching* between instances is *idempotent*: the composition of two Abstract instances yields yet another Abstract instance.
2. BFT is nothing but a special Abstract instance — *one that never aborts*.

[4] Our code counts are in fact conservative since they do not discount for the libraries shared between *ZLight*, *Quorum* and *Chain*, which amount to 10% of a state-of-the-art BFT protocol.

[5] For space constraints, the formal specification of Abstract and full implementation details are postponed to a companion technical report [17]. This also includes the details on model checking Abstract idempotency using TLA+ tools [23], pseudo-code and proofs of our protocols, and additional performance measurements.

A correct implementation of an Abstract instance always preserves BFT safety — this extends to any composition thereof. The designer of a BFT protocol only has to make sure that: a) individual Abstract implementations are correct, *irrespectively of each other*, and b) the composition of the chosen instances is live: i.e. that every request will eventually be committed. We exemplify this later, in Sections 3 and 4. In this paper, we highlight the main characteristics of Abstract. For space limitations, precise specification of Abstract and our theorem on Abstract switching idempotency are postponed to [17].

Switching. Every Abstract instance has a unique identifier (instance number) i. When an instance i commits a request, i returns a state-machine reply to the invoking client. Like any SMR scheme, i establishes a total order on all committed requests according to which the reply is computed for the client. If, however, i aborts a request, it returns to the client a digest of the history of requests h that were committed by i (possibly along with some uncommitted requests); this is called an *abort history*. In addition, i returns to the client the identifier of the next instance ($next(i)$) which should be invoked by the client: $next$ is the same function across all abort indications of instance i, and we say instance i *switches* to instance $next(i)$. In the context of BFT protocols presented in this paper, we consider $next$ to be a predetermined function (e.g., known to servers implementing a given Abstract instance); we talk about *deterministic* or *static* switching. However, this is not required by our specification; $next(i)$ can be computed "on-fly" by the Abstract implementation (e.g., depending on the current workload, or possible failures or asynchrony) as long as $next$ remains a function. In this case, we talk about *dynamic switching*; this is discussed in Section 5.

The client uses abort history h of i to invoke $next(i)$; in the context of $next(i)$, h is called an *init* history. Roughly speaking, $next(i)$ is initialized with an init history, before it starts committing/aborting requests. The initialization serves to transfer to instance $next(i)$ the information about the requests committed within instance i, in order to preserve total order among committed requests across instances.

Once i aborts some request and switches to $next(i)$, i cannot commit any subsequently invoked request. We impose *switching monotonicity*: for all i, $next(i) > i$. Consequently, Abstract instance i that fails to commit a request is abandoned and all clients go from there on to the next instance, never re-invoking i.

Illustration. Figure 1 depicts a possible run of a BFT system built using Abstract. To preserve consistency, Abstract properties ensure that, at any point in time, only one Abstract instance, called *active*, may commit requests. Client A starts sending requests to the first Abstract instance. The latter commits requests #1 to #49 and aborts request #50, becoming inactive. Abstract appends to the abort indication an (unforgeable) history ($hist\_1$) and the information about the

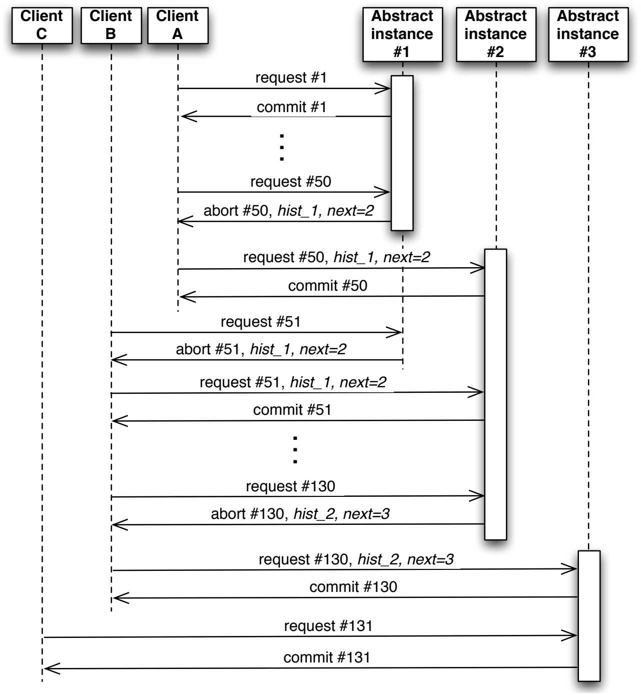

Figure 1. Abstract operating principle.

next Abstract instance to be used ($next = 2$). Client A sends to the new Abstract instance both its uncommitted request ($\#50$) and the history returned by the first Abstract instance. Instance $\#2$ gets initialized with the given history and executes request $\#50$. Later on, client B sends request $\#51$ to the first Abstract instance. The latter returns an abort indication with a possibly different history than the one returned to client A (yet both histories must contain previously committed requests #1 to #49). Client B subsequently sends request $\#51$ together with the history to the second abstract instance. The latter being already initialized, it simply ignores the history and executes request $\#51$. The second abstract instance then executes the subsequent requests up to request $\#130$ which it aborts. Client B uses the history returned by the second abstract instance to initialize the third abstract instance. The latter executes request $\#130$. Finally, Client C, sends request $\#131$ to the third instance, that executes it. Note that unlike Client B, Client C directly accesses the currently active instance. This is possible if Client C knows which instance is active, or if all three Abstract instances are implemented over the same set of replicas: replicas can then, for example, 'tunnel' the request to the active instance.

A view-change perspective. In a sense, an Abstract instance number can be seen as a view number, used in classical BFT protocols (including [7, 20]).[6] Like in these protocols, which merely reiterate the exact same sub-protocol across views (possibly changing the server acting as *leader*), the same Abstract implementations can be re-used (with increasing instance numbers). However, unlike classical protocols, Abstract compositions *allow entire sub-protocols to be changed on a 'view-change'* (i.e., during switching).

[6] The opposite however does not hold, since multiple views of a given BFT protocol can be captured within a single Abstract instance.

Byzantine clients. Clients that fail to comply with the switching mechanism (e.g., by inventing/forging an init history) cannot violate the Abstract specification. Indeed, to be considered valid, an init history of $next(i)$ must be previously returned by the preceding Abstract i as an abort history. To enforce this causality, in practice, our Abstract compositions (see Sec. 3 and Sec. 4) rely on unforgeable digital signatures to authenticate abort histories in the presence of potentially Byzantine clients. View-change mechanisms employed in existing BFT protocols [7, 20], have similar requirements: they exchange digitally signed messages. We further discuss Byzantine clients in Section 5.

3. Illustrating Abstract: AZyzzyva

We illustrate how Abstract significantly eases the design, implementation, and proof of BFT protocols with *AZyzzyva*. This is a full fledged BFT protocol that mimics Zyzzyva [20] in its "common case" (also called "best-case" i.e., when there are no link or server failures). In "other cases" we rely on *Backup*, an Abstract implementation with strong progress guarantees that can be implemented on top of *any* existing BFT protocol. In our implementation, we chose PBFT [7] for it has been extensively tested and proved correct. We chose to mimic Zyzzyva, for it is known to be efficient, yet very difficult to implement [11]. Using Abstract, we had to write, prove and test less than 24% of the Zyzzyva code to obtain *AZyzzyva*.

In the "common case", Zyzzyva executes the fast speculative path depicted in Figure 2. A client sends a request to a designated server, called *primary* (r_1 in Fig. 2). The primary appends a sequence number to the request and broadcasts the request to all replicas. Each replica speculatively executes the request and sends a reply to the client. All messages in the above sequence are authenticated using MACs rather than (more expensive) digital signatures. The client commits the request if it receives the same reply from all $3f+1$ replicas. Otherwise, Zyzzyva executes a second phase that aims at handling the case with link/server/client failures ("worst-case"). Roughly, this phase (that *AZyzzyva* avoids to mimic) consists of considerable modifications to PBFT [7], which arise from the "profound effects" [20], that the Zyzzyva "common-case" optimizations have on its "worst-case". The second phase is so complex that, as confessed by the authors themselves [11], it is not entirely implemented in the current Zyzzyva prototype. In fact, when this second phase is stressed, due to its complexity and the inherent bugs that it contains, the throughput of Zyzzyva drops to 0.

In the following, we describe how we build *AZyzzyva*, assess the qualitative benefit of using Abstract and discuss the performance of *AZyzzyva*.

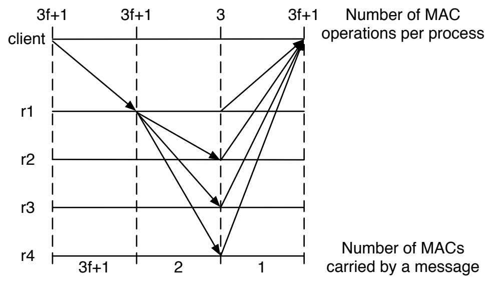
3f+1
3f+1
3
3f+1
Number of MAC operations per process
client
r1
r2
r3
r4
3f+1
2
1
Number of MACs carried by a message

Figure 2. Communication pattern of *ZLight*.

3.1 Protocol overview

Our goal when building *AZyzzyva* using Abstract is to show that we can completely *separate the concerns* of handling the "common-case" and the "worst-case". We thus use two different Abstract implementations: *ZLight* and *Backup*. Roughly, *ZLight* is a Abstract that guarantees progress in the Zyzzyva "common-case". On the other hand, *Backup* is an Abstract with strong progress: it guarantees to commit an exact certain number of requests k (k is itself configurable) before it starts aborting.

We then simply construct *AZyzzyva* such that every odd (resp., even) Abstract instance is *ZLight* (resp., *Backup*). *ZLight* is first executed. When it aborts, it switches to *Backup*, which commits the next k requests. *Backup* then aborts subsequent requests and switches to (a new instance of) *ZLight*, and so on.

Note that *ZLight* uses a lightweight checkpoint protocol (shared with *Aliph*'s *Quorum* and *Chain*, Sec. 4) triggered every 128 messages to truncate histories [17]. This protocol is very similar to the one used in [7, 20].

In the following, we briefly describe *ZLight* and *Backup*; full details and correctness proofs can be found in [17].

3.2 ZLight

ZLight implements Abstract with the following progress property: it commits requests when (a) there are no server or link failures, and (b) no client is Byzantine (simple client crash failures are tolerated). When this property holds, *ZLight* implements *Zyzzyva*'s "common-case" pattern (Fig. 2), described earlier. Outside the "common-case", when a client does not receive $3f + 1$ consistent replies, the client sends a PANIC message to replicas. Upon reception of this message, replicas stop executing requests and send back a signed message containing their history (replicas will now send the same abort message for all subsequent requests). When the client receives $2f + 1$ signed messages containing replica histories, it can generate an abort history and switch to *Backup*.

3.3 Backup

Backup is an Abstract implementation with a progress property that guarantees exactly $k \geq 1$ requests to be committed, where k is a generic parameter (we explain our configuration for k at the end of this section). We employ *Backup* in *AZyzzyva* (and *Aliph*) to ensure progress outside "common-cases" (e.g., under replica failures).

We implemented *Backup* as a very thin wrapper (around 600 lines of C++ code) that can be put around *any existing* BFT protocol. In our C/C++ implementations, *Backup* is implemented over PBFT [7], for PBFT is the most extensively tested BFT protocol and it is proven correct. Other existing BFT protocols that provide robust performance under failures, like Aardvark [11], are also very good candidates for the *Backup* basis (unfortunately, the code of Aardvark is not yet publicly available).

To implement *Backup*, we exploit the fact that any BFT can totally order submitted requests and implement any functionality on top of this total order. In our case, *Backup* is precisely this functionality. *Backup* works as follows: it ignores all the requests delivered by the underlying BFT protocol until it receives a request containing a valid init history, i.e. an unforgeable abort history generated by the preceding Abstract (*ZLight* in the case of *AZyzzyva*). At this point, *Backup* sets its state by executing all the requests contained in a valid init history it received. Then, it simply executes the first k requests ordered by BFT (neglecting subsequent init histories) and commits these requests. After committing the k^{th} request, *Backup* aborts all subsequent requests returning the signed sequence of executed requests as the abort history (replica digital signature functionality assumed here is readily present in all existing BFT protocols we know of).

Parameter k is generic and is an integral part of the *Backup* progress guarantees. Our default configuration increases k exponentially, with every new instance of *Backup*. This ensures the liveness of the composition, which might not be the case with, say, a fixed k in a corner case with very slow clients (see [17] for more details on this issue). More importantly, in the case of failures, we actually do want to have a *Backup* instance remaining active for long enough, since *Backup* is precisely targeted to handle failures. On the other hand, to reduce the impact of transient link failures, which can drive k to high values and thus confine clients to *Backup* for a long time after the transient failure disappears, we flatten the exponential curve for k to maintain $k = 1$ during some targeted outage time. In our implementation, we also periodically reset k. Dynamically adapting k to fit the system conditions is appealing but requires further studies and is out of the scope of this paper.

3.4 Qualitative assessment

In evaluating the effort of building *AZyzzyva*, we focus on the cost of *ZLight*. Indeed, *Backup*, for which the additional effort is small (around 600 lines of C++ code), can be reused for other BFT protocols in our framework. For instance, we use *Backup* in our *Aliph* protocol as well (Sec. 4).

Table 1 compares the number of pages of pseudo-code, proofs and lines of code of Zyzzyva and *ZLight*. [7] The code line comparison shows that to build *ZLight* we needed less than 24% of the *Zyzzyva* line count (14,339 lines).

Using the same syntax as the one used in the original Zyzzyva paper [20], *ZLight* requires approximately half a page of pseudo-code, its plain-english proof requires about 1 page [17]. In comparison, the pseudo-code of Zyzzyva (without checkpointing) requires 4.5 pages [21], making it about 9 times bigger than that of *ZLight*. Due to the complexity of Zyzzyva, the authors first presented a version using signatures and then explained how to modify it to use MACs. The correctness proof of the *Zyzzyva* signature version requires 4 (double-column) pages, whereas the proof for the MAC version is only sketched.

	Zyzzyva	*ZLight*
Pages of pseudo-code	4,5	0,5
Pages of proofs	> 4	1
Lines of code	14,339	3,358

Table 1. Complexity comparison of Zyzzyva and *ZLight*.

3.5 Performance evaluation

We have compared the performance of *AZyzzyva* and Zyzzyva in the "common-case", using the benchmarks described in Section 4.2. Not surprisingly, *AZyzzyva* and Zyzzyva have the exact same performance in this case. In this section, we do thus focus on the cost induced by our switching mechanism when the operating conditions are outside the common-case (and *ZLight* aborts a request). We could not compare against Zyzzyva. Indeed, as explained above, it has bugs in the second phase in charge of handling faults, which makes its impossible to evaluate the current prototype outside the "common-case".

To assess the switching cost, we perform the following experiments: we feed the request history of *ZLight* with r requests of size $1kB$. We then issue 10,000 successive requests. To isolate the cost of the switching mechanism, we do not execute the *ZLight* common case; the measured time comprises the time required (1) by the client to send a PANIC message to *ZLight* replicas, (2) by the replicas to generate and send a signed message containing their history, (3) by the client to invoke Backup with the abort/init history, and (4) by the (next) client to get the abort history from Backup and initialize the next *ZLight* instance. Note that we deactivate the functions in charge of updating the history of *ZLight*, so that we ensure that for each aborted request, the history contains r requests. We reproduced each experiment three times and observed a negligible variance.

Figure 3 shows the switching time (in ms) as a function of the history size when the number of tolerated faults equals 1. As mentioned above, *ZLight* uses a checkpointing mechanism triggered every 128 requests. To account for requests that might be received by servers while they are performing a checkpoint, we assume that the history size can grow up to 250 requests. We plot two different curves: one corresponds to the case when replicas do not miss any request. The other one corresponds to the case when replicas miss requests. More precisely, we assess the performance when 30% of the requests are absent from the history of at least one replica. Not surprisingly, we observe that the switching cost increases with the history size and that it is slightly higher in the case when replicas miss requests (as replicas need to fetch the requests they miss). Interestingly, we see that the switching cost is very reasonable. It ranges between $19.7ms$ and $29.2ms$, which is low provided faults are supposed to be rare in the environment for which Zyzzyva has been devised.

Furthermore, we observe that the switching cost grows faster than linearly. We argue that this is not an issue since the number of requests in histories is bounded by checkpointing. Finally, the switching cost could easily be higher in the case of a real application performing actual computations on requests that are reordered by the switching mechanism. However, it is important to notice that this extra-cost would also be present in Zyzzyva, induced by the request replay during view-changes.

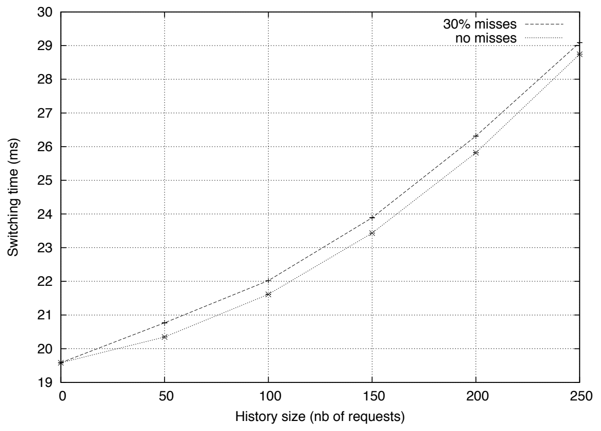

Figure 3. Switching time in function of history size and percentage of missing requests in replica histories.

[7] Needless to say, such metrics have to be taken with a grain of salt: (i) these protocols have been developed and proved by different researchers (note however that Zyzzyva and *ZLight* do use the same code base, inherited from PBFT [7]), and (ii) Zyzzyva does not fully implement the code required to handle faults. Yet, we believe these metrics provide a useful intuition of the difference in code and algorithmic complexity between Zyzzyva and *ZLight*. We also managed a library of contains cryptographic functions, networking code (to send/receive messages and manage sockets), and data structures (e.g. maps, sets) and ported Zyzzyva on this library (roughly 7,500 lines of code).

4. Putting Abstract to Work: Aliph

In this section, we demonstrate how we can build novel, very efficient BFT protocols, using Abstract. Our new protocol, called *Aliph*, achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. The development of *Aliph* consisted in building two new instances of Abstract, each requiring less than 25% of the code of state-of-the-art protocols, and reusing *Backup* (Sec. 3.3). In the following, we first describe *Aliph* and then we evaluate its performance.

4.1 Protocol overview

The characteristics of *Aliph* are summarized in Table 2, considering the metrics of [20]. In short, *Aliph* is the first optimally resilient protocol that achieves a latency of 2 one-way message delays when there is no contention. It is also the first protocol for which the number of MAC operations at the bottleneck replica tends to 1 (under high contention when batching of messages is enabled): 50% less than required by state-of-the-art protocols.

Aliph uses three Abstract implementations: *Backup* (introduced in Sec. 4.3), *Quorum* and *Chain* (both described below). A *Quorum* instance commits requests as long as there are no: (a) server/link failures, (b) client Byzantine failures, and (c) contention. *Quorum* implements a very simple communication pattern and gives *Aliph* the low latency flavor when its progress conditions are satisfied. On the other hand, *Chain* provides exactly the same progress guarantees as *ZLight* (Sec. 3.2), i.e., it commits requests as long as there are no server/link failures or Byzantine clients. *Chain* implements a pipeline pattern and, as we show below, allows *Aliph* to achieve better peak throughput than all existing protocols. Both *Quorum* and *Chain* share the panicking mechanism with *ZLight*, which is invoked by the client when it fails to commit the request.

Aliph uses the following static switching ordering to orchestrate its underlying protocols: *Quorum-Chain-Backup-Quorum-Chain-Backup−etc.* Initially, *Quorum* is active. As soon as it aborts (e.g., due to contention), it switches to *Chain*. *Chain* commits requests until it aborts (e.g., due to asynchrony). *Aliph* then switches to *Backup*, which executes k requests (see Sec. 3.3). When *Backup* commits k requests, it aborts, switches back to *Quorum*, and so on.

In the following, we first describe *Quorum* (Sec. 4.1.1) and *Chain* (Sec. 4.1.2) (full details and correctness proofs can be found in [17]). Then, we discuss some system-level optimizations of *Aliph* (Sec. 4.1.3).

4.1.1 Quorum

Quorum implements a very simple communication pattern (see Fig. 4); it requires only one round-trip of message exchange between a client and replicas to commit a request. Namely, the client sends the request to all replicas that speculatively execute it and send a reply to the client. As in *ZLight*, replies sent by replicas contain a digest of their history. The client checks that the histories sent by the $3f + 1$ replicas match. If that is not the case, or if the client does not receive $3f + 1$ replies, the client invokes a panicking mechanism. This is the same as in *ZLight* (Sec. 3.2): (i) the client sends a PANIC message to replicas, (ii) replicas stop executing requests on reception of a PANIC message, (iii) replicas send back a signed message containing their history. The client collects $2f + 1$ signed messages containing replica histories and generates an abort history. Note that, unlike *ZLight*, *Quorum* does not tolerate contention: concurrent requests can be executed in different orders by different replicas. This is not the case in *ZLight*, as requests are ordered by the primary.

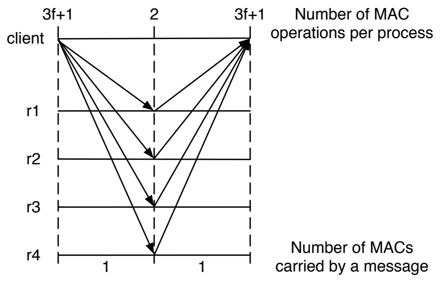
3f+1
2
3f+1
Number of MAC operations per process
client
r1
r2
r3
r4
1
1
Number of MACs carried by a message

Figure 4. Communication pattern of *Quorum*.

The implementation of *Quorum* is very simple. It requires 3200 lines of C code (including panicking and checkpoint libraries). *Quorum* makes *Aliph* the first BFT protocol to achieve a latency of 2 one-way message delays, while only requiring $3f + 1$ replicas (Q/U [1] has the same latency but requires $5f + 1$ replicas). Given its simplicity and efficiency, it might seem surprising not to have seen it published earlier. We believe that Abstract made that possible because we could focus on weaker (and hence easier to implement) Abstract specifications, without caring about (numerous) difficulties outside the *Quorum* "common-case".

4.1.2 Chain

Chain organizes replicas in a pipeline (see Fig. 5). All replicas know the fixed ordering of replica IDs (called *chain order*); the first (resp., last) replica is called the *head* (resp., the *tail*). Without loss of generality we assume an ascending ordering by replica IDs, where the head (resp., tail) is replica r_1 (resp., r_{3f+1}).

In *Chain*, a client invokes a request by sending it to the head, who assigns sequence numbers to requests. Then, each replica r_i forwards the message to its *successor* $\overrightarrow{r_i}$, where $\overrightarrow{r_i} = r_{i+1}$. The exception is the tail whose successor is the client: upon receiving the message, the tail sends the reply to the client. Similarly, replica r_i in *Chain* accepts a message only if sent by its predecessor $\overleftarrow{r_i}$, where $\overleftarrow{r_i} = r_{i-1}$; the exception is the head, which accepts requests only from the client.

	PBFT	Q/U	HQ	Zyzzyva	Aliph
Number of replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1**
Throughput (MAC ops at bottleneck server)	2+$\frac{8f}{b}$	2+4f	2+4f	2+$\frac{3f}{b}$	**1+$\frac{f+1}{b}$**
Latency (1-way messages in the critical path)	4	**2**	4	3	**2**

Table 2. Characteristics of state-of-the-art BFT protocols. Row 1 is the number of replicas. Row 2 is the throughput in terms of number of MAC operations at the bottleneck replica (assuming batches of b requests). Row 3 is the latency in terms of number of 1-way messages in the critical path. Bold entries denote protocols with the lowest known cost.

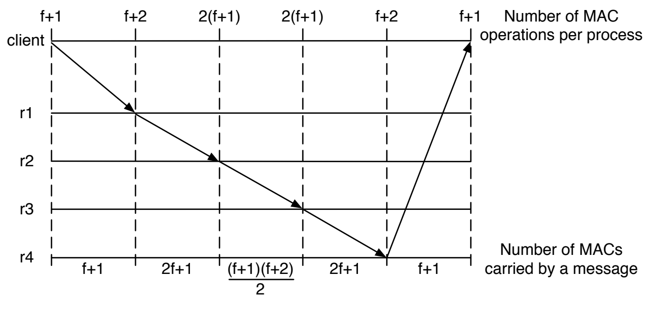
f+1
f+2
2(f+1)
2(f+1)
f+2
f+1
Number of MAC operations per process
client
r1
r2
r3
r4
f+1
2f+1
(f+1)(f+2)/2
2f+1
f+1
Number of MACs carried by a message

Figure 5. Communication pattern of *Chain*.

The behavior of *Chain*, as described so far, is very similar to the crash-tolerant protocol described in [29]. We tolerate Byzantine failures by ensuring: (1) that the content of a message is not modified by a malicious replica, (2) that no replica in the chain is bypassed, and (3) that the reply sent by the tail is correct. To provide those guarantees, our *Chain* relies on a novel authentication method we call *chain authenticators (CAs)*. CAs are lightweight MAC authenticators, requiring processes to generate (at most) $f + 1$ MACs (in contrast to $3f + 1$ in traditional authenticators). CAs guarantee that, if a client commits request req, every correct replica executed req. CAs, along with the inherent throughput advantages of a pipeline pattern, are key to *Chain*'s dramatic throughput improvements over other BFT protocols. We describe below how CAs are used in *Chain*.

Processes generate CAs in order to authenticate the messages they send. Each CA contains MACs for a set of processes called *successor* set. The successor set of clients consists of the $f + 1$ first replicas in the chain. The successor set of replica r_i depends on its position i: (a) for the first $2f$ replicas, the successor set comprises the next $f + 1$ replicas in the chain, whereas (b) for other replicas ($i > 2f$), the successor set comprises all subsequent replicas in the chain, as well as the client. Dually, when a process receives a message m it *verifies* m, i.e., it checks whether m contains a correct MAC from the processes it is in the successor set of. For instance, process p_1 verifies that the message contains a valid MAC from process p_0 and the client, whereas the client verifies that the reply it gets contains a valid MAC from the last $f+1$ replicas in the chain. Finally, to make sure that the reply sent by the tail is correct, f processes that precede the tail in the chain append a digest of the response to the message.

When the client receives a correct reply, it commits it. On the other hand, when the reply is not correct, or when it does not receive any reply (e.g., due to the Byzantine tail which discards the request), the client broadcasts a PANIC message to replicas. As in *ZLight* and *Quorum*, when replicas receive a PANIC message, they stop executing requests and send back a signed message containing their history. The client collects $2f + 1$ signed messages containing replica histories and generates an abort history.

Chain's implementation requires 3300 lines of code (with panic and checkpoint libraries). Moreover, it is the first protocol in which the number of MAC operations at the bottleneck replica tends to 1. This comes from the fact that, under contention, the head of the chain can batch requests. Head and tail do thus need to read (resp. write) a MAC from (resp. to) the client, and write (resp. read) $f + 1$ MACs for a batch of requests. Thus for a single request, head and tail perform $1+\frac{f+1}{b}$ MAC operations. Note that all other replicas process requests in batch, and have thus a lower number of MAC operations per request. State-of-the-art protocols [7, 20] do all require at least 2 MAC operations at the bottleneck server (with the same assumption on batching). The reason why this number tends to 1 in *Chain* can be intuitively explained by the fact that these are two distinct replicas that receive the request (the head) and send the reply (the tail).

4.1.3 Optimizations

When a *Chain* instance is executing in *Aliph*, it commits requests as long as there are no server or link failures. In the *Aliph* implementation we benchmark in the evaluation, we slightly modified the progress property of *Chain* so that it aborts requests as soon as replicas detect that there is no contention (i.e. there is only one active client since at least $2s$). Moreover, *Chain* replicas add an information in their abort history to specify that they aborted because of the lack of contention. We slightly modified *Backup*, so that in such case, it only executes one request and aborts. Consequently, *Aliph* switches to *Quorum*, which is very efficient when there is no contention. Finally, in *Chain* and *Quorum* we use the same checkpoint protocol as in *ZLight*.

4.2 Evaluation

This section evaluates the performance of *Aliph*. For lack of space, we focus on experiments without failures (of processes or links), since we compare to protocols that are known to perform well in the common-case — PBFT [7], Q/U [1] and Zyzzyva [20]. Assessment of the behavior of *Aliph* when faults occur can be found in [17].

We first study latency, throughput, and fault scalability using microbenchmarks [7, 20], varying the number of clients. In these microbenchmarks clients invoke requests in closed-loop, i.e., a client does not invoke a new request before it gets a reply for a previous one.[8] The benchmarks are denoted x/y, where x is the request payload size (in kB) and y is the reply payload size (in kB). We then proceed by studying the performance of *Aliph* under faults. Finally, we perform an experiment in which the input load dynamically varies.

We evaluate PBFT and Zyzzyva because the former is considered the "baseline" for practical BFT implementations, whereas the latter is considered state-of-the-art. Moreover, Zyzzyva systematically outperforms HQ [20]; hence, we do not evaluate HQ. Finally, we benchmark Q/U as it is known to provide better latency than Zyzzyva under certain condition. Note that Q/U requires $5f + 1$ servers, whereas other protocols we benchmark only require $3f + 1$ servers.

PBFT and Zyzzyva implement two optimizations: request batching by the primary, and client multicast (in which clients send requests directly to all the servers and the primary only sends ordering messages). All measurements of PBFT are performed with batching enabled as it always improves performance. This is not the case in Zyzzyva. Therefore, we assess Zyzzyva with or without batching depending on the experiment. As for the client multicast optimization, we show results for both configurations every time we observe an interesting behavior.

PBFT code base underlies both Zyzzyva and *Aliph*. To ensure that the comparison with Q/U is fair, we evaluate its simple best-case implementation described in [20].

We ran all our experiments on a cluster of 17 identical machines, each equipped with a 1.66GHz bi-processor and 2GB of RAM. Machines run the Linux 2.6.18 kernel and are connected using a Gigabit ethernet switch.

4.2.1 Latency

We first assess the latency in a system without contention, with a single client issuing requests. The results for all microbenchmarks (0/0, 0/4 and 4/0) are summarized in Table 3 demonstrating the latency improvement of *Aliph* over Q/U, PBFT, and Zyzzyva. We show results for a maximal number of server failures f ranging from 1 to 3. Our results show that *Aliph* consistently outperforms other protocols, since *Quorum* is active when there is no contention. These results confirm the theoretical expectations (see Table 2, Sec. 4.1). The results show that Q/U also achieves good latency with $f = 1$, as Q/U and *Quorum* use the same communication pattern. However, when f increases, performance of Q/U decreases significantly. The reason is that Q/U requires $5f + 1$ replicas and both clients and servers perform additional MAC computations compared to *Quorum*. Moreover, the significant improvement of *Aliph* over Zyzzyva (31% at $f = 1$) can be easily explained by the fact that Zyzzyva requires 3-one-way message delays in the common case, whereas *Aliph* (*Quorum*) only requires 2-one-way message delays.

4.2.2 Throughput

In this section, we present results obtained running the 0/0 and 4/0 microbenchmarks under contention (for lack of space, results for the 0/4 benchmark are presented in [17]). We do not present the results for Q/U since it is known to perform poorly under contention. Notice that in all the experiments presented in this section, *Chain* is active in *Aliph*. The reason is that, due to contention, there is always a point in time when a request sent to *Quorum* reaches replicas in a different order, which results in a switch to *Chain*. As there are no failures in the experiments presented in this section, *Chain* executes all the subsequent requests.

Our results show that *Aliph* consistently and significantly outperforms other protocols starting from a certain number of clients that depends on the benchmark. Below this threshold, Zyzzyva achieves higher throughput than other protocols.

0/0 benchmark. Figure 6 plots the throughput achieved in the 0/0 benchmark by various protocols when $f = 1$. We ran Zyzzyva with and without batching. For PBFT, we present only the results with batching, since they are substantially better than results without batching. We observe that Zyzzyva with batching performs better than PBFT, which itself achieves higher peak throughput than Zyzzyva without batching (this is consistent with the results of [20, 28]).

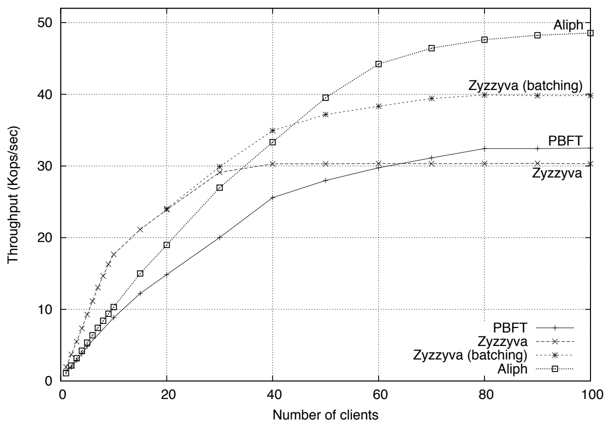

Figure 6. Throughput for the 0/0 benchmark (f=1).

Moreover, Figure 6 shows that with up to 40 clients, Zyzzyva achieves the best throughput. With more than 40 clients, *Aliph* starts to outperform Zyzzyva. The peak throughput achieved by *Aliph* is 21% higher than that of

[8] Although closed-loop microbenchmarks are not always representative of the behavior of real systems [27], we use these microbenchmarks to enable fair comparison with previously reported results, e.g. [7, 11, 20].

	0/0 benchmark			4/0 benchmark			0/4 benchmark		
	f=1	f=2	f=3	f=1	f=2	f=3	f=1	f=2	f=3
Q/U	8 %	14,9%	33,1%	6,5 %	13,6%	22,3%	4,7%	20,2%	26%
Zyzzyva	31,6 %	31,2%	34,5%	27,7 %	26,7%	15,6%	24,3%	26%	15,6%
PBFT	49,1%	48,8%	44,5%	36,6 %	38,4 %	26%	37,6%	38,2%	29%

Table 3. Latency improvement of *Aliph* for the 0/0, 4/0, and 0/4 benchmarks.

Zyzzyva. The reason is that *Aliph* executes *Chain*, which uses a pipeline pattern to disseminate requests. This pipeline pattern brings two benefits: reduced number of MAC operations at the bottleneck server, and better network usage: servers send/receive messages to/from a single server.

Nevertheless, the *Chain* is efficient only if its pipeline is fed, i.e. the link between any server and its successor in the chain must be saturated. There are two ways to feed the pipeline: using large messages (see the next benchmark), or a large number of small messages (this is the case of 0/0 benchmark). Moreover, as in the microbenchmarks clients invoke requests in closed-loop, it is necessary to have a large number of clients to issue a large number of requests. This explains why *Aliph* starts outperforming Zyzzyva only with more than 40 clients.

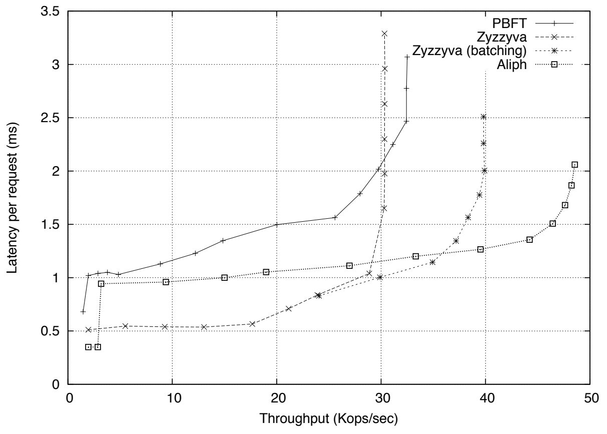

Figure 7. Response-time vs. throughput for the 0/0 benchmark (f=1).

Figure 7 plots the response-time of Zyzzyva (with and without batching), PBFT and *Aliph* as a function of the achieved throughput. We observe that *Aliph* achieves consistently lower response-time than PBFT. This stems from the fact that the message pattern of PBFT is a very complex one, which increases the response time and limits the throughput of PBFT. Moreover, up to the throughput of 37Kops/sec, *Aliph* has a slightly higher response-time than Zyzzyva. The reason is the pipeline pattern of *Chain* that results in a higher response time for low to medium throughput, which stays reasonable nevertheless. Moreover, *Aliph* scales better than Zyzzyva: from 37Kops/sec, it achieves lower response time, since the messages are processed faster due to the higher throughput ensured by *Chain*. Hence, messages spend less time in waiting queues. Finally, we observe that for very low throughput, *Aliph* has lower response time than Zyzzyva. This comes from the fact that *Aliph* uses *Quorum* when there is no contention, which significantly improves the response-time of the protocol.

4/0 benchmark. Figure 8 shows the results of *Aliph*, PBFT and Zyzzyva for the 4/0 microbenchmark with $f = 1$. Notice the logarithmic scale for the X axis, that we use to better highlight the behavior of various protocols with small numbers of clients. For PBFT and Zyzzyva, we plot curves both with and without client multicast optimization. The graph shows that with up to 3 clients, Zyzzyva outperforms other protocols. With more than 3 clients, *Aliph* significantly outperforms other protocols. Its peak throughput is about 360% higher than that of Zyzzyva. The reason why *Aliph* is very efficient under high load and when requests are large was explained earlier in the context of the 0/0 benchmark.

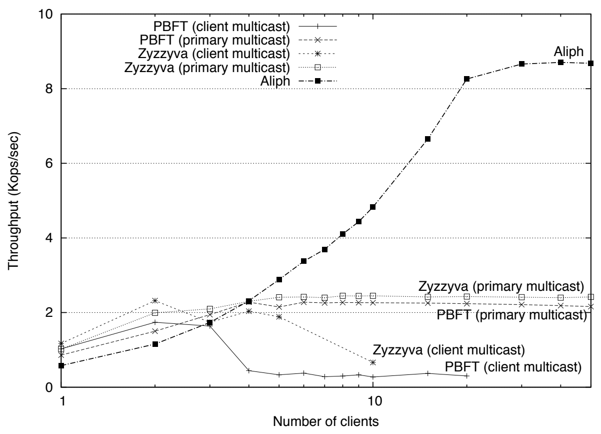

Figure 8. Throughput for the 4/0 benchmark (f=1).

Notice also the interesting drop in the performance of Zyzzyva and PBFT when client multicast optimization is used (Fig. 8). This is to be contrasted with the case when the primary forwards requests, where the performance of PBFT and Zyzzyva remain almost constant after the peak throughput has been reached. These results may seem surprising given that [7, 20] recommend to use the client multicast optimization when requests are large, in order to spare the primary of costly operations request forwarding. Nevertheless, these results can be explained by the fact that simultaneous multicasts of large messages by different clients result in collisions and buffer overflows, thus requiring many message retransmissions[9] (there is no flow control in UDP). This explains why enabling the concurrent client multicasts drastically reduces performance. On the other hand, when the primary forwards messages, there are fewer collisions, which explains the better performance we observe.

Impact of the request size. In this experiment we study how protocols are impacted by the size of requests. Figure 9 shows the peak throughput of *Aliph*, PBFT and Zyzzyva as a function of the request size for $f = 1$. To obtain the peak throughput of PBFT and Zyzzyva, we benchmarked both protocols with and without client multicast optimization and with different batching sizes for Zyzzyva. Interestingly, the behavior we observe is similar to that observed using simulations in [28]: differences between PBFT and Zyzzyva diminish with the increase in payload. Indeed, starting from 128B payloads, both protocols have almost identical performance. Figure 9 also shows that *Aliph* sustains high peak throughput with all message sizes, which is again the consequence of *Chain* being active under contention.

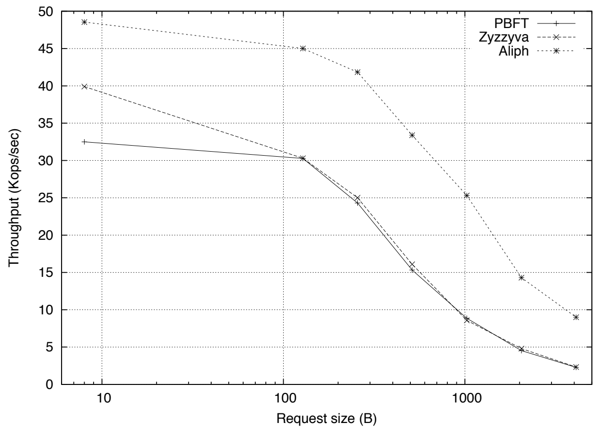

Figure 9. Peak throughput in function of request size (f=1).

Fault scalability. One important characteristic of BFT protocols is their behavior when the number of tolerated server failures f increases. Figure 10 depicts the performance of *Aliph* for the 4/0 benchmark when f varies between 1 and 3. We do not present results for PBFT and Zyzzyva as their peak throughput is known to suffer only a slight impact [20]. Figure 10 shows that this is also the case for *Aliph*. The peak throughput at $f = 3$ is only 3,5% lower than that achieved at $f = 1$. We also observe that the number of clients that *Aliph* requires to reach its peak throughput increases with f. This can be explained by the fact that *Aliph* uses *Chain* under contention. The length of the pipeline used in *Chain* increases with f. Hence, more clients are needed to feed the *Chain* and reach the peak throughput.

[9] Note that similar performance drops with large UDP packets have already been observed in the context of broadcast protocols. For instance, a recent study made by the authors of the JGroups toolkit showed that with 5K messages, their TCP stack achieves up to 5 times the throughput of their UDP stack, even if the latter includes some flow control mechanisms.

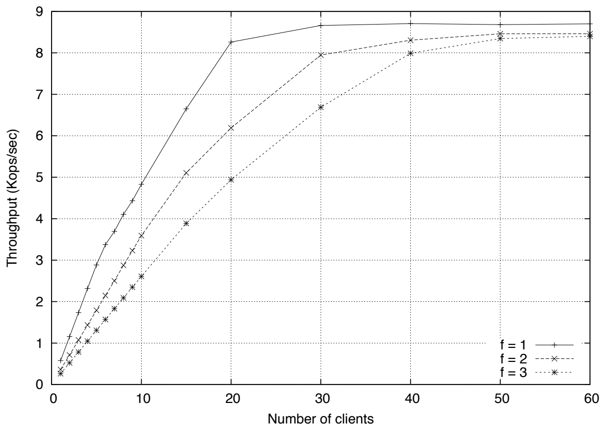

Figure 10. Impact of the number of tolerated failures f on the *Aliph* throughput.

4.2.3 Performance in case of faults

In this section, we assess the behavior of *Aliph* when a replica crashes. The experiment proceeds as follows. We consider 4 replicas, ($f = 1$) and one client that issues 15,000 requests. After the client sends 2,000 requests, we crash one replica, which recovers 10s later. Consequently, during 10s, only 3 replicas are active. We compare two strategies: in the first strategy, when *Aliph* switches to *Backup*, *Backup* always executes $k = 1$ request. In the second strategy, when *Aliph* switches to *Backup*, it executes $k = 2^i$, where i is the number of invocations of *Backup* since the beginning of the experiment. In principle, *Aliph* combines both strategies by exponentially increasing k, while maintaining the exponential curve initially very flat for reasons discussed in Section 3.3.

The behavior of *Aliph* with the first strategy is depicted in Figure 11. When only 3 replicas are active, *Quorum* and *Chain* cannot commit requests and *Aliph* switches to *Backup* for every single request. We depict on the Y axis both the throughput achieved by *Aliph* and the periods during which *Backup* is active. Not surprisingly, the throughput of *Aliph* is very low in this case.

Figure 12 shows the behavior of *Aliph* with the second strategy. The throughput is significantly higher because *Backup* is used to process an exponentially increasing number of requests. We can also observe that, although the 4 replicas are active at time $t = 11s$, *Aliph* switches back to *Quorum* only around time $t = 14s$. This is due to the fact that *Backup* had to process 8,192 requests before *Aliph* could switch. We point out that if the replica is down for a long time, *Aliph* will end up executing *Backup* for a very large number of requests. This means that, during a very long time period, the performance of *Aliph* will be that of *Backup*. We therefore periodically reset the number k of requests that *Backup* processes before aborting.

We would also like to remark that when many other kinds of faults occur (e.g. malformed requests, MAC attacks [11]),

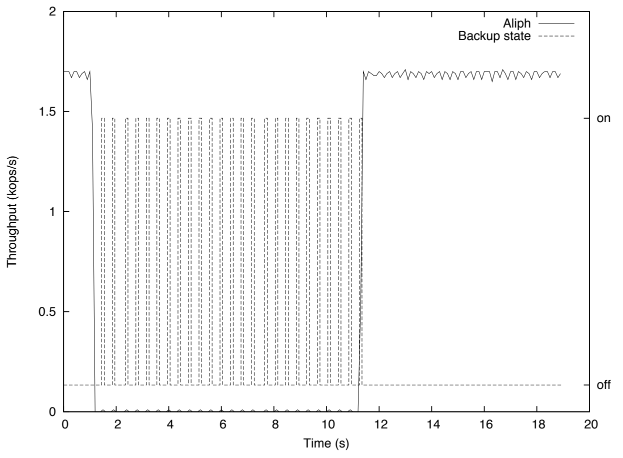

Figure 11. Throughput under faults, when system switches to *Backup* for one request.

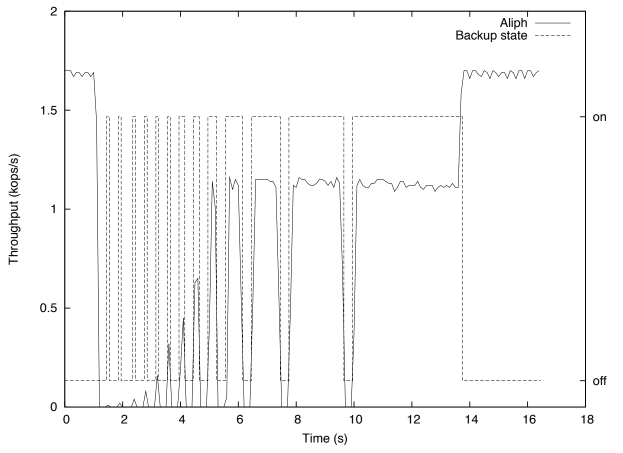

Figure 12. Throughput under faults, when system switches to *Backup* for 2^i requests.

a similar behavior would be observed. This is further discussed in Section 5.

4.2.4 Dynamic workload

Finally, we study the performance of *Aliph* under dynamic workload (i.e., fluctuating contention). We compare its performance to that achieved by Zyzzyva and by *Chain* alone. We do not present results for *Quorum* alone as it does not perform well under contention. The experiments consists in having 30 clients issuing requests of different sizes, namely, 0k, 0.5k, 1k, 2k, and 4k. Clients do not send requests all at the same time: the experiment starts with a single client issuing requests. Then we progressively increase the number of clients until it reaches 10. We then simulate a load spike with 30 clients simultaneously sending requests. Finally, the number of clients decreases, until there is only one client remaining in the system.

Figure 13 shows the performance of *Aliph*, Zyzzyva, and *Chain*. For each protocol, clients were invoking the same number of requests. Moreover, requests were invoked after the preceding clients have completed their bursts. First, we observe that *Aliph* is the most efficient protocol: it completes the experiment in $42s$, followed by Zyzzyva ($68.1s$), and *Chain* ($77.2s$). Up to time $t = 15.8s$, *Aliph* uses *Quorum*, which performs much better than Zyzzyva and *Chain*. Starting at $t = 15.8$, contention becomes too high for *Quorum*, which switches to *Chain*. At time $t = 31.8s$, there is only one client in the system. After $2s$ spent with only one client in the system, *Chain* in *Aliph* starts aborting requests due to the low load optimization (Sec. 4.1.3). Consequently, *Aliph* switches to *Backup* and then to *Quorum*. This explains the increase in throughput observed at time $t = 33.8s$. We also observe on the graph that *Chain* and *Aliph* are more efficient than Zyzzyva when there is a load spike: they achieve a peak throughput about three times higher than that of Zyzzyva. On the other hand, *Chain* and *Aliph* have slightly lower performance than Zyzzyva under medium load (i.e. from $16s$ to $26s$ on the *Aliph* curve). This suggests an interesting BFT protocol that would combine *Quorum*, Zyzzyva, *Chain* and *Backup*. However, this requires smart choices for dynamic switching, e.g., between Zyzzyva and *Chain*. We believe that building such a protocol is an interesting research topic.

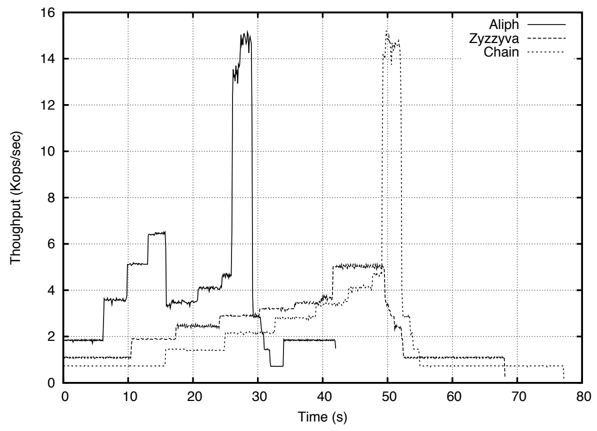

Figure 13. Throughput under dynamic workload.

5. Discussion

Byzantine clients. Our speculative Abstract implementations, *ZLight*, *Quorum* and *Chain*, do not ensure progress if clients are Byzantine. In fact it is very simple for a Byzantine client to force switching in our speculative Abstract implementations by simply sending PANIC messages to replicas (or by using big MAC attack [11]).[10] However, this is intentional. Namely, our lightweight Abstract instances focus on guaranteeing progress in "common" cases typically considered in literature and their implementations are optimized for such cases. Outside of these cases, e.g., in case of a Byzantine client, the goal of *AZyzzyva* and *Aliph* is to switch to

[10] It is important to notice here that Abstract specification does not require that clients initiate switching.

Backup and stay there as long as possible, if the Byzantine failure persists, to stabilize the system (cf. exponential increase of *Backup*'s parameter k, Sec. 3.3).

The throughput of *Backup* is hence crucial for the throughput of *AZyzzyva* and *Aliph* in presence of Byzantine clients. In this light, it would be interesting to replace PBFT in *Backup* with a more *robust* BFT protocol such as Aardvark [11] (this can be done modularly in our *Backup* construction) with the goal of improving *Aliph*'s performance under attacks (unfortunately, Aardvark code was not available). In addition, if clients' Byzantine failures are indeed a norm in a given system, new Abstract instances may be designed with a goal of optimizing efficiency under failures; these are outside of the scope of this paper.

Finally, Byzantine clients may undertake the attacks mentioned above in attempt to launch many speculative instances in order to simply subvert the performance of *AZyzzyva* and *Aliph*. To cope with this, we use a garbage collection mechanism; for lack of space, this is explained in [17].

Switching through replicas. Abstract switching is performed through clients, who receive an abort indication containing an unforgeable abort history. A possible alternative design is switching through replicas. While the full formal treatment of this alternative is beyond the scope of this paper, it is possible to generalize Abstract interface to accommodate this. In short, the idea would use one or more *arbiters*, processes (e.g., replicas) that would receive the full abort indication with abort histories instead of a client, who would receive a simple abort "signal", without abort histories.

Dynamic switching. *AZyzzyva* and *Aliph* use static switching, i.e, a predetermined order in which Abstract instances are switched among. Abstract specification envisions the possibility of dynamical choice of the next Abstract instance. This could lead to further performance improvements: e.g., a dynamic switching scheme could sense the current system conditions and switch to the seemingly most appropriate Abstract instance. It is important to stress that a prospective dynamic switching scheme would have to ensure that the id of the next instance remains the same across all abort indications of a given Abstract instance. While the dynamic switching is outside the scope of this paper, we highlight this as the first item on the Abstract research agenda.

Failure independence. To maintain the assumption of a threshold f of replica failures realistic, BFT systems need to ensure failure independence. An established technique used in ensuring failure independence is n-version programming which mandates a different BFT implementation for each replica, with the goal of reducing the probability of identical software faults across replicas. While Abstract does not alleviate the need for n-version programming, this may reveal less costly and more feasible due to the inherently reduced code sizes and complexities involved with Abstract implementations. In addition, abstractions like BASE [26], that enable reuse of off-the-shelf service implementations, can be used complementary to our approach.

6. Concluding remarks

In this paper, we introduced Abstract, a novel abstraction that simplifies the design, implementation, testing and verification of BFT protocols. In a sense, Abstract is an abortable state machine that enables to build a BFT protocol as the composition of as many (gracefully degrading) phases as desired, each with a "standard" interface. These phases are Abstract instances and each of them can be designed, implemented, tested and proved independently. This allows for an unpreceded flexibility in BFT protocol design that we illustrated with *Aliph*, a BFT protocol that combines three different phases.

The idea of aborting if "something goes wrong" is old. It underlies for instance the seminal two-phase commit protocol [16]: abort can be decided if there is a failure or some database server votes "no". The idea was also explored in the context of mutual exclusion: a process in the *entry section* can abort if it cannot enter the critical section [19]. Abortable consensus was proposed in [9] and [4]. In the first case, a process can abort if a majority of processes cannot be reached whereas, in the second, a process can abort if there is contention. The latter idea was generalized for arbitrary shared objects in [3] and then [2]. In [2], a process can abort and then query the object to seek whether the last query of the process was performed. This query can however abort if there is contention. Our notion of abortable state machine replication is different. First, the condition under which Abstract can abort is a generic parameter: it can express for instance contention, synchrony or failures. Second, in case of abort, Abstract returns (without any further query) what is needed for recovery in a Byzantine context; namely, an unforgeable history. This, in turn, can then be used to invoke another, possibly stronger, Abstract. This ability is key to the composability of Abstract instances.

Abstractions for Byzantine fault tolerance have been proposed in a weaker adversarial model that assumes trusted components that always remain beyond adversary control. Such architectures (see e.g., [10]) limit, by construction, the potential of the adversary to create deviations from the "common" case. Intuitively, protocols designed using these abstractions may handle Byzantine failures in a more efficient manner. While BFT protocols that we present in this paper do not assume such a weaker adversarial model, our approach is orthogonal: it does not prevent trusted components from being used in building Abstract instances.

Compositional approach to building BFT protocols is also not new. Several examples of protocols distinguishing an optimistic phase from a recovery one, were discussed in the survey of Pedone [25]. Such a bimodal protocol was also proposed in [22] in the context of Byzantine fault-tolerant atomic broadcast, where the optimistic phase consists of

Bracha broadcast [5] and the recovery phase uses a probabilistic multivalued Byzantine agreement protocol. Composition ideas were also used in the context of BFT state machine replication, e.g., in HQ [13]. HQ is similar to the construction of our *AZyzzyva* in a sense that its ("quorum-based") optimistic phase is followed by PBFT [7] as the recovery phase. However, these protocols lack the modularity and the flexibility of Abstract: e.g., it is not trivial to reuse the code and proofs of HQ or [22] while replacing optimistic and/or recovery phase by a different protocol, nor is it clear how to combine more than two phases. In contrast, we are the first to clearly separate the phases and encapsulate them within first class, well-specified, modules, that can each be designed, tested and proved independently.

Several directions can be interesting to explore, like using the concepts that underly Abstract in the context of Byzantine-resilient storage [18], or the possibilities for signature-free switching, to obtain practical BFT protocols that do not rely on signatures [12]. Moreover, we believe that an interesting research challenge lies in devising effective heuristics for dynamic switching among Abstract instances. While we described *Aliph* and showed that, albeit simple, it outperforms existing BFT protocols, *Aliph* is simply the starting point for Abstract. The idea of dynamic switching depending on the system conditions seems very promising.

Acknowledgements

We are grateful to Neeraj Suri for his help in improving the presentation of this paper, as well as Lorenzo Alvisi and Ramakrishna Kotla for their help with Zyzzyva. We also thank Hagit Attiya, Christian Cachin, Idit Keidar, Dejan Kostić, Marc Shapiro, and Jialin Zhang for their useful feedback.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[2] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[3] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC*, 2005.

[4] R. Boichat, P. Dutta, S. Frölund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News in Distributed Computing*, 34(1):47–67, 2003.

[5] G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *PODC*, 1984.

[6] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *PaCT*, 2001.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[8] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[9] W. Chen. Abortable consensus and its application to probabilistic atomic broadcast. Technical Report MSR-TR-2006-135, 2007.

[10] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, 2007.

[11] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[12] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.*, 49(1):82–96, 2006.

[13] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[14] R. De Prisco. *On building blocks for distributed systems*. PhD thesis, 2000. Massachusetts Institute of Technology.

[15] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *DSN*, 2006.

[16] J. Gray. Notes on database operating systems. In *Operating Systems — An Advanced Course*, number 66. 1978.

[17] R. Guerraoui, V. Quéma, and M. Vukolić. The next 700 BFT protocols. Technical Report LPD-REPORT-2008-008, EPFL.

[18] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP*, 2007.

[19] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, 2003.

[20] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, 2007.

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. Technical Report UTCS-TR-07-40, University of Texas at Austin, Austin, TX, USA, 2007.

[22] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *ICALP*, 2005.

[23] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[24] L. Lamport. Lower bounds for asynchronous consensus. In *FuDiCo*, 2003.

[25] F. Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[26] R. Rodrigues, M. Castro, and B. Liskov. Base: using abstraction to improve fault tolerance. In *SOSP*, 2001.

[27] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI*, pages 18–18, 2006.

[28] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *NSDI*, 2008.

[29] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.