

The Next 700 BFT Protocols

PIERRE-LOUIS AUBLIN, INSA Lyon
 RACHID GUERRAOU, EPFL
 NIKOLA KNEŽEVIĆ, IBM Research Zurich
 VIVIEN QUEMA, Grenoble INP
 MARKO VUKOLIĆ, Eurécom

We present Abstract (ABortable STate mACHine replicaTion), a new abstraction for designing and reconfiguring generalized replicated state machines that are, unlike traditional state machines, allowed to *abort* executing a client's request if "something goes wrong".

Abstract can be used to considerably simplify the incremental development of efficient Byzantine fault-tolerant state machine replication (BFT) protocols that are notorious for being difficult to develop. In short, we treat a BFT protocol as a composition of Abstract instances. Each instance is developed and analyzed independently, and optimized for specific system conditions. We illustrate the power of Abstract through several interesting examples.

We first show how Abstract can yield benefits of a state-of-the-art BFT protocol in a less painful and error-prone manner. Namely, we develop *AZyzyyva*, a new protocol that mimics the celebrated best-case behavior of *Zyzyyva* using less than 35% of the *Zyzyyva* code. To cover worst-case situations, our abstraction enables to use in *AZyzyyva* any existing BFT protocol.

We then present *Aliph*, a new BFT protocol that outperforms previous BFT protocols both in terms of latency (by up to 360%) and throughput (by up to 30%). Finally, we present *R-Aliph*, an implementation of *Aliph* that is *robust*, i.e., whose performance degrades gracefully in presence of Byzantine replicas and Byzantine clients.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems

General Terms: Design, Algorithms, Performance, Fault-tolerance

Additional Key Words and Phrases: Abstract, Byzantine Fault-Tolerance, composability, optimization, robustness

ACM Reference Format:

Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M., 2014. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 0, 0, Article 0 (2014), 46 pages.

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 0734-2071/2014/-ART0 \$15.00

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

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 [Toueg 1984; Lamport 2003]). These are called Byzantine Fault Tolerant SMR protocols, or simply *BFT* protocols,¹ e.g., PBFT, QU, HQ, Zyzzyva, Spinning, Prime, and Aardvark [Castro and Liskov 2002; Abd-El-Malek et al. 2005; Cowling et al. 2006; Kotla et al. 2010; Veronese et al. 2009; Amir et al. 2011; Clement et al. 2009]. The ultimate goal of the designer of a BFT protocol is not only to provide robustness to Byzantine faults and asynchrony, but also 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. According to our own experience, the performance differences among the protocols can be heavily impacted by the actual network, the size of 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 [Singh et al. 2008] which concluded for instance that *“PBFT [Castro and Liskov 2002] offers more predictable performance and scales better with payload size compared to Zyzzyva [Kotla et al. 2010]; in contrast, Zyzzyva offers greater absolute throughput in wider-area, lossy networks”*. In fact, besides all BFT protocols mentioned above, there are good reasons to believe that we could design new protocols outperforming all others under specific circumstances. We do indeed present examples of such protocols 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 sometimes algorithmically intuitive, is very painful. Moreover, the changes of the protocol needed to optimize for the “common” case have sometimes strong impacts on the parts of the protocol used in other cases (e.g., “view-change” in Zyzzyva). Finally, proving that a BFT protocol is correct is notoriously difficult: the only complete proof of a BFT protocol we knew of is that of PBFT and it involves 35 pages.² This difficulty, together with the impossibility of exhaustively testing distributed protocols [Chandra et al. 2007] would rather plead for never changing a protocol that was widely tested, e.g., PBFT.

We propose in this paper a way to have the cake and eat a big chunk of it. We present Abstract (ABortable STate mACHine replicaTion), a new abstraction for designing and reconfiguring generalized state machines that look like traditional state machines with one exception: they may sometimes *abort* a client’s request. Following the divide-and-conquer principle, we then use Abstract to build BFT protocols as

¹Whereas BFT generally refers to a broader set of Byzantine fault tolerant techniques, in this paper, that focuses on BFT state-machine replication (SMR), we use BFT as a shorthand for BFT SMR.

²It 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 Byzantine faults.

compositions of instances of our abstraction, each instance targeted and optimized for specific system conditions.

The progress condition under which an Abstract instance should not abort is a generic parameter.³ An extreme instance of Abstract is one that never aborts: this is exactly a traditional (replicated) state machine. 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 a request history that is used by the client (proxy) to “recover” by switching to another instance of Abstract, e.g., one with a stronger progress condition. This new instance will commit subsequent requests until it itself aborts. This paves the path to *composability* and flexibility of BFT protocol design using Abstract. Indeed, the composition of any two Abstract instances is *idempotent*, yielding 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 Byzantine fault-tolerant Abstract instances. This is usually way much simpler than tolerating Byzantine faults within a full-fledged, monolithic state machine replication protocol and allows for very effective schemes. A single Abstract instance can be crafted solely with its progress in mind, irrespective of other instances.
- Implementing a switching mechanism to glue an together different Abstract instances. This typically involves devising a library that exposes the abort subprotocol of the aborting Abstract instance to the next Abstract instance. In this way, the next Abstract instance is initialized using abort indications of the aborting Abstract instance.
- Ensuring that a request is not aborted by all Abstract instances. This can be made very simple by reusing, as a black-box, an existing BFT protocol within one of the instances, without indulging into complex modifications.

To demonstrate the benefits of Abstract, we present three BFT protocols, each we believe is interesting in its own right:

- (1) *AZyzyyva*, a protocol that illustrates the ability of using Abstract to significantly ease the development of BFT protocols. *AZyzyyva* is the composition of two Abstract instances: (i) *ZLight*, which mimics *Zyzyyva* [Kotla et al. 2010] 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 which was widely tested. The code line count to obtain *AZyzyyva* is, conservatively, around one third of that of *Zyzyyva*, while keeping the same optimizations (such as batching and read-only request handling). In some sense, had Abstract been identified several years ago, the designers of *Zyzyyva* would have had a much easier task devising a correct protocol exhibiting the performance they were seeking.
- (2) *Aliph*, a protocol that demonstrates the ability of using 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, besides the *Backup* instance used in *AZyzyyva* (to handle the cases with asynchrony/failures), 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 asynchrony/failures. *Quorum* has a very low-latency (like e.g., [Brasileiro

³Abstract can be viewed as a *virtual type*; each specification of the 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).

et al. 2001; Abd-El-Malek et al. 2005; Dobre and Suri 2006]) 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. It 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 [Kotla et al. 2010]. Interestingly, each of *Quorum* and *Chain* could be developed independently and required less than 35% of the code needed to develop state-of-the-art BFT protocols⁴.

- (3) *R-Aliph*, a protocol, based on *Aliph*, that achieves about the same performance as *Aliph* in the absence of faults, and that performs significantly better than *Aliph* under attack, i.e., when Byzantine replicas and Byzantine clients act maliciously in order to decrease the performance of the system. Similarly as *Aliph*, *R-Aliph* uses the *Quorum* and *Chain* protocols when there are no attacks. In order to achieve good performance under attack, *R-Aliph* monitors the progress of *Chain* and *Quorum*, implements various mechanisms to bound the time required to switch from one protocol to another protocol, and uses the Aardvark protocol [Clement et al. 2009] as *Backup* instance. Aardvark is a BFT protocol that has been specifically designed to sustain good performance despite attacks.

The rest of the paper is organized as follows. Section 2 describes the system model. Section 3 presents Abstract. Afterwards, we describe our new BFT protocols: *AZyzyva* in Section 4, *Aliph* in Section 5, and *R-Aliph* in Section 6. Section 7 discusses the related work. Finally, Section 8 discusses future work and concludes the paper. Correctness arguments and proofs of our implementations are postponed to Appendix A.

2. SYSTEM MODEL

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). However, we assume fair-loss links: a message sent an infinite number of times between two correct processes will be eventually received. Processes are Byzantine fault-prone; processes that do not fail are said to be correct. A process is called *benign* if it is correct or if it fails by simply crashing. In our algorithms, we assume that any number of clients and up to f out of $n = 3f + 1$ servers can be Byzantine. We assume a strong adversary that can coordinate faulty nodes; however, we assume that the adversary cannot violate cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), and signatures.

We further assume that during *synchronous* periods there are no link failures, i.e., that correct processes can communicate and process messages in a timely manner. More specifically, we assume that during synchronous periods: (i) any message m sent between two correct processes is delivered within a bounded delay Δ_c , (ii) any message received by a correct process is processed (including possible application-level execution) within a bounded delay Δ_p , and (iii) Δ_c and Δ_p are known to all correct processes. Intuitively, bounds Δ_c and Δ_p only serve to conveniently define timers that are assumed not to expire in the “common” case; our BFT protocol constructions from Abstract work correctly in the traditional partially synchronous model [Dwork et al. 1988]. We also denote $\Delta_p + \Delta_c$ as Δ .

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

Finally, we declare *contention* in an Abstract instance whenever there are two concurrent requests such that both requests are invoked but not yet committed/aborted.

3. ABSTRACT

In this section, we present our new approach for the design and reconfiguration of abortable state machine replication protocols. We start by an overview of Abstract, the new abstraction we propose. Then, we illustrate the way Abstract can be used to design BFT protocols. Finally, we provide the formal specification of Abstract and state and prove the Abstract composability theorem.

3.1. Overview

Abstract has been devised with Byzantine faults in mind, but it is not restricted to them. In fact, Abstract specification does not explicitly refer to any specific kind of faults.

Individually, each Abstract instance behaves just like a (replicated) state machine: it *commits* client's requests, returning state machine dependent replies to clients. Each reply is a function of a sequence of clients' requests called a *commit history* [van Renesse and Guerraoui 2010]. Like in traditional state machine replication [Schneider 1990], commit histories are totally ordered — we refer to this property of Abstract as *Commit Order*. However, an Abstract instance needs to commit a request only if certain *Progress* conditions are met; otherwise, an Abstract instance may *abort* a client's request. These *Progress* conditions, determined by the developer of the particular Abstract instance, might depend on the design goals and the environment in which a particular instance is to be deployed. Intuitively, designing a variant of a replicated state machine, in particular a Byzantine fault tolerant one, that needs to make progress only when certain conditions are met, is often considerably simpler than designing a full-fledged replicated state machine that makes progress under all system conditions.

A single Abstract instance is not particularly interesting on its own. That is why the Abstract framework comes with a mechanism and formalism for *reconfiguring* Abstract instances. Abstract reconfiguration resembles traditional state machine reconfiguration [Lamport et al. 2010; Birman et al. 2010], with the difference that its properties are tailored for reconfiguration of abortable state machines rather than full-fledged, traditional ones. In a sense, Abstract provides a rigorous framework of reconfiguration as an object, using shared-memory style.

More specifically, for the sake of reconfiguration, every Abstract instance has a unique identifier (called *instance number*) i ; this instance number abstracts away a set (and a number) of replicas implementing a given instance, the protocol used, possible protocol internals such as a protocol leader, etc. For example, in the BFT protocols that we present in this paper, Abstract instance number i captures the protocol used and its *Progress* conditions. Following the classical state machine reconfiguration approach [Lamport et al. 2010; Birman et al. 2010], Abstract reconfiguration can be divided into three steps: (i) stopping the current Abstract instance, (ii) choosing the next Abstract instance, and (iii) combining the request sequences (i.e., commit histories) of separate Abstract instances into a single sequence.

- (i) An Abstract instance is automatically stopped as soon as it aborts a single request. This is captured by the Abstract *Abort Order* property. Namely, when a client's request aborts, along the abort indication, the Abstract instance also returns a sequence of requests called *abort history*; The *Abort Order* property mandates that each *abort history* of a given Abstract instance i contains as its prefix every *commit history* of instance i (possibly along with some uncommitted requests). In a sense, Abstract *Abort Order* allows a certain request index to become a stopping-

index — any attempt to order requests higher than this index is guaranteed to abort, effectively causing a given Abstract instance to “stop”. As we will see later on, an *abort history* is used as an input to the next Abstract instance.

- (ii) Within an abort indication, an aborting Abstract instance i also returns an identifier of the next instance $next(i)$ — we say instance i *switches* to instance $next(i)$. Like in the reconfiguration of classical state machines [Lamport et al. 2010], the Abstract switching requires consensus on the next Abstract instance; hence, we require $next$ to be a function, i.e., to have $next(i)$ denote the same instance across all abort indications of instance i . Moreover, since we stop aborting instances and to avoid “switching loops”, we require monotonically increasing instance numbers, i.e., for every instance i , $next(i) > i$.

In the context of the protocols presented in this paper, we consider $next(i)$ to be a pre-determined function (e.g., known to servers implementing a given Abstract instance); we talk about *deterministic* or *static* switching. Concretely, in our protocols, we simply fix $next(i) = i + 1$. However, this is not required by our specification; $next(i)$ can be computed “on-the-fly” by an Abstract implementation (e.g., depending on the current workload, or possible failures or asynchrony). In this case, we talk about *dynamic switching*; this is out of the scope of this paper.

- (iii) Finally, “state transfer”, i.e., migration of histories between Abstract instances, reveals critical Abstract subtleties. In short, the client uses the abort history h of an aborting Abstract instance i to invoke $next(i)$; in the context of $next(i)$, h is called an *init* history. Roughly speaking, init histories are used to initialize instance $next(i)$, before $next(i)$ starts committing/aborting clients’ requests.

However, Abstract mandates *no explicit agreement* on the initial state (i.e., initial sequence of requests) of the next instance $next(i)$. More specifically, there is no agreement across abort histories of an aborting instance i . Abort histories need only contain commit histories as their prefix — there is no mutual order imposed on any two abort histories, which are free to contain arbitrary uncommitted requests at their tails.

Similarly, in the context of $next(i)$, no specific init history (recall that init history of instance $next(i)$ is an abort history of instance i) needs to be a prefix of commit/abort histories of $next(i)$. Abstract only requires *implicit agreement* on the initial sequence of requests of $next(i)$. This is captured by the Abstract *Init Order* property which mandates that, for any Abstract instance, a *longest common prefix* of init histories is a prefix of any commit/abort history. Intuitively, since every commit history of an aborting instance i is contained as a prefix in every abort history of i , we do not need a stronger property than *Init Order* to maintain total order of commit histories across Abstract instances.

Such weak, implicit, ordering of abort/init histories enables very efficient Abstract implementations. Since no explicit agreement is required neither across abort histories, nor on the “first” init history, Abstract implementations do not need to solve consensus outside *Progress* conditions. In fact, and as our Abstract implementations (*ZLight*, *Quorum* and *Chain*) exemplify, some Abstract specifications with “weak” *Progress* can be implemented in the asynchronous model despite Byzantine faults, circumventing the FLP impossibility result [Fischer et al. 1985].

With such a design, our Abstract framework has the following appealing properties that simplify the modular and incremental design of state machine replication, and BFT protocols in particular:

- (1) *Switching* between instances is *idempotent*: the composition of two Abstract instances yields another Abstract instance.

- (2) A correct implementation of an Abstract instance always preserves the safety of a state machine, i.e., the total order across committed requests [Schneider 1990; van Renesse and Guerraoui 2010]. This extends to any composition of Abstract instances.
- (3) A (replicated) state machine is nothing but a special Abstract instance — *one that never aborts*.

Consequently, the designer of a state machine replication (e.g., BFT) protocol has only to make sure that: (a) individual Abstract implementations are correct, *irrespective of each other*, and (b) the composition of the chosen instances is live: i.e., that every request will eventually be committed.

3.2. Illustration

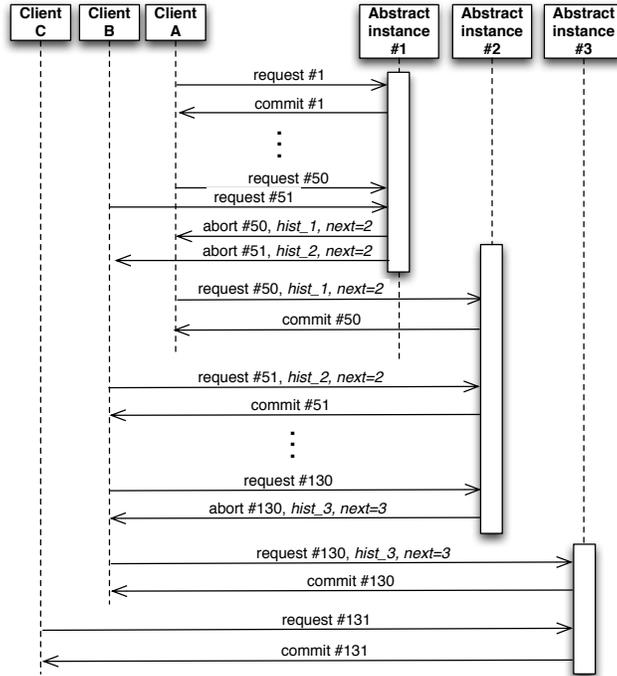


Fig. 1. Abstract operating principle.

Figure 1 depicts a possible run of a BFT protocol built using Abstract. As we have seen before, to preserve consistency, Abstract properties ensure that, at any point in time, only one Abstract instance may commit requests. We say that this instance is *active*. Client A starts sending requests to the first Abstract instance. The latter commits requests #1 to #49, aborts request #50 and stops committing further requests. Abstract appends to the abort indication an (unforgeable) history (*hist_1*) and the information about the next Abstract instance to be used (*next = 2*). Concurrently with request #50, client B sends request #51 to the first Abstract instance. Similarly to request #50, request #51 aborts and B obtains an (unforgeable) history *hist_2*. This abort indication contains the same, previously committed requests #1 to #49, just like history *hist_1*, but appended information on aborted requests differs.

Client A then 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 commits request #50. 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 commits request #51. The second abstract instance then commits 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 commits request #130. Finally, Client C, sends request #131 to the third instance, that commits 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.

3.3. Specification

In this section, we provide the specification of Abstract. We model every Abstract instance as a concurrent shared object, where every instance has a unique identifier $i \in \mathbb{N}$ (a natural number). The *type* (i.e., the set of possible values) of every Abstract instance is a *history*; a history $h \in H$, where $H = REQ^*$, is a (possibly empty) sequence of *requests* $req \in REQ$, where REQ is a set of all possible requests. More specifically, $REQ = C \times CMD \times \mathbb{N}$, where C is the set of clients’ ids, CMD is the set of all possible state machine commands, whereas the third element (a natural number) in a request is called *request identifier*.

Abstract i exports one operation: $Invoke_i(req, h_I)$, where $req \in REQ$ is a client’s request, and $h_I \in H$, an (optional) sequence of requests called *init history*; we say the client *invokes* request req (with init history h_I). By convention, when $i = 1$, the invocation never contains an init history.

On the other hand, Abstract instance i returns one of the possible two *indications* to the client invoking req (with h_I):

- (1) $Commit_i(req, rep(h_{req}))$, where $h_{req} \in H$ is a sequence of requests called *commit history* (of i) that contains req . Here, $rep(h_{req})$ represents the output function of the (replicated) state machine; basically, $rep(h_{req})$ represents the *replies* that the state machine outputs to clients.
- (2) $Abort_i(req, h_A, next(i))$, where $h_A \in H$ is a sequence of requests called *abort history* (of i) and $next$ is a function that returns an integer, where $next(i) > i$.

We say that a client *commits* or *aborts* a request req , respectively. In the case of an abort, we also say that instance Abstract i *switches* to instance $next(i)$. Intuitively, and as detailed below, abort histories of Abstract instance i are prefixed by commit histories of i and are used as init histories in instance $next(i)$.

We model an execution of a concurrent system composed of the set of Abstract instances and clients using *traces*.⁵ A trace is a finite sequence of Abstract invocation and indication events. A subtrace of a trace T is a subsequence of the events of T . An indication *matches* an invocation if their Abstract instance, request and the client ids are the same. A (sub)trace T is *sequential* if it starts with an invocation event and each invocation (except possibly the last) is immediately followed by a matching indication. An *instance subtrace*, denoted $T|i$, of a trace T is the subsequence of all events in T pertaining to Abstract instance i . Similarly, a *client subtrace*, denoted $T|c$, of a trace T is the subsequence of all events in T pertaining to client c . Client c is *well-formed* in trace T if client subtrace $T|c$ of T is sequential and all requests invoked by c in T

⁵Execution traces are sometimes also called execution histories (see e.g., [Herlihy and Wing 1990]). We use the term traces to avoid the possible confusion with Abstract commit/abort/init histories.

are unique (i.e., given two requests $req_0 = \langle c, cmd_0, rid_0 \rangle$ and $req_1 = \langle c, cmd_1, rid_1 \rangle$, we have $cmd_0 \neq cmd_1$ or $rid_0 \neq rid_1$). We assume all correct clients to be well-formed in all traces.

Given trace T and an invocation/indication event $ev \in T$, a history h is called a *valid init history* (VIH) for Abstract instance i at event ev , if and only if ev follows some indication $Abort_j(*, h, i)$ in T , for some Abstract instance j (i.e., such that h is an abort history of j and $next(j) = i$). Furthermore, req is called a *valid init request* (VIR) for Abstract instance i if and only if req is invoked with a VIH for i at invocation of req .

Finally, we define requests *valid* for Abstract instance i . In the special case where $i = 1$, any invoked request is valid. In case $i > 1$, valid requests are: (i) VIR requests for i and (ii) informally, the requests invoked after instance i is “initialized”, i.e., after i commits/aborts some VIR request. More formally, given trace T , if the invocation of req follows an indication of a request req' in the instance subtrace $T|i$ of T , where req' is a VIR for i , then req is valid for i .

In addition, for every instance i , if client c is Byzantine, then every request in $\{c\} \times CMD \times \mathbb{N}$ is valid for i .

With the above definitions, we are ready to state the properties of Abstract instance i (parameterized by a predicate P_i that reflects progress). In the following, “prefix” refers to a non-strict prefix.

- (1) (*Validity*) For any commit/abort event ev and the corresponding commit/abort history h of i , no request appears twice in h and every request in h is a valid request for i , or an element of a valid init history for i at ev .
- (2) (*Termination*) If a correct client c invokes a valid request req , then c eventually commits or aborts req (i.e., i eventually returns a matching indication).
- (3) (*Progress*) If some predicate P_i holds, a correct client never aborts a request.
- (4) (*Init Order*) For any commit/abort event ev and the corresponding commit/abort history h of i , the longest common prefix of all valid init histories for i at ev is a prefix of h .
- (5) (*Commit Order*) Let h_{req} and $h_{req'}$ be any two commit histories of i : either h_{req} is a prefix of $h_{req'}$ or vice versa.
- (6) (*Abort Order*) Every commit history of i is a prefix of every abort history of i .

It is important to note that Abstract is a strict generalization of a state machine. Namely, a state machine is precisely an Abstract instance (with id $i = 1$) that never aborts. In this case, the *Abort Order* and *Init Order* properties become irrelevant.

3.4. Abstract composability

The key invariant in the Abstract framework is *idempotence*: a composition of any two Abstract instances is, itself, an Abstract instance. Two Abstract instances are composed by feeding an abort history of a given instance i to instance $next(i)$ as the latter’s init history, i.e., by having a client that receives $Abort_i(req, h, next(i))$ invoke $Invoke_{next(i)}(req, h)$. The abort/init history h is included in a client’s c invocation of $next(i)$ only once, with the first invocation of instance $next(i)$ by c . Intuitively, the composed instance can typically have a different *Progress* property than the original two, but is never “weaker” than any of them.

More precisely, consider two Abstract instances: instance 1 that switches to i (denoted, for clarity, by $1 \rightarrow i$) and instance i , that switches to j (denoted by $i \rightarrow j$). Using these two instances, we can implement a single Abstract instance that inherits instance number 1 and switches to j (denoted by $1 \rightarrow j$). This implementation is a simple client-side protocol that we call Abstract composition protocol (ACP), which proceeds as follows.

Initially, with each invocation of the implemented instance $1 \rightarrow j$ ($Invoke_{1 \rightarrow j}(req)$) a client simply invokes instance $1 \rightarrow i$ ($Invoke_{1 \rightarrow i}(req)$). If a client receives $Commit_{1 \rightarrow i}(req, h_{req})$, it outputs $Commit_{1 \rightarrow j}(req, h_{req})$. If, however, a client in ACP receives $Abort_{1 \rightarrow i}(req, h, i)$ for the first time, it immediately feeds h to instance $i \rightarrow j$ with $Invoke_{i \rightarrow j}(req, h)$, without “exposing” the *Abort* indication. From this point on, for every following invocation of the implemented instance $1 \rightarrow j$ ($Invoke_{1 \rightarrow j}(req)$), a client simply invokes $Invoke_{i \rightarrow j}(req)$ and never invokes again $1 \rightarrow i$. Finally, for every received $Commit_{i \rightarrow j}(req, h_{req})$, a client outputs $Commit_{1 \rightarrow j}(req, h_{req})$ and for every $Abort_{i \rightarrow j}(req, h_A, j)$, a client outputs $Abort_{1 \rightarrow j}(req, h_A, j)$.

Abstract composability is captured by the following theorem:

THEOREM 3.1. [*Abstract composability theorem*] *Given Abstract instance 1 that switches to i and instance i that switches to j, the ACP protocol implements Abstract instance 1 that switches to j.*

In the following, we prove Theorem 3.1, which, by induction, extends to Abstract compositions of arbitrary length. We have also specified Abstract and ACP in TLA+/PlusCal [Lampert 2009] and model-checked the Abstract composability theorem using the TLC model checker. The details are available in the technical report [Guerraoui et al. 2008].

PROOF. In the following, we prove the Abstract properties of instance $1 \rightarrow j$. Notice that Init Order is not relevant for instances with instance number 1, including $1 \rightarrow j$. We, however, use Init Order of instance $i \rightarrow j$ to prove the other properties of instance $1 \rightarrow j$. We denote the set of commit (resp., abort) histories of Abstract instance x by CH_x (resp., AH_x).

We first prove that individual commit/abort indications of Abstract instance $1 \rightarrow j$ conform to the specification. For commit indications, it is straightforward to see from ACP that $CH_{1 \rightarrow j} = CH_{1 \rightarrow i} \cup CH_{i \rightarrow j}$. Since $CH_{1 \rightarrow i} \subset H$ and $CH_{i \rightarrow j} \subset H$ (by specifications of $1 \rightarrow i$ and $i \rightarrow j$), this implies that for every request req committed by instance $1 \rightarrow j$, its commit history $h_{req} \in CH_{1 \rightarrow j}$ is in H , such that $req \in h_{req}$. Similarly, it is easy to see that $AH_{1 \rightarrow j} = AH_{i \rightarrow j} \subset H$. Moreover, recall that Abstract instance numbers should monotonically increase; in this case, $1 < j$ follows directly from $1 < i$ and $i < j$, by specifications of Abstract instances $1 \rightarrow i$ and $i \rightarrow j$.

To prove Validity, observe that no request appears twice in any commit/abort history in $CH_{1 \rightarrow j} \cup AH_{1 \rightarrow j}$, since, by Validity of $1 \rightarrow i$ and $i \rightarrow j$, no request appears twice in $CH_{1 \rightarrow i} \cup CH_{i \rightarrow j} \cup AH_{i \rightarrow j}$. Moreover, every request in every commit/abort history of $1 \rightarrow j$ is an invoked request or a request by a Byzantine client — since the id of Abstract instance $1 \rightarrow j$ is equal to 1, all such requests are by definition valid.

Termination also follows directly from Termination of instances $1 \rightarrow i$ and $i \rightarrow j$. Notice that a request req that a correct client c executing ACP invokes on $i \rightarrow j$ without an init history (i.e., such that req is not a VIR for $i \rightarrow j$) is indeed valid for $i \rightarrow j$ (and hence the invocation of req terminates); namely, before c invokes req , c already received an indication for its VIR invocation of $i \rightarrow j$. Indeed, by ACP, the first invocation of $i \rightarrow j$ is a VIR invocation that contains a valid init history, and, by well-formedness of c , c does not invoke req before it receives the indication matching its first invocation.

To prove Commit Order, we focus on the case where req and req' are originally committed by *different* instances, i.e., we focus on showing that for any $h_{req} \in CH_{1 \rightarrow i}$ and $h_{req'} \in CH_{i \rightarrow j}$, h_{req} is a prefix of $h_{req'}$ — other cases directly follow from the Commit Order properties of instances $1 \rightarrow i$ and $i \rightarrow j$.

By definition, every VIH for instance $i \rightarrow j$ is an abort history of instance $1 \rightarrow i$. By Abort Order of $1 \rightarrow i$, every commit history of $1 \rightarrow i$, including h_{req} , is a prefix of any abort history of $1 \rightarrow i$. Hence, h_{req} is a prefix of every VIH for instance $i \rightarrow j$. By Init

Order of $i \rightarrow j$, the longest common prefix of VIHs of instance $i \rightarrow j$ is a prefix of all histories in $CH_{i \rightarrow j} \cup AH_{i \rightarrow j}$, including $h_{req'}$. This implies that $h_{req'}$ is a prefix of h_{req} .

The proof of Abort Order follows the same reasoning as the above proof of Commit Order.

Finally, we show that the Progress of $1 \rightarrow j$ holds for some predicate $P_{1 \rightarrow j}$, which, intuitively, is not “weaker” than any of the respective predicates $P_{1 \rightarrow i}$ and $P_{i \rightarrow j}$. We do not attempt, however, to express $P_{1 \rightarrow j}$ in terms of $P_{1 \rightarrow i}$ and $P_{i \rightarrow j}$.

If $P_{1 \rightarrow i}$ holds during an entire execution, clearly $1 \rightarrow j$ never aborts at a correct client as $1 \rightarrow i$ never aborts. On the other hand, if $P_{i \rightarrow j}$ holds during an entire execution, instance $1 \rightarrow i$ might abort the request; however, a correct client executing ACP does not output these abort indications, but immediately invokes instance $i \rightarrow j$. On the other hand, since $P_{i \rightarrow j}$ holds, instance $i \rightarrow j$ never aborts a correct client’s request. \square

To summarize, ACP involves Abstract switching through clients, who receive an abort indication containing an abort history and invoke the next instance. This approach allows for a streamlined specification of Abstract that does not reason about any specific process beyond clients. Indeed, in the Abstract specification we did not have to mention replicas implementing Abstract, nor any other process apart from clients. This shared-memory style of Abstract is intentional. In the remainder of this section we briefly discuss some of the aspects of Abstract composability and its proof.

Alternatives to switching through clients. In practice, due to e.g., performance or security concerns, a need may arise for switching through other processes in the system, beyond clients. For example, it might be useful to switch through a replica implementing Abstract, e.g., to save on bandwidth, or through a dedicated *reconfigurator* process to, e.g., (logically) centralize management. One way of achieving this would be to generalize the Abstract specification to accommodate for this. Notice here that our proof of Theorem 3.1 relies on clients *only* for Progress and Termination, to ensure that the state is actually transferred to the next Abstract instance. Instead, for example, the Abstract specification could be generalized so that one or more processes, reconfigurators, receive the full abort indication with abort histories instead of a client. Formalizations of such generalizations of our specification are, however, out of the scope of this paper.

That said, it is perfectly possible to achieve *switching through replicas* and still reap benefits of the composability proofs we present here, without introducing new processes to the Abstract specification. To this end, it is sufficient to allow a replica implementing Abstract to act as a client of Abstract, by invoking a special *noop* request that does not modify the state of the replicated state machine. That way, the replica can perform switching itself.⁶ This can be further complemented by disallowing *application clients* from transferring the state themselves. Such switching through replicas is illustrated in *R-Aliph*, our robust BFT protocol that we present in Section 6.3. Before that, we also discuss an important state transfer performance optimization used in all our Abstract implementations in Section 4.4.

Byzantine clients. Abstract composability theorem (ACT, Theorem 3.1) holds despite Byzantine clients. Whereas Byzantine clients are not very prominent in our proof of the ACT, notice that this is only so because ACT relies on the composition of implementations of individual Abstract instances $1 \rightarrow i$ and $i \rightarrow j$ that are assumed to be correct despite Byzantine clients. For example, a correct implementation of instance

⁶Of course, in general, depending on Progress of a specific Abstract instance i , up to $f + 1$ of such replica *noop* invocations might be required to actually perform switching (e.g., if instance i guarantees Progress despite f Byzantine replicas).

$i \rightarrow j$ must guarantee that a Byzantine client cannot successfully forge an init history, i.e., correct implementations of $1 \rightarrow i$ and $i \rightarrow j$ must ensure that any valid init history h for $i \rightarrow j$ was indeed previously output as an abort history of $1 \rightarrow i$. How specific Abstract implementations achieve this causal dependency has no effect on the correctness of ACT; however, in the rest of this paper, we will demonstrate how such correct implementations can be achieved. For example, our implementations use unforgeable digital signatures to enforce causality between abort histories and init histories.

4. SIMPLE ILLUSTRATION: AZYZZYVA

In this section, we illustrate how Abstract significantly eases the design, implementation, and proof of BFT protocols. We describe *AZyzyyva*, a full fledged BFT protocol that mimics *Zyzyyva* [Kotla et al. 2010] in its “common case” (i.e., when there are no link or server failures). In “other cases”, *AZyzyyva* relies on *Backup*, an Abstract implementation with strong progress guarantees that can be implemented on top of *any* existing BFT protocol. We chose to mimic *Zyzyyva*, for it is known to be efficient, yet very difficult to implement [Clement et al. 2009]. Using Abstract, we had to write and test less than 30% of the *Zyzyyva* code to obtain *AZyzyyva*. Abstract also considerably simplified the proof of *AZyzyyva* compared to that of *Zyzyyva*, due to the composability of Abstract instances and the straightforward (and reusable) proof of the *Backup* module.

We start this section with an overview of *AZyzyyva*. We then describe the Abstract instances it relies on. Finally, we provide a qualitative assessment, as well as a performance evaluation of *AZyzyyva*.

4.1. Protocol overview

As mentioned before, *AZyzyyva* is a BFT protocol that mimics *Zyzyyva* [Kotla et al. 2010]. *Zyzyyva* requires $3f + 1$ replicas. It works as follows. In the “common case”, *Zyzyyva* 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.

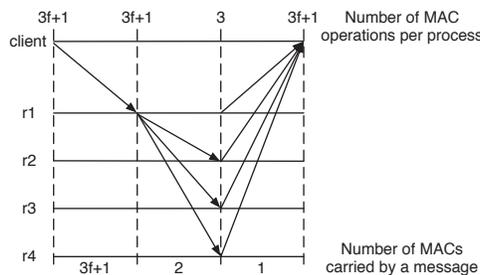


Fig. 2. Communication pattern of *ZLight* (fast path of *Zyzyyva*).

When a client does not receive the same reply from all $3f + 1$ replicas, *Zyzyyva* executes a second phase. This second phase aims thus at handling the case with link/server/client failures (“worst-case”). Roughly, this phase (that *AZyzyyva* avoids to mimic) consists of considerable modifications to PBFT [Castro and Liskov 2002], which

arise from the “profound effects” [Kotla et al. 2010], that the Zyzyva “common-case” optimizations have on PBFT’s “worst-case”.

Our goal when building *AZyzyva* using Abstract is to show that we can completely separate the concerns of handling the “common-case” and the “worst-case”. *AZyzyva* uses two different Abstract implementations: *ZLight* and *Backup*. Roughly, *ZLight* is an Abstract that guarantees progress in the Zyzyva “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. *AZyzyva* works as follows: every odd (resp., even) Abstract instance is *ZLight* (resp., *Backup*). This means that *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.

In the following, we describe *ZLight* and *Backup*. Then, we assess the qualitative benefit of using Abstract. Finally, we discuss the performance of *AZyzyva*.

4.2. ZLight

ZLight implements Abstract with the following progress property which reflects Zyzyva’s “common case”: it commits requests when (a) there are no server or link failures, and (b) no client is Byzantine (crash failures are tolerated). When this property holds, *ZLight* implements Zyzyva’s “common-case” pattern (Fig. 2), described earlier. Outside the “common-case”, i.e., when a client does not receive $3f + 1$ consistent replies, *ZLight* performs additional steps, depicted in Figure 3 (see the last two communication steps). The client sends a PANIC message to replicas. Upon reception of this message, replicas stop executing requests and send back a signed abort 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. It will then use this abort history to switch to *Backup*.

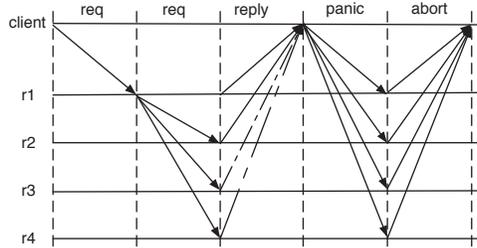


Fig. 3. Communication pattern of *ZLight* outside the “common-case” (the three first rounds correspond to the “common-case”).

In the remainder of this section, we give the pseudo-code of *ZLight*. We first present the code executed in the common-case. For ease of presentation, we do not mention init histories. We do then present the pseudo-code executed when *ZLight* aborts. Follows a presentation of the pseudo-code executed to initialize a new *ZLight* instance. Finally, we conclude by presenting the pseudo-code of a checkpointing protocol that is used in *ZLight* to garbage collect some data stored by replicas.

4.2.1. Common case. This section describes the pseudo-code executed by *ZLight* in the common-case. A message m sent by a process p to a process q and authenticated with a MAC is denoted by $\langle m \rangle_{\mu_{p,q}}$. A process p can use vectors of MACs (called authenticators [Castro and Liskov 2002]) to simultaneously authenticate a message m for multiple

recipients belonging to a set S ; we denote such a message, which contains $\langle m \rangle_{\mu_{p,q}}$, for every $q \in S$, by $\langle m \rangle_{\alpha_{p,S}}$. In addition, we denote the digest of a message m by $D(m)$, and $\langle m \rangle_{\sigma_p}$ denotes a message that contains $D(m)$ signed by the private key of process p and the message m . Notations for message fields and client/replica local variables is shown in Figure 4. To help distinguish clients' requests for the same command o , we assume that client c calls $Invoke_i(req)$, where $req = \langle c, o, t_c \rangle$ and where t_c is a unique, monotonically increasing client's timestamp. A replica r_j logs req by appending it to its local history, denoted LH_j . A replica r_j executes req by applying it to the state of the replicated state machine and by calculating an application level reply (i.e., the reply of a replicated state machine) for the client.

Σ - the set of all replicas
 i - current Abstract id
 c/r_j - client (resp., replica) ID
 t_c - local timestamp at client c
 $t_j[c]$ - the highest t_c seen by replica r_j
 o - command invoked by the client
 LH_j - a local history at replica r_j
 $reply_j$ - application reply (which is a function of LH_j)
 sn_j - sequence number at replica r_j

Fig. 4. Message fields and local variables.

We detail below the various steps that are executed when a client invokes a request. For the sake of brevity, we omit to mention that upon receiving a message m , a process p first checks that m has a valid authenticator, and does not process the message if that is not the case.

Step Z1. On $Invoke_i(req)$, client c sends a message $m' = \langle \text{REQ}, req, i \rangle_{\alpha_{c,\Sigma}}$ to the primary (say r_1) and triggers timer T set to 3Δ .

Step Z2. The primary r_1 on receiving $m' = \langle \text{REQ}, req, i \rangle_{\alpha_{c,\Sigma}}$, if $req.t_c$ is higher than $t_1[c]$, then (i) it updates $t_1[c]$ to $req.t_c$, (ii) increments sn_1 , and (iii) sends $\langle \langle \text{ORDER}, req, i, sn \rangle_{\mu_{r_1,r_j}}, MAC_j \rangle$ to every replica r_j , where MAC_j is the MAC entry for r_j in the client's authenticator for m' .

Step Z3. Replica r_j on receiving (from primary r_1) $\langle \langle \text{ORDER}, req, i, sn' \rangle_{\mu_{r_1,r_j}}, MAC_j \rangle$, if (i) MAC_j authenticates req and i , (ii) $sn' = sn_j + 1$, and (iii) $t_j[c] < req.t_c$, then it (i) updates sn_j to sn' and $t_j[req.c]$ to $req.t_c$, (ii) logs and then executes req , and (iii) sends $\langle \text{RESP}, reply_j, D(LH_j), i, req.t_c, r_j \rangle_{\mu_{r_j,c}}$ to c .⁷ If MAC_j verification fails, r_j stops executing Step Z3 in instance i .

Step Z4. If client c receives $3f + 1$ $\langle \text{RESP}, reply, LHDigest, i, req.t_c, * \rangle_{\mu_{*,c}}$ messages from different replicas before expiration of T , with identical digests of replicas' local history ($LHDigest$) and identical replies (or digests thereof), then the client commits req

⁷Here (see also Fig. 4) $reply_j = rep(LH_j)$, where rep is the function within a replicated state machine (i.e., application) that computes the reply to the client. As an optimization, all but one designated replica can send reply digests $D(reply_j)$ instead of $reply_j$ within a RESP message.

with *reply*. Otherwise, the client triggers the panicking mechanism as explained in section 4.2.2 (Step P1).

4.2.2. Aborting. In this section, we describe the pseudo-code of the panicking mechanism that clients trigger when they do not receive $3f + 1$ consistent replies (in Step Z4). As soon as the execution of this mechanism completes, the client is able to generate an abort history that it can use to switch to another Abstract instance (*Backup* in the case of the *AZyzyva* protocol).

Step P1. If the client does not commit request *req* by the expiration of timer *T* (triggered in Step Z1), *c* panics, i.e., it sends a $\langle \text{PANIC}, req.t_c \rangle_{\mu_c, r_j}$ message to every replica r_j . Since messages may be lost, the client periodically sends PANIC messages, until it aborts the request.

Step P2. Replica r_j , on receiving a $\langle \text{PANIC}, req.t_c \rangle_{\mu_c, r_j}$ message, stops executing new requests (i.e., stops executing Step Z3) and sends $\langle \text{ABORT}, req.t_c, LH_j, next(i) \rangle_{\sigma_{r_j}}$ to *c* (with periodic retransmission).

Step P3. When client *c* receives $2f + 1$ $\langle \text{ABORT}, req.t_c, *, next(i) \rangle_{\sigma_*}$ messages with correct signatures from different replicas and the same value for *next(i)*, the client collects these messages into a set $Proof_{AH_i}$, and extracts the abort history AH_i from $Proof_{AH_i}$ as follows: first, *c* generates history *AH* such that $AH[j]$ equals the value that appears at position $j \geq 1$ of $f + 1$ different histories LH_j that appear in $Proof_{AH_i}$; if such a value does not exist for position *x* then *h* does not contain a value at position *x* or higher. Then, *c* extracts the abort history AH_i by taking the longest prefix of *AH* in which no request appears twice.

4.2.3. Initializing a ZLight instance. In this section, we describe the way a *ZLight* instance is initialized. We assume that a client extracted an abort history AH_i and that it invokes instance $i' = next(i)$. To do so, it accompanies *req* with init history $IH_{i'} = AH_i$ and $Proof_{AH_i}$. We summarize below the additional actions performed by processes in the various steps of *ZLight* to take into account init histories.

Step Z1+. On $Invoke_{i'}(req, IH)$, the message(s) sent by the client also contains *IH* and the set of signed ABORT messages $Proof_{IH}$ returned by the preceding Abstract *i*, where $i' = next(i)$.

Step Z2+. If its local history LH_1 is empty, the primary/head r_1 executes the step only if (i) *IH* can be verified against $Proof_{IH}$, following the algorithm given in Step P3 (Section 4.2.2), and (ii) ABORT messages in $Proof_{AH_i}$ indeed declare i' as $next(i)$.

Step Z3+. If its local history LH_j is empty, the replica r_j executes the step only if (i) *IH* can be verified against $Proof_{IH}$, following the algorithm given in Step P3 (Section 4.2.2), and (ii) ABORT messages in $Proof_{AH_i}$ indeed declare i' as $next(i)$. If so, then (before executing *req*) r_j logs all the requests contained in *IH* (i.e., r_j sets LH_j to *IH*); then r_j logs *req* unless *req* was already in *IH*.

Step P1+. On sending PANIC messages for a request that was invoked with an init history, client also includes *IH* and the set of signatures $Proof_{IH}$ returned by the preceding Abstract *i* within a PANIC message.

Step P2+. If its local history LH_j is empty, replica r_j , executes the step only if (i) IH can be verified against $Proof_{IH}$, following the algorithm given in Step P3 (Section 4.2.2), and (ii) ABORT messages in $Proof_{AH_i}$ indeed declare i' as $next(i)$. Then, before executing the step as described in Section 4.2.2, r_j first sets LH_j to IH .

4.2.4. Checkpointing. *ZLight* uses a *lightweight checkpoint subprotocol* (LCS) to truncate histories every CHK requests (in our performance evaluation, $CHK = 128$). LCS is very similar to checkpoint protocols used in [Castro and Liskov 2002; Kotla et al. 2010]. Its operating principle is the following:

- (1) Every replica r_j increments a checkpoint counter cc and sends it along with the digest of its local state to every other replica (using simple point-to-point MACs), when its (non-checkpointed suffix of) local history reaches CHK requests. Then, r_j triggers a checkpoint timer.
- (2) If the timer expires and there is no checkpoint, the replica stops executing all requests and retransmits its last checkpoint message to every other replicas.
- (3) If replica r_j receives the digest of the same state st_{cc} with the same checkpoint counter number cc greater than $lastcc$ (initially $lastcc = 0$) from *all* replicas, r_j : (a) truncates its local history and checkpoints its state to st_{cc} , and (b) stores cc to variable $lastcc$. Checkpointed state st_{cc} becomes a prefix of replicas' local histories to which new requests are appended and is treated as such in all operations on local histories in our algorithms. Moreover, every abort or commit history of length at most $cc * CHK$ is considered to be a prefix of st_{cc} .

With LCS, Step P2 of the panicking mechanism (Sec. 4.2.2) is modified so that a digest of the last checkpointed state, the last checkpoint number, and the non-checkpointed suffix of the local history are propagated by an aborting replica r_j in place of a complete local history LH_j . Note however that LCS has an additional slight impact on the panicking/aborting mechanism. Consider, for example, replicas performing a checkpoint $cc = 34$ concurrently with some client panicking. What may happen is that when the PANIC message is received by, say replicas r_1 and r_2 , the local state of replica r_1 is already truncated to st_{34} , whereas the local state of replica r_2 is still st_{33} followed by CHK requests. — we refer to this effect as *partial checkpoint*. If r_1 and r_2 send such (seemingly different) information to the panicking client within ABORT messages, a client might not be able to tell that the local states and local histories of replicas r_1 and r_2 are, in fact, identical. This may in turn impact the panicking/aborting subprotocol as already described. To this end, when: (a) the non-checkpointed suffix of local history of replica r_j contains CHK requests or more, but a replica r_j did not yet perform a checkpoint cc , and (b) replica r_j is about to send an ABORT message, then replica r_j tentatively performs checkpoint cc and calculates st_{cc} prior to sending an ABORT. Then, r_j adds the digest of st_{cc} to its ABORT message that already contains the digest of st_{cc-1} and the non-checkpointed suffix, as explained earlier. This allows the client to avoid the apparent ambiguity in identifying the same histories in Step P3, Sec. 4.2.2, in case of a partial checkpoint.

4.3. Backup

Backup is an Abstract implementation with a progress property that guarantees that exactly $k \geq 1$ requests will be committed, where k is a generic parameter (we explain our configuration for k at the end of this section). We employ *Backup* in *AZyzyva* 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 [Castro and Liskov 2002], 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 [Clement et al. 2009], are also very good candidates for the *Backup* basis, as illustrated in Section 6.3.

To implement *Backup*, we exploit the fact that any BFT protocol can totally order requests submitted to it 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 *AZyzyva*). At this point, *Backup* sets its state by executing all the requests contained in the valid init history it received. Then, it simply executes the first k requests ordered by the underlying BFT protocol (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. A client can switch from *Backup* as soon as it receives $f + 1$ signed messages from different replicas, containing an identical abort history and the same next Abstract instance id i' . This is a reasonable requirement on the BFT protocol that underlies *Backup*, since any BFT protocol must anyway provide an identical reply from at least $f + 1$ replicas; in the case of *Backup* abort history, we just require this particular reply to be signed (all existing BFT protocols we know of provide a way to digitally sign messages).

The parameter k used in *Backup* 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.⁸ 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.⁹ 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.

4.4. State transfer optimization

Sending entire local histories to clients within ABORT messages in Step P2 of the panicking/aborting subprotocol (Sec. 4.2.2) might reveal expensive, even if local histories are checkpointed as described in Section 4.2.4. To this end, all our Abstract implementations presented in this paper, except *Backup*, implement the following *state transfer optimization* when switching to the next Abstract instance.¹⁰

Upon receiving a PANIC message from a client in Step P2, replica r_j sends an ABORT message to the client containing the signed history of *digests* of (non-checkpointed)

⁸In short, k requests committed by a single *Backup* instance i might all be invoked by the same, fast client. A slow client can then get its request aborted by i . The same can happen with a subsequent *Backup* instance, etc. This issue can be avoided by exponentially increasing k (for any realistic load that does not increase faster than exponentially) or by having the replicas across different Abstract instances share a client input buffer.

⁹For example, using $k = \lceil C * 2^m \rceil$, where m is incremented with every new Abstract instance, with the rough average time of 50ms for switching between 2 consecutive *Backup* instances in *AZyzyva*, we can maintain $k = 1$ during 10s outages with $C = 2^{-200}$.

¹⁰In *Backup*, to minimize the size of state transfer through clients, we simply align switching with a checkpoint, since the former can be performed at will.

requests, in place of the signed history of (non-checkpointed) requests. The client performs Step P3 normally, except that it extracts an abort history which contains digests of requests, rather than an abort history that contains full requests.

A client normally uses such an abort history (together with the corresponding proof of $2f + 1$ ABORT messages) as an init history of the next Abstract instance. During initialization of the next instance, the replicas verify the abort history normally. Often, a replica being initialized will have all the requests locally and will initialize its local state accordingly. However, a replica might not have some requests whose digests appear in the init history — we speak of *missing requests*. If a replica misses requests at initialization, it simply asks other replicas for inter-replica state transfer of missing requests. This procedure is guaranteed to terminate since each digest of the request in the init history is vouched for by at least $f + 1$ replicas; for each request, at least one of these replicas is correct and supplies the missing request.

4.5. Qualitative assessment

In evaluating the effort of building *AZyzyyva*, we focus on the cost of *ZLight* in terms of lines of code. 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* and *R-Aliph* protocols as well (Section 5 and 6).

Our main observation is that it took 4086 lines of code to implement *ZLight*, which is to be contrasted to 14339 of lines of code needed for *Zyzyyva*. Note that we implemented and used an external library that contains cryptographic functions, networking code (to send/receive messages and manage sockets), and data structures (e.g., maps, sets). This library roughly contains 7,500 lines of code that are not taken into account. To provide a fair comparison, we ported *Zyzyyva* on this library. In other words, to build *ZLight*, we needed less than 30% of the *Zyzyyva* line count (14,339 lines).

Needless to say, this line-of-code comparison has to be taken with a grain of salt: (i) these protocols have been developed and proved by different researchers, and (ii) *Zyzyyva* does not fully implement the code required to handle faults. Yet, we believe that the line-of-code metric provides a useful intuition of the difference in code and algorithmic complexity between *Zyzyyva* and *ZLight* since both implementations use the same code base, inherited from PBFT [Castro and Liskov 2002].

As another benefit of using Abstract comes also the fact that we did not have to prove from scratch the correctness of *AZyzyyva* in all possible executions. Namely (see also Appendix A), to prove *AZyzyyva* we needed to concentrate our effort only on proving the correctness of *ZLight*. The general Abstract composability theorem 3.1 and the straightforward proof of the reusable *Backup* module complement the entire proof.

Finally, it is fair to note that incremental development using Abstract may, in fact, increase the code base that needs to be maintained. Indeed, in our case, instead of maintaining only *Zyzyyva* or PBFT, we would need to maintain both *ZLight* and PBFT/*Backup*, which together have more lines of code than any of *Zyzyyva* or PBFT individually. In this sense, there is no free lunch: adaptive performance of Abstract has a certain price with respect to monolithic BFT protocols. While more lines of code might seemingly imply more vulnerabilities for Byzantine attacks and more maintenance effort, we believe that this is not the case due to simplified design, proofs and testing of individual modules. Such modular designs are widely used in traditional software engineering, where monolithic solutions that might have less lines of code, often have more bugs and are more difficult to maintain.

4.6. Performance evaluation

We have compared the performance of *AZyzyyva* and *Zyzyyva* in the “common-case”, using the benchmarks described in Section 5.4. Not surprisingly, *AZyzyyva* and

Zyzyva have identical performance in this case, since in the “common-case” the two protocols are the same. 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).

To assess the switching cost, we perform the following experiments: we feed the request history of *ZLight* with r requests of size 1 kB. 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 ensure that for each aborted request, the history contains r requests. We reproduced each experiment three times and observed a variance of less than 3%.

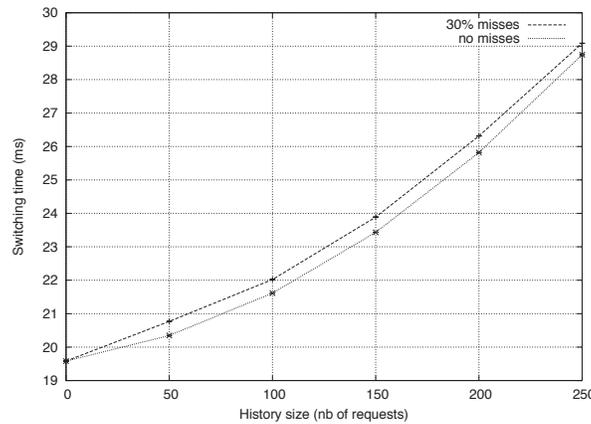


Fig. 5. Switching time as a function of the history size and the percentage of missing requests in replica histories.

Figure 5 shows the switching time (in ms) as a function of the history size when the number of tolerated faults equals 1. As described in Section 4.2.4, *ZLight* uses a checkpointing mechanism triggered every 128 requests. Moreover, to account for requests that can be received while a replica is performing a checkpoint, we assume that the history size can grow up to 250 requests. Note that in our current implementation, the history size is actually bounded: when the history is full, requests are blocked until a checkpoint is completed. We plot two different curves: one corresponds to the case when replicas do not miss any request. The other one corresponds to the case of *missing requests*, described in Section 4.4. More precisely, we assess the performance when 30% of the requests are absent from the history of at least one replica upon receiving an init history containing signed message digests. 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 some requests (as replicas need to fetch the requests they miss). Interestingly, we see that the switching cost is low. It ranges between $19.7ms$ and $29.2ms$. This is very reasonable provided that faults are supposed to be rare in the environment for which *Zyzyva* 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 the checkpointing protocol. 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 *Zyzyva*, induced by the request replay during view-changes.

5. A NEW BFT PROTOCOL: ALIPH

In this section, we demonstrate how we can build novel, very efficient BFT protocols, using Abstract. We present a new protocol, called *Aliph*, that achieves up to 30% lower latency and up to 25% higher throughput than state-of-the-art protocols. The development of *Aliph* consisted in building two new instances of Abstract, each requiring less than 30% of the code of state-of-the-art protocols, and reusing *Backup* (Section 4.3). In the following, we start by an overview of *Aliph*. We then present the two new Abstract instances it relies on. Finally, we assess its performance.

5.1. Protocol overview

Aliph is a new BFT protocol that uses three Abstract implementations: *Backup*, *Quorum* and *Chain*. The *Backup* protocol has been introduced in Section 4.3. 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 (one round-trip of message exchange) that is very efficient when there is no contention. The *Chain* protocol provides exactly the same progress guarantees as *ZLight* (Section 4.2), i.e., it commits requests as long as there are no server/link failures or Byzantine clients. *Chain* implements a pipeline pattern that is very efficient under contention, unlike *Quorum*. *Aliph* uses the following static switching ordering to orchestrate its underlying protocols: *Quorum-Chain-Backup-Quorum-Chain-Backup-...* In other words, *Quorum* is initially 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 commits k requests (see Section 4.3). When *Backup* commits k requests, it aborts, switches back to *Quorum*, and so on.

The characteristics of *Aliph* are summarized in Table I, considering the metrics of [Kotla et al. 2010]. As we can see, *Aliph* achieves better throughput and latency than existing protocols and is optimally resilient. More precisely, *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.

Table I. Characteristics of state-of-the-art BFT protocols.

	PBFT	QU	HQ	Zyzyva	Aliph
Number of replicas	3f+1	5f+1	3f+1	3f+1	3f+1
Number of MAC operations at the bottleneck replica	$2 + \frac{8f}{b}$	2+4f	2+4f	$2 + \frac{3f}{b}$	$1 + \frac{2f+1}{b}$
Number of 1-way messages in the critical path	4	2	4	3	2

Note: Bold entries denote protocols with the lowest known cost.

In the next two sections, we describe *Quorum* and *Chain*. We focus on the common-case as both *Quorum* and *Chain* use the same panicking mechanism and checkpointing protocol as *ZLight* that we presented in Sections 4.2.2 and 4.2.4, respectively.

5.2. Quorum

In this section we present *Quorum*, an Abstract implementation used in the *Aliph* protocol to guarantee low latency when there is no contention. We first describe the *Quorum* protocol. We then give its pseudo-code.

5.2.1. Protocol description. The communication pattern implemented in *Quorum* is depicted in Figure 6. This pattern is very simple: 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* (Section 4.2.2): (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, inducing the current *Quorum* instance to abort. This is not the case in *ZLight*, as requests are ordered by the primary.

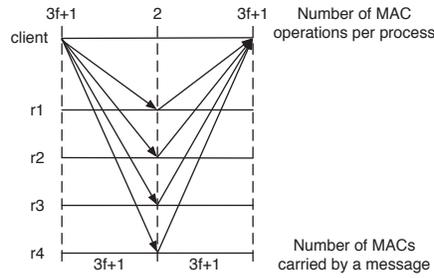


Fig. 6. Communication pattern of *Quorum*.

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 [Abd-El-Malek et al. 2005] 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 the weaker (and hence easier to implement) Abstract specification, without having to consider the numerous difficult corner cases that occur outside the “common-case”.

Finally, let us note that the implementation of *Quorum* is very simple. It requires only 3200 lines of C/C++ code (including the code for panicking and checkpointing that is the same as the respective code used in the *ZLight* and *Chain* protocols).

5.2.2. Pseudo-code. We give below the pseudo-code of *Quorum* using the same notations as the ones presented in Figure 4. Moreover, as for *ZLight*, we omit to mention, for the sake of brevity, that upon receiving a message m , a process p first checks that m has a valid authenticator. We do not describe the pseudo-code of the panicking and checkpointing mechanisms as they are shared between *Quorum* and *ZLight*. Finally, we maintain the definitions of a replica *logging* and *executing* a request from *ZLight* (Sec. 4.2).

Step Q1. On $Invoke_i(req)$, client c sends message $\langle REQ, req, i \rangle_{\mu_c, \Sigma}$ to all replicas and triggers timer T set to 2Δ .

Step Q2. Replica r_j on receiving $\langle REQ, req, i \rangle_{\mu_c, \Sigma}$ from client c , if $req.t_c$ is higher than $t_j[c]$, then it (i) updates $t_j[c]$ to $req.t_c$, (ii) logs and executes req , and (iii) sends

$\langle \text{RESP}, \text{reply}_j, D(LH_j), i, \text{req.t}_c, r_j \rangle_{\mu_{r_j, c}}$ to c .

Step Q3. Identical to Step Z4 of *ZLight*.

5.3. Chain

In this section we present *Chain*, an Abstract implementation used in the *Aliph* protocol to ensure high throughput under contention. The *Chain* protocol shares similarities with the protocol presented in [van Renesse and Schneider 2004]. Nevertheless, there is a significant difference between the two protocols: the protocol presented in this paper tolerates Byzantine faults, whereas the protocol presented in [van Renesse and Schneider 2004] only tolerates crash faults, which makes it significantly simpler. This section is organized as follows: we first describe the *Chain* protocol. We then give its pseudo-code.

5.3.1. Protocol description. The communication pattern implemented in *Chain* is presented in Figure 7. As we see, *Chain* organizes replicas in a pipeline. 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}).

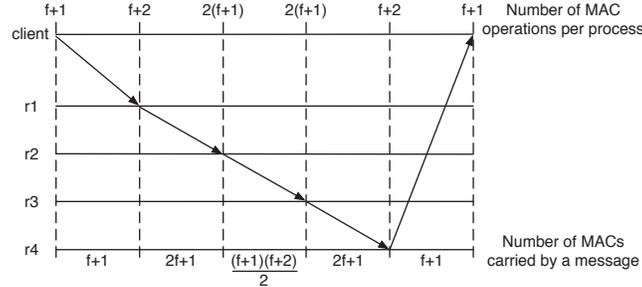


Fig. 7. Communication pattern of *Chain*.

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* \vec{r}_i , where $\vec{r}_i = r_{i+1}$. 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.

Chain tolerates Byzantine failures by ensuring: (1) that the content of a message is not modified by a Byzantine replica before being forwarded, (2) that no replica in the chain is bypassed, and (3) that the reply sent by the tail is correct. To provide those guarantees, *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 logs req (i.e., appends it to its local history). 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*.

Replicas and clients 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 order. The successor set of a 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 process p receives a message m it *verifies* m , i.e., it checks whether m contains a correct MAC from the processes from p 's *predecessor set* (a set of processes q such that p is in q 's successor set). For instance, replica r_2 verifies that the message contains a valid MAC from the replica r_1 (i.e., the head) 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 order.

Whereas all *Chain* replicas log a request by appending it to their local history, only $f + 1$ last replicas execute requests and calculate the application level reply. The reply is sent to the client only by the tail. To make sure that the reply sent by the tail is correct, the f processes that precede the tail in the chain order append a digest of the reply to the message.¹¹

When the client receives a correct reply, it commits it. On the other hand, when the reply is not correct, or when the client does not receive any reply (e.g., due to a Byzantine replica which discards the request), the client sends a PANIC message to all replicas. Just like 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 makes *Aliph* the first protocol in which the number of MAC operations at the bottleneck replica tends to 1. Indeed, the bottleneck replica in *Chain* is the $f + 1$ -st replica. This replica needs to (i) read a MAC from the client, (ii) read a MAC from its f predecessors in the chain, and (iii) write a MAC for its $f + 1$ first successors in the chain. Replicas in the chain can forward multiple requests in a single batch, and can generate a single MAC for the batch of requests. The first $f + 1$ replicas do nevertheless need to read the MAC written by the client. Consequently, the bottleneck replica (i.e., the $f + 1$ -st replica) will perform $1 + \frac{2f+1}{b}$ MAC operations per request, with b being the number of requests per batch. State-of-the-art protocols [Kotla et al. 2010; Castro and Liskov 2002] 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 intuitively be explained by the fact that these are two distinct set of replicas that read a MAC from the client (the $f + 1$ first replicas in the chain) and write a MAC to the client (the $f + 1$ last replicas in the chain). In contrast, state-of-the-art protocols require some replicas to both read a MAC from the clients and write a MAC to the clients.

We have implemented *Chain* in C/C++. The implementation requires about 4300 lines of code (including the panicking and checkpointing code). This is about 30% of the code size of state-of-the-art protocols.

5.3.2. Pseudo-code. We describe below the pseudo-code of *Chain*. We use the same notations as for *ZLight* and *Quorum*. These notations are summarized in Figure 4. Moreover, we assume that every CHAIN message sent by a process p contains the chain authenticator (CA) generated by p , as well as the MACs p received from its predecessor \overleftarrow{p} and that are destined to processes in p 's successor set. Finally, we do not describe the pseudo-code of the panicking and checkpointing mechanisms which are shared between *Chain* and *ZLight*.

¹¹In some cases, sequential request execution by $f + 1$ *Chain* replicas might impact latency of *Chain* for large values of f . On the other hand, (i) values of f are typically small, and (ii) there are practical workloads with processing times much lower than the communication latency, e.g., when a metadata service is replicated.

Step C1. On $Invoke_i(req)$, client c sends the message $m' = \langle \text{CHAIN}, req, i \rangle$ to the head (say r_1) and triggers the timer T set to $(n + 1)\Delta$.

Step C2. The head r_1 , on receiving $m = \langle \text{CHAIN}, req, i \rangle$ from client c , if (i) $req.t_c$ is higher than $t_1[c]$, and (ii) the head can verify client's MAC (otherwise the head discards m), then the head (i) updates $t_1[c]$ to $req.t_c$, (ii) increments sn_1 , and (iii) sends $\langle \text{CHAIN}, req, i, sn_1, \perp, \perp \rangle$ to $\vec{r}_1 = r_2$.

Step C3. Replica r_j on receiving $m = \langle \text{CHAIN}, req, i, sn, \text{REPLY}, \text{LHDigest} \rangle$ from \vec{r}_j , if (i) it can verify MACs from all processes from its predecessor set against the content of m , (ii) $sn = sn_j + 1$, and (iii) $req.t_c$ is higher than $t_j[c]$, then it (i) updates sn_j to sn and $t_j[c]$ to $req.t_c$, (ii) if r_j is one of the first $2f$ replicas, it logs req , otherwise r_j logs and executes req , and (iii) r_j sends $\langle \text{CHAIN}, req, i, sn, \text{REPLY}, \text{LHDigest} \rangle$ to \vec{r}_j , where $\text{REPLY} = \text{LHDigest} = \perp$ in case of the first $2f$ replicas, $\text{REPLY} = D(reply_j)$ and $\text{LHDigest} = D(LH_j)$ in case $j \in \{2f + 1 \dots 3f\}$, and $\text{REPLY} = reply_j$ and $\text{LHDigest} = D(LH_j)$ in case r_j is tail. In case MAC verification mentioned above fails, replica stops executing Step C3 in instance i .

Step C4. If client c receives $\langle \text{CHAIN}, req, i, *, reply, \text{LHDigest} \rangle$ from the tail before expiration of T_{chain} , and with MACs from last $f + 1$ replicas that authenticate $req, i, \text{LHDigest}$ and $D(reply)$ (or $reply$ itself), then c commits req with $reply$. Otherwise, the client triggers the panicking mechanism explained in Section 4.2.2 (Step P1).

5.4. Performance evaluation

In this section, we evaluate the performance of *Aliph*. 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.

We first study the latency, throughput, and fault scalability using microbenchmarks [Castro and Liskov 2002; Kotla et al. 2010], 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.¹² 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 Zyzyva because the former is considered the “baseline” for practical BFT implementations, whereas the latter is considered state-of-the-art. Moreover, Zyzyva systematically outperforms HQ [Kotla et al. 2010]; hence, we do not evaluate HQ. Finally, we benchmark Q/U as it is known to provide better latency than Zyzyva under certain conditions. Note that Q/U requires $5f + 1$ servers, whereas other protocols we benchmark only require $3f + 1$ servers.

PBFT and Zyzyva 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 Zyzyva. Therefore, we assess Zyzyva with or without batching depending on the

¹²Although closed-loop microbenchmarks are not always representative of the behavior of real systems [Schroeder et al. 2006], we use these microbenchmarks to enable fair comparison with previously reported results, e.g. [Castro and Liskov 2002; Kotla et al. 2010; Clement et al. 2009].

experiment. As for the client multicast optimization, we show results for both configurations every time we observe an interesting behavior.

Aliph also implements two optimizations. First, the progress property of *Chain* is slightly different from the one described earlier in the paper: *Chain* aborts requests as soon as replicas detect that there is no contention (i.e. there is only one active client since at least $2s$). This avoids executing *Chain* when there is no contention. Second, *Chain* replicas add an information in their abort history to specify that they aborted because of the lack of contention. We modified *Backup*, so that in such case, it only commits one request and aborts. Consequently, when there is no contention, *Aliph* switches to *Quorum*, which is very efficient in such a case.

Finally, let us note that the PBFT code base underlies both *Zyzyva* and *Aliph*, which ensures a fair comparison between these three protocols. To also ensure that the comparison with Q/U is fair, we evaluate a simple best-case implementation that uses the same code base and that is described in [Kotla et al. 2010]. Consequently, all protocols are implemented in C++, rely on MD5 for computing message digests, and use UMAC as MAC type. Finally, regarding networking primitives, all protocols use UDP and IP Multicast, as described in the respective papers. The only exception is *Chain* that uses TCP. As our experiments demonstrate, the pipeline pattern of *Chain* coupled with the use of TCP gives *Chain* significant advantage over other protocols under high load.

5.4.1. Latency. We first assess the latency in a system without contention, with a single client issuing requests. The improvement of *Aliph* over Q/U, PBFT, and *Zyzyva* is reported in Table II for all microbenchmarks (0/0, 0/4 and 4/0) and for a maximal number of server failures f ranging from 1 to 3. We observe that *Aliph* consistently outperforms other protocols. The reason why the latency achieved by *Aliph* is very low is that it uses *Quorum* when there is no contention. These results confirm the theoretical analysis (see Table I, Section 5.1). The results show that Q/U also achieves a good latency with $f = 1$. This is not surprising provided that Q/U and *Quorum* use the same communication pattern. Nevertheless, when f increases, the 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 *Zyzyva* (31% at $f = 1$) can be easily explained by the fact that *Zyzyva* requires 3-one-way message delays in the common case, whereas *Aliph* (*Quorum*) only requires 2-one-way message delays.

Table II. Latency improvement of *Aliph* for the 0/0, 4/0, and 0/4 benchmarks, without contention.

	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%
Zyzyva	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%

5.4.2. Throughput. In this section, we present throughput results obtained running the 0/0, 0/4, and 4/0 microbenchmarks under contention. We do not report 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, *Aliph* never switches to *Backup*. Consequently, *Chain* commits all the subsequent requests.

The results presented in this section show that *Aliph* consistently and significantly outperforms other protocols, starting from a certain number of clients that depends on

the benchmark. Below this threshold, Zyzyzyva achieves higher throughput than other protocols.

0/0 benchmark. Figure 8 plots the throughput achieved with the 0/0 benchmark by the various protocols when $f = 1$. We ran Zyzyzyva with and without batching. For PBFT, we present only the results with batching, since they are substantially better than those obtained without batching. We observe that Zyzyzyva with batching performs better than PBFT, which itself achieves higher peak throughput than Zyzyzyva without batching (this is consistent with the results of [Kotla et al. 2010; Singh et al. 2008]).

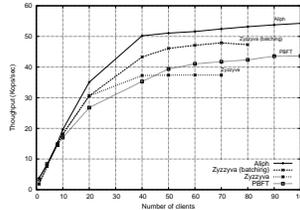


Fig. 8. Throughput for the 0/0 benchmark ($f=1$).

Moreover, Figure 8 shows that with up to 40 clients, Zyzyzyva achieves the best throughput. With more than 40 clients, *Aliph* starts to outperform Zyzyzyva. The peak throughput achieved by *Aliph* is 21% higher than that of Zyzyzyva. 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* protocol is efficient only if its pipeline is fed — the link between any server and its successor in the chain is 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 the 0/0 benchmark). Provided that 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 Zyzyzyva only with more than 40 clients.

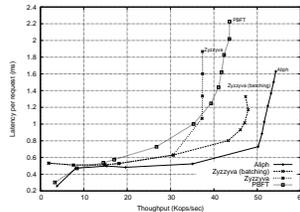


Fig. 9. Response time vs. throughput for the 0/0 benchmark ($f=1$).

Figure 9 plots the response time of Zyzyzyva (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 implemented by PBFT is very complex, 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 Zyzyzyva. The reason is the pipeline pattern of *Chain* that results in a higher response time for low to medium throughput,

but stays nevertheless reasonable. Moreover, *Aliph* scales better than *Zyzyzyva*: 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 *Zyzyzyva*. This comes from the fact that *Aliph* uses *Quorum* when there is no contention, which significantly improves the response time of the protocol.

0/4 benchmark. Figure 10 shows the throughput of the various protocols for the 0/4 microbenchmark when $f = 1$. PBFT and *Zyzyzyva* are using the client multicast optimization. We observe that with up to 15 clients, *Zyzyzyva* outperforms other protocols. Starting from 20 clients, *Aliph* outperforms PBFT and *Zyzyzyva*. Nevertheless, the gain in peak throughput (7,7% over PBFT and 9,8% over *Zyzyzyva*) is lower than the gain we observed with the 0/0 microbenchmark. This can be explained by the fact that the dominating cost is now in sending replies to clients. This cost partly masks the cost of request ordering. In all protocols, there is only one server sending a full reply to the client (other servers send only a digest of the reply), which explains why the various protocols achieve a pretty close throughput.

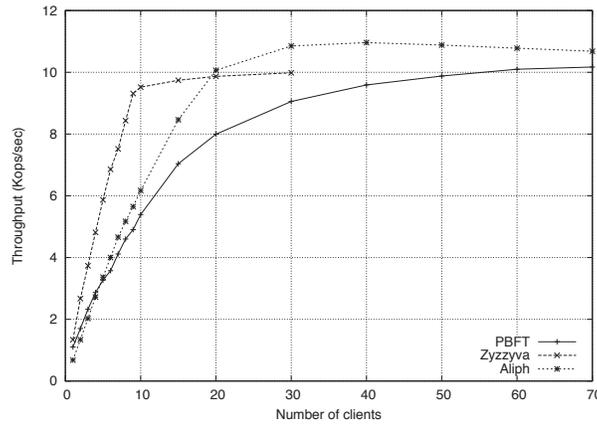
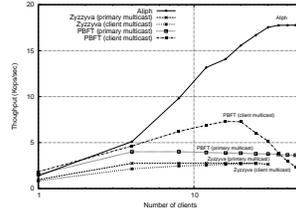
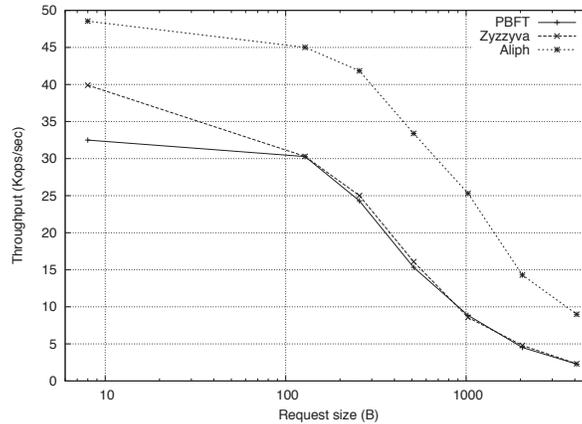


Fig. 10. Throughput for the 0/4 benchmark ($f=1$).

4/0 benchmark. Figure 11 shows the results of *Aliph*, PBFT and *Zyzyzyva* 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 the various protocols with small numbers of clients. For PBFT and *Zyzyzyva*, we plot curves both with and without client multicast optimization. The graph shows that with up to 3 clients, *Zyzyzyva* outperforms other protocols. With more than 3 clients, *Aliph* significantly outperforms other protocols. Its peak throughput is about 360% higher than that of *Zyzyzyva*. 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. We attribute the poor performance of PBFT and *Zyzyzyva* to the fact that when IP Multicast is used with large messages, this induces message losses that are inefficiently handled by the available prototypes. Moreover, we explain the performance drop observed for *Zyzyzyva* and PBFT when the client multicast optimization is used (Fig. 11) by the fact that enabling this optimization increases the number of message losses (due to a higher number of message collisions).

Fig. 11. Throughput for the 4/0 benchmark ($f=1$).

5.4.3. *Impact of the request size.* In this experiment, we study how protocols are impacted by the size of requests. Figure 12 shows the peak throughput of *Aliph*, PBFT and Zyzyyva as a function of the request size for $f = 1$ (the response size is kept at 0k). To obtain the peak throughput of PBFT and Zyzyyva, we benchmarked both protocols with and without the client multicast optimization and with different batching sizes for Zyzyyva. Interestingly, the behavior we observe is similar to that observed using simulations in [Singh et al. 2008]: the performance gap between PBFT and Zyzyyva diminishes with the increase in payload. Indeed, starting from 128B payloads, both protocols have almost identical performance. Figure 12 also shows that *Aliph* sustains high peak throughput with all message sizes, which is again the consequence of *Chain* being active under contention.

Fig. 12. Peak throughput as a function of the request size ($f=1$).

5.4.4. *Fault scalability.* One important characteristic of BFT protocols is their behavior when the number of tolerated server failures f increases. Figure 13 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 Zyzyyva as their peak throughput is known to suffer only a slight impact [Kotla et al. 2010]. Figure 13 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.

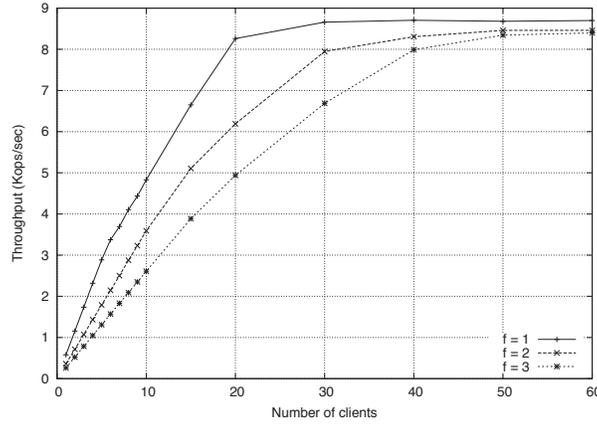
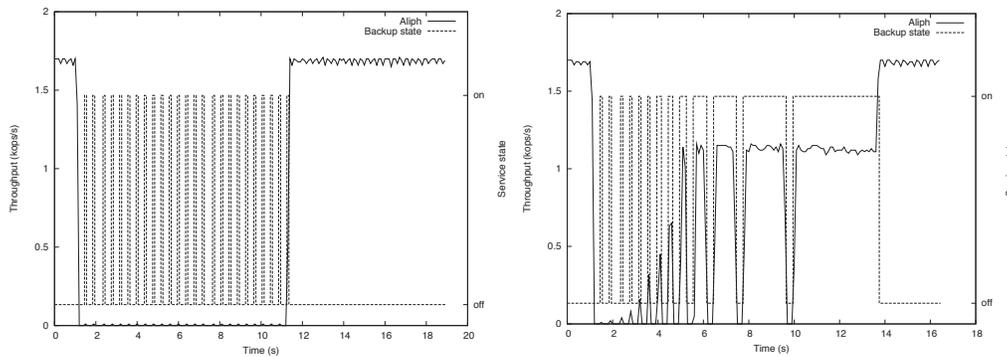


Fig. 13. Impact of the number of tolerated failures f on the *Aliph* throughput.

5.4.5. Behavior of *Aliph* in case of faults. In this section, we assess the behavior of *Aliph* when one 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 commits $k = 1$ request. In the second strategy, when *Aliph* switches to *Backup*, it commits $k = 2^i$, where i is the number of invocations of *Backup* since the beginning of the experiment. Note that in its current version, *Aliph* combines both strategies by exponentially increasing k , while maintaining the exponential curve initially very flat for reasons discussed in Section 4.3.

The behavior of *Aliph* with the first strategy is depicted in Figure 14(a). 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.



(a) Behavior under faults, when *Aliph* switches to *Backup* for one request. (b) Behavior under faults, when *Aliph* switches to *Backup* for 2^i requests.

Fig. 14. Behavior of *Aliph* in case of faults.

Figure 14(b) shows the behavior of *Aliph* with the second strategy. We observe that 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.

5.4.6. Dynamic workload. Finally, we study the performance of *Aliph* under a dynamic workload (i.e., fluctuating contention). We compare its performance to that achieved by *Zyzyva* 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, with response size kept at 0k. 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 15 shows the performance of *Aliph*, *Zyzyva*, and *Chain*. For each protocol, clients were invoking the same number of requests. Moreover, requests were invoked after the preceding clients had completed their bursts. First, we observe that *Aliph* is the most efficient protocol: it completes the experiment in 42s, followed by *Zyzyva* (68.1s), and *Chain* (77.2s). Up to time $t = 15.8s$, *Aliph* uses *Quorum*, which performs much better than *Zyzyva* 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 described earlier. 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 *Zyzyva* when there is a load spike: they achieve a peak throughput about three times higher than that of *Zyzyva*. On the other hand, *Chain* and *Aliph* have slightly lower performance than *Zyzyva* under medium load (i.e., from 16s to 26s on the *Aliph* curve). This suggests an interesting BFT protocol that would combine *Quorum*, *Zyzyva*, *Chain* and *Backup*.

6. MAKING ALIPH ROBUST: R-ALIPH

The *Aliph* protocol presented in Section 5 achieves excellent performance in the “common” case, i.e., when the network is synchronous and when both clients and replicas are benign. Unfortunately, as we show in this section, a Byzantine client or a Byzantine replica can attack this protocol and drastically reduce its performance. This fragility of BFT protocols is well-known and motivated the development of three so-called *robust protocols*: Spinning [Veronese et al. 2009], Prime [Amir et al. 2011], and Aardvark [Clement et al. 2009]. In this section, we first study the performance achieved by *Aliph* when clients or replicas attack the protocol (Sec. 6.1). We then briefly describe existing robust protocols and compare their performance under the same attacks (Sec. 6.2). We then describe *R-Aliph*, a *robust* version of the *Aliph* protocol (Sec. 6.3). Finally, we evaluate the performance of *R-Aliph* both with and without attacks (Sec. 6.4).

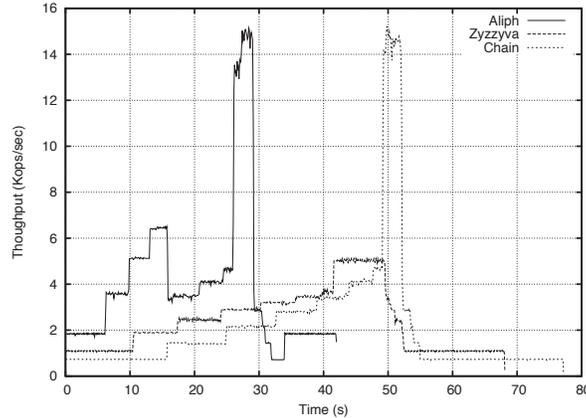


Fig. 15. Throughput under dynamic workload.

6.1. *Aliph* under attack

We study the performance of *Aliph* under four different, representative attacks inspired from those used in [Clement et al. 2009]:

- *Client flooding*: in this attack, one of the clients is Byzantine: it implements a brute force denial of service attack by repeatedly sending 9kB messages to the replicas.
- *Malformed client requests*: in this attack, one of the clients is Byzantine: it sends requests with an invalid authenticator that can only be authenticated by a subset of the replicas (including the primary in *Backup* and the head in *Chain*).
- *Processing delay*: in this attack, one Byzantine replica (the primary in *Backup* (PBFT), the head in *Chain*, and a randomly chosen replica in *Quorum*) delays the ordering of requests it receives from clients by 10ms.
- *Replica flooding*: in this attack, one of the replicas (different from the primary in *Backup* and the head in *Chain*) is Byzantine: it fails to process protocol messages and implements a brute force denial of service attack by repeatedly sending 9kB messages to other replicas.

Table III. Peak throughput (req/s) of *Aliph* both in the absence of attacks and under various attacks (0/0 micro-benchmark).

	Without attack	Client flooding	Malformed requests	Processing delay (10ms)	Replica flooding
<i>Aliph</i>	55575	30733 (-44,7%)	0 (-100%)	2629 (-95,3%)	0 (-100%)

The performance results for the 0/0 micro-benchmark are reported in Table III (we observed similar behavior with the 4/0 and 0/4 micro-benchmarks).¹³

We observe that the throughput of *Aliph* drops to 0 when malformed requests are sent or when one replica floods other replicas. This is explained by the fact that in both cases, *Quorum* and *Chain* are not able to commit requests, which induces a switching to *Backup*. *Backup* relies on PBFT which is unable to sustain a non-null throughput in these cases as already observed in [Clement et al. 2009].

¹³Notice that in order to accommodate the networking requirements of protocols that are studied later in this section, these experiments have been performed on another experimental testbed made of 8-core machines. Each machine has ten 1GB network cards and runs Linux 2.6.35. This explains the slight performance difference with results reported in previous sections.

Regarding the “processing delay” attack, the throughput of *Aliph* drops to about 2629 requests per seconds. This result is explained by the fact that requests are invoked in a closed-loop manner, i.e., a client does not invoke a new request before it gets a reply for a previous one. Consequently, delaying the ordering of requests decreases the throughput achieved by the protocol. Moreover, the overall latency perceived by the clients is below the threshold beyond which they timeout, panic and trigger a switching to *Backup*. Consequently, the throughput remains at this very low value.

Finally, regarding the “client flooding” attack, the throughput of *Aliph* decreases, but less than with the other attacks. This is explained as follows. When running under this attack, *Quorum* is not able to commit requests, which induces a switch to *Chain*. The latter relies on TCP, and, consequently, correct clients can issue requests at a high throughput despite the fact that a malicious client is trying to flood the network.

6.2. A brief overview of “robust” BFT protocols

The fact that efficient BFT protocols are often fragile (i.e., perform poorly under attack) motivated the development of so-called *robust* BFT protocols: Spinning [Veronese et al. 2009], Prime [Amir et al. 2011], and Aardvark [Clement et al. 2009]. These protocols aim at achieving good performance when the network is synchronous, *despite* the presence of Byzantine faulty clients and replicas. Below, we briefly describe these three protocols.

Spinning [Veronese et al. 2009] is a robust BFT protocol based on PBFT [Castro and Liskov 2002]. The idea underlying Spinning is to perform regular primary changes after each (fixed-size) batch of requests in order to limit the impact a Byzantine primary can have. Moreover, to further limit the impact of Byzantine primaries, Spinning uses a blacklisting mechanism that works as follows. Requests are sent to all replicas and, as soon as a non-primary replica receives a request, it starts a timer ($S_{timeout}$) and waits for a request ordering message from the primary for this request. In case of a timeout, the current primary is blacklisted (i.e., it will no longer become a primary in the future¹⁴) and $S_{timeout}$ is doubled.

Another robust protocol that has been designed is Prime [Amir et al. 2011]. In Prime, clients can send their requests to any replica in the system. Replicas periodically exchange the requests they receive from clients. Consequently, replicas are aware of the set of requests that should be ordered (i.e., for which they expect ordering messages from the primary). Moreover, even when there are no requests to order, the primary must periodically send (empty) ordering messages. That way, non-primary replicas expect to receive ordering messages at a given frequency. To improve the accuracy of the expected frequency, replicas monitor the network performance. Specifically, replicas periodically measure the round-trip time between each pair of them. This measure allows replicas to compute the maximum delay that should elapse between two consecutive sending of ordering messages by a correct primary. If the primary becomes slower than what is expected by the replicas, then it is replaced.

Finally, a third robust protocol that has been designed is Aardvark [Clement et al. 2009], also based on PBFT. Aardvark implements a number of mechanisms to ensure good performance despite the presence of Byzantine clients and replicas. First, the protocol limits the degradation that Byzantine clients can cause by isolating their traffic and by implementing a smart authentication mechanism that limits the impact of malformed client requests. Second, the protocol limits the impact that a faulty replica can have by isolating the traffic induced by the different replicas: namely, each replica uses a distinct NIC to communicate with every other replica. Third, the protocol lim-

¹⁴If f replicas are already blacklisted, then the oldest one is removed from the blacklist, to ensure the liveness of the system.

its the damage that a faulty primary may cause. To this end, all replicas monitor the throughput at which the primary is ordering requests. During the first five seconds of the primary being active, the primary is expected to achieve a throughput at least equal to 90% of the maximum throughput achieved by the primary replicas of the n preceding views (where n is the number of replicas). The expected throughput is then periodically raised by a factor of 0.01. As soon as a primary does not meet the throughput expectations, a primary change occurs.

Table IV. Peak throughput (req/s) of Spinning, Prime, and Aardvark both in the absence of attacks and under various attacks (0/0 micro-benchmark).

	Without attack	Client flooding	Malformed requests	Processing delay (10ms)	Replica flooding
Spinning	37116	19164 (-48,4%)	36986 (-0,3%)	18529 (-50,1%)	21975 (-40,8%)
Prime	6682	1445 (-78,4%)	6596 (-1,3%)	3648 (-45,4%)	0 (-100%)
Aardvark	31510	30280 (-3.9%)	31336 (-0,1%)	25997 (-17.5%)	28599 (-9,2%)

In Table IV, we report the performance of Spinning, Prime, and Aardvark for the 0/0 micro-benchmark, both in the absence of attacks and under the three attacks described in the previous section¹⁵. We do use a slightly modified “malformed request” attack due to the fact that the three studied protocols use signatures to authenticate messages rather than MAC authenticators. Hence, in this section, a malformed request is a request with an invalid signature.

We can make several observations. First, in the absence of attacks, robust protocols are much less efficient than fast BFT protocols, namely *ZLight* and *Chain*. For instance, the throughput of *Chain* is 76% higher than that of *Aardvark*. This performance difference between fast and robust protocols is expected and well-known [Clement et al. 2009]: it is due to the performance overhead induced by the various mechanisms that robust protocols implement to limit the impact of Byzantine clients and replicas. Second, we observe that, overall, the performance of robust protocols is less impacted by attacks than that of *Aliph*. Third, we observe that there are significant differences among robust protocols. *Aardvark* consistently achieves better performance under attack than both *Spinning* and *Prime*. The reason is that *Aardvark* combines accurate monitoring of the primary progress and network isolation between pairs of replicas, whereas *Spinning* and *Prime* only monitor the primary progress.

6.3. *R-Aliph*, a robust version of *Aliph*

The conclusion we can draw from the two previous sections is that distributed system designers seemingly have the choice between either (i) a BFT protocol (e.g., *Aliph*) that achieves very good performance when there are no attacks, but that achieves very bad or no throughput under attacks, or (ii) a BFT protocol (e.g. *Aardvark*) that achieves a lower throughput when there are no attacks (about 43 % less efficient than *Aliph* in the 0/0 micro-benchmark), but that is only slightly impacted by attacks.

In this section, we show how, using *Abstract*, we designed *R-Aliph*, a protocol that almost achieves the best of both worlds: *R-Aliph* is almost as efficient as *Aliph* when there are no attacks, and as efficient as *Aardvark* when there are attacks. To achieve this goal, we built *R-Aliph* following four main principles:

- (Principle P1) *Backup* is implemented on top of *Aardvark* (and is thus resilient to attacks),

¹⁵We use the original code bases for the three protocols. Nevertheless, we improved the networking stack of the *Spinning* protocol and enabled batching in order to increase its performance.

- (Principle P2) *Quorum* and *Chain* are only executed if they sustain a better throughput than the one *Backup* (i.e., Aardvark) would sustain,
- (Principle P3) *Quorum* and *Chain* are only executed if they are *fair* with respect to different clients, and
- (Principle P4) The time needed to switch among the three protocols is not impacted by the presence of Byzantine clients and Byzantine replicas.

It is trivial to enforce Principle P1. We describe below how the remaining three principles are enforced.

In order to make sure that *R-Aliph* executes *Quorum* or *Chain* only if they sustain a better throughput than the one *Backup* would sustain (see Principle P2), replicas executing *Quorum* or *Chain* periodically monitor the throughput achieved by the protocol, and check that it is higher than an expected throughput. If a replica detects that this is not the case, it becomes *unhappy*, stops processing requests and triggers a protocol switching (we explain this later on, under Principle P4). In the following, we explain how replicas in *R-Aliph*: (a) set their throughput expectations, and (b) monitor the current throughput.

- (a) In setting throughput expectation thresholds, *R-Aliph* replicas leverage the throughput expectations that are computed inside *Backup* (i.e., Aardvark) when it is executed. More specifically, when *R-Aliph* switches to *Quorum* or *Chain*, the throughput that each replica r_i expects is the maximum over all throughput expectations r_i computed when executing *Backup* (i.e., Aardvark).
- (b) When executing *Quorum* or *Chain* (just like with *Zyzyva/ZLight*), a replica cannot accurately compute the throughput based on what it observes locally, i.e., neither based on the requests it orders, nor on the checkpoint messages it receives from other replicas. For instance, a Byzantine replica in *Quorum* (or *Zyzyva/ZLight*) can send a checkpoint message pretending that it executed a request, but postpone the sending of the reply to that request, thus reducing the overall throughput without being caught by the client (e.g., if the client receives the reply before the timer expires, as in the “processing delay” attack described in Section 6.1). Similarly, the tail replica in *Chain* can send a checkpoint message pretending that it replied to a client, but postpone the sending of the reply to the client.

Consequently, the only way for replicas to accurately monitor the throughput in *R-Aliph* is to ask clients to send to replicas feedback messages confirming they committed the requests they previously sent.¹⁶ In *R-Aliph*, *Quorum* and *Chain* replicas calculate the throughput using feedback messages every 128 committed requests with a replica taking feedback into account only for a request it previously executed. In our prototype, to limit the overhead of feedback messages, clients only send them every 5 requests they commit. Moreover, in the case of *Quorum*, feedback messages are piggybacked to “common-case” requests within REQ messages. In addition, if a replica does not perform another throughput calculation in time *Backup* would have committed 128 requests (see also point (a)), a replica immediately becomes unhappy and stops processing further requests.

In order to make sure that *R-Aliph* executes *Quorum* and *Chain* only in case clients are treated *fairly* (Principle P3), the replicas implement the following mechanism (inspired by Aardvark): replicas track client requests: they check that, after having

¹⁶Notice here that some BFT protocols can rely on replicas only to monitor throughput — this is the case with protocols that have an explicit commit phase among replicas, as in PBFT and Aardvark.

received a request req , no single client issues feedback for two requests received after req (which might be a sign that one of the replicas is unfair and delays req). If a replica detects that this is not the case, it stops processing requests and triggers a protocol switching (see Principle P4 implementation below for details on switching). Observe that replicas in *Quorum* receive all requests directly from clients and can thus accurately track requests. This is not the case for *Chain*: the head can forward requests in an arbitrary order, thus preventing accurate tracking. The only way to make sure that replicas accurately track requests is to have clients send feedback messages to indicate they issued some requests. To limit the overhead of these messages in *Chain*, they are piggybacked to feedback messages containing information about committed requests.

Finally, in order to ensure that the time needed to switch between the three protocols is not impacted by Byzantine clients and/or Byzantine replicas (Principle P4), *R-Aliph* relies on three main ideas.

- (a) First, each replica bounds the number of uncheckpointed requests it adds to its local history. This in turn bounds the amount of state that might need to be transferred during the switching. In our implementation, this bound is set to 384 requests, which is not limiting performance (i.e., setting this bound to a higher value does not increase the peak throughput achieved by *R-Aliph*).
- (b) Second, *R-Aliph* leverages the isolation of networking and processing resources that Aardvark (that *R-Aliph* uses in *Backup*) anyway requires. More precisely, as illustrated in Figure 16, (i) each replica uses a dedicated NIC to communicate with clients – note that this particular NIC is also used in *Chain* to exchange common-case protocol messages among replicas, (ii) each replica uses a dedicated set of NICs to communicate with other replicas, (iii) each replica picks messages on the set of NICs dedicated to other replicas in a round-robin manner, and (iv) each replica disables a NIC dedicated to communicating with another replica if the latter sends invalid messages or much more messages than the other replicas on average. This combination of mechanisms ensures a robust implementation of point-to-point channels between each pair of replicas: i.e., even if a Byzantine client or a Byzantine replica floods the network, this will not prevent any pair of correct replicas to communicate efficiently.
- (c) Third, clients are *not* involved in the switching protocol (although they can panic if they want) — otherwise, since a replica is connected to all clients using a single NIC, a Byzantine client could perform a denial of service attack and arbitrarily delay the protocol switching. This is achieved as follows. A replica wishing to trigger protocol switching (because it detects that *Quorum* or *Chain* either do not sustain adequate throughput or are unfair, or if some client panics) acts itself as a client: it invokes a *noop* request on the current Abstract (i.e., *Quorum* or *Chain*) and immediately panics (without waiting for the timer in Steps Q1/C1). The replica then completes Step P3 of the panicking/aborting subprotocol and switches, acting as a client, to the next Abstract instance. Then, since (i) the size of local histories is limited, (ii) clients are not involved in the aborting protocol (beyond the ability to panic), and (iii) each pair of replicas is connected by a dedicated point-to-point channel, we are guaranteed that Byzantine clients and/or Byzantine replicas cannot impact the upper bound on time required to perform a protocol switching.

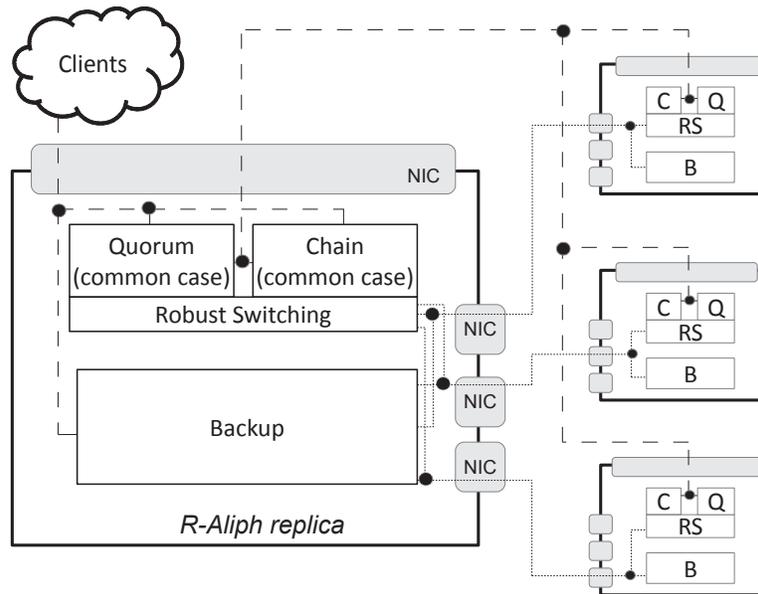


Fig. 16. Architecture of *R-Aliph* replicas.

6.4. Evaluation

In this section, we evaluate *R-Aliph*. We first assess its overhead with respect to *Aliph*. We then study its behavior under attacks. Finally, we assess the worst case switching time under attack.

6.4.1. Overhead of *R-Aliph*. *R-Aliph* has an overhead that is mainly caused by feedback messages sent by clients to notify replicas of the *Chain* protocol that they sent and committed requests. To assess this overhead, we run an experiment without attacks and compare the throughput of *R-Aliph* to that achieved by *Aliph*. We vary the request size from 0kB to 10kB and we use null replies (we obtained comparable results with non-null reply sizes). Results are reported in Figure 17. We can make two observations. First, we observe that the maximum throughput decrease is below 6%. This is very reasonable: in “common” case, *R-Aliph* is still 65% more efficient than Aardvark. Second, we observe that the overhead of *R-Aliph* decreases when the size of requests increases. For instance, with 4kB requests, the throughput decrease of *R-Aliph* with respect to *Aliph* is below 3%. The reason why the overhead decreases when increasing the request size is that the relative size of feedback client messages becomes lower.

6.4.2. Behavior of *R-Aliph* under attack. We assessed the performance of *R-Aliph* under the attacks described in Section 6.1. Figure 18 presents the behavior of *R-Aliph* during the “processing delay” attack. We have observed similar behaviors for other attacks. Therefore, to avoid redundancy, we do not provide figures for other attacks. On the X axis, we report the time (in seconds). On the Y axis, we report both the throughput that was expected by replicas (dashed line) and the throughput that *R-Aliph* actually sustained (solid line). The dotted vertical lines represent protocol switchings. The experiment is as follows. 100 clients inject 8B requests in a closed-loop manner and receive 8B replies during all the experiment. Initially, *R-Aliph* executes *Backup*,

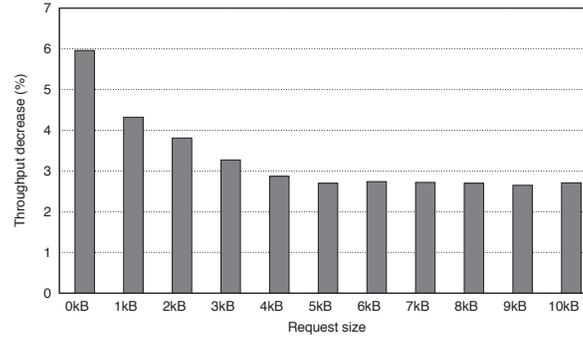


Fig. 17. Throughput decrease of *R-Aliph* with respect to *Aliph* for various request sizes and null replies.

configured to execute a fixed amount of requests. There is no ongoing attack yet and *Backup* sustains a throughput of about 31500 *req/s*, which is in line with what we previously reported in Table IV. Then, at time 55s, *R-Aliph* switches to *Quorum*. As there is contention, *Quorum* is not able to commit requests and immediately aborts. Subsequently, *R-Aliph* switches to *Chain*. The latter sustains a much higher throughput than *Backup* (and thus a much higher throughput than that expected by replicas). At time 114s, the attack starts. The replica acting as head in *Chain* adds a 10ms processing delay to all messages it receives. We have profiled the system and observed that a correct replica in the *Chain* starts noticing that *Chain* is not behaving properly and triggers a switching about 7ms after the attack started. Again, system profiling shows that it takes about 63ms for *Chain* to switch to *Backup*. The latter executes requests despite the attack. Its throughput is slightly impacted (-21% on average). Indeed, we can observe periodic performance drops that are explained by the fact that the Aardvark protocol used in *Backup* regularly changes the primary replica and one fourth of the time, the elected primary is the Byzantine replica that adds a 10ms processing delay. This induces a short performance drop before a new primary is elected. At time 187s, after *Backup* executed a fixed amount of requests, *R-Aliph* switches to *Quorum*. The latter is not able to commit requests (because of contention). *R-Aliph* subsequently switches to *Chain* that does not sustain the required throughput (because of the attack). Profiling reveals that it again takes about 5ms for a correct replica in the *Chain* to notice that *Chain* is slow and about 20ms to switch to *Backup*. Note that switching at around 187s is faster compared to the switching at around 114s when the attack was initially launched. Profiling reveals that this is due to the fact that the history of replicas contains a much smaller amount of requests.

6.4.3. Worst case switching time. In this section, we assess the worst case switching time both without attacks and under the different attacks described in Section 6.1. The time we measure corresponds to the elapsed time between the creation of the first panicking message and the time when the replicas have switched to the next protocol. To assess the worst case switching time, we perform several switching in a loop, as explained in Section 4.6, with half of the replicas having a full request history (384 10kB requests) and half of the replicas having an empty request history. This configuration induces the largest possible state transfer among replicas. Results are reported in Table V. We observe that the worst case switching time is low and that it is only marginally impacted in the presence of attacks. This is easily explained by the fact that (i) the processing and networking resources needed by the switching mechanism are isolated, and (ii) clients are not involved in the switching protocol. Therefore, correct replicas

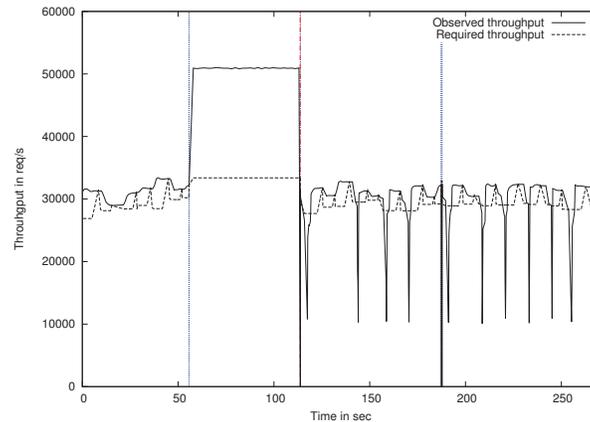


Fig. 18. Behavior of *R-Aliph* under a “processing delay” attack.

can perform switching without being impacted by the presence of Byzantine replicas and clients.

Table V. Worst case switching time (in ms) of *R-Aliph* both in the absence of attacks and under various attacks.

Without attack	Client flooding	Replica is unfair	Processing delay (10ms)	Replica flooding
60.36	62.49	60.52	63.92	63.24

7. RELATED WORK

The idea of aborting if “something goes wrong” is old. It underlies for instance the seminal two-phase commit protocol [Gray 1978]: 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 [Jayanti 2003]. Abortable consensus was proposed in [Chen 2007] and [Boichat et al. 2003]. In the first case, a process can abort if a majority of processes cannot be reached whereas, in the second case, a process can abort if there is contention. The latter idea was generalized for arbitrary shared objects in [Attiya et al. 2005] and then [Aguilera et al. 2007]. In [Aguilera et al. 2007], 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 be used to invoke another, possibly stronger, Abstract. This ability is key to the composability of Abstract instances.

Several examples of speculative protocols, distinguishing an optimistic phase from a recovery one, were discussed in the survey of Pedone [Pedone 2001]. These speculation ideas were used in the context of Byzantine state machine replication, e.g., in HQ [Cowling et al. 2006] and Zyzzyva [Kotla et al. 2010]. We are however 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. In a sense, Abstract enables the construction of a BFT protocol as the composition of as many (gracefully degrading) phases as desired, each with a “standard” interface. This allows for an unprecedented flexibility in BFT protocol design that we illustrated with *Aliph*, a BFT protocol that combines three different phases. Similarly, with *R-Aliph* we illustrated how one can quickly devise and implement a BFT protocol ensuring good performance despite attacks. While we described *Aliph* and *R-Aliph* and showed that, albeit simple, they outperform existing BFT protocols, *Aliph* and *R-Aliph* are simply the starting point for Abstract.

To maintain the assumption of a threshold f of replica failures realistic, BFT systems need to ensure failure independence [Gashi et al. 2007; Garcia et al. 2011]. 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 [Castro et al. 2003], that enable reuse of off-the-shelf service implementations, can be used complementarily to our approach.

Since the publication of the preliminary, conference version of this work [Guerraoui et al. 2010], several papers that exploit Abstract-like reconfiguration have been published. In particular, CheapBFT [Kapitza et al. 2012] implements Abstract-like reconfiguration with switching through replicas assuming an FPGA-based trusted subsystem to reduce the resource overhead of BFT. In the common case, CheapBFT relies on a novel, optimistic protocol that uses $f + 1$ replicas whereas, in the case of failures, CheapBFT falls back to a protocol called MinBFT [Veronese et al. 2013] (itself also based on a trusted subsystem) that uses $2f + 1$ replicas. It is precisely for such usages demonstrated by CheapBFT, that diverge from the classical BFT model we assume in this paper, that we defined Abstract as a specification with properties that reason about clients’ histories rather than about the state and the number of replicas that implement Abstract. Indeed, Abstract specification deliberately does not reason about the number of replicas, their local state or state relative to other replicas, nor about the fault model. By adopting such an approach, we do not restrict the use of Abstract to the classical BFT model, but also allow for the use of Abstract in the trusted BFT model [Kapitza et al. 2012], vanilla crash-failure model, or any other failure model.

8. CONCLUSION AND FUTURE WORK

Byzantine fault tolerant state machine replication (BFT) protocols are notoriously difficult to design, implement and prove correct. In this paper, we presented Abstract, a framework for the design and reconfiguration of *abortable* replicated state machines. Using Abstract, we incrementally developed new BFT protocols with a fraction of the complexity required to develop full-fledged BFT protocols. We build BFT protocols as sequences of Abstract instances, each designed, implemented and proved independently. Such protocols are not only simpler to design but also efficient and robust.

In this paper, we have implemented several BFT protocols (*AZyzyva*, *Aliph*, and *R-Aliph*) that consist of lightweight Abstract implementations (*ZLight*, *Quorum* and *Chain*) designed to be efficient in the “common” case, typically when the system is synchronous and there are no replica faults. In all protocols we reused existing BFT protocols such as PBFT [Castro and Liskov 2002] and Aardvark [Clement et al. 2009] to handle the faulty case, by wrapping them into a powerful Abstract instance called *Backup*.

In future work, several directions can be interesting to explore, e.g., using the concepts that underly Abstract in the context of Byzantine-resilient storage [Hendricks et al. 2007], or devising signature-free switching. Moreover, we believe that an interesting research challenge lies in devising effective heuristics for dynamic switching among Abstract instances. While we described *Aliph* and *R-Aliph*, and showed that, albeit simple, they outperform existing BFT protocols, *Aliph* and *R-Aliph* are simply the starting point for Abstract. The idea of dynamic switching depending on the system conditions seems very promising; such a scheme could monitor the current system conditions and implement smart heuristics to switch to the seemingly most appropriate Abstract instance.

References

- Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM.
- Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. 2007. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the ACM Symposium on Principles of distributed computing (PODC)*.
- Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication under Attack. *IEEE Trans. Dependable Sec. Comput.* 8, 4 (2011), 564–577.
- Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. 2005. Computing with Reads and Writes in the Absence of Step Contention. In *Proceedings of the International Conference on Distributed Computing (DISC)*.
- Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. 2010. *Virtually Synchronous Methodology for Dynamic Service Replication*. Technical Report MSR-TR-2010-151.
- Romain Boichat, Partha Dutta, Svend Frölund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *SIGACT News in Distributed Computing* 34, 1 (2003), 47–67. DOI: <http://dx.doi.org/10.1145/637437.637447>
- Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. 2001. Consensus in One Communication Step. In *Proceedings of the International Conference on Parallel Computing Technologies (PaCT)*.
- Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. DOI: <http://dx.doi.org/10.1145/571637.571640>
- Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. 2003. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems* 21 (August 2003), 236–269. Issue 3. DOI: <http://dx.doi.org/10.1145/859716.859718>
- Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM. DOI: <http://dx.doi.org/10.1145/1281100.1281103>
- Wei Chen. 2007. *Abortable consensus and its application to probabilistic atomic broadcast*. Technical Report MSR-TR-2006-135.
- Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*.
- James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association. <http://portal.acm.org/citation.cfm?id=1298455.1298473>
- Dan Dobre and Neeraj Suri. 2006. One-step Consensus with Zero-Degradation. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*.
- Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35 (April 1988), 36. Issue 2. DOI: <http://dx.doi.org/10.1145/42282.42283>
- Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382.
- Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2011. OS diversity for intrusion tolerance: Myth or reality?. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN '11)*. IEEE Computer Society, Washington, DC, USA, 383–394. DOI: <http://dx.doi.org/10.1109/DSN.2011.5958251>

- Illir Gashi, Peter T. Popov, and Lorenzo Strigini. 2007. Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers. *IEEE Trans. Dependable Sec. Comput.* 4, 4 (2007), 280–294.
- Jim Gray. 1978. Notes on Database Operating Systems. In *Operating Systems — An Advanced Course*.
- Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2008. *The Next 700 BFT Protocols*. Technical Report LPD-REPORT-2008-008. EPFL.
- Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The Next 700 BFT Protocols. In *Proceedings of the ACM European conference on Computer systems (EuroSys)*. <http://eurosys2010.sigops-france.fr/>
- James Hendricks, Gregory R. Ganger, and Michael K. Reiter. 2007. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM.
- Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
- Prasad Jayanti. 2003. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*.
- Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 295–308. DOI: <http://dx.doi.org/10.1145/2168836.2168866>
- Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.* 27, 4, Article 7 (Jan. 2010), 39 pages. DOI: <http://dx.doi.org/10.1145/1658357.1658358>
- Leslie Lamport. 2003. Lower Bounds for Asynchronous Consensus. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*.
- Leslie Lamport. 2009. The PlusCal Algorithm Language. In *ICTAC*. 36–60.
- Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a state machine. *SIGACT News* 41, 1 (2010), 63–73.
- Fernando Pedone. 2001. Boosting System Performance with Optimistic Distributed Protocols. *Comput. J.* 34, 12 (2001), 80–86. DOI: <http://dx.doi.org/10.1109/2.970581>
- Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. DOI: <http://dx.doi.org/10.1145/98163.98167>
- Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus closed: a cautionary tale. In *NSDI*. 18–18.
- Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT protocols under fire. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.
- Sam Toueg. 1984. Randomized Byzantine Agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*. 163–178.
- Robbert van Renesse and Rachid Guerraoui. 2010. Replication Techniques for Availability. In *Replication*. 19–40.
- Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.
- Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One’s Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society. DOI: <http://dx.doi.org/10.1109/SRDS.2009.36>
- Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Computers* 62, 1 (2013), 16–30.

Received May 2012; revised February 2014; accepted July 2014

A. CORRECTNESS PROOFS

In this Appendix, we give the correctness proofs of *ZLight* (Sec. 4.2), *Quorum* (Sec. 5.2), and *Chain* (Sec. 5.3). We omit the correctness proof of *Backup* (Sec. 4.3) which is straightforward due to the properties of the underlying BFT protocol. Finally, the proofs of liveness of our compositions trivially rely on the assumption of an exponentially increasing *Backup* configuration parameter k (see Sec. 4.3).

Since *ZLight* and *Quorum* share many similarities, we give their correctness proof together. This is followed by the proof of *Chain*.

A.1. ZLight and Quorum

In this Section, we prove that *ZLight* and *Quorum* implements Abstract. We first prove the common properties of the two implementations and then focus on the only different property (Progress).

Well-formed commit indications. It is easy to see that the reply returned by a commit indication for a request req always equals $rep(h_{req})$, where (commit history) h_{req} is a uniquely defined sequence of requests. Indeed, by Step Z4/Q3, in order to commit a request, a client needs to receive identical digests (*LHDigest*) of some history h' and identical reply digests from all $3f + 1$ replicas, including from at least $2f + 1$ correct replicas. By Step Z3 of *ZLight* (resp., Step Q2 of *Quorum*), a digest of the reply sent by a correct replica is $D(rep(h'))$. Hence, h' is exactly the commit history h_{req} and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica logs and then executes an invoked request req before sending a RESP message in Step Z4 (resp., Q3), it is straightforward to see that if req is committed with a commit history h_{req} , then req is in h_{req} . \square

Validity. For any request req to appear in a commit or abort history, at least $f + 1$ replicas must have sent a history (or a digest of a history) containing req to the client (see Step Z4/Q3 for commit histories, and Step P3 for abort histories). Hence, at least one correct replica appended req to its local history. By Step Z3/Q2, the correct replica r_j appends req to its local history only if r_j receives a REQ message with a valid MAC from a client. This MAC is, in turn, present only if some client invoked req , or if req is contained in some verified (valid) init history.

Moreover, by Step Z3/Q2, no replica logs/executes the same request twice (since every replica maintains $t_j[c]$). Hence, no request appears twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step P3 Sec. 4.2.2). \square

Termination. By assumption of a quorum of $2f + 1$ correct replicas and fair-loss links: since correct clients (resp., replicas) periodically retransmit the PANIC (resp., ABORT) messages (Step P1): (1) correct replicas eventually receive the PANIC message sent by correct client c and (2) c eventually receives $2f + 1$ ABORT messages from correct replicas (sent in Step P2). Hence, if correct client c panics, it eventually aborts invoked request req , in case c does not commit req beforehand.

To prove Commit and Abort Ordering we first prove the following Lemma.

LEMMA A.1. *Let r_j be a correct replica and LH_j^{req} the state of LH_j upon r_j logs req . Then, LH_j^{req} remains a prefix of LH_j forever.*

PROOF. A correct replica r_j modifies its local history LH_j only in Step Z3/Q2 by sequentially appending requests to LH_j . Hence, LH_j^{req} remains a prefix of LH_j forever. \square

Commit Order. Assume, by contradiction, that there are two different committed requests req (by benign client c) and req' (by benign client c'), with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. Since a benign client commits a request only if it receives in Step Z4/Q3 identical digests of replicas' local histories from all $3f+1$ replicas, there must be a correct replica r_j that sent $D(h_{req})$ to c and $D(h_{req'})$ to c' such that $h(req)$ is not a prefix of $h_{req'}$ nor vice versa. A contradiction with Lemma A.1. \square

Abort Order. First, we show that for every committed request req with the commit history h_{req} and any ABORT message m sent by a correct replica r_j containing a (digest of a) local history LH_j^m , h_{req} is a prefix of LH_j^m . Assume, by contradiction, that there are request req' , correct replica $r_{j'}$ and ABORT message m' such that the above does not hold. Then, since a benign client needs to receive identical history digests from all replicas to commit a request (Step Z4/Q3), and since $r_{j'}$ stops executing new requests before sending any ABORT message (Step P2), $r_{j'}$ logged and executed req before sending m' . However, by Lemma A.1, $h_{req'}$ is a prefix of $LH_{j'}^{m'}$ — a contradiction.

By Step P3, a client that aborts a request waits for $2f+1$ ABORT messages including at least $f+1$ from correct replicas. Since any commit history h_{req} is a prefix of every history sent in an ABORT message by any correct replica (as shown above), at least $f+1$ received histories will contain h_{req} as a prefix, for any committed request req . Hence, by construction of abort histories (Step P3 Sec. 4.2.2) every commit history h_{req} is a prefix of every abort history. \square

Init Order. By Step Z3+, Sec. 4.2.3,¹⁷ and Step P2, every correct replica must initialize its local history (with some valid init history) before sending any RESP or ABORT message. Since any common prefix CP of all valid init histories is a prefix of any particular init history I , CP is a prefix of every local history sent by a correct replica in an RESP or ABORT message. Init Order for commit histories immediately follows. In the case of abort histories, notice that out of $2f+1$ ABORT messages received by a client on aborting a request in Step P3, at least $f+1$ are sent by correct replicas and contain local histories that have CP as a prefix. Hence, by Step P3, CP is a prefix of any abort history. \square

ZLight Progress. Recall that *ZLight* guarantees to commit clients' requests if: there are no replica/link failures and Byzantine client failures. Recall also that, with no link failures, message propagation time between two correct processes is bounded by Δ_c and that message processing time is bounded by Δ_p . Also, in Step Z1, a client triggers a timer T set to 3Δ (where $\Delta = \Delta_c + \Delta_p$). Then, to prove Progress, we prove a stronger property that no client executes Step P1 and panics (consequently no client ever aborts and Progress follows from Termination).

Assume by contradiction that there is a client c that panics and denote the first such time by t_{PANIC} . Since no client is Byzantine, c must be benign and c invoked request req at $t = t_{PANIC} - 3\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC} . Then, it is not difficult to see, since there are no link failures,

¹⁷Notice that, for *Quorum*, Step Z3+ defines additional actions performed during Step Q2.

that: (i) by $t + \Delta_c$ the primary receives req and by time $t + \Delta$ primary sends ORDER messages in Step Z2 and (ii) by time $t + 2\Delta < t_{PANIC}$ the replicas send RESP messages in Step Z3 for req . Since the primary is correct all replicas execute all requests received before t_{PANIC} in the same order (established by the sequence numbers assigned by the primary). Hence, by $t + 3\Delta = t_{PANIC}$, c receives and processes $3f + 1$ identical replies (Step Z4), commits req and never panics. A contradiction. \square

Quorum Progress. Recall that *Quorum* guarantees to commit clients' requests only if:

- there are no replica/link failures,
- no client is Byzantine, and
- there is no contention.

We assume that the timer T triggered in Step Q1 is set to 2Δ (where $\Delta = \Delta_c + \Delta_p$). Like in the proof of ZLight Progress, we prove a stronger property that no client executes Step P1 and panics.

Assume by contradiction that there is a client c that panics and denote the first such time by t_{PANIC} . Since no client is Byzantine, c must be benign and c invoked request req at $t = t_{PANIC} - 2\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC} . Then, it is not difficult to see, since there are no link failures, that by time $t + \Delta < t_{PANIC}$ all replicas receive req and send RESP message in Step Q2. Since there is no contention and all replicas are correct, all replicas order all requests in the same way and send identical histories to the clients. Hence, by $t + 2\Delta = t_{PANIC}$, c receives and processes $3f + 1$ identical replies (Step Q3), commits req and never panics. A contradiction. \square

A.2. Chain

In this Section, we prove that *Chain* implements Abstract with Progress equivalent to that of ZLight (see also Appendix A.1).

We denote the *predecessor* (resp., *successor*) set of the replica r_j , by \overleftarrow{R}_j (resp., \overrightarrow{R}_j). We also denote by Σ_{last} the set of the last $f + 1$ replicas in the chain order, i.e., $\Sigma_{last} = \{r_j \in \Sigma : j > 2t\}$. In addition, we say that correct replica r_j logs (resp., executes) req at position pos if $sn_j = pos$ when r_j logs/executes req .

Before proving Abstract properties, we first prove two auxiliary lemmas. Notice also that Lemma A.1, Section A.1, extends to *Chain* as well.

LEMMA A.2. *If correct replica r_j logs req (at position sn , at time t_1), then all correct replicas s_j , $1 \leq j < i$ log req (at position sn , before t_1).*

PROOF. By contradiction, assume the lemma does not hold and fix r_j to be the first correct replica that logs req (at position sn), such that there is a correct replica r_x ($x < j$) that never logs req (at position sn); we say r_j is the first replica for which req skips. Since CHAIN messages are authenticated using CAs, r_j logs req at position sn only if r_j receives a CHAIN message with MACs authenticating req and sn from all replicas from \overleftarrow{R}_j authenticate req and sn , i.e., only after all correct replicas from \overleftarrow{R}_j log req at position sn . If $r_x \in \overleftarrow{R}_j$, r_x must have logged req at position sn — a contradiction. On the other hand, if $r_x \notin \overleftarrow{R}_j$, then r_j is not the first replica for which req skips, since req skips for any correct replica (at least one) from \overleftarrow{R}_j — a contradiction. \square

LEMMA A.3. *If benign client c commits req with history h_{req} (at time t_1), then all correct replicas in Σ_{last} execute req (before t_1) and the state of their local history upon executing req is h_{req} .*

PROOF. To prove this lemma, notice that correct replica $r_j \in \Sigma_{last}$ generates a MAC for the client authenticating req and $D(h')$ for some history h' (Step C3): (1) only after r_j logs and executes req and (2) only if the state of LH_j upon execution of req equals h' . Moreover, by Step C3, no correct replica executes the same request twice. By Step C4, a benign client cannot commit req with h_{req} unless it receives a MAC authenticating req and $D(h')$ from every correct replica in Σ_{last} . Hence the lemma. \square

Well-formed commit indications. By Step C4, in order to commit a request req a client needs to receive MACs authenticating $LHDigest = D(h')$ for some history h' and the reply digest from all replicas from Σ_{last} , including at least one correct replica. By Step C3, a digest of the reply sent by a correct replica is $D(rep(h'))$. Hence, h' is exactly a commit history h_{req} and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica in Σ_{last} logs and executes an invoked request before sending a CHAIN message in Step C3, it is straightforward to see that if req is committed with a commit history h_{req} , then req is in h_{req} . Namely, notice that a client needs to receive the MAC for the same local history digest $D(h_{req})$ from all $f + 1$ replicas from Σ_{last} including at least one correct replica r_j . By Step C3, r_j logs req and appends it to its local history LH_j before authenticating the digest of LH_j ; hence, $req \in h_{req}$. \square

Validity. For any request req to appear in an abort (resp., commit) history h , at least $f + 1$ replicas must have sent h (resp., a digest of h) in Step P2 (resp., Step C3) such that $req \in h$. Hence, at least one correct replica logged req .

Now, we show that correct replicas log only requests invoked by clients. By contradiction, assume that some correct replica logged a request not invoked by any client and let r_j be the first correct replica to log such a request req' in Step C3 of *Chain*. In case $j < f + 1$, r_j logs req' only if r_j receives a CHAIN message with a MAC from the client, i.e., only if some client invoked req , or if req is contained in some valid init history. On the other hand, if $j > f + 1$, Lemma A.2 yields a contradiction with our assumption that r_j is the first correct replica to log req' .

Moreover, by Step C3, no replica logs the same request twice (every replica maintains $t_j[c]$). Hence, no request appears twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction (see Step P3 Sec. 4.2.2). \square

Termination. Since *Chain* uses the same panicking/aborting mechanism as *Quorum/ZLight*, the proof of Termination for *ZLight/Quorum* (Sec. A.1) applies. \square

Commit Order. Assume, by contradiction, that there are two different committed request req (by benign client c) and req' (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. By Lemma A.3, there is correct replica $r_j \in \Sigma_{last}$ that logged and executed req and req' such that the state of LH_j upon executing these requests is h_{req} and $h_{req'}$, respectively. A contradiction with Lemma A.1 (recall that this lemma extends to *Chain* as well). \square

Abort Order. Assume, by contradiction, that there is committed request req_C (by some benign client) with commit history h_{req_C} and aborted request req_A (by some benign client) with abort history h_{req_A} , such that h_{req_C} is not a prefix of h_{req_A} . By Lemma A.3 and the assumption of at most f faulty replicas, all correct replicas (at least one) from Σ_{last} log and execute req_C and their state upon executing req_C is h_{req_C} . Let $r_j \in \Sigma_{last}$ be a correct replica with the highest index j among all correct replicas in

Σ_{last} . By Lemma A.2, all correct replicas log all the requests in h_{req_C} at the same positions these requests have in h_{req_C} . In addition, a correct replica logs all the requests belonging to h_{req_C} before sending any ABORT message in Step P2; indeed, before sending any ABORT message, a correct replica must stop further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an ABORT message, h_{req_C} is a prefix of LH_j .

Finally, by Step P3, a client that aborts a request waits for $2f + 1$ ABORT messages including at least $f + 1$ from correct replicas. By construction of abort histories (Step P3), every commit history, including h_{req_C} is a prefix of every abort history, including h_{req_A} , a contradiction. \square

Init Order. The proof is identical to the proof of *ZLight/Quorum* Init Order. \square

Progress (sketch). *Chain* guarantees to commit clients' requests under the same conditions as *ZLight*, i.e., if there are no replica/link failures and no Byzantine client failures. Assuming that the message processing at processes takes Δ_p , and communication time is bounded by Δ_c , it is sufficient that clients set the timer T triggered in Step C1 to $(3f + 2)\Delta$, where $\Delta = \Delta_p + \Delta_c$. Then, Progress of *Chain* is very simple to show, along the lines of *ZLight* Progress (Sec. A.1). \square