The Next 700 BFT Protocols

Rachid Guerraoui,
Nikola Knežević

EPFL

rachid.guerraoui@epfl.ch,
nikola.knezevic@epfl.ch

Vivien Quéma

CNRS

vivien.quema@inria.fr

Marko Vukolić

IBM Research - Zurich

mvu@zurich.ibm.com

Abstract

Modern Byzantine fault-tolerant state machine replication (BFT) protocols involve about 20.000 lines of challenging C++ code encompassing synchronization, networking and cryptography. They are notoriously difficult to develop, test and prove. We present a new abstraction to simplify these tasks. We treat a BFT protocol as a composition of instances of our abstraction. Each instance is developed and analyzed independently.

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

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

1. Introduction

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

despite contention and failures, replicas perform client requests in the same order.

Two objectives underly the design and implementation of a SMR protocol: *robustness* and *performance*. Robustness conveys the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures and asynchrony. On the other hand, performance measures the time it takes to respond to a request (latency) and the number of requests that can be processed per time unit (throughput). The most robust protocols are those that tolerate (a) arbitrarily large periods of asynchrony, during which communication delays and process relative speeds are unbounded, and (b) arbitrary (Byzantine) failures of any client as well as up to one-third of the replicas (this is the theoretical lower bound [17]). These are called Byzantine-Fault-Tolerance SMR protocols, or simply *BFT* protocols, e.g., PBFT, QU, HQ and Zyzzyva [1, 6, 11, 15]. The ultimate goal of the designer of a BFT protocol is to exhibit comparable performance to a non-replicated server under "common" circumstances that are considered the most frequent in practice. The notion of "common" circumstance might depend on the application and underlying network, but it usually means network synchrony, rare failures, and sometimes also the absence of contention.

Not surprisingly, even under the same notion of "common" case, there is no "one size that fits all" BFT protocol. According to our own experience, the performance differences among the protocols can be heavily impacted by the actual network, the size of the messages, the very nature of the "common" case (e.g, contention or not); the actual number of clients, the total number of replicas as well as the cost of the cryptographic libraries being used. This echoes [20] which concluded for instance that *"PBFT [6] offers more predictable performance and scales better with payload size compared to Zyzzyva [15]; 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 an example of a such protocol in this paper.

To deploy a BFT solution, a system designer will hence certainly be tempted to adapt a state-of-the-art BFT protocol to the specific application/network setting, and possibly keep adapting it whenever the setting changes. But this can rapidly turn into a nightmare. All protocol implementations we looked at involve around 20.000 lines of (nontrivial) C++ code, e.g., PBFT and Zyzzyva. Any change to an existing protocol, although algorithmically intuitive, is very painful. The changes of the protocol needed to optimize for the "common" case have sometimes strong impacts on the part of the protocol used in other cases (e.g., "view-change" in Zyzzyva). The only complete proof of a BFT protocol we knew of is that of PBFT and it involves 35 pages (even without using any formal language).[1] This difficulty, together with the impossibility of exhaustively testing distributed protocols [7] 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 *A*bortable Byzantine faulT-toleRant stAte maChine replicaTion (we simply write Abstract): a new abstraction to reduce the development cost of BFT protocols. Following the divide-and-conquer principle, we view BFT protocols as a composition of instances of our abstraction, each instance targeted and optimized for specific system conditions. An instance of Abstract looks like BFT state machine replication, with one exception: it may sometimes *abort* a client's request.

The progress condition under which an Abstract instance should not abort is a generic parameter.[2] An extreme instance of Abstract is one that never aborts: this is exactly BFT. Interesting instances are "weaker" ones, in which an abort is allowed, e.g., if there is asynchrony or failures (or even contention). When such an instance aborts a client request, it returns 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. Indeed, the composition of any two Abstract instances is *idempotent*, yielding yet another Abstract instance. Hence, and as we will illustrate in the paper, the development (design, test, proof and implementation) of a BFT protocol boils down to:

- Developing individual Abstract instances. This is usually way much simpler than developing a full-fledged BFT protocol and allows for very effective schemes. A single

Abstract instance can be crafted solely with its progress in mind, irrespective of other instances.

- Ensuring that a request is not aborted by all instances. This can be made very simple by reusing, as a black-box, an existing BFT protocol as one of the instances, without indulging into complex modifications.

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

1. *AZyzzyva*, a protocol that illustrates the ability of Abstract to significantly ease the development of BFT protocols. *AZyzzyva* is the composition of two Abstract instances: (i) *ZLight*, which mimics Zyzzyva [15] 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, but could replace it with any BFT protocol. The code line count and proof size required to obtain *AZyzzyva* are, conservatively, less than $1/4$ than those of Zyzzyva. In some sense, had Abstract been identified several years ago, the designers of Zyzzyva would have had a much easier task devising a correct protocol exhibiting the performance they were seeking. Instead, they had to hack PBFT and, as a result, obtained a protocol that is way less stable than PBFT.

2. *Aliph*, a protocol that demonstrates the ability of Abstract to develop novel efficient BFT protocols. *Aliph* achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. *Aliph* uses, besides the *Backup* instance used in *AZyzzyva* (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 failures/asynchrony. *Quorum* has a very low-latency (like e.g., [1, 5, 12]) 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 [15]. Interestingly, each of *Quorum* and *Chain* could be developed independently and required less than 25% of the code needed to develop state-of-the-art BFT protocols.[3]

The rest of the paper is organized as follows. Section 2 presents Abstract. After describing our system model in Section 3, we describe and evaluate our new BFT protocols: *AZyzzyva* in Section 4 and *Aliph* in Section 5. Sec-

---

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

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

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

tion 6 discusses the related work and concludes the paper. For better readability, details are postponed to appendices. Appendix A contains the formal specification of Abstract the details on model checking Abstract idempotency using TLA+ tools [16]. Appendix B contains protocol details with the correctness proofs given separately in Appendix C.

2. Abstract

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

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

2. BFT is nothing but a special Abstract instance — *one that never aborts*.

A correct implementation of an Abstract instance always preserves BFT safety — this extends to any composition thereof. The designer of a BFT protocol only has to make sure that: a) individual Abstract implementations are correct, *irrespectively of each other*, and b) the composition of the chosen instances is live: i.e. that every request will eventually be committed. We exemplify this later, in Sections 4 and 5. In the following, we highlight the main characteristics of Abstract. For better readability, precise specification of Abstract, our theorem on Abstract switching idempotency and model checking details are postponed to Appendix A.

Switching. Every Abstract instance has a unique identifier (instance number) i. When an instance i commits a request, i returns a state-machine reply to the invoking client. Like any SMR scheme, i establishes a total order on all committed requests according to which the reply is computed for the client. If, however, i aborts a request, it returns to the client a digest of the history of requests h that were committed by i (possibly along with some uncommitted requests); this is called an *abort history*. In addition, i returns to the client the identifier of the next instance ($next(i)$) which should be invoked by the client: $next$ is the same function across all abort indications of instance i, and we say instance i *switches* to instance $next(i)$. In the context of this paper, we consider $next$ to be a pre-determined function (e.g., known to servers implementing a given Abstract instance); we talk about *deterministic* or *static* switching. However, this is not
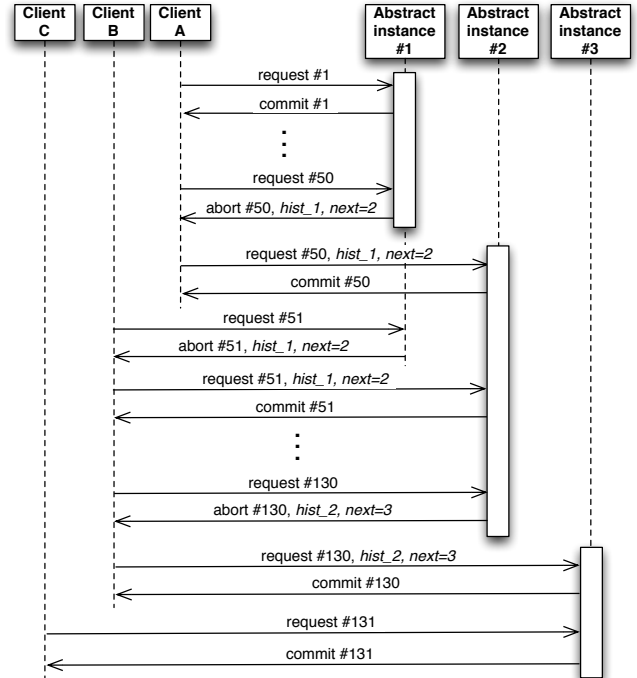

Figure 1. Abstract operating principle.

required by our specification; $next(i)$ can be computed "on-fly" by the Abstract implementation (e.g., depending on the current workload, or possible failures or asynchrony) as long as $next$ remains a function. In this case, we talk about *dynamic switching*; this is out of the scope of this paper.

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

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

Illustration. Figure 1 depicts a possible run of a BFT system built using Abstract. To preserve consistency, Abstract properties ensure that, at any point in time, only one Abstract instance, called *active*, may commit requests. Client A starts sending requests to the first Abstract instance. The latter commits requests #1 to #49 and aborts request #50, becoming inactive. Abstract appends to the abort indication an (unforgeable) history ($hist\_1$) and the information about the next Abstract instance to be used ($next = 2$). Client A sends to the new Abstract instance both its uncommitted request

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

A view-change perspective. In some sense, an Abstract instance number can be seen as a view number, e.g., in PBFT [6].[4] Like in existing BFT protocols, which merely reiterate the exact same sub-protocol across the views (possibly changing the server acting as *leader*), the same Abstract implementations can be re-used (with increasing instance numbers). However, unlike existing BFT protocols, Abstract compositions *allow entire sub-protocols to be changed on a 'view-change'* (i.e., during switching).

Misbehaving clients. Clients that fail to comply with the switching mechanism (e.g., by inventing/forging an init history) cannot violate the Abstract specification. Indeed, to be considered valid, an init history of $next(i)$ must be previously returned by the preceding Abstract i as an abort history. To enforce this causality, in practice, our Abstract compositions (see Sec. 4 and Sec. 5) rely on unforgeable digital signatures to authenticate abort histories in the presence of potentially Byzantine clients. View-change mechanisms employed in existing BFT protocols [6, 15], have similar requirements: they exchange digitally signed messages.

3. 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 $3f + 1$ 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 (i.e., when there are no link failures) any message m sent between two correct processes is delivered within a bounded delay Δ (known to sender and receiver) if the sender retransmits m until it is delivered.

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.

4. Putting Abstract to Work: AZyzzyva

We illustrate how Abstract significantly eases the design, implementation, and proof of BFT protocols with *AZyzzyva*. This is a full fledged BFT protocol that mimics Zyzzyva [15] in its "common case" (i.e., when there are no link or server failures). In "other cases" we rely on *Backup*, an Abstract implementation with strong progress guarantees that can be implemented on top of *any* existing BFT protocol. In our implementation, we chose PBFT [6] for it has been extensively tested and proved correct. We chose to mimic Zyzzyva, for it is known to be efficient, yet very difficult to implement [9]. Using Abstract, we had to write, prove and test less than 24% of the Zyzzyva code to obtain *AZyzzyva*.

In the "common case", Zyzzyva executes the fast speculative path depicted in Figure 2. A client sends a request to a designated server, called *primary* (r_1 in Fig. 2). The primary appends a sequence number to the request and broadcasts the request to all replicas. Each replica speculatively executes the request and sends a reply to the client. All messages in the above sequence are authenticated using MACs rather than (more expensive) digital signatures. The client commits the request if it receives the same reply from all $3f + 1$ replicas. Otherwise, Zyzzyva executes a second phase that aims at handling the case with link/server/client failures ("worst-case"). Roughly, this phase (that *AZyzzyva* avoids to mimic) consists of considerable modifications to PBFT [6], which arise from the "profound effects" [15], that the Zyzzyva "common-case" optimizations have on its "worst-case". The second phase is so complex that, as confessed by the authors themselves [9], it is not entirely implemented in the current Zyzzyva prototype. In fact, when this second phase is stressed, due to its complexity and the inherent bugs that it contains, the throughput of Zyzzyva drops to 0.

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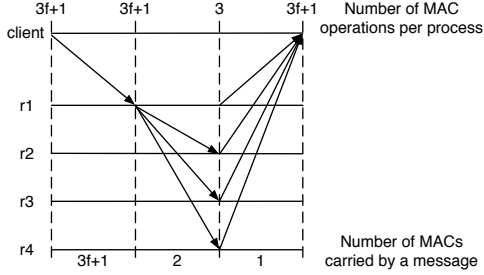
---

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

Figure 2. Communication pattern of *ZLight*.

4.1 Protocol overview

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

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

Note that *ZLight* uses a lightweight checkpoint protocol (shared with *Aliph*'s *Quorum* and *Chain*, Sec. 5) triggered every 128 messages to truncate histories (see Sec. B.7).

In the following, we briefly describe *ZLight* and *Backup*. Details are postponed to Appendix B, whereas correctness proofs can be found in Appendix C.

4.2 ZLight

ZLight implements Abstract with the following progress property which reflects Zyzzyva "common case": it commits requests when (a) there are no server or link failures, and (b) no client is Byzantine (simple client crash failures are tolerated). When this property holds, *ZLight* implements Zyzzyva "common-case" pattern (Fig. 2), described earlier. Outside the "common-case", when a client does not receive $3f + 1$ consistent replies, the client sends a PANIC message to replicas. Upon reception of this message, replicas stop executing requests and send back a signed message containing their history (replicas will now send the same abort message for all subsequent requests). When the client receives $2f + 1$ signed messages containing replica histories, it can generate an abort history and switch to *Backup*. Client generates abort history ah such that $ah[j]$ equals the value that appears at position $j \geq 1$ of $f + 1$ different replica histories (the details of the panic and switching mechanisms are in the appendix, in Sec. B.5 and B.6, respectively).

4.3 Backup

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

We implemented *Backup* as a very thin wrapper (around 600 lines of C++ code) that can be put around *any existing* BFT protocol. In our C/C++ implementations, *Backup* is implemented over PBFT [6], 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 [9], are also very good candidates for the *Backup* basis (unfortunately, the code of Aardvark is not yet publicly available).

To implement *Backup*, we exploit the fact that any BFT can totally order 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 *AZyzzyva*). At this point, *Backup* sets its state by executing all the requests contained in a valid init history it received. Then, it simply executes the first k requests ordered by BFT (neglecting subsequent init histories) and commits these requests. After committing the kth request, *Backup* aborts all subsequent requests returning the signed sequence of executed requests as the abort history (replica digital signature functionality assumed here is readily present in all existing BFT protocols we know of).

The parameter k is generic and is an integral part of the *Backup* progress guarantees. Our default configuration increases k exponentially, with every new instance of *Backup*. This ensures the liveness of the composition, which might not be the case with, say, a fixed k in a corner case with very slow clients.[5] 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. [6] In our implementation, we also periodically reset k. Dynami-

---

[5] In 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.

[6] For 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

cally adapting k to fit the system conditions is appealing but requires further studies and is out of the scope of this paper.

4.4 Qualitative assessment

In evaluating the effort of building *AZyzzyva*, we focus on the cost of *ZLight*. Indeed, *Backup*, for which the additional effort is small (around 600 lines of C++ code), can be reused for other BFT protocols in our framework. For instance, we use *Backup* in our *Aliph* protocol as well (Sec. 5).

Table 1 compares the number of pages of pseudo-code, pages of proofs and lines of code of Zyzzyva and *ZLight*. The comparison in terms of lines of code is fair, since *Zyzzyva* and all protocols presented in this paper use the same code base (inherited from PBFT [6]). Notice that, all implementations in the PBFT code base, share about 7500 lines of code implementing cryptographic functions, data structures (e.g. maps, sets), etc. We do not count these lines, which we packaged in a separate library. The code line comparison shows that to build *ZLight* we needed less than 24% of the *Zyzzyva* line count (14339 lines). This is, however, conservative since we needed only about 14% to implement *ZLight* "common case"; the remaining 10% (1391 lines) are due to panicking and checkpointing mechanisms, which are all shared among *ZLight*, *Quorum* and *Chain* (the latter two are used in *Aliph*, Sec. 5). The difference in the *ZLight* vs. *Zyzzyva* code size is that *ZLight* aborts as soon as the system conditions fall outside the "common-case" (in which case *AZyzzyva* shifts the load to *Backup*). Hence, we avoid the "common-case"/"worst-case" code dependency that plagued Zyzzyva.

Using the same syntax as the one used in the original Zyzzyva paper [15], *ZLight* requires approximately half a page of pseudo-code, its plain-english proof requires about 1 page (see Sec. C.1). In comparison, the pseudo-code of Zyzzyva (without checkpointing) requires 4.5 pages, making it about 9 times bigger than that of *ZLight*. Due to the complexity of Zyzzyva, the authors first presented a version using signatures and then explained how to modify it to use MACs. The correctness proof of the *Zyzzyva* signature version requires 4 (double-column) pages, whereas the proof for the MAC version is only sketched.

	Zyzzyva	ZLight
Pages of pseudo-code	4,5	0,5
Pages of proofs	> 4	1
Lines of code	14339	3358

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

4.5 Performance evaluation

We have compared the performance of AZyzzyva and Zyzzyva in the "common-case", using the benchmarks described in Section 5.2. Not surprisingly, AZyzzyva and

---

between 2 consecutive *Backup* instances in *AZyzzyva*, we can maintain $k = 1$ during $10s$ outages with $C = 2^{-200}$.

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

To assess the switching cost, we perform the following experiments: we feed the request history of *ZLight* with r requests of size $1kB$. We then issue 10000 successive requests. To isolate the cost of the switching mechanism, we do not execute the *ZLight* common case; the measured time comprises the time required (1) by the client to send a PANIC message to *ZLight* replicas, (2) by the replicas to generate and send a signed message containing their history, (3) by the client to invoke Backup with the abort/init history, and (4) by the (next) client to get the abort history from Backup and initialize the next *ZLight* instance. Note that we deactivate the functions in charge of updating the history of *ZLight*, so that we ensure that for each aborted request, the history contains r requests. We reproduced each experiment three times and observed a negligible variance.

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

5. Putting Abstract to Really Work: Aliph

In this section, we demonstrate how we can build novel, very efficient BFT protocols, using Abstract. Our new protocol, called *Aliph*, achieves up to 30% lower latency and up to 360% higher throughput than state-of-the-art protocols. The development of *Aliph* consisted in building two new instances of Abstract, each requiring less than 25% of the code of state-of-the-art protocols, and reusing *Backup* (Sec. 4.3). In the following, we first describe *Aliph* and then we evaluate its performance.
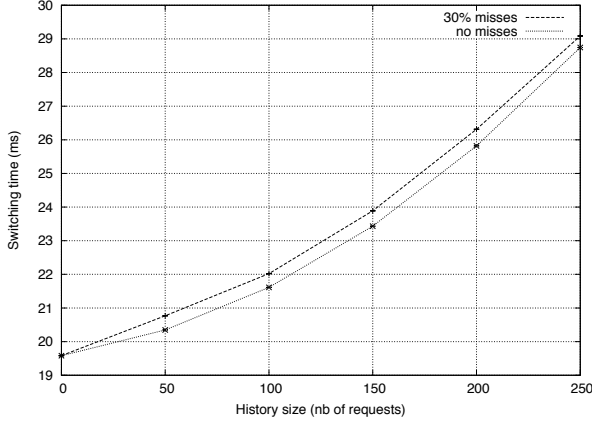
Figure 3. Switching time in function of history size and percentage of missing requests in replica histories.

5.1 Protocol overview

The characteristics of *Aliph* are summarized in Table 2, considering the metrics of [15]. In short, *Aliph* is the first optimally resilient protocol that achieves a latency of 2 one-way message delays when there is no contention. It is also the first protocol for which the number of MAC operations at the bottleneck replica tends to 1 (under high contention when batching of messages is enabled): 50% less than required by state-of-the-art protocols.

	PBFT	Q/U	HQ	Zyzzyva	Aliph
1	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1**
2	$2+\frac{8f}{b}$	2+4f	2+4f	$2+\frac{3f}{b}$	$\mathbf{1+\frac{f+1}{b}}$
3	4	**2**	4	3	**2**

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

Aliph uses three Abstract implementations: *Backup* (introduced in Sec. 4.3), *Quorum* and *Chain* (both described below). A *Quorum* instance commits requests as long as there are no: (a) server/link failures, (b) client Byzantine failures, and (c) contention. *Quorum* implements a very simple communication pattern and gives *Aliph* the low latency flavor when its progress conditions are satisfied. On the other hand, *Chain* provides exactly the same progress guarantees as *ZLight* (Sec. 4.2), i.e., it commits requests as long as there are no server/link failures or Byzantine clients. *Chain* implements a pipeline pattern and, as we show below, allows *Aliph* to achieve better peak throughput than all existing protocols. Both *Quorum* and *Chain* share the panicking mechanism with *ZLight*, which is invoked by the client when it fails to commit the request.

Aliph uses the following static switching ordering to orchestrate its underlying protocols: *Quorum-Chain-Backup-Quorum-Chain-Backup−etc*. Initially, *Quorum* is active. As soon as it aborts (e.g., due to contention), it switches to *Chain*. *Chain* commits requests until it aborts (e.g., due to asynchrony). *Aliph* then switches to *Backup*, which executes k requests (see Sec. 4.3). When *Backup* commits k requests, it aborts, switches back to *Quorum*, and so on.

In the following, we first describe *Quorum* (Sec. 5.1.1) and *Chain* (Sec. 5.1.2) (full details and correctness proofs can be found in Appendix B and C, respectively). Then, we discuss some system-level optimizations of *Aliph* (Sec. 5.1.3).

5.1.1 Quorum

Quorum implements a very simple communication pattern (see Fig. 4); it requires only one round-trip of message exchange between a client and replicas to commit a request. Namely, the client sends the request to all replicas that speculatively execute it and send a reply to the client. As in *ZLight*, replies sent by replicas contain a digest of their history. The client checks that the histories sent by the $3f + 1$ replicas match. If that is not the case, or if the client does not receive $3f + 1$ replies, the client invokes a panicking mechanism. This is the same as in *ZLight* (Sec. 4.2): (i) the client sends a PANIC message to replicas, (ii) replicas stop executing requests on reception of a PANIC message, (iii) replicas send back a signed message containing their history. The client collects $2f + 1$ signed messages containing replica histories and generates an abort history. Note that, unlike *ZLight*, *Quorum* does not tolerate contention: concurrent requests can be executed in different orders by different replicas. This is not the case in *ZLight*, as requests are ordered by the primary.
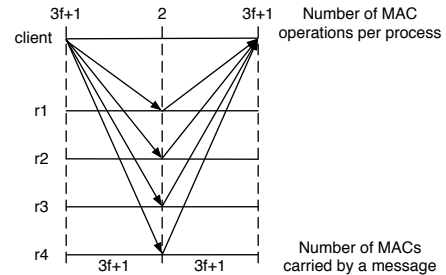

Figure 4. Communication pattern of *Quorum*.

The implementation of *Quorum* is very simple. It requires 3200 lines of C code (including panicking and checkpoint libraries). *Quorum* makes *Aliph* the first BFT protocol to achieve a latency of 2 one-way message delays, while only requiring $3f + 1$ replicas (Q/U [1] has the same latency but requires $5f + 1$ replicas). Given its simplicity and efficiency, it might seem surprising not to have seen it published earlier. We believe that Abstract made that possible because we could focus on weaker (and hence easier to implement)

Abstract specifications, without caring about (numerous) difficulties outside the *Quorum* "common-case".

5.1.2 Chain

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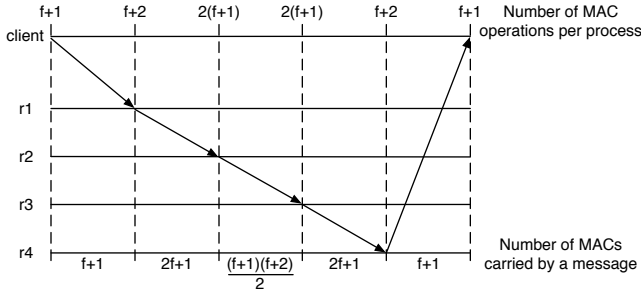

Figure 5. Communication pattern of *Chain*.

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

The behavior of *Chain*, as described so far, is very similar to the protocol described in [21]. That protocol, however, did only tolerate crash faults. We tolerate Byzantine failures by ensuring: (1) that the content of a message is not modified by a malicious 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, our *Chain* relies on a novel authentication method we call *chain authenticators (CAs)*. CAs are lightweight MAC authenticators, requiring processes to generate (at most) $f + 1$ MACs (in contrast to $3f + 1$ in traditional authenticators). CAs guarantee that, if a client commits request req, every correct replica executed req. CAs, along with the inherent throughput advantages of a pipeline pattern, are key to *Chain*'s dramatic throughput improvements over other BFT protocols. We describe below how CAs are used in *Chain*.

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. 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, process p_1 verifies that the message contains a valid MAC from process p_0 and the client, whereas the client verifies that the reply it gets contains a valid MAC from the last $f + 1$ replicas in the chain. Finally, to make sure that the reply sent by the tail is correct, the f processes that precede the tail in the chain append a digest of the response to the message.

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

Chain's implementation requires 3300 lines of code (with panic and checkpoint libraries). Moreover, it is the first protocol in which the number of MAC operations at the bottleneck replica tends to 1. This comes from the fact that, under contention, the head of the chain can batch requests. Head and tail do thus need to read (resp. write) a MAC from (resp. to) the client, and write (resp. read) $f + 1$ MACs for a batch of requests. Thus for a single request, head and tail perform $1 + \frac{f+1}{b}$ MAC operations. Note that all other replicas process requests in batch, and have thus a lower number of MAC operations per request. State-of-the-art protocols [6, 15] do all require at least 2 MAC operations at the bottleneck server (with the same assumption on batching). The reason why this number tends to 1 in *Chain* can be intuitively explained by the fact that these are two distinct replicas that receive the request (the head) and send the reply (the tail).

5.1.3 Optimizations

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

5.2 Evaluation

This section evaluates the performance of *Aliph*. For lack of space, we focus on experiments without failures (of processes or links), since we compare to protocols that are known to perform well in the common-case — PBFT [6], Q/U [1] and Zyzzyva [15].

We first study latency, throughput, and fault scalability using Castro's microbenchmarks [6, 15], varying the number of clients. Clients invoke requests in closed-loop (meaning that a client does not invoke a new request before it commits a previous one). In the x/y microbenchmark, clients send xkB requests and receive ykB replies. We also perform an experiment in which the input load dynamically varies.

We evaluate PBFT and Zyzzyva because the former is considered the "baseline" for practical BFT implementations, whereas the latter is considered state-of-the-art. Moreover, Zyzzyva systematically outperforms HQ [15]; hence, we do not evaluate HQ. Finally, we benchmark Q/U as it is known to provide better latency than Zyzzyva under certain condition. Note that Q/U requires $5f + 1$ servers, whereas other protocols we benchmark only require $3f + 1$ servers.

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

PBFT code base underlies both Zyzzyva and *Aliph*. To ensure that the comparison with Q/U is fair, we evaluate its simple best-case implementation described in [15].

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.

5.2.1 Latency

We first assess the latency in a system without contention, with a single client issuing requests. The results for all microbenchmarks (0/0, 0/4 and 4/0) are summarized in Table 3 demonstrating the latency improvement of *Aliph* over Q/U, PBFT, and Zyzzyva. We show results for a maximal number of server failures f ranging from 1 to 3. Our results show that *Aliph* consistently outperforms other protocols, since *Quorum* is active when there is no contention. These results confirm the theoretical expectations (see Table 2, Sec. 5.1). The results show that Q/U also achieves a good latency with $f = 1$, as Q/U and *Quorum* use the same communication pattern. Nevertheless, when f increases, performance of Q/U decreases significantly. The reason is that Q/U requires $5f + 1$ replicas and both clients and servers perform addi-

tional MAC computations compared to *Quorum*. Moreover, the significant improvement of *Aliph* over Zyzzyva (31% at $f = 1$) can be easily explained by the fact that Zyzzyva requires 3-one-way message delays in the common case, whereas *Aliph* (*Quorum*) only requires 2-one-way message delays.

5.2.2 Throughput

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

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

0/0 benchmark. Figure 6 plots the throughput achieved in the 0/0 benchmark by various protocols when $f = 1$. We ran Zyzzyva with and without batching. For PBFT, we present only the results with batching, since they are substantially better than results without batching. We observe that Zyzzyva with batching performs better than PBFT, which itself achieves higher peak throughput than Zyzzyva without batching (this is consistent with the results of [15, 20]).
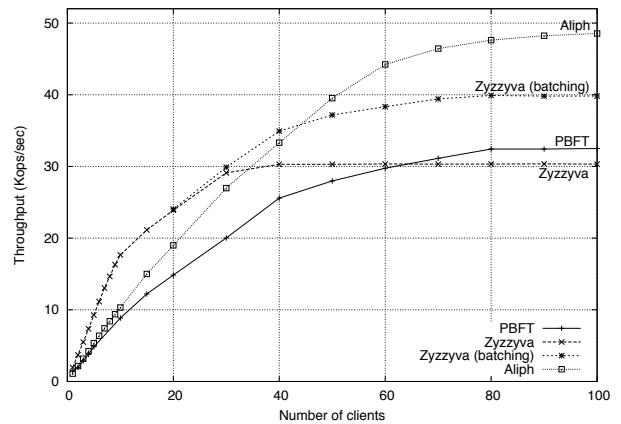
Figure 6. Throughput for the 0/0 benchmark (f=1).

Moreover, Figure 6 shows that with up to 40 clients, Zyzzyva achieves the best throughput. With more than 40 clients, *Aliph* starts to outperform Zyzzyva. The peak throughput achieved by *Aliph* is 21% higher than that of Zyzzyva. The reason is that *Aliph* executes *Chain*, which uses a pipeline pattern to disseminate requests. This pipeline pattern brings two benefits: reduced number of MAC op-

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

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

erations at the bottleneck server, and better network usage: servers send/receive messages to/from a single server.

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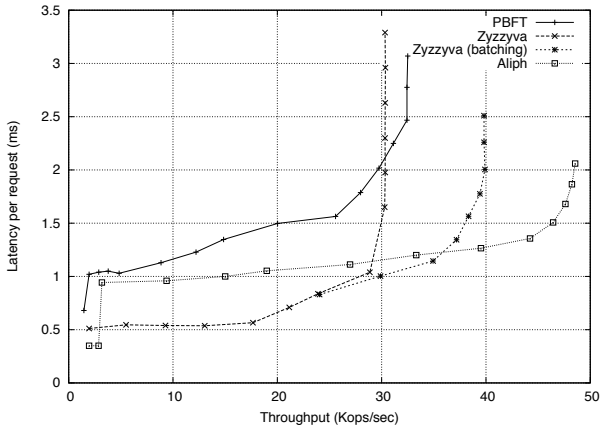

Figure 7. Response-time vs. throughput for the 0/0 benchmark (f=1).

Figure 7 plots the response-time of Zyzzyva (with and without batching), PBFT and *Aliph* as a function of the achieved throughput. We observe that *Aliph* achieves consistently lower response-time than PBFT. This stems from the fact that the message pattern of PBFT is a very complex one, which increases the response time and limits the throughput of PBFT. Moreover, up to the throughput of 37Kops/sec, *Aliph* has a slightly higher response-time than Zyzzyva. The reason is the pipeline pattern of *Chain* that results in a higher response time for low to medium throughput, which stays reasonable nevertheless. Moreover, *Aliph* scales better than Zyzzyva: from 37Kops/sec, it achieves lower response time, since the messages are processed faster due to the higher throughput ensured by *Chain*. Hence, messages spend less time in waiting queues. Finally, we observe that for very low throughput, *Aliph* has lower response time than Zyzzyva. This comes from the fact that *Aliph* uses *Quorum* when there

is no contention, which significantly improves the response-time of the protocol.

0/4 benchmark. Figure 8 shows the throughput of the various protocols for the 0/4 microbenchmark when $f = 1$. PBFT and Zyzzyva are using the client multicast optimization. We observe that with up to 15 clients, Zyzzyva outperforms other protocols. Starting from 20 clients, *Aliph* outperforms PBFT and Zyzzyva. Nevertheless, the gain in peak throughput (7,7% over PBFT and 9,8% over Zyzzyva) is lower than the gain we had with the 0/0 microbenchmark. This can be explained by the fact that the dominating cost is now sending replies to clients, partly masking the effect of request processing and request/sequence number forwarding. In all protocols, there is only one server sending a full reply to the client (other servers send only a digest of the reply). We were expecting PBFT and Zyzzyva to outperform *Aliph* (which executes *Chain* when there is load), since the server that sends a full reply in PBFT and Zyzzyva changes on a per-request basis. Nevertheless, this is not the case. We again attribute this result to the fact that *Chain* uses a pipeline pattern: the last process in the chain replies to clients at the throughput of about 391MB/sec.
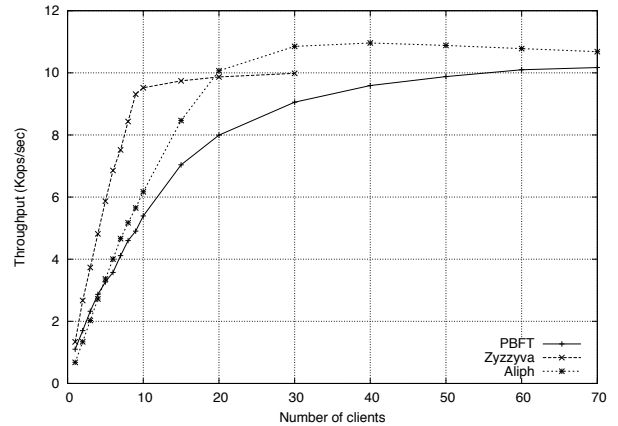
Figure 8. Throughput for the 0/4 benchmark (f=1).

4/0 benchmark. Figure 9 shows the results of *Aliph*, PBFT and Zyzzyva for the 4/0 microbenchmark with $f = 1$. Notice the logarithmic scale for the X axis, that we use to better highlight the behavior of various protocols with small numbers of clients. For PBFT and Zyzzyva, we plot curves both with and without client multicast optimization. The graph shows that with up to 3 clients, Zyzzyva outperforms

other protocols. With more than 3 clients, *Aliph* significantly outperforms other protocols. Its peak throughput is about 360% higher than that of Zyzzyva. The reason why *Aliph* is very efficient under high load and when requests are large was explained earlier in the context of the 0/0 benchmark.

Figure 9. Throughput for the 4/0 benchmark (f=1).

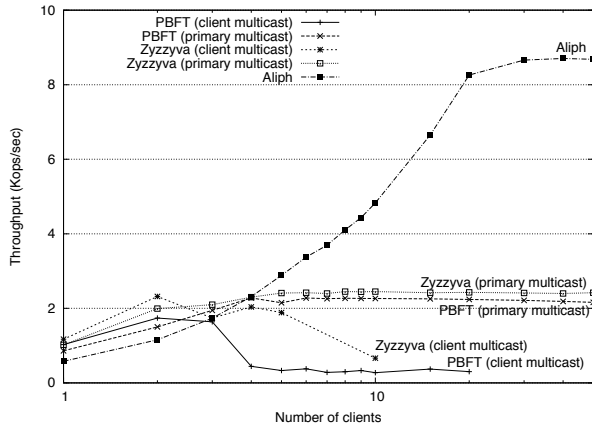Notice also the interesting drop in the performance of Zyzzyva and PBFT when client multicast optimization is used (Fig. 9). This is to be contrasted with the case when the primary forwards requests, where the performance of PBFT and Zyzzyva remain almost constant after the peak throughput has been reached. These results may seem surprising given that [6, 15] recommend to use the client multicast optimization when requests are large, in order to spare the primary of costly operations request forwarding. Nevertheless, these results can be explained by the fact that simultaneous multicasts of large messages by different clients result in collisions and buffer overflows, thus requiring many message retransmissions[7] (there is no flow control in UDP). This explains why enabling the concurrent client multicasts drastically reduces performance. On the other hand, when the primary forwards messages, there are fewer collisions, which explains the better performance we observe.

Impact of the request size. In this experiment we study how protocols are impacted by the size of requests. Figure 10 shows the peak throughput of *Aliph*, PBFT and Zyzzyva as a function of the request size for $f = 1$. To obtain the peak throughput of PBFT and Zyzzyva, we benchmarked both protocols with and without client multicast optimization and with different batching sizes for Zyzzyva. Interestingly, the behavior we observe is similar to that observed using simulations in [20]: differences between PBFT and Zyzzyva diminish with the increase in payload. Indeed, starting from 128B payloads, both protocols have almost identical performance.

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

Figure 10 also shows that *Aliph* sustains high peak throughput with all message sizes, which is again the consequence of *Chain* being active under contention.
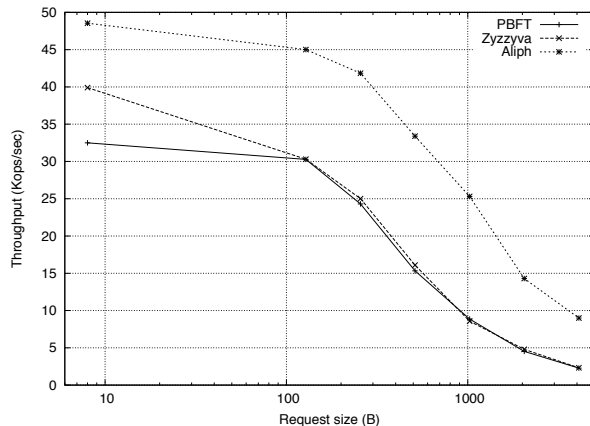
Figure 10. Peak throughput in function of request size (f=1).

Fault scalability. One important characteristic of BFT protocols is their behavior when the number of tolerated server failures f increases. Figure 11 depicts the performance of *Aliph* for the 4/0 benchmark when f varies between 1 and 3. We do not present results for PBFT and Zyzzyva as their peak throughput is known to suffer only a slight impact [15]. Figure 11 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.

Figure 11. Impact of the number of tolerated failures f on the *Aliph* throughput.

5.2.3 Performance in case of faults

As mentioned in Section 4.3, when *Aliph* switches to *Backup*, it executes k requests, and then switches back to *Quorum*.

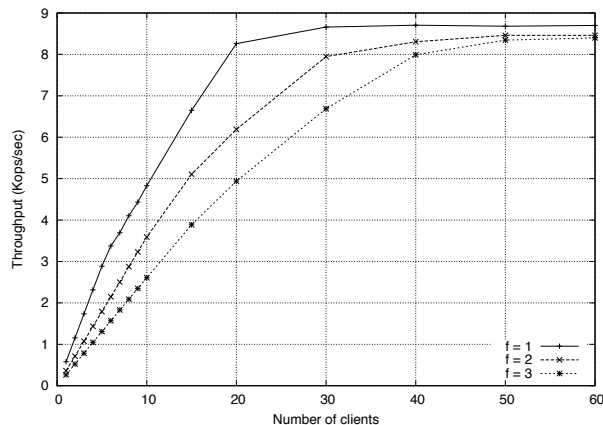To asses the impact of k on system's performance, we compared performance for two different values of k – in the first case, $k = 1$, and in the second case, $k = 2^i$, where i is the number of invocations of *Backup* since beginning.

In this experiment, we skip execution in *Chain*, and focus only on *Quorum* and *Backup*. There is one client, which issues 15.000 requests in total. If run just under *Quorum*, system would process these requests in 7s. After client sends 2.000 requests, one of the replicas goes down, and remains down for 10s. During this time, only three replicas are active. Hence, as *Quorum* protocol requires all replicas to respond, it will not execute any request. Figure 12 shows the throughput of the system, when *Aliph* switches to *Backup* for a single request. As discovery of replica failure in *Quorum* is done with timers, only handful of requests will be serviced (by *Backup*) while one replica is down. On the other hand, Figure 13 shows the behavior of the system if it stays in *Backup* for 2^i requests. Although it takes less time to finish the experiment, system may stay for too long in *Backup*. Replica came back up at $t = 11s$ in the experiment, but switch from *Backup* to *Quorum* occurred around $t = 14s$, because *Backup* had to process 8192 requests.
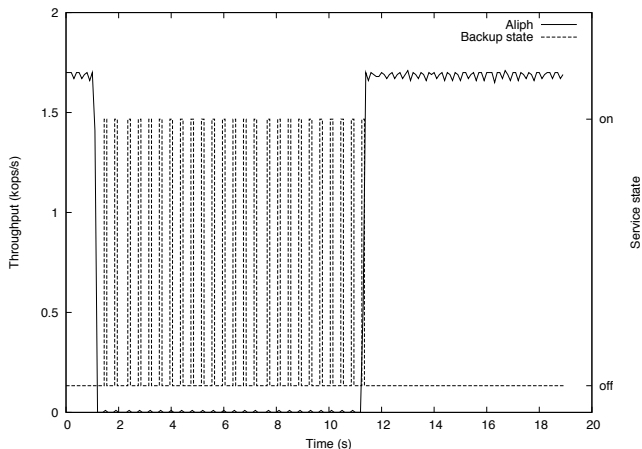

Figure 12. Throughput under faults, when system switches to *Backup* for one request.

5.2.4 Dynamic workload

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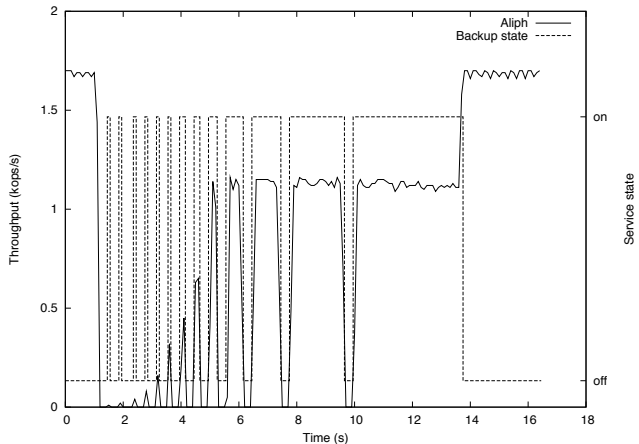

Figure 13. Throughput under faults, when system switches to *Backup* for 2^i requests.

Figure 14 shows the performance of *Aliph*, Zyzzyva, and *Chain*. For each protocol, clients were invoking the same number of requests. Moreover, requests were invoked after the preceding clients have completed their bursts. First, we observe that *Aliph* is the most efficient protocol: it completes the experiment in $42s$, followed by Zyzzyva ($68.1s$), and *Chain* ($77.2s$). Up to time $t = 15.8s$, *Aliph* uses *Quorum*, which performs much better than Zyzzyva and *Chain*. Starting at $t = 15.8$, contention becomes too high for *Quorum*, which switches to *Chain*. At time $t = 31.8s$, there is only one client in the system. After $2s$ spent with only one client in the system, *Chain* in *Aliph* starts aborting requests due to the low load optimization (Sec. 5.1.3). Consequently, *Aliph* switches to *Backup* and then to *Quorum*. This explains the increase in throughput observed at time $t = 33.8s$. We also observe on the graph that *Chain* and *Aliph* are more efficient than Zyzzyva when there is a load spike: they achieve a peak throughput about three times higher than that of Zyzzyva. On the other hand, *Chain* and *Aliph* have slightly lower performance than Zyzzyva under medium load (i.e. from $16s$ to $26s$ on the *Aliph* curve). This suggests an interesting BFT protocol that would combine *Quorum*, Zyzzyva, *Chain* and *Backup*. However, this requires smart choices for dynamic switching, e.g., between Zyzzyva and *Chain*. We believe that building such a protocol is an interesting research topic.

6. Concluding Remarks

The idea of aborting if "something goes wrong" is old. It underlies for instance the seminal two-phase commit protocol [13]: 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 [14]. Abortable consensus was proposed in [8] and [4]. In the first case, a process can abort if a majority of processes cannot be reached whereas, in the second, a process can abort if there
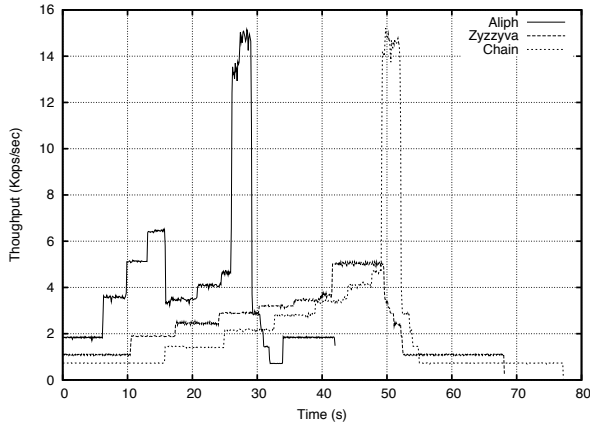
Figure 14. Throughput under dynamic workload.

is contention. The latter idea was generalized for arbitrary shared objects in [3] and then [2]. In [2], a process can abort and then query the object to seek whether the last query of the process was performed. This query can however abort if there is contention. Our notion of abortable state machine replication is different. First, the condition under which Abstract can abort is a generic parameter: it can express for instance contention, synchrony or failures. Second, in case of abort, Abstract returns (without any further query) what is needed for recovery in a Byzantine context; namely, an unforgeable history. This, in turn, can then be used to invoke another, possibly stronger, Abstract. This ability is key to the composability of Abstract instances.

Several examples of speculative protocols, distinguishing an optimistic phase from a recovery one, were discussed in the survey of Pedone [19]. These speculation ideas were used in the context of Byzantine state machine replication, e.g., in HQ [11] and Zyzzyva [15]. 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 to build a BFT protocol as the composition of as many (gracefully degrading) phases as desired, each with a "standard" interface. This allows for an unpreceded flexibility in BFT protocol design that we illustrated with *Aliph*, a BFT protocol that combines three different phases.

Several directions can be interesting to explore. It would be interesting to devise efficient Abstract implementations for other interesting definitions of the progress property, e.g., implementations that perform well despite failures. It would be also interesting to explore possibilities for signature-free switching, to obtain practical BFT protocols that do not rely on signatures [10]. Moreover, we believe that an interesting research challenge is to define and evaluate effective heuristics for dynamic switching among Abstract instances. While we described *Aliph* and showed that, albeit simple, it outperforms existing BFT protocols, *Aliph* is simply the starting point for Abstract. The idea of dynamic switching depending on the system conditions seems very promising.

Acknowledgements

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] L. Lamport. The +CAL algorithm language. In *NCA*, 2006.

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

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

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

A. Abstract **specification and composition**

Before diving into the precise specification of Abstract, we first introduce some notations and definitions. We denote the output function of the (replicated) state machine by $rep(h)$, where h is a sequence of requests called *commit history*: the initial state is an implicit argument of $rep()$. Basically, $rep(h)$ represents *replies* that state machine outputs to clients. We assume that every Abstract instance has its own unique id i (a natural number) that uniquely determines the implementation and the set of underlying servers.

Abstract i exports one operation: $Invoke_i(m, [h])$, where m is a request, and h a (optional) sequence of requests called *init history*; we say the client *invokes* request m (with init history h). By convention, when $i = 1$, the invocation never contains an init history. Abstract i returns two indications to the client:

1. $Commit_i(m, rep(h))$, where commit history h contains m;

2. $Abort_i(m, h, next(i))$, where the sequence of requests h is called an *abort history* and $next$ is a function that returns an integer.

Respectively, we say that a client *commits/aborts* the request m. In the case of an abort, we also say that instance Abstract i *switches* to instance $next(i)$. We also define a *valid init history* (VIH) as follows: init history h is a VIH if there is an Abstract j that returned $Abort_j(*, h, i)$ (i.e., such that $next(j) = i$). Similarly, we define a *valid init request* (VIR) as follows: (1) if $i = 1$, any invoked request is a VIR; (2) if $i \neq 1$, m is a VIR if and only if m is invoked with a VIH. We say that an instance $i > 1$ is *initialized* upon it commits or aborts some VIR. Finally, we say that a request m is *valid* if (1) m is a VIR, or (2) m is invoked after i gets initialized.

We are now ready to specify the properties of Abstract (parametrized by a predicate P). In the following, "prefix" refers to a non-strict prefix.

1. *(Validity)* In every commit/abort history, no request appears twice and every request is a valid request, or an element of a valid init history.

2. *(Termination)* If a correct client c invokes a valid request m, then c eventually commits or aborts m.

3. *(Progress)* If a correct client c invokes a valid request m and some predicate P holds, then c commits m.

4. *(Init Order)* The longest common prefix of all valid init histories is a prefix of any commit or abort history.

5. *(Commit Order)* Let h and h' be any two commit histories: either h is a prefix of h' or vice versa.

6. *(Abort Order)* Every commit history is a prefix of every abort history.

7. *(Switching Monotonicity)* For every Abstract instance i, $i < next(i)$.

It is important to see that Abstract is a strict generalization of BFT. Namely, BFT is precisely an Abstract (with id $i = 1$) that never aborts. In this case, *Abort Order* and *Switching Monotonicity* become irrelevant, and *Init Order* trivially holds.

We build BFT protocols by composing Abstract instances. At the heart of the composition lies a simple scheme (we refer to as Abstract composition algorithm (ACA)), given in Figure 15, where: a) correct clients use an abort history of an aborting Abstract instance (e.g., i) as the init history for the next instance ($next(i)$), and b) a given client invokes $next(i)$ with an init history included only once (with its first invocation of $next(i)$): subsequently, it invokes $next(i)$ without any init history.

Figure 15. Abstract composition algorithm (ACA). Invocations/indications of the composed Abstract are in uppercase.

The key invariant in the Abstract framework is *idempotence*: a composition of *any two* Abstract instances is, itself, an Abstract instance. More precisely:

Theorem 1. *Given any two Abstract instances i and i' and integer i'', such that $i' = next(i)$ and $next(i') = i''$, ACA implements a single* Abstract *instance with the instance id i, such that $next(i) = i''$.*

By induction, the theorem extends to a composition of *any* number of Abstract instances, which yields again an Abstract instance. To prove our composition theorem, we specified ACA and Abstract properties in +CAL/TLA+ and leveraged their correctness using the TLC model checker [16, 18].

Details of the model checking are given in Figures 16-18. In Figures 16- 18, for simplicity, we focused on model checking the case where $i = 1$ to avoid simulating input init histories (not used for instance $i = ABSID = 1$). Moreover, we assumed (without loss of generality) $next(i) = i + 1$. We model checked the configuration with two clients, with 1 or 2 requests per client ($MaxReq = 1$ or $MaxReq = 2$).

Output of the TLC model checker for the evaluated configuration is given in Figure 18.

```
 1 ┌─────────────────────── MODULE Abstract ───────────────────────┐
 2 EXTENDS Integers, TLC, FiniteSets, Sequences
 3 CONSTANTS MaxReq, ABSID, Abstract, Client, LCPrefix(_), Set2Sequence(_, _), InitHistories(_), Tails(_)

 5 Requests ≜ 1 .. MaxReq * Cardinality(Client)  ⟨contains all possible requests⟩

 7 prefix(m, h) ≜ (Len(m) ≤ Len(h)) ∧ (∀ i ∈ 1 .. Len(m) : m[i] = h[i])  ⟨Definition of a non-strict prefix⟩

 9    ⟨Checks whether commit history ch satisfies Commit Order⟩
10 CommitOrder(ch, ex) ≜
11    ∀ m ∈ ex : m.type = "COMMIT" ⇒ (prefix(m.history, ch) ∨ prefix(ch, m.history))

13    ⟨Checks whether commit history ch satisfies Abort Order⟩
14 AbortOrder4Commit(ch, ex) ≜ ∀ m ∈ ex : m.type = "ABORT" ⇒ prefix(ch, m.history)

16    ⟨Checks whether abort history ah satisfies Abort Order⟩
17 AbortOrder(ah, ex) ≜ ∀ m ∈ ex : m.type = "COMMIT" ⇒ prefix(m.history, ah)

19    ⟨Recursive functions are prefixed with R⟩
20 RTails(histories) ≜
21    IF histories = {} THEN {}
22                      ELSE  LET h ≜ CHOOSE m ∈ histories : m ∈ histories
23                            IN  {Tail(h)} ∪ Tails(histories \ {h})
24 RLCPrefix(histories) ≜    ⟨Returns a largest common prefix of a set of histories⟩
25    IF histories = {} THEN ⟨⟩
26                      ELSE  LET hist ≜ CHOOSE m ∈ histories : ∀ mm ∈ histories : Len(m) ≤ Len(mm)
27                            IN  IF (hist = ⟨⟩) ∨ (∃ m1 ∈ histories : Head(hist) ≠ Head(m1))
28                                THEN ⟨⟩
29                                ELSE ⟨Head(hist)⟩ ∘ LCPrefix(Tails(histories))
30 RInitHistories(inv) ≜    ⟨Returns a set of init histories from a set of invoked requests inv⟩
31    IF ∀ mm ∈ inv : mm.type ≠ "INIT" THEN {}
32                            ELSE  LET init ≜ CHOOSE m ∈ inv : m.type = "INIT"
33                                  IN  {init.history} ∪ InitHistories(inv \ {init})
34 InitOrder(h, inv) ≜ prefix(LCPrefix(InitHistories(inv)), h)  ⟨Checks whether commit/abort history h satisfies Init Order⟩

36 Response(m, ex) ≜ {mm ∈ ex : m.req = mm.req}

38    ⟨Checks whether h is a valid init histories (VIH) for instance AbsID⟩
39 VIH(h, AbsID, ex) ≜ ∃ precID ∈ Abstract : ∃ m ∈ ex[precID] :  ∧ m.type = "ABORT"
40                                                               ∧ m.history = h
41                                                               ∧ m.Next = AbsID

43    ⟨Returns a set of valid requests for AbsID that were invoked but not yet committed/aborted⟩
44 NewValidRequest(AbsID, inv, ex) ≜ {m ∈ inv[AbsID] : m.valid = TRUE ∧ Response(m, ex[AbsID]) = {}}

46    ⟨Histories(ValidRequests()) returns a set of histories that conform with Abstract Validity⟩
47 Exists(req, h) ≜ ∃ i ∈ 1 .. Len(h) : h[i] = req
48 ValidRequests(AbsID, inv, ex) ≜ {req ∈ Requests : ∃ mm ∈ inv :  ∨ (mm.req = req ∧ mm.valid = TRUE)
49                                                                  ∨ ( ∧ mm.type = "INIT"
50                                                                      ∧ Exists(req, mm.history)
51                                                                      ∧ VIH(mm.history, AbsID, ex))}
52 SetPermutations(len, S) ≜
53    {f ∈ [1 .. len → S] : ∀ v, w ∈ 1 .. len : (v = w) ∨ f[w] ≠ f[v]}
54 RSet2Sequence(len, S) ≜
55    IF len = 0 THEN {} ELSE SetPermutations(len, S) ∪ Set2Sequence(len − 1, S)
56 Histories(invokedReqs) ≜ Set2Sequence(Cardinality(invokedReqs), invokedReqs)

58 validCommitHistories(newReq, AbsID, inv, ex) ≜    ⟨Returns all possible commit histories for a given newReq⟩
59    {ch ∈ Histories(ValidRequests(AbsID, inv[AbsID], ex)) :  ∧ Exists(newReq.req, ch)
60                                                             ∧ CommitOrder(ch, ex[AbsID])
61                                                             ∧ AbortOrder4Commit(ch, ex[AbsID])
62                                                             ∧ InitOrder(ch, inv[AbsID])}

                                          1
```

Figure 16. Abstract model checking in +CAL/TLA+ 1/3.

64 $validAbortHistories(newReq, AbsID, inv, ex) \triangleq$ Returns all possible abort histories for a given $newReq$
65 $\{ah \in Histories(ValidRequests(AbsID, inv[AbsID], ex)) : \wedge AbortOrder(ah, ex[AbsID])$
66 $\wedge InitOrder(ah, inv[AbsID])\}$

68 $next(AbsID) \triangleq AbsID + 1$ Sample $next()$ function

```
--algorithm ACA{
  variables \ * Global variables for asserting correctness
        Invoked = [i ∈ Abstract ↦ {}]; Executed = [i ∈ Abstract ↦ {}];
        INVOKED = {}; EXECUTED = {};
        UniqueReqID = 0;
        finished = [i ∈ Client ↦ FALSE];

  macro RETURN(m){EXECUTED := EXECUTED ∪ {m};}
  macro INVOKE(m){
    if(m.type = "NORMAL")
      INVOKED := INVOKED ∪ {[type ↦ "NORMAL", req ↦ m.req, valid ↦ (ABSID = 1) ∨ (EXECUTED ≠ {})]}}}

  macro Return(AbsID, m){Executed[AbsID] := Executed[AbsID] ∪ {m}}
  macro Invoke(AbsID, m){
    if(m.type = "NORMAL")
      Invoked[AbsID] := Invoked[AbsID] ∪ {[type ↦ "NORMAL", req ↦ m.req, valid ↦ (AbsID = 1) ∨ (Executed[AbsID] ≠ {})]}
    else Invoked[AbsID] := Invoked[AbsID] ∪ {[type ↦ "INIT", req ↦ m.req, history ↦ m.history,
                                valid ↦ Executed[AbsID] ≠ {} ∨ VIH(m.history, AbsID, Executed)]}}}

  process(clnt ∈ Client)
    variables absReq; active = 1;
          localts = 0; \ * localts : counter for local requests for model checking purposes
    {C : while(localts < MaxReq){
        localts := localts + 1;
        UniqueReqID := UniqueReqID + 1;
        \ * Requests without init histories are tagged as "NORMAL".
        absReq := [type ↦ "NORMAL", req ↦ UniqueReqID];
        INVOKE(absReq);
absInv : Invoke(active, absReq);
absResp : with(m ∈ Response(absReq, Executed[active]))
        if(m.type = "ABORT" ∧ active = 1){
            active := m.Next;
            \ * Requests with init histories are tagged as "INIT"
            absReq := [type ↦ "INIT", req ↦ absReq.req, history ↦ m.history];
            goto absInv; }
        else{
            RETURN(m);
            goto C; }};
    finished[self] := TRUE} \ * end process(clnt)

  process(AbsID ∈ Abstract){
S : while(∃ c ∈ Client : finished[c] = FALSE){
      with(newReq ∈ NewValidRequest(self, Invoked, Executed)){
        either{
        with(hist ∈ validCommitHistories(newReq, self, Invoked, Executed))
        Return(self, [type ↦ "COMMIT", req ↦ newReq.req, history ↦ hist]); }
        or{
        with(hist ∈ validAbortHistories(newReq, self, Invoked, Executed))
        Return(self, [type ↦ "ABORT", req ↦ newReq.req, Next ↦ next(self), history ↦ hist])}}}}
} \ * algorithm ACA
```

125 BEGIN TRANSLATION
126 END TRANSLATION

128 $Perms \triangleq Permutations(Client)$

130 Abstract properties
131 $VALIDITY(id, ex, inv) \triangleq \forall m \in ex : (\text{LET } h \triangleq m.history \text{ IN}$
132 $(\forall i \in 1 .. Len(h) : \wedge h[i] \in ValidRequests(id, inv, ex)$
133 $\wedge (\forall k \in i + 1 .. Len(h) : h[i] \neq h[k])))$

2

Figure 17. Abstract model checking in +CAL/TLA+ 2/3.

135    $INITORDER \triangleq$ TRUE Since we implement *Abstract* with *id* 1, *Init* Order trivially holds
136    $COMMITORDER(ex) \triangleq \forall m \in ex : (m.type = \text{"COMMIT"} \Rightarrow CommitOrder(m.history, ex))$
137    $ABORTORDER(ex) \triangleq \forall m \in ex : (m.type = \text{"ABORT"} \Rightarrow AbortOrder(m.history, ex))$
138    $SWITCHING(ex) \triangleq \forall m \in ex : (m.type = \text{"ABORT"} \Rightarrow m.Next = next(next(1)))$

140    $Safety \triangleq \land VALIDITY(ABSID, EXECUTED, INVOKED)$
141                $\land INITORDER$
142                $\land COMMITORDER(EXECUTED)$
143                $\land ABORTORDER(EXECUTED)$
144                $\land SWITCHING(EXECUTED)$

146    $TERMINATION(ex, inv) \triangleq (\forall self \in ProcSet : pc[self] = \text{"Done"}) \Rightarrow$
147                                 $(\forall m \in inv : \exists mm \in ex : m.req = mm.req)$

149    $Liveness \triangleq TERMINATION(EXECUTED, INVOKED)$
150

```
\ * Results SPECIFICATION Spec
CONSTANT
    ABSID = 1 \ * implemented Abstract instance
    Abstract = {1, 2}
    LCPrefix ← RLCPrefix
    Set2Sequence ← RSet2Sequence
    InitHistories ← RInitHistories
    Tails ← RTails

    MaxReq = 1
    Client = {p1, p2}

SYMMETRY Perms INVARIANT Safety PROPERTY Liveness
```

Model checking completed. No error has been found. Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint:
   calculated (optimistic): $1.1906511475541504E-12$ based on the actual fingerprints: $7.615995759161076E-12$
9398 states generated, 5041 distinct states found, 0 states left on queue. The depth of the complete state graph search is 19.

With $MaxReq = 2$

Model checking completed. No error has been found. Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint:
   calculated (optimistic): $9.900040347878242E-5$ based on the actual fingerprints: $6.460365656696169E-5$
85564169 states generated, 44799917 distinct states found, 0 states left on queue. The depth of the complete state graph search is 27.

Figure 18. Abstract model checking in +CAL/TLA+ 3/3.

B. Detailed protocol descriptions

In this Appendix, we give the details of *ZLight*, *Quorum*, *Chain* and their shared panicking mechanism (Sec. B.2 - B.5). For ease of presentation, we first give details about handling invocations without init histories and then (Sec. B.6) we show how we handle init histories and switch between Abstract instances in *AZyzzyva* and *Aliph*. Finally, we give the details of our checkpointing subprotocol in Section B.7.

Correctness proofs can be found in Appendix C.

B.1 Notation

A message m sent by process p to the 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 [6]) to simultaneously authenticate the message m for multiple recipients belonging to the 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 the message m by $D(m)$, whereas $\langle m \rangle_{\sigma_p}$ denotes a message that contains $D(m)$ signed by the private key of process p and the message m. All processes are assumed to own the public key of every other process. Finally, we denote the set of all $(3f + 1)$ replicas by Σ.

---

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 - operation invoked by the client
LH_j - a local history at replica r_j
$reply_j$ - $rep(LH_j)$ (application reply in function of LH_j)
sn_j - sequence number at replica r_j (not used in *Quorum*)

Figure 19. Message fields and local variables.

Notation for message fields and client/replica local variables used in *ZLight*, *Quorum* and *Chain* is shown in Figure 19. To help distinguish clients' requests for the same operation o, we assume that client c calls $Invoke_i(req)$, where $req = \langle o, t_c, c \rangle$ and where t_c is a unique, monotonically increasing client's timestamp. A replica r_j *executes* req by appending it to LH_j.

B.2 ZLight details

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

Step Z2. The primary r_1 on receiving $m' = \langle REQ, req, i \rangle_{\alpha_{c,\Sigma}}$, if:

- $req.t_c$ is higher than $t_1[c]$,

then it:

- updates $t_1[c]$ to $req.t_c$,
- increments sn_1, and

- sends $\langle\langle ORDER, req, i, sn\rangle_{\mu_{r_1,r_j}}, MAC_j\rangle$ 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) $\langle\langle ORDER, req, i, sn'\rangle_{\mu_{r_1,r_j}}, MAC_j\rangle$, if:

- MAC_j authenticates req and i,
- $sn' = sn_j + 1$, and
- $t_j[c] < req.t_c$,

then it:

- updates sn_j to sn' and $t_j[req.c]$ to $req.t_c$,
- executes req, and
- sends $\langle RESP, reply_j, D(LH_j), i, req.t_c, r_j \rangle_{\mu_{r_j,c}}$ to c.[8]

Moreover, if MAC_j verification fails, r_j stops executing Step Z3 in instance i.

Step Z4. If client c receives $3f+1$ $\langle RESP, reply, LHDigest, i, req.t_c, * \rangle_{\mu_{*,c}}$ 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 with $reply$. Otherwise, the client triggers the panicking mechanism explained in Section B.5 (Step P1).

B.3 Quorum details

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.

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:

- updates $t_j[c]$ to $req.t_c$,
- executes req, and
- sends $\langle RESP, reply_j, D(LH_j), i, req.t_c, r_j \rangle_{\mu_{r_j,c}}$ to c.

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

B.4 Chain details

In the following, we assume that every CHAIN message sent by process p contains its respective chain authenticator (CA), as well as MACs p received from its predecessor $\overleftarrow{p}$ destined to processes in p's successor set (see Sec. 5.1.2).

Step C1. On $Invoke_i(req)$, client c sends the message $m' = \langle CHAIN, req, i \rangle$ to the head (say r_1) and triggers the timer T.

---

[8] As an optimization (which also applies to Step Q2 of *Quorum*), all but one designated replica can send reply digests $D(reply_j)$ instead of $reply_j$.

Step C2. The head r_1, on receiving $m = \langle \text{CHAIN}, req, i \rangle$ from client c, if:

- $req.t_c$ is higher than $t_1[c]$, and
- the head can verify client's MAC (otherwise the head discards m),

then the head:

- updates $t_1[c]$ to $req.t_c$,
- increments sn_1, and
- sends $\langle \text{CHAIN}, req, i, sn_1, \perp \rangle$ to $\overrightarrow{r_1} = r_2$.

Step C3. Replica r_j on receiving $m = \langle \text{CHAIN}, req, i, sn, REPLY \rangle$ from $\overleftarrow{r_j}$, if

- it can verify MACs from all processes from its predecessor set against the content of m,
- $sn = sn_j + 1$, and
- $req.t_c$ is higher than $t_j[c]$,

then it:

- updates sn_j to sn and $t_j[c]$ to $req.t_c$,
- (ii) executes req, and
- (iii) sends $\langle \text{CHAIN}, req, i, sn, REPLY, LHDigest \rangle$ to $\overrightarrow{r_j}$, where $REPLY = LHDigest = \perp$ in case of the first $2f$ replicas, $REPLY = D(reply_j)$ and $LHDigest = D(LH_j)$ in case $i \in \{2f+1 \ldots 3f\}$, or $REPLY = reply_j$ and $LHDigest = D(LH_j)$ 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 $\langle \text{CHAIN}, req, i, *, reply, LHDigest \rangle$ from the tail before expiration of T, and with MACs from last $f+1$ replicas that authenticate $req, i, LHDigest$ and $D(reply)$ (or $reply$ itself), then c commits req with $reply$. Otherwise, the client triggers the panicking mechanism explained in the following section (Step P1, Sec. B.5).

B.5 Panicking mechanism

This is the mechanism through which we initiate the switching from *ZLight* (resp., *Quorum*, *Chain*) in Step Z4 (resp., Q3, C4).

Step P1. If the client does not commit request req by the expiration of timer T (triggered in Steps Z1/Q1/C1), 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, client periodically 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/Q2/C3) and sends $\langle \text{ABORT}, req.t_c, LH_j, next(i) \rangle_{\sigma_{r_j}}$

to c.

Step P3. When client c receives $2f + 1$ $\langle \text{ABORT}, req.t_c, *, next(i) \rangle$ messages with correct signatures from different replicas and the same value for $next(i)$, the client collects these messages into the set $Proof_{AH_i}$, and extracts the abort history AH_i from $Proof_{AH_i}$ as follows:

- First, c generates history h 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.
- Finally, AH_i is the longest prefix of h in which no request appears twice.

B.6 Handling init histories and switching

To switch from *ZLight*, *Quorum* or *Chain* instance i, client invokes instance $i' = next(i)$ by accompanying req with init history $IH_{i'} = AH_i$ and $Proof_{AH_i}$. Then a replica running instance i', executing its first request in i' (e.g., in Step C3 of *Chain*), simply makes the library call to verify IH'_i against $Proof_{AH_i}$ following the algorithm given in Sec. B.5 and verifies that ABORT messages in $Proof_{AH_i}$ indeed declare i' as $next(i)$. In the case of switching to *Backup*, this check is simply a part of the functionality implemented on top of the underlying BFT.

To switch from *Backup*[9], *Backup* replicas must provide the client with $f + 1$ different signatures of the identical abort history and the next Abstract instance id i'. This is a reasonable requirement on the BFT that underlies *Backup*, since any BFT protocol must anyway provide an identical reply from at least $f + 1$ replica; in the case of *Backup* abort history, we just require this particular reply to be signed (we trivially implemented this in PBFT). Then, the client includes these signatures with req in its invocations of an Abstract i' (e.g., *ZLight*) and replicas running i', before executing the request (e.g., in Step Z3 of *ZLight*), simply verify the signatures against the submitted init history to find $f + 1$ matching ones.

In all cases of switching, a replica running instance i' executes all the requests contained in the *first verified* init history IH'_i before executing the invoked request itself. Replicas simply ignore all subsequent init histories. Below we summarize the additional init history related actions performed by processes in steps of *ZLight*, *Quorum* and *Chain*. In the following, we assume that the verification of init histories is performed as described above.

Step Z1/Q1/C1. On $Invoke_{i'}(req, IH)$, the message(s) sent by the client contain also IH and the set of signatures $Proof_{IH}$ returned by the preceding Abstract i, where

---

[9] In *AZyzzyva*, *Backup* switches to *ZLight*, whereas in *Aliph* it switches to *Quorum*.

$i' = next(i)$.

Step Z2/C2. If its local history LH_1 is empty, the primary/head r_1 executes the step only if IH can be verified against $Proof_{IH}$.

Step Z3/Q2/C3. If its local history LH_j is empty, the replica r_j executes the step only if it receives IH that can be verified against $Proof_{IH}$. If so, then (before executing req) r_j executes all the requests contained in IH (i.e., r_j sets LH_j to IH); then r_j executes 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 IH can be verified against $Proof_{IH}$. Then, before executing the step as described in Section B.5, r_j first sets LH_j to IH.

B.7 Lightweight checkpointing subprotocol

In *ZLight*, *Quorum* and *Chain* we use a *lightweight checkpoint subprotocol* (LCS) to truncate histories every CHK requests (in our evaluations, $CHK = 128$), similarly to checkpoint protocols used in [6, 15]. Here, we explain our simple LCS and its impact on our implementations as presented earlier in this appendix.

LCS consists in the following:

1. every replica r_j increments the 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.

3. If replica r_j receives the digest of the same state st 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 and (b) stores cc to variable $lastcc$. Such a checkpointed state (referred to as 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}.

LCS has no impact on *ZLight*, *Quorum* and *Chain* as described earlier in this appendix, with a single exception, related to client extraction of abort histories from the received ABORT messages (see Step P3, Sec. B.5). Namely, if the client receives a history from some replica r_j consisting of a checkpointed state followed by CHK requests, the client will first collapse all such histories into the single checkpointed state (i.e., the client will perform the checkpoint on behalf of the replica). Only in case the client cannot retrieve $t + 1$ confirmations of (some) checkpointed state when executing Step P3 in this way, the client will repeat the procedure described in this step with replica histories as received from replicas, i.e., precisely as described in Step P3, Section B.5.

C. Correctness proofs

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

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

C.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).

<u>Well-formed commit indications.</u> It is easy to see that the reply returned by a commit indication always equals $rep(h))$, where (commit history) h 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 of some history h' and identical reply digests from all $3f+1$ replicas including at least $2f+1$ correct ones. By Step Z3 of *ZLight* (reps., Q2 of *Quorum*), a digest of the reply sent by a correct replica is $D(rep(h')$. Hence, h' is exactly a commit history h and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica executes an invoked request before sending a RESP message in Step Z3 (resp., Q2), it is straightforward to see that if req is committed with a commit history h_{req}, then req is in h_{req}. □

<u>Validity.</u> 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 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 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. B.5). □

<u>Termination.</u> By assumption of a quorum of $2f+1$ correct replicas and fair-loss links: (1) correct replicas eventually receive the PANIC message sent by correct client c (in Step P1) 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 1. *Let r_j be a correct replica and LH_j^{req} the state of LH_j upon r_j executes 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. □

<u>Commit Order.</u> Assume, by contradiction, that there are two committed requests req (by benign client c) and $req' \neq req$ (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 1. □

<u>Abort Order.</u> 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 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), $r_{j'}$ executed req before sending m'. However, by Lemma 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. B.5) every commit history h_{req} is a prefix of every abort history. □

<u>Init Order.</u> By the clarifications of Step Z3/Q2 and Step P2 given in Section B.6, 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 at least 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 processes and contain

local histories that have CP as a prefix. Hence, by Step P3, CP is a prefix of any abort history. □

ZLight Progress. Recall that *ZLight* guarantees to commit clients' requests if: there are no replica/link failures and Byzantine client failures. We assume that the message processing at processes takes negligible time and that clients set the timer T triggered in Step Z1 to 3Δ. 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, that: (i) by $t + \Delta$ the primary receives req and take Step Z2 and (ii) by time $t + 2\Delta < t_{PANIC}$ the replicas take 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 $3f + 1$ identical replies (Step Z4), commits req and never panics. A contradiction. □

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 message processing at processes takes negligible time and that the timer T triggered in Step Z1 to 2Δ. 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 take 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 $3f + 1$ identical replies (Step Q3), commits req and never panics. A contradiction. □

C.2 Chain

In this Section, we prove that *Chain* implements Abstract with Progress equivalent to that of *ZLight*.

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., $\Sigma_{last} = \{r_j \in \Sigma : i > 2t\}$. In addition, we say that correct replica r_j

executes req *at position pos* if $sn_j = pos$ when r_j executes req.

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

Lemma 2. *If correct replica r_j executes req (at position sn, at time t_1), then all correct replicas s_j, $1 \le j < i$ execute 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 executes req (at position sn), such that there is a correct replica r_x ($x < j$) that never executes 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 executes 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 $\overleftarrow{R_j}$ execute req at position sn. If $r_x \in \overleftarrow{R_j}$, r_x must have executed 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. □

Lemma 3. *If benign client c commits req with history h (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.*

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 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 (resp., replica) cannot commit req with h unless it receives a MAC authenticating req and $D(h')$ from every correct replica in Σ_{last}. Hence the lemma.

□

Well-formed commit indications. By Step C4, in order to commit a request 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 and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica 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}. □

Validity. For any request req to appear in a abort (resp., commit) history h, at least $f + 1$ replicas must have 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 executed req.

Now, we show that correct replicas execute only requests invoked by clients. By contradiction, assume that some correct replica executed a request not invoked by any client and let r_j be the first correct replica to execute such a request req' in Step C3 of *Chain*. In case $j < f + 1$, r_j executes 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 2 yields a contradiction with our assumption that r_j is the first correct replica to execute req'.

Moreover, by Step C3, no replica executes 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(ii) Sec. B.5). $\square$

Termination. See the proof of Termination for *ZLight*/*Quorum* (Sec. C.1).

Moreover, to see that a committed request req must be in its commit h_{req}, notice that a client needs to receive the MAC for the same local history digest $D(h_{req})$ from all $f + 1$ from Σ_{last} including at least one correct replica r_j. By Step C3, r_j executes req and appends it to its local history LH_j before authenticating the digest of LH_j; hence, $req \in h_{req}$. $\square$

Commit Order. Assume, by contradiction, that there are two committed request req (by benign client c) and $req' \neq req$ (by benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. By Lemma 3, there is correct replica $r_j \in \Sigma_{last}$ that 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 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 commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. By Lemma 3 and the assumption of at most f faulty replicas, all correct replicas (at least one) from Σ_{last} 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 ind among all replicas in Σ_{last}. By Lemma 2, all correct replicas execute all the requests in h_{req_C} at the same positions these requests have in h_{req_C}. In addition, a correct replica executes 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.

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