Trees and Turtles: Modular Abstractions for State Machine Replication Protocols

Natalie Neamtu
Microsoft Corporation
Redmond, WA, USA
nan55@cornell.edu

Haobin Ni
Cornell University
Ithaca, NY, USA
haobin@cs.cornell.edu

Robbert van Renesse
Cornell University
Ithaca, NY, USA
rvr@cs.cornell.edu

Abstract

We present two abstractions for designing modular state machine replication (SMR) protocols: *trees* and *turtles*. A tree captures the set of possible state machine histories, while a turtle represents a subprotocol that tries to find agreement in this tree. We showcase the applicability of these abstractions by constructing crash-tolerant SMR protocols out of abstract tree turtles and providing examples of tree turtle implementations. The modularity of tree turtles allows a generic approach for adding a leader for liveness. We expect that these abstractions will simplify reasoning and formal verification of SMR protocols as well as facilitate innovation in protocol designs.

1 Introduction

State machine replication (SMR) is a widely-used paradigm in distributed and decentralized services, wherein a set of processors provides an abstraction of a single state machine with an ever-growing history [12]. In the face of possible processor failures and unbounded communication delays, the challenge lies in ensuring that nodes always agree on the history while allowing updates to be made in as timely a manner as possible.

Traditional SMR protocols are usually constructed around the notion of an unbounded sequence of slots. The goal of such a protocol is to fill the slots with values. A typical protocol consists of an unbounded series of rounds where the contents of at most one slot may be decided in a round. Some protocols are only able to fill a single slot (i.e., [1, 5]), thus an unbounded number of instances of the protocol must be used to implement SMR. Higher throughput is achieved by running multiple instances in parallel, either independently [4, 9] or in a pipelined fashion [8, 13], or by putting a batch of values in each slot.

We present here two abstractions that break this slot-by-slot paradigm: *trees* and *turtles*. Referring to a sequence of values as a *chain*, the set of such chains forms a tree under the *is-a-prefix-of* relation. As was proposed in [10], we generalize the slot-based scheme for constructing SMR protocols to allow entire chains to be decided at once, extending the state machine history down a path through the tree. We then generalize the notion of a round to an abstract subprotocol which can be used to decide one chain. We refer to such protocols as turtles because they are stacked *in infinitum* to construct an SMR protocol. Taken together, the result is protocols called *tree turtles*.

Using chains requires our subprotocols to form a consensus out of a set with richer algebraic structures than the traditional set of singlar values. In this case, the algebraic structure we use is the meet-semilattice formed by the ancestor relation between the nodes of the tree as the partial order and the lowest common ancestor of a set of nodes as the meet operator. This structure is utilized in the tree turtle protocols and their proofs of correctness that we present in this paper.

We expect that our abstractions can lead to various advantages over traditional SMR approaches. Proposing chains allows processors to specify preferred orderings of values in the state machine history. Reasoning about SMR protocols is made simpler with tree turtles because the never-terminating execution is factored out; this can lead to more reusable proofs in both an informal an formal setting. Tree turtles themselves have simple proofs when compared to existing SMR protocols. The modularity of tree turtles also enables the design of heterogeneous protocols that dynamically adapt to their workloads or operating conditions.

For liveness, we show how to compose our turtle abstraction with a *leader* which attempts to eliminate contention between processors. Different from traditional approaches, our protocol does not require a non-faulty leader to make progress under favorable conditions.

2 Trees and Turtles

Put simply, the goal of SMR is to allow a set of processors agree on an ever-growing sequence of values in a fault-tolerant manner. Rather than focusing on the individual values in a sequence, we will consider how an agreement can be formed on an entire sequence, or a *chain*, at once. Doing so utilizes the tree structure present in sets of chains.

Consider a processor p that believes the state machine history is represented by a chain c. If another processor p' believes the state machine is represented by a different chain c', then p and p' should be able to agree on a common history. Beyond the simplest case where $c = c'$, consider whether one chain is a *prefix* of the other. If this is the case, the processor who proposed the shorter chain could later “catch up"

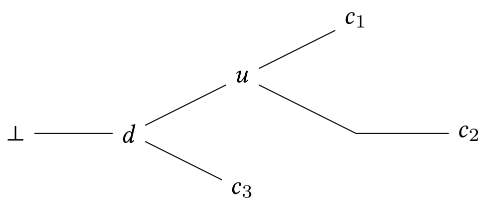

Figure 1. Illustrating the tree structure formed by trees and the is-a-prefix-of relation. Chain u is the longest common prefix of chains c_1 and c_2. Chain d is the longest common prefix of chains c_1, c_2, and c_3.

by extending its chain to the longer chain, without needing to modify the earlier state machine history.

If p and p' attempt to establish agreement on an extension of the state machine history by proposing chains c and c' to each other, then we could consider that p and p' agree on the *longest common prefix* of c and c'. This is the (possibly empty) longest identical subsequence that can be found starting from the beginning of each chain. The longest common prefix can readily be generalized to any number of chains; Figure 1 depicts a tree formed by three chains where the longest common prefixes are ancestor nodes.

Trees[1], then, can be viewed as fundamental to state machine replication: processors continually propose new chains to each other in order to keep extending the longest agreed-upon path through the tree. Table 1 summarizes the notation which we will use for chains in this paper.

Table 1. Summary of notation for chains.

$\bot$	the empty chain
$c \preceq c'$	c is a *prefix* of c' (or, c' is an *extension* of c)
$c \simeq c'$	$c \preceq c'$ or $c' \preceq c$ (we say that c and c' *agree*)
$c \sqcap c'$	the longest common prefix of c and c'

Now we will turn our attention to the structure of an SMR protocol. To perform SMR, processors alternate between proposing new extensions to the longest agreed-upon chain, collecting proposals from other processors, and deciding what the new longest agreed-upon chain is. This process repeats forever to ensure that the longest agreed-upon chain is ever-growing.

Thus, we may extract a natural building block from this structure: a subprotocol in which processors propose and subsequently decide on a single chain. While many subprotocol implementations are possible, we establish a common specification for the properties they should have. Once we have one (or many) such subprotocols, constructing a SMR protocol is simple: we stack these subprotocols on top of each other in an unending sequence. Owing to this infinite repetition, and inspired by the saying *"turtles all the way down,"* we refer to our building blocks of SMR protocols as *turtles.* Since our protocols combine the ideas of trees and turtles, we will call them *tree turtles* (Figure 2).

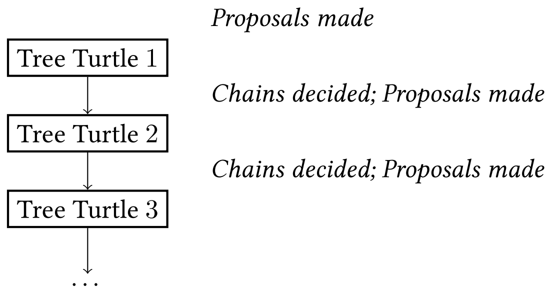
Proposals made
Tree Turtle 1
Chains decided; Proposals made
Tree Turtle 2
Chains decided; Proposals made
Tree Turtle 3
...

Figure 2. Tree turtle protocols are stacked to implement a state machine replication protocol.

Tree turtles reduce the problem of solving SMR into making a single proposal and decision. The specification for tree turtles requires them to always terminate (Section 5), while SMR requires that the protocol goes on forever (Section 4). The tree turtle implementations are encapsulated from the correctness of the SMR protocol proved in Section 6 making those proofs reusable across implementations. This also readily demonstrates the potential of heterogeneous SMR protocols composed from multiple types of tree turtles.

3 System Model

We will denote the set of processors participating in a protocol as $\mathcal{P}$. Processors can communicate with all other processors by sending messages over a network.

Failures. Each processor p can either be correct or faulty. Faulty processors may fail by crashing, at which point they may stop executing indefinitely. Which processors are faulty is not known *a priori.*

Network. The network is assumed to be *reliable* in the following sense: if a processor p sends a message to a correct processor q, then q eventually receives that message. We also assume that the network doesn't forge or garble messages, meaning that if a processor q receives a message from a processor p, then p actually sent that message.

Asynchrony. We assume that there are no upper bounds on the difference between processor speeds or network latency. This means that processors are not able to distinguish between faulty processors and correct processors that are simply slow or whose messages have yet to be delivered.

Quorums. A processor can never expect to receive a message from all other processors due to the possibility that some have crashed. To address this, a processor instead waits

[1] In addition to being like trees in a loose conceptual sense, chains, taken with the partial order $\preceq$, satisfy the set-theoretic definition of a tree. This is because for each chain c, the set of its prefixes $\{c' \mid c' \preceq c\}$ is totally ordered by $\preceq$. As mentioned in the introduction, it is possible to generalize trees even further to a semilattice.

for a *quorum* which is a subset of the processors in $\mathcal{P}$. A *quorum system* Q is a set of quorums ($Q \subseteq 2^{\mathcal{P}}$) which satisfies the following:

- We assume that any quorum system Q contains a special quorum, which we will denote Q^*, that consists entirely of correct processors.
- We say that a quorum system satisfies k*-intersection* if any set of k quorums have at least one processor in common [7].

Appendix B shows how $k-$intersecting quorum systems can be implemented using threshold quorums if $|\mathcal{P}| > k \cdot f$, where f is the maximum number of faulty processors.

4 State Machine Replication Specification

Below we give the requirements for a SMR protocol, in which processors in $\mathcal{P}$ alternate *proposing* and *deciding* chains.

Definition 4.1 (State Machine Replication Specification)**.**

- *SMR-Agreement*: if a processor p decides chain d and a processor p' decides chain d', then $d \simeq d'$ (even if $p = p'$).
- *SMR-Validity*: if a processor decides chain d, then some processor proposed a chain c such that $d \preceq c$.
- *SMR-Relay*: if a correct processor decides chain d, then eventually all correct processors decide d or an extension of d.
- *SMR-Monotonicity*: if a processor decides chain d and later another chain d', then $d \preceq d'$.
- *SMR-Progress*: if a correct processor proposes a chain that it has not decided before, then that processor eventually decides a chain it has not decided before.

Note that it is impossible for a protocol to satisfy all properties if the system is asynchronous [6]. For our tree turtle protocols in Section 7, we will focus on the first four properties. We will return to SMR-Progress in Section 8 when we introduce synchrony assumptions.

5 Tree Turtle Specification

A tree turtle is a fault-tolerant protocol executed by the processors in $\mathcal{P}$. Processors construct an input and produce an output for each tree turtle. An input to a tree turtle is a chain c, which we will denote in brackets $\langle c \rangle$. An output of a tree turtle is a pair of chains $\langle d, u \rangle$. Since tree turtles are subprotocols, we do not assume here that all processors—including non-crashed ones—will execute a given tree turtle. We show in Section 6 that in our construction, all correct processors will in fact execute each tree turtle.

A tree turtle must satisfy the following properties:

Definition 5.1 (Tree Turtle Specification)**.**

- *Turtle-Termination*: if each correct processor constructs an input, then eventually each correct processor produces an output.
- *Turtle-Agreement*: for any two outputs $\langle d, u \rangle$ and $\langle d', u' \rangle$, $d \preceq u'$ and $d' \preceq u$.
- *Turtle-Unanimity*: for any chain w, if $w \preceq c$ for all inputs $\langle c \rangle$, then $w \preceq d$ for all outputs $\langle d, u \rangle$.
- *Turtle-Validity*: if some processor produces an output $\langle d, u \rangle$, then some processor must have produced an input $\langle c \rangle$ such that $u \preceq c$.

Turtle-Agreement ensures agreement between any two outputs of a turtle: note that if both $d \preceq u$ and $d' \preceq u$, then either $d \preceq d'$ or $d' \preceq d$. Further, the case where $\langle d, u \rangle = \langle d', u' \rangle$ implies that $d \preceq u$ for each output. Turtle-Unanimity ensures that if there is an agreement in the inputs (i.e., proposals) to a turtle, then that agreement is reflected in the outputs. Turtle-Validity means that each output of a turtle be a prefix of an input to that turtle. Unlike SMR protocols, we are able to guarantee the liveness properties for tree turtle protocols (Turtle-Termination).

6 Tree Turtles All The Way Down

We will now show that tree turtle protocols can be composed with each other to implement a protocol satisfying SMR-Agreement, SMR-Validity, SMR-Relay, and SMR-Monotonicity.

The construction works as follows. All processors in $\mathcal{P}$ are configured with instructions to execute the same unbounded sequence of tree turtles numbered $1, 2, 3, \dots,$ and so on. For convenience, we will extend the tree turtle inputs and outputs $\langle i, c \rangle$ and $\langle i, d, u \rangle$ to now include the tree turtle number i. We assume that each processor is initialized with the tuple $\langle 0, \bot, \bot \rangle$ as the output for the non-existent tree turtle 0. Then the processors begin executing the tree turtles in succession. If a processor produces tree turtle output $\langle i, d, u \rangle$, that processor decides the chain d. It then proposes a new chain c, selecting c such that $u \preceq c$, and constructs an input $\langle i + 1, c \rangle$ for the next tree turtle $i + 1$.

By choice of $u \preceq c$, this construction ensures the following lemma:

Lemma 6.1. *If a processor produces the output $\langle i, d, u \rangle$ for tree turtle i, then for all inputs to subsequent tree turtles $\langle j, c \rangle$ where $i < j$, it must be that $d \preceq c$.*

Proof. Suppose $\langle j, c \rangle$ is an input to tree turtle j made by a processor p. Proceed by induction on $j - i$. If $j = i + 1$, then by the above construction, p must have produced $\langle i, d, u \rangle$ as the output of tree turtle i for some chain $u \preceq c$. By Turtle-Agreement, we also know that $d \preceq u$ which implies $d \preceq c$. In the inductive case, we again know that p must have produced an output $\langle j - 1, d', u' \rangle$ with $d' \preceq u' \preceq c$ in the previous tree turtle. The inductive hypothesis gives us that $d \preceq c'$ for all inputs to tree turtle $j - 1$ $\langle j - 1, c' \rangle$. Thus, the condition for Turtle-Unanimity is satisfied for tree turtle $j - 1$, and so it must be that $d \preceq d' \preceq c$. □

Further, we can use an inductive argument to see that by Turtle-Termination, each correct processor will eventually complete each tree turtle.

Lemma 6.2. *Every correct processor eventually produces an output for each tree turtle.*

Proof. Trivially, each non-crashed processor constructs an input to tree turtle 1. Thus, Turtle-Termination ensures that the base case of the induction is satisfied. In the inductive case, each non-crashed processor will use its output of tree turtle i to construct an input to tree turtle $i + 1$, and so an analogous argument applies. □

Now we proceed to the main proof:

Theorem 6.3. *The composition of tree turtles implements a protocol satisfying SMR-Agreement, SMR-Validity, SMR-Relay, and SMR-Monotonicity.*

SMR-Agreement. Suppose that two processors p and p' decide chains based on their outputs $\langle i, d, u\rangle$ and $\langle j, d', u'\rangle$, respectively. First, suppose that $i = j$. By Turtle-Agreement, $d \preceq u'$, and we also know that $d' \preceq u'$. Then, since d and d' are prefixes of the same chain u', it must be that either $d \preceq d'$ or $d' \preceq d$. Thus, the decided values agree. Now suppose that $i < j$. By Lemma 6.1, we know that $d \preceq c$ for all inputs $\langle j, c\rangle$ to tree turtle j. By Turtle-Unanimity, we have that $d \preceq d'$.

SMR-Validity: Suppose that some processor p decides d. This means that p produces an output $\langle i, d, u\rangle$ for some tree turtle i and chain u. By Turtle-Validity, there must exist an input $\langle i, c\rangle$ to the same tree turtle such that $u \preceq c$. So, c was proposed by some processors. And since we must have $d \preceq u$, we know $d \preceq c$.

SMR-Relay: If a correct processor decides d as a result of its output of tree turtle i, then by Lemma 6.1 and Turtle-Unanimity, any processor that completes tree turtle $i+1$ will decide d or an extension of d, and Lemma 6.2 gives us that all correct processors will do exactly as such.

SMR-Monotonicity: If a processor decides d as a result of its output of tree turtle i, then by Lemma 6.1 and Turtle-Unanimity, any processor that completes tree turtle $i+1$ will decide d or an extension of d. So by induction, any later decision will be monotonically extending d.

□

The above protocol does not, by itself, satisfy SMR-Progress. However, the fact that the correct processors eventually complete each turtle (Lemma 6.2) can be used to make an auxiliary argument for SMR-Progress for a specific protocol (for instance, using a probabilistic termination argument).

7 Tree Turtle Implementations

Here we present two possible tree turtle implementations.

7.1 One-Step Tree Turtle

Our first tree turtle protocol uses only a single round of communication between processors, making it a *one-step* protocol [3]. It requires a quorum system satisfying 3-intersection.

A processor p executing the One-Step Tree Turtle protocol proceeds as follows:

1a. p produces an input $\langle c\rangle$ and broadcasts it to all processors, including itself;
1b. p waits to receive inputs $\langle c_s\rangle$ from all processors s in any quorum Q_p;
1c. p produces $\langle d, u\rangle$, where:
  i. $d = \bigsqcap_{s\in Q_p} c_s$;
  ii. Let $C_p = \left\{ \bigsqcap_{s\in Q_p\cap Q} c_s \mid Q \in \mathcal{Q}\right\}$. Then $u = \max(C_p)$.

That is, d is simply the longest common prefix of the received proposals. To compute u, p considers all quorums Q. For each such quorum, p determines the longest common prefix on the proposals it received from the processors in Q. We show below that all these subchains agree with one another. p then selects the longest.

Lemma 7.1. *All elements of C_p agree.*

Proof. All of the elements in C_p are computed by taking the intersection of Q_p with another quorum. Let Q, Q' be any two quorums and x, x' elements of C_p where $x = \bigsqcap_{s\in Q_p\cap Q} c_s$, and $x' = \bigsqcap_{s\in Q_p\cap Q'} c_s$. Since $\mathcal{Q}$ satisfies 3-intersection, $Q_p \cap Q \cap Q'$ is non-empty. Let r be some processor in this intersection. By the use of $\sqcap$, we have that $x \preceq c_r$ and $x' \preceq c_r$. This means that either $x \preceq x'$ or $x' \preceq x$. Thus, all elements of C_p agree with each other. □

Since $\preceq$ is transitive, C_p has a maximum element according to $\preceq$, and so the computation of u is well-defined. Further, we can show a stronger property about C_p.

Lemma 7.2. *For all $x \in C_p$, $d \preceq x$.*

Proof. For any quorum Q, we can observe that $Q_p \cap Q \subseteq Q_p$. The longest common prefix over a subset of inputs belonging to processors Q_p is at least as long as the longest common prefix over Q_p. Thus, $d \preceq x$ for any $x \in C_p$. □

Now we proceed to show that this protocol implements a tree turtle.

Theorem 7.3. *The One-Step Tree Turtle protocol satisfies the tree turtle specification (Definition 5.1).*

Turtle-Termination: Suppose that all correct processors construct an input to the tree turtle. The only point in the protocol where a given correct processor p will wait is to receive messages from a quorum of processors at step 1b. Since there is assumed to be a quorum Q^* that consists entirely of correct processors, and the network reliably delivers messages between correct processors, p will need to wait no longer than it takes for the messages from all processors in Q^* to be delivered. Thus, p will be able to complete the turtle at step 1c.

Turtle-Agreement: Suppose that two processors p and p' produce $\langle d, u\rangle$ and $\langle d', u'\rangle$, respectively. For all processors $r \in Q_p \cap Q_{p'}$, both p and p' received the proposal c_r from r. Letting $x = \bigsqcap_{r \in Q_p \cap Q_{p'}} c_r$, we see that x is present in both C_p and $C_{p'}$. By Lemma 7.2, $d \preceq x$, and by the maximality of u' over $C_{p'}$, $x \preceq u'$. So, $d \preceq u'$ by transitivity. The same argument can be used to show that $d' \preceq u$.

Turtle-Unanimity: Suppose that there exists a common prefix $w \preceq c$ for all inputs $\langle c\rangle$ to the turtle. Because of step 1b in the protocol, all of the c_s values used to compute $\langle d, u\rangle$ came from inputs $\langle c_s\rangle$. This means that w is a prefix of each c_s, and so w must be a (not necessarily strict) prefix of the longest common prefix $d = \bigsqcap_{s \in Q_p} c_s$ of the proposals.

Turtle-Validity: Suppose that p produces the output $\langle d, u\rangle$. We know that $u = \bigsqcap_{s \in Q_p \cap Q} c_s$ is a prefix of all proposals made by processors in $Q_p \cap Q$ for some quorum Q. So, taking any processor r in the intersection $Q_p \cap Q$, we know that $u \preceq c_r$, where $\langle c_r\rangle$ was the input produced by r. □

7.2 Lower-Bound Tree Turtle

Now we will present a second tree turtle protocol, the Lower-Bound Tree Turtle. This protocol meets the lower bound on the intersection properties of the quorum system needed to solve SMR: namely, 2-intersection in the crash failure case [2]. With these weaker assumptions about the quorum system, we can design a protocol that makes it safe for processors to output a chain based on messages from all processors in a quorum under the condition that the chains in all such messages agree. Satisfying this condition requires an additional round of communication.

A processor p executing the Lower-Bound Tree Turtle protocol proceeds as follows:

1a. p produces an input $\langle c\rangle$ and broadcasts it to all processors, including itself;
1b. p waits to receive inputs $\langle c_s\rangle$ from all processors s in any quorum Q_p^1;
1c. p computes $x = \bigsqcap_{s \in Q_p^1} c_s$;
2a. p broadcasts $\langle x\rangle$ to all processors, including itself;
2b. p waits to receive messages $\langle x_s\rangle$ from all processors s in a quorum Q_p^2;
2c. p produces $\langle d, u\rangle$, where $d = \min_{s \in Q_p^2} x_s$ and $u = \max_{s \in Q_p^2} x_s$.

Appendix A contains the proof of correctness for the Lower-Bound Tree Turtle.

7.3 Message Size

The protocol we discussed uses messages containing chains that represent the entire state machine history. As the size of this history grows, this quickly becomes impractical. However, it is not necessary for a processor to broadcast a chain in subsequent turtles once it has decided that chain following the construction in Section 6.

If processor p decides chain d as a result of its output of a tree turtle, then any other processor p' that outputs $\langle d', u'\rangle$ from the same tree turtle will have $d \preceq u'$ by Turtle-Agreement. Thus, p' already knows the contents of the chain d, and p may omit that chain prefix in its proposals to subsequent turtles.

7.4 Heterogeneous Protocols

The simplest way to construct an SMR protocol using the proposed abstractions is to use a single tree turtle protocol, instantiated an unbounded number of times. There are other options, however, briefly discussed in this section.

Different tree turtle protocols may have different normal case or worst case performance properties. The Lower-Bound Tree Turtle, when combined with a leader as discussed in Section 8, has good normal case performance properties, but it relies on synchrony assumptions for liveness. A similar protocol, borrowing ideas from the Ben-Or protocol using randomness [1], can provide termination almost surely but has bad normal case performance. By alternating between the two protocols, we can achieve the best of both worlds.

8 Leaders as an Abstraction

Processors may make different proposals to a tree turtle, preventing them from being able to decide new chains. This issue of contention has been addressed previously in SMR protocols by using a *leader* which drives all processors to use the same proposals. In existing leader-based consensus protocols, the leader lies in the critical path of the protocol: without a functioning leader, no decisions may be made. We show that using tree turtles, leaders can be easily *factored out* so that their only role is to help the protocol towards making decisions.

Leaders can be introduced to an existing tree turtle protocol as follows:

- The leader ℓ for tree turtle i is processor with identifier $i \pmod{|\mathcal{P}|}$, meaning the role of leader rotates through all of the processors.
- The leader ℓ for tree turtle i broadcasts its input c_ℓ to turtle i.
- All processors set a timer and wait for the leader's message. If a processor p receives c_ℓ before the timer expires, then it uses c_ℓ as its input to tree turtle i. Otherwise, it proceeds normally.
- The processors double the length of the timer in each tree turtle.

Under synchronous conditions, the leader is able to eliminate contention. The proof requires reasoning about quorums and messages, but these concepts generalize beyond the protocols presented in Section 7.

Lemma 8.1. *Using the above construction and the protocols in Section 7, if there exists an upper bound Δ such that a message sent between two non-crashed processors is delivered and* *processed within Δ, then there will be an unbounded number of tree turtles where the leader's message is received before all timers expire.*

Proof. Consider any tree turtle i after the point in the execution where the timers have become larger than $\Delta \cdot 2^{|\mathcal{P}|}$. In general, the leader for a tree turtle may not have started executing that tree turtle at the point when other processors begin waiting for its message. Applying Lemma 6.2, consider a processor p that has begun tree turtle $i + 1$. Then there must be a quorum of processors who have begun tree turtle i. This is because in all of our protocols, p must wait to receive messages from a quorum before completing the protocol. Let Q be this quorum. Since $Q \cap Q^*$ is non-empty, there must be a tree turtle in the next $|\mathcal{P}|$ instances whose leader is a correct processor in Q. Let j be the first such instance and let ℓ be the leader for tree turtle j.

Since ℓ has already begun tree turtle i, it must catch up at most $|\mathcal{P}|$ tree turtles to reach j. Let t be the timer length for tree turtle j. Since all timer lengths are greater than 2Δ, ℓ will have received messages from a quorum for every tree turtle up to j. A period of Δ is sufficient for other processors to receive ℓ's input to tree turtle j. However, this does not include the timers that ℓ must set for tree turtles $i, \ldots, j-1$. Since the timer lengths are doubled after each tree turtle, the total time required is $\Delta + (t/2^{|\mathcal{P}|} + t/2^{|\mathcal{P}|-1} \cdots + t/2) = \Delta + (1-2^{-|\mathcal{P}|})t$. Since $t > \Delta \cdot 2^{|\mathcal{P}|}$, ℓ's message will be received before the timers expire. For any $|\mathcal{P}|$ turtles, except for a constant number of turtles in which failures happen, there must exist a distinct turtle that satisfies the above property, thus there will be an unbounded number of such turtles. □

Theorem 8.2. *Using the above construction and the protocols in Section 7, if there exists an upper bound Δ such that a message sent between two non-crashed processors is delivered and processed within Δ, then SMR-Progress is satisfied for the composition of tree turtles (Section 6).*

Proof. According to Lemma 8.1 there will be an unbounded number of tree turtles where the leader's message is received before all timers expire. Turtle-Unanimity provides that the processors who complete the turtle will decide the leader's chain. It follows that correct processors will always eventually be able to decide a new chain.

□

9 Related Work

Abstracting the problem of achieving consensus on a single value to entire sequences was previously applied to Paxos in Generalized Paxos [10]. This work further generalizes chains to partial-orders of values in which non-interfering values commute. The HotStuff protocol [13] conceptualizes the state machine history as a tree, but it only extends the tree by a single node at a time. The same can be said of blockchain protocols.

The idea of heterogeneous SMR protocols is similar to protocols which have different "modes". Typically there is one mode which is considered the normal operation of the protocol and another designed for fast-tracking decisions under best-case conditions. One such example is Fast Paxos [11] which is able to skip a round of communication in periods without contention. These modes, however, are usually considered to be part of a single protocol instead of separate protocols satisfying a common specification.

10 Conclusion & Future Work

SMR protocols have been around for over thirty years. We revisit the structure of these protocols and propose new abstractions—*trees and turtles*—for the design of modular SMR protocols.

While this paper did not discuss the performance of tree turtle protocols, we believe that they have potential to be performant through their ability to drive long extensions to the state machine history in a few rounds of communication. Future work could include an empirical analysis of their performance. Further techniques for optimization can also be investigated such as pipelining chains from different rounds of the protocol simultaneously.

There is work in progress by the authors on building formally verified consensus and SMR protocols in both Dafny and Coq based on the presented abstractions due to their simplicity and efficiency. We also expect our abstractions to support different flavors of SMR, such as ordered consensus and heterogeneous consensus, by utilizing the additional structure of trees and the flexibility of the tree turtle composition. Further generalizations of trees and chains into more general algebriac structures are also possible.

Acknowledgments

The authors would like to thank Pierre Sutra and the anonymous reviewers for their helpful suggestions.

References

[1] M. Ben-Or. 1983. Another advantage of free choice: Completely Asynchronous Agreement Protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*. ACM SIGOPS-SIGACT, ACM Press, Montreal, Quebec, 27–30.

[2] G. Bracha and S. Toueg. 1983. Resilient Consensus Protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*. ACM SIGOPS-SIGACT, Montreal, Quebec, 12–26.

[3] F.V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. 2001. Consensus in One Communication Step. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*. Springer-Verlag, London, UK, 42–50.

[4] M. Castro and B. Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX, New Orleans, LA.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[6] M.J. Fischer, N.A. Lynch, and M.S. Patterson. 1983. Impossibility of Distributed Consensus with one Faulty Process. In *Proceedings of the*

2nd Symposium on Principles of Database Systems (PODS'83). ACM SIGACT-SIGMOD, Atlanta, GA.

[7] F.P. Junqueira and K. Marzullo. 2005. Replication predicates for dependent-failure algorithms. In *Proceedings of the 11th Euro-Par Conference (Lecture Notes on Computer Science, 3648).* Springer-Verlag, Monte de Caparica, Portugal, 617–632.

[8] F.P. Junqueira, B.C. Reed, and M. Serafini. 2011. ZAB: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN).* 245–256. https://doi.org/10.1109/DSN.2011.5958223

[9] L. Lamport. 1998. The Part-Time Parliament. *Trans. on Computer Systems* 16, 2 (1998), 133–169.

[10] L. Lamport. 2005. Generalized consensus and Paxos. (2005).

[11] L. Lamport. 2006. Fast Paxos. *Distrib. Comput.* 19, 2 (oct 2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

[12] F.B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Comput. Surveys* 22, 4 (Dec. 1990), 299–319.

[13] M. Yin, D. Malkhi, M.K. Reiter, G.G. Gueta, and I. Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19).* Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

Appendix A: Correctness of Lower-Bound Tree Turtle

First we show that all chains computed in step 1c agree:

Lemma 10.1. *If processors p and p' compute x and x' in step 1c respectively, then $x \simeq x'$.*

Proof. Since there must be some processor r in the intersection $Q_p^1 \cap Q_{p'}^1$, both p and p' received the same chain c_r from r. Then, since both x and x' are prefixes of c_r, x and x' must agree. □

Note that the preceding lemma shows that the rules to compute d and u are well-defined since all of the chains x_s for $s \in Q_p^2$ must agree. The next lemma is trivial and shows that a common prefix to all proposals is preserved in the chains computed at step 1c.

Lemma 10.2. *If there exists a chain w such that $w \preceq c$ for all inputs $\langle c \rangle$, then for any processor p that computes x in step 1c, $w \preceq x$.*

We now show that the Lower-Bound protocol implements the specification of a tree turtle.

Theorem 10.3. *The Lower-Bound Tree Turtle protocol satisfies the tree turtle specification (Definition 5.1).*

Turtle-Agreement: Suppose that two processors p and p' produce $\langle d, u \rangle$ and $\langle d', u' \rangle$, respectively. There must be some processor r in the intersection $Q_p^2 \cap Q_{p'}^2$, and both p and p' received the same chain x_r from r. By Lemma 10.1 and the minimality of d, it must be that $d \preceq x_r$. And by the maximality of u', we have that $x_r \preceq u'$. Therefore, $d \preceq u'$. The same argument can be used to show that $d' \preceq u$.

Turtle-Unanimity: Suppose that there exists a chain w such that $w \preceq c$ for all inputs $\langle c \rangle$, and processor p produces output $\langle d, u \rangle$. Lemma 10.2 shows that w will be a prefix of the minimum chain d received by p in step 2b. □

Turtle-Termination and ***Turtle-Validity*** are similar to the proof for One-Step Tree Turtle (Section 7.1).

Appendix B: Threshold Quorum Systems

For completeness, we include how a k-intersecting quorum system can be implemented (based on [7]). A *threshold quorum system* constructs quorums from all subsets of $\mathcal{P}$ above a particular size. Let $n = |\mathcal{P}|$ be the number of processors. Let f be the maximum number of processors that may be faulty. We can construct a k-intersecting quorum system for $k \geq 1$ such that $n > k \cdot f$ as follows: the quorums are all subsets of $\mathcal{P}$ that have at least $n - f$ processors. Then Q^* is the quorum that contains the $n - f$ correct processors. In a k-intersecting quorum system implemented with a threshold quorum system, increasing the value for k results in an increase in the number of correct processors that are assumed to exist in the system.