The Building Blocks of Consensus*

Yee Jiun Song[1], Robbert van Renesse[1], Fred B. Schneider[1], and Danny Dolev[2]

[1] Cornell University
[2] The Hebrew University of Jerusalem

Abstract. Consensus is an important building block for building replicated systems, and many consensus protocols have been proposed. In this paper, we investigate the building blocks of consensus protocols and use these building blocks to assemble a skeleton that can be configured to produce, among others, three well-known consensus protocols: Paxos, Chandra-Toueg, and Ben-Or. Although each of these protocols specifies only one quorum system explicitly, all also employ a second quorum system. We use the skeleton to implement a replicated service, allowing us to compare the performance of these consensus protocols under various workloads and failure scenarios.

1 Introduction

Computers will fail, and for many systems it is imperative that such failures be tolerated. Replication, a general approach for supporting fault tolerance, requires a protocol so replicas will agree on values and actions. The *agreement* or *consensus* problem was originally proposed in [1]. Many variants and corresponding solutions have followed (see [2] for a survey of just the first decade, containing well over 100 references).

This paper focuses on protocols for Internet-like systems — systems in which there are no real-time bounds on execution or message latency. Such systems are often termed *asynchronous*. The well-known FLP impossibility [3] result proved that consensus cannot be solved even if only one process can fail. Practical consensus algorithms sidestep this limitation using one of two approaches: i) *leader-based* algorithms use a failure detector that captures eventual timing assumptions, and ii) *randomized* algorithms solve a non-deterministic version of consensus and eventually decide with probability 1.

Guerraoui and Raynal [4] point out significant similarities between different consensus protocols. They provide a generic framework for consensus algorithms and show that differences between the various algorithms can be factored out into a function called `Lambda`. Each consensus algorithm employs rather different implementations of `Lambda`. Later, Guerraoui and Raynal [5] show that leader-based algorithms can be factored into an `Omega` module and an `Alpha` module, where all differences are captured by differences in `Omega`.

This paper is a next step in unifying consensus algorithms. By breaking down consensus algorithms into building blocks, we show that different consensus algorithms can be instantiated from a single *skeletal algorithm*:

- Going beyond the work reported in [5], we present the building blocks of consensus algorithms and how they can be used to build a skeletal consensus algorithm. The skeletal algorithm provides insight into how consensus protocols work, and we show that consensus requires not one but two separate quorum systems.
- We show that both leader based and randomness based algorithms can be instantiated from our skeletal algorithm by configuring the two quorums systems that are used and the way instances are started. This approach can be used to instantiate three well-known consensus protocols: Paxos [6], Chandra-Toueg [7], and Ben-Or [8];

* This work is supported by AFOSR grants FA8750-06-2-0060, FA9550-06-1-0019, FA9550-06-1-0244, the National Science Foundation under grants 0424422, 0430161 and CCF-0424422 (TRUST), a gift from Microsoft Corporation, and ISF, ISOC, and CCR. Any opinions expressed in this publications are those of the authors and do not necessarily reflect the views of the funding agencies.

– The skeleton provides a natural platform for the implementation of multiple consensus protocols from a single code base. We implement our approach and present a performance comparison of these protocols under varying workload and crash failures. The implementation reveals interesting trade-offs between various design choices in consensus algorithms.

The rest of this paper is organized as follows. Section 2 describes the consensus problem and proposes terminology. Next, we present the building blocks of consensus protocols in Section 3; these building blocks are used to build a skeletal consensus algorithm in Section 4. Section 5 illustrates the instantiation of particular consensus algorithms using the skeletal algorithm. Section 6 describes the implementation of the skeleton and compares the performance of three well-known consensus protocols. Section 7 concludes.

2 The Consensus Problem

Computers that run programs – nodes – are either *honest*, executing programs faithfully, or *Byzantine* [9], exhibiting arbitrary behavior. We will also use the terms *correct* and *faulty*, but not as alternatives to honest and Byzantine. A correct node is an honest node that always eventually makes progress. A faulty node is a Byzantine node or an honest node that has crashed or will eventually crash. Note that honest and Byzantine are mutually exclusive, as are correct and faulty. However, a node can be both honest and faulty.

We assume that each pair of nodes is connected by a *link*, which is a bi-directional reliable virtual circuit and therefore messages sent on this link are delivered, eventually, and in the order in which they were sent (*i.e.*, an honest sender keeps retransmitting a message until it receives an acknowledgment or crashes). A receiver can tell who sent a message (*e.g.*, using MACs), so a Byzantine node cannot forge a message so it is indistinguishable from a message sent by an honest node.

Because our model is asynchronous, we do not assume timing bounds on execution of programs or on latency of communication. We also do not assume that a node on one side of a link can determine whether the node on the other side of the link is correct or faulty. Timeouts cannot reliably detect faulty nodes in an asynchronous system, even if only crash failures are allowed.

Nodes run *actors*, which are either *proposers* or *deciders*. A node may run both a proposer and a decider—in practice, the proposer often would like to learn the outcome of the agreement. In the consensus problem there is a set of proposers, each of which proposes a *proposal*, and a set of deciders, each of which *decides* one of the proposals. Assuming there exists at least one correct proposer, the goal of a consensus protocol is to ensure each non-faulty decider decides the same proposal, even in the face of faulty proposers.

Why is the consensus problem hard? Consider the following strawman protocol: each decider collects proposals from all proposers, determines the minimum proposal from among the proposals it receives (in case it received multiple proposals), and decides on that one. If no nodes were faulty, such a protocol would work, albeit limited in speed by the slowest node or link.

Unfortunately, even if only crash failures are possible, deciders do not know how long to wait for proposers. If deciders use time-outs, then each might time-out on different sets of proposers, so these deciders could decide different proposals. Thus, each decider has no choice but to wait until it has received a proposal from all proposers. But if one of the proposers is faulty, such a decider will wait forever and never decide.

In an asynchronous system with crash failures (Byzantine failures comprise crash failures), there exists no deterministic protocol in which some correct deciders eventually decide [3]. We might circumvent this limitation by allowing some correct deciders not to decide. Instead, we will embrace a slightly stronger requirement: that the consensus protocol never reach a state in which some correct decider can never decide. Since the strawman protocol of deciding the minimum proposal can reach a state in which deciders wait indefinitely for a faulty proposer, it is not a consensus protocol, even with respect to the relaxed requirement.

Formally, a protocol that solves the consensus problem must satisfy:

Definition 1. *Agreement If two honest deciders decide, then they decide the same proposal.*

Definition 2. *Unanimity If all honest proposers propose the same proposal v, then an honest decider that decides will decide v.*

Definition 3. *Validity If a honest decider decides v, then v was proposed by some proposer.*

Definition 4. *Non-Blocking For any run of the protocol that reaches a state in which a particular correct decider has not yet decided, there exists a continuation of the run in which that decider does decide on a proposal.*

Agreement is a safety property that captures what is informally meant by "consensus"; **Unanimity** and **Validity** are non-triviality properties; and **Non-Blocking** is a weaker version of the non-triviality requirement that all correct deciders eventually decide. **Non-Blocking** makes consensus solvable without trivializing the problem. Such a weakening of the problem is present in all algorithms that "solve" the consensus problem, since there cannot exist a solution to consensus with a strong liveness requirement [3].

3 Building Blocks

The strawman (viz, decide the minimum proposal) protocol presented in Section 2 is not a solution to the consensus problem because a faulty proposer can cause correct deciders to wait indefinitely, violating **Non-Blocking**. To remedy this, a consensus protocol might invoke multiple *instances*, where an instance is an execution of a subprotocol that itself might not decide. Such instances have also been called *rounds*, *phases*, or *ballots*. Ensuring consistency among decisions made by the multiple instances is central to the design of consensus protocols. In this section, we give building blocks in common to different consensus protocols; in the next section, we show how these building blocks can be combined to create full consensus protocols.

3.1 Instances

Instances may run in parallel with other instances. An instance decides a proposal if an honest decider in that instance decides a proposal. All honest deciders that decide in an instance must be guaranteed to decide the same proposal. An instance may not necessarily decide any proposals. With multiple instances, if one instance does not decide, then future ones may decide. If multiple instances decide, they must decide the same proposal.

Instances are identified by instance identifiers $r, ...$ from a totally ordered set $\bar{\mathbb{N}}$ (which can be, but does not have to be, the set $\mathbb{N}$ on naturals). Instance identifiers induce an ordering on instances, and we say that one instance is *before* or *after* another instance, even though instances may execute concurrently.

We name proposals $v, w, ...$ Within an instance, proposals are paired with instance identifiers. A pair (r, v) is called a *suggestion*, if v is a proposal and r an instance identifier. A special suggestion $\perp$ is used to indicate the absence of a specific proposal.

3.2 Actors

We employ two new types of actors in addition to proposers and deciders: *archivers* and *selectors*.[3] A proposer sends its proposal to the selectors. Selectors and archivers exchange messages and occasionally archivers inform deciders about potential values for decision. Deciders apply a filter to reach a decision.

Selectors *select* proposals, and archivers *archive* suggestions. Each archiver stores the last suggestion that it has archived. The initial archived suggestion of an archiver is $\perp$. *The objective of selectors is to reach a decision within an instance, while the objective of archivers is to maintain a collective memory that ensures decisions are remembered across instances and therefore conflicting decisions are avoided.*

At any point in time, a selector or archiver executes within a single instance; it sends and receives messages that are part of the instance execution. Archivers may progress to later instances, but must abort the current instance before starting in a new instance. Archivers always keep their state on non-volatile storage. Selectors, on the other hand, can lose their state on a crash and subsequently join any instance upon recovery, even a prior one.

[3] A node may run multiple actors, although each node can run at most one archiver and at most one selector.

3.3 Extended Quorum Systems

In order to ensure consistency in decisions, actors in a consensus protocol use *quorums*. An *extended quorum system* is a quadruple $(\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$. $\mathcal{P}$ is a set of nodes called the *participants*. $\mathcal{M}, \mathcal{Q}$, and $\mathcal{G}$ are each a collection of subsets of participants (that is, each is a subset of $2^{\mathcal{P}}$). $\mathcal{M}$ is the collection of *maximal-wait sets*, $\mathcal{Q}$ the collection of *quorum sets*, and $\mathcal{G}$ the collection of *guarded sets*. Each is defined below.

Crashed or Byzantine participants might never respond to requests. In an instance, an actor often tries to collect as many responses to a broadcast request as possible; it stops awaiting responses when it is in danger of waiting indefinitely. $\mathcal{M}$ characterizes this — it is a set of subsets of $\mathcal{P}$, none contained in another, such that some $M \in \mathcal{M}$ contains all the correct nodes.[4] An actor stops waiting for responses after receiving replies from all participants in M.

A *quorum set* is a subset of $\mathcal{P}$ such that the intersection of any two quorum sets must contain at least one correct node.

A subset of $\mathcal{P}$ is a *guarded set* if and only if it is guaranteed to contain at least one honest participant. Note, a guarded set may consist of a single participant that could be crashed but is not Byzantine.

An extended quorum system must satisfy the follow properties:

Definition 5. *Consistency The intersection of any two quorum sets (including a quorum set with itself) is guaranteed to contain a correct participant.*

	Crash	Byzantine
guarded set (in $\mathcal{G}$)	> 0	$> t$
quorum set (in $\mathcal{Q}$)	$> n/2$	$> (n+t)/2$
maximal-wait set (in $\mathcal{M}$)	$= n - t$	$= n - t$
set of participants ($\mathcal{P}$)	$> 2t$	$> 5t$

Table 1. Size requirements for Threshold Quorum Systems that satisfy consistency and opaqueness.

Definition 6. *Opaqueness Each maximal-wait set contains a quorum consisting entirely of honest participants.*

The simplest example of extended quorum system are threshold quorum systems; Table 1 summarizes requirements for $\mathcal{P}$, $\mathcal{M}$, $\mathcal{Q}$, and $\mathcal{G}$ in (n, t)-threshold systems. Other quorum systems may be more appropriate for particular applications. See [11] and [10] for advantages and disadvantages of various quorum systems for crash and arbitrary failure models respectively.

One degenerate extended quorum system, used in some well-known consensus protocols, is a *leader extended quorum system*: it involves one participant (the leader), and that participant by itself forms the only maximal-wait set in $\mathcal{M}$, quorum in $\mathcal{Q}$, and guarded set in $\mathcal{G}$. Because quorum sets have to satisfy consistency, the leader has to be honest.

3.4 Guarded Proposal

Selectors must be careful about selecting proposals that can conflict with prior decisions. Before selecting a proposal in an instance, a selector obtains a set L of suggestions from each participant in a maximal-wait set of archivers. A proposal v is considered a *potential-proposal* if L contains suggestions containing v from a guarded set and, therefore, at least one honest archiver sent a suggestion containing v. The selector identifies a *guarded proposal* of L, if any, as follows:

1. Consider each potential-proposal v separately:
   (a) Consider all subsets of suggestions containing v from guarded sets of archivers. The minimum instance identifier in a subset is called a *guarded-instance-identifier*;

[4] For those familiar with Byzantine Quorum Systems [10], $\mathcal{M}$ is the set of complements of the fail-prone system $\mathcal{B}$. For the purposes of this paper, it is often more convenient to talk about maximal-wait sets.

(b) The maximum among the guarded-instance-identifiers for v is called the *associated-instance-identifier* of v. (Note, because v is a potential-proposal, at least one guarded-instance-identifier exists and thus the maximum is well-defined.) The *support-sets* for v are those subsets of suggestions for which the guarded-instance-identifier is the associated-instance-identifier;

2. Among the potential-proposals, select all proposals with the maximal associated-instance-identifier. If there is exactly one such potential-proposal v', and $v' \neq \perp$, then this is the guarded proposal. Otherwise there is no guarded proposal.

If a decider obtains suggestions (r, v) from a quorum of archivers (and consequently decides), then any honest selectors in instances at least r are guaranteed to compute a guarded proposal v' such that $v' = v$ (unless they crash). If a selector fails to compute a guarded proposal in a particular instance, then this is both evidence that no prior instance can have decided and a guarantee that no prior instance will ever decide. However, the reverse is not true. If a selector computes a guarded proposal v', it is not guaranteed that v' is or will be decided.

4 Assembling The Pieces

The building blocks described in the previous section can be used to populate a *skeletal algorithm*, which in turn can be instantiated to obtain particular consensus algorithms. The skeletal algorithm specifies the interaction between the actors. It does not, however, define the quorums that are used, the mechanisms for invoking new instances, or other protocol-specific details. A consensus protocol must specify these details, and some options are described in Section 5.

4.1 The Skeletal Algorithm

The skeletal algorithm defines actions by actors in each instance. Figure 1 shows the behavior of each actor.

Each selector, archiver, and decider participates in an extended quorum system and exchanges messages of the form $\langle$*message-type*, *instance*, *source*, *suggestion*$\rangle$.

An extended quorum system $\mathcal{E} = (\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$ has the following interface:

- $\mathcal{E}$.*broadcast*(*m*): send message m to all participants in $\mathcal{P}$;
- $\mathcal{E}$.*wait*(*pattern*): wait for messages, matching the given pattern (specifies, for example, the message type and instance number). When the sources of the collected messages form an element or a superset of an element of $\mathcal{M}$, return the set of collected messages;
- $\mathcal{E}$.*uni-quorum*(*set of messages*): if the set of messages contains the same suggestion from a quorum, then return that suggestion.[5] Otherwise, return $\perp$;
- $\mathcal{E}$.*guarded-proposal*(*set of messages*): return the guarded proposal among these messages, or $\perp$ if there is none.

The skeletal algorithm uses two separate extended quorum systems. Archivers form one extended quorum system $\mathcal{A}$ that is the same for all instances; selectors use $\mathcal{A}$ to find the guarded proposal, preventing selection of proposals that conflict with decisions in earlier instances. Selectors form a second extended quorum system $\mathcal{S}^r$, which may be different for each instance r; archivers in instance r use quorums of $\mathcal{S}^r$ to prevent two archivers from archiving different suggestions in the same instance.

Deciders, although technically not part of an instance, try to obtain the same suggestion from a quorum of archivers in each instance. For simplicity of presentation, we associate deciders with instances and have them form a third extended quorum system, $\mathcal{D}$.

Returning to Figure 1, archivers start a new instance by sending their currently archived suggestion c_i to the selectors (A.1). Each selector awaits `select` messages from a maximal wait set (S.1) and determines if one of the suggestions it receives could have been decided in a previous instance (S.2). If so, it selects the corresponding proposal. If not, it selects one of the proposals issued by the proposers (S.3). The selector composes a suggestion from the selected proposal using the current instance identifier, and sends that suggestion to the archivers (S.4).

[5] Quorum consistency ensures at most one such suggestion.

If an archiver receives the same suggestion from a quorum of selectors (A.3), it (i) archives that suggestion (A.4), and (ii) broadcasts the suggestion to the deciders (A.5). If a decider receives the same suggestion from a quorum of archivers (D.2), the decider decides the corresponding proposal in those suggestions (D.3).

Each selector i maintains a set P_i containing proposals it has received (across instances). A selector waits for at least one proposal before participating in the rest of the protocol, so P_i is never empty during execution of the protocol. (Typically, P_i first contains a proposal from the proposer on the same node as selector i.) For simplicity, we assume an honest proposer sends a single proposal. The details of how P_i is formed and used differ across consensus protocols, so this is discussed below when full protocols are presented. P_i has an operation $P_i.pick(r)$ that returns either a single proposal from the set or some value as a function of r. Different protocols use different approaches for selecting that value, and these too are discussed below. Note, selectors may lose their state, starting again with an empty P_i.

Archivers' states survive crashes and recoveries. So, an archiver j running on an honest node maintains: r_j, the current instance identifier and c_j, the last archived suggestion, which is initialized with the value $\perp$.

Note that steps (A.1), (S.1), and (S.2) can be skipped in the lowest numbered instance, because c_i is guaranteed to be $\perp$ for all archivers. This is an important optimization in practice and eliminates one of the three message rounds necessary for a proposal to be decided in the normal (failure-free) case.

At the **start of instance** r, each **archiver** i executes:

(A.1) send c_i to all participants (selectors) in $\mathcal{S}^r$:
$\mathcal{S}^r.broadcast(\langle \texttt{select}, r, i, c_i \rangle)$

Each **selector** j in $\mathcal{S}^r$ executes:

(S.1) wait for `select` messages from archivers:
$L_j^r := \mathcal{A}.wait(\langle \texttt{select}, r, *, * \rangle)$;
(S.2) see if there is a guarded proposal:
$v_j^r := \mathcal{A}.guarded\text{-}proposal(L_j^r)$;
(S.3) if not, select from received proposals instead:
if $v_j^r = \perp$ **then** $v_j^r := P_j.pick(r)$ **fi**;
(S.4) send a suggestion to all archivers:
$\mathcal{A}.broadcast(\langle \texttt{archive}, r, j, (r, v_j^r) \rangle)$;

Each **archiver** i (still in instance r) executes:

(A.2) wait for `archive` messages from selectors:
$M_i^r := \mathcal{S}^r.wait(\langle \texttt{archive}, r, *, * \rangle)$;
(A.3) unanimous suggestion from a quorum?
$q_i^r := \mathcal{S}^r.uni\text{-}quorum(M_i^r)$;
(A.4) *archive* the suggestion:
$c_i :=$ **if** $q_i^r = \perp$ **then** $(r, \perp)$ **else** q_i^r **fi**;
(A.5) send the suggestion to all deciders:
$\mathcal{D}.broadcast(\langle \texttt{decide}, r, i, c_i \rangle)$

Each **decider** k executes:

(D.1) wait for `decide` messages from archivers:
$N_k^r := \mathcal{A}.wait(\langle \texttt{decide}, r, *, * \rangle)$;
(D.2) unanimous suggestion from a quorum?
$d_k^r := \mathcal{A}.uni\text{-}quorum(N_k^r)$;
(D.3) if there is, and not $\perp$, decide:
if ($d_k^r = (r, v')$ **and** $v' \neq \perp$) **then**
decide v' **fi**;

Fig. 1. The skeletal algorithm of consensus protocols.

4.2 Agreement

We now show that the skeletal algorithm satisfies **Agreement**, that is, if two honest deciders decide, then they decide the same proposal. We omit the proofs of lemmas that are relatively straightforward. For complete proofs please refer to [12].

Lemma 1. *In the skeletal algorithm of Figure 1:*

(a) if any honest archiver i computes a suggestion $q_i^r \neq \perp$ in Step (A.3) of instance r, then any honest archiver that computes a non-$\perp$ suggestion in that step of that instance, computes the same suggestion.
(b) if any honest decider k computes a suggestion $d_k^r \neq \perp$ in Step (D.2) of instance r, then any honest decider that computes a non-$\perp$ suggestion in that step of that instance, computes the same suggestion.

Note that Step (S.2) does *not* satisfy (a) and (b) of Lemma 1. because selectors do not try to obtain a unanimous suggestion from a quorum.

Corollary 1. *In the skeletal algorithm of Figure 1, if any honest archiver archives a suggestion (r, v) with $v \neq \perp$ in Step (A.4) of instance r, then any honest archiver that archives a suggestion with non-$\perp$ proposal in that step of that instance, archives the same suggestion.*

Lemma 2. *In the skeletal algorithm of Figure 1, if any honest archiver sends a suggestion $(\bar{r}, v)$ with $v \neq \perp$ in Step (A.1) of instance r then any honest archiver that sends a suggestion $(\bar{r}, v')$ with $v' \neq \perp$ in that step of that instance, sends the same proposal, i.e., $v = v'$.*

Lemma 3. *In the skeletal algorithm of Figure 1:*

(a) if each honest selector that completes Step (S.4) of instance r sends the same suggestion, then any honest archiver that completes Step (A.3) of that instance computes the same suggestion;
(b) if each honest archiver that completes Step (A.4) of instance r sends the same suggestion, then any honest decider that completes Step (D.2) of that instance computes the same suggestion;
(c) if each honest archiver that completes Step (A.1) of instance r sends the same suggestion, then any honest selector that completes Step (S.2) of that instance computes the same proposal.

The most important property we need to prove is:

Lemma 4. *In the skeletal algorithm of Figure 1, if r' is the earliest instance in which a proposal w is decided by some honest decider, then for any instance r, $r > r'$, if an honest archiver archives a suggestion in Step (A.4), then it is (r, w).*

Proof. Since instances are totally ordered, any subset of them are totally ordered. The proof will be by induction on all instances, past instance r', in which eventually some honest archiver archives a suggestion.

Let $w \neq \perp$ be the proposal decided by an honest decider in Step (D.4) of instance r'. Let $Q^{r'} \in \mathcal{A}$ be the quorum in instance r' whose suggestions caused the decider to decide w.

Let $r_1 > r'$ be the first instance past r' at which some honest archiver eventually completes Step (A.4). Since this archiver completes Step (A.4), it must have received `archive` messages from a maximal-wait set of selectors following Step (A.2) of instance r_1. Each honest selector that sent such a message received `select` messages from a maximal-wait set of archivers that were sent in their Step (A.1) of instance r_1. Each honest archiver that completes Step (A.1) did not archive any new suggestion in any instance r'' where $r' < r'' < r_1$ holds, because r_1 is the first such instance. Moreover, the archiver will not archive such a suggestion in the future, since all such instances r'' aborted before sending `select` messages in Step (A.1) of instance r_1.

In Step (A.1), an archiver sends the last suggestion it archived. Some archivers may send suggestions they archived prior to instance r' while other archivers send suggestions they archived in Step (A.5) of instance r'.

Each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j contains suggestions from a quorum Q^{r_1} consisting entirely of honest archivers (by the opaqueness property of $\mathcal{A}$). By the consistency property of $\mathcal{A}$, the intersection of Q^{r_1} and $Q^{r'}$ contains a guarded set, and thus Q^{r_1} contains suggestions from a guarded set of honest archivers that archived (r', w). There cannot be such a set of suggestions for a later instance, prior to r_1. By Corollary 1 and Lemma 2, there cannot be any suggestions from a guarded set for a different proposal in instance r'. Thus, each honest selector will select a non-$\perp$ proposal and those proposals are identical.

By Lemma 3, every honest archiver that completes Step (R.4) will archive the same suggestion. Thus the proof holds for r_1.

Now assume that the claim holds for all instances r'' where $r' < r'' < r$ holds; we will prove the claim for instance r.

There is an honest archiver that completes Step (A.4) in instance r and archives (r, w). Following Step (A.2) of instance r, it must have received `archive` messages from a maximal-wait set of selectors. Each honest selector that sent such a message received `select` messages from a maximal-wait set of archivers in Step (A.1) of instance r.

Each honest archiver sends the last suggestion it archived. Some honest archivers might send suggestions they archived prior to instance r', while other honest archivers send suggestions archived in Step (A.5) of instance r'', where $r' \leq r'' < r$ holds. By the induction hypothesis, all honest archivers that send a suggestion archived by an instance ordered after instance r' use proposal w in their suggestions.

In instance r, each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j has to contain suggestions from a quorum Q^r consisting entirely of honest archivers (by the opaqueness property of $\mathcal{A}$). By the consistency property of $\mathcal{A}$, the intersection of Q^r and $Q^{r'}$ contains a guarded set, so Q^r has to contain suggestions from a guarded set of honest archivers that archived (r', w) in instance r' and that might have archived (r'', w) in some later instance. Therefore, selector j obtains w as a possible potential-proposal. Since all honest archivers that archive a suggestion past instance r' archive the same proposal, there is a support-set for w with associated-instance-identifier $\bar{r} \geq r'$.

There cannot be any other possible potential-proposal with an associated-instance-identifier ordered larger than r' since, by induction, no honest archiver archives a suggestion with a different proposal later than r'. Therefore, each honest selector selects proposal w. By Lemma 3, every honest archiver that completes Step (A.4) archivew the same suggestion. Thus, the proof holds for r.

Theorem 1 (Agreement). *If two honest deciders decide, then they decide the same proposal.*

Proof. If the deciders decide in the same instance, the result follows from Lemma 1. Say one decider decides v' in instance r', and another decider decides v in instance r, with $r' < r$. By Lemma 4, all honest archivers that archive in instance r archive (r, v'). By the consistency property of $\mathcal{A}$, an honest decider can only decide (r, v') in instance r, so $v = v'$.

5 Full Protocols

The skeletal algorithm described above does not specify how instances are created, how broadcasts are done in steps (A.1), (S.4), and (A.5), what specific extended quorum systems to use for $\mathcal{A}$ and $\mathcal{S}^r$, how a selector j obtains proposals for P_j, or how j selects a proposal from P_j. We now show how Paxos [6], the algorithm by Chandra and Toueg [7], and the early protocol by Michael Ben-Or [8] resolve these questions.

5.1 Paxos

Paxos [6] was originally designed only for honest systems. In Paxos, any node can create an instance r at any time, and that node becomes the *leader* of the instance. The leader creates a unique instance-identifier r from its node identifier along with a sequence number per node that is incremented for each new instance created on that node. The leader runs both a proposer and a selector. $\mathcal{S}^r$ is a leader extended quorum system consisting only of that selector.

The leader starts the instance by broadcasting a `prepare` message containing the instance identifier to all archivers. Upon receipt, an archiver i checks that $r > r_i$, and, if so, sets r_i to r and proceeds with Step (A.1). Since there is only one participant in $\mathcal{S}_r$, the broadcast in (A.1) is actually a point-to-point message back to the leader, now acting as selector. In Step (S.3), if the leader has to pick a proposal from P_j, it selects the proposal by the local proposer. Thus, there is no need for proposers to send their proposals to all selectors.

Unanimity and **Validity** follow directly from the absence of Byzantine participants. To see why Paxos is **Non-Blocking**, consider a state in which some correct decider has not yet decided. Now consider the following continuation of the run: one of the correct nodes creates a new instance with an instance identifier higher than used before. Because there are always correct nodes and there is an infinite number of instance identifiers, this is always possible. The node sends a `prepare` message to all archivers. All honest archivers start in Step (A.1) of the instance on receipt, so the selector at the leader will receive enough `select` messages in Step (S.1) to continue. Due to Lemma 3 and there being only one selector in $\mathcal{S}^r$, all honest archivers archive the same suggestion in Step (A.4). The deciders will each receive a unanimous suggestion from a quorum of archivers in Step (D.1) and decide in Step (D.3).

5.2 Chandra-Toueg

The Chandra-Toueg algorithm [7] is another consensus protocol designed for honest systems. It requires a coordinator in each instance; the role of the coordinator is similar to the leader in Paxos. Unlike Paxos, Chandra-Toueg instances are consecutively numbered $0, 1, \ldots$. The coordinator of each instance is determined by the instance number modulo the number of nodes in the system, so the role of the coordinator shifts from

node to node at the end of each instance. Each node in the system is both a proposer and a archiver. For each instance r, selector quorum $\mathcal{S}^r$ is the extended quorum consisting only of the coordinator of that instance.

To start the protocol, a proposer sends a message containing a proposal to all nodes. Upon receiving the first proposal, an archiver starts in instance 0 and executes (A.1). The coordinator of each instance starts (S.1) upon receiving a `select` message for that instance. In (S.3), $P_i.pick(r)$ returns the first proposal received by the coordinator. Archivers that successfully complete (A.2-5) move to the next instance. Archivers must be prepared to time-out while awaiting an `archive` message from the selector of a particular instance, because the selector can fail. When this happens, archivers proceed to (A.1) in the next instance.

When an archiver receives an `archive` message with a larger instance number than it has thus far received, it aborts the current instance and skips forward to the instance of the `archive` message.

In the original description of the Chandra-Toueg algorithm, the coordinator for an instance is the only decider for that instance. This necessitates an additional round of communication, where the coordinator broadcasts a decision so that all nodes become aware of the decision. The Chandra-Toueg algorithm can be changed so that all nodes are deciders in all instances without affecting the rest of the protocol. This eliminates one round of communication while increasing the number of messages sent in (A.5) of the skeletal algorithm. This is similar to the algorithm proposed in [13]. A comparison of the original Chandra-Toueg algorithm and this modified version is given in [14].

As in the case of Paxos, **Unanimity** and **Validity** follow directly from the absence of Byzantine participants. **Non-blocking** follows from that fact that a honest, correct selector can always receive sufficient `select` messages in (S.1) to continue. All honest archivers will always receive the same suggestion in (A.3), since there is only one selector in each instance. If the coordinator for an instance fails, then archivers for that instance will time-out and move to the next instance.

5.3 Ben-Or

In this early protocol [8], each node runs a proposer, a selector, an archiver, and a decider. Instances are numbered with consecutive integers. Proposals are either "0" or "1" (that is, this is a binary consensus protocol), and each $P_i = \{0, 1\}$. $P_i.pick(r)$ selects the local proposer's proposal for the first instance, or a random one in later instances.

Each of the selectors, archivers, and deciders starts in instance 1 and loops. The loop at selector j consists of steps (S.1) through (S.4), with r_j incremented right after Step (S.4). The loop at archiver i consists of steps (A.1-5), with r_i incremented after Step (A.4). The broadcasts in steps (A.1) and (A.5) are to the same destination nodes and happen in consecutive steps, so they can be merged into a single broadcast, resulting in just two broadcasts per instance. Finally, the loop at decider k consists of steps (D.1) through (D.3), with r_k incremented after Step (D.3).

$\mathcal{S}^r$ is the same extended quorum system as $\mathcal{A}$ for every instance r; both consist of all nodes and use a threshold quorum system. Ben-Or works equally well in honest and Byzantine environments as long as opaqueness is satisfied. It is easily shown that if a decider decides, then all other deciders decide either in the same or the next instance.

Unanimity follows from the rule that selectors select the locally proposed proposal in the first instance: if all selectors select the same proposal v, then by Lemma 3 the archivers archive v, and, by opaqueness of $\mathcal{A}$, the deciders decide v. **Validity** is ensured by the rule that selectors pick the local proposal in the first instance and random proposals in subsequent instances. Selectors have to pick random proposals in an instance iff there was not a unanimous suggestion computed in (A.3) of the previous instance. This can only happen if both of the binary proposals have been proposed by some proposer. **Non-Blocking** follows from the rule that honest selectors pick their proposals at random in all but the first instance, so it is always possible that they pick the same proposal, after which a decision in Step (D.3) is guaranteed because of opaqueness for $\mathcal{A}$.

6 Implementation and Evaluation

The descriptions of the Paxos, Chandra-Toueg, and Ben-Or protocols above show that these protocols share common building blocks. Having observed their similarities, we now investigate how their differences affect

their performance. To do this, we implemented the skeletal algorithm, and built each of the three protocols on top of it. In this section, we present that implementation and the performance of the three instantiations.

6.1 Implementation

We built a replicated state machine to provide a simple logging service for remote clients. The service consists of a set of servers that run a consensus protocol. Clients submit values to any server; that server then attempts to have that value decided in an *epoch*. To decide a value, a server submits that value as a *proposal* associated with the current epoch. When a value is decided in an *epoch*, the client that submitted the value is informed of the epoch number in which the value was decided, and all servers move to the next epoch. Each server maintains an internal queue of values it has received from clients, and each attempts to get them decided in FIFO fashion.

Paxos requires a leader election mechanism th at was not described in the original protocol [6]. We explored two different leader election mechanisms. First, we built a version of Paxos where each node that wants to propose a value simply makes itself the leader. By having each node pick instance numbers for instances where it is the leader from a disjoint set of instance numbers, we ensure that each instance can only have one unique leader. We call this version of Paxos *GreedyPaxos*.

We also built a variant of Paxos that uses a token-passing mechanism to determine the leader. We call this version of Paxos *TokenPaxos*. The current leader holds a token that is passed to other nodes when the leader no longer has any local requests to commit. Token request and token passing messages are piggy-backed on `select` and `archive` messages. Further details of this protocol are outside the scope of this paper.

For the implementation of Chandra-Toueg, we modified the original algorithm to have all nodes be deciders in all instances. As described in Section 5.2, this avoids requiring deciders to broadcast a decision when a value is decided, thus improving the performance of our particular application where all servers need to learn about decisions.

All of our implementations use a simple threshold quorum system for the archiver and decider quorums, as well as for Ben-Or's selector quorums.

6.2 Experimental Setup

We evaluate the protocols using simulation. In our experiments, the logging service consists of a set of 10 servers. The workload is sent to the servers by a set of 10 clients. Each client sends requests to the servers according to a Poisson distribution with a mean λ_c requests per minute. Each client chooses a server at random and sends its requests to that server. All client to server and server to server messages have a latency that is given by a lognormal distribution with mean 100 ms and standard deviation 20 ms. For each set of experiments, we measure the elapsed time between when a server first receives a value from a client until the time that the server learns the value has been decided.

6.3 Results

In the first set of experiments, we ran the server against clients with varying loads until 100 values were decided by the logging service. We vary request rate λ_c from each client from 0.5 requests per minute to 14 requests per minute. We report the mean and median values of 100 decisions averaged over 8 runs of each experiment.

Figure 2 and Figure 3 show the mean and the median latency for a single value to be decided. The graphs show that as load increases, the time it takes for a value to be decided increases gradually. At low loads, the performance of all four algorithms is quite close. This is because in the ideal case, all four algorithms take four rounds of communication for a value to be decided.

As load increases, performance degrades, because of contention between servers trying to commit different values. GreedyPaxos consistently outperforms TokenPaxos, particularly under heavy load. This is because GreedyPaxos does not need to wait (i.e., for the token) before proposing a value. Under heavy load, each GreedyPaxos node sends a `prepare` message in the beginning of each epoch without having to wait. The node with the largest instance number wins and gets its value decided. TokenPaxos, on the other hand, will always decide values of the node with the token before passing the token to the next node with requests.

This has 2 implications: i) if the leader keeps getting new requests, other nodes can starve, and ii) one round of communication overhead is incurred for passing the token.

Figure 4 shows the number of messages that each protocol uses to commit 100 values under different request rates. Ben-Or is seen to incur a much larger overhead than the other protocols. This is because Ben-Or uses a selector quorum that consists of all nodes rather than just a leader/coordinator, so (A.1) and (S.4) of the skeletal algorithm send n^2 messages in each instance, rather than just n messages in Paxos and Chandra-Toueg.

Also observe in Figure 4 that compared to TokenPaxos, GreedyPaxos sends more messages as load increases. Under heavy load, each GreedyPaxos node will broadcast a `prepare` message to all other nodes in the beginning of every round. This results in n^2 messages being sent rather than the n `prepare` messages that are sent in the case of TokenPaxos.

In order to investigate the performance of each protocol under crash failures, we simulated these failures. We modeled failure event occurences as a Poisson distributed rate λ_f failures per minute. When a failure event occurs, we fail a random server until the end of the epoch. To ensure that the system is able to make progress, we limit the number of failures in an epoch to be less than half the number of servers in the system. Keeping the request rate from clients steady at 7 requests per minute per client, we vary the failure rate from 0.5 failures per minute to 12 failures per minute.

Figure 5 and Figure 6 show mean and median decision latency, resp., for the four protocols under varying failure rates. Note that GreedyPaxos and Ben-Or are not affected significantly by server failures. Chandra-Toueg and TokenPaxos, on the other hand, see significant performance degradation as the failure rate increases. This is because Chandra-Toueg and TokenPaxos both depend on time-out to recover from failures of particular nodes. In the case of Chandra-Toueg, failure of the coordinator requires that all archivers time-out and move to the next instance; in the case of TokenPaxos, if the node that is holding the token crashes, then a time-out is required to generate a new token.

A comparison study presented by Hayabashibara et al. [15] found that Paxos outperforms Chandra-Toueg under crash failures. We find that this result depends on the leader election protocol used by Paxos. In our experiments, GreedyPaxos outperforms Chandra-Toueg, but TokenPaxos performs worse under certain failure scenarios.

Figure 7 shows the message overhead of each protocol under varying failure rates, clearly showing that the number of messages sent is not affected by failures.

7 Conclusion

We investigated several well-known consensus protocols and showed that they share the same basic building blocks. We demonstrated that these building blocks can be used to describe a single skeletal algorithm that can be instantiated to obtain Paxos, Chandra-Toueg, and Ben-Or consensus protocols simply by configuring the quorum systems that are used, the way instances are started, and other protocol-specific details. We implemented the skeletal algorithm and used it to instantiate Ben-Or, Chandra-Toueg, and two variants of the Paxos algorithm. Simulation experiments using those implementations allowed the performance differences between these algorithms to be measured for different workloads and crash failures. This approach thus provides a basis for understanding consensus protocols and comparing their performance. The skeletal algorithm also provides a novel platform for exploring other possible consensus protocols.

References

1. M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.
2. M. Barborak and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, 1993.
3. M. Fischer, N. Lynch, and M. Patterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.
4. R. Guerraoui and M. Raynal, "The information structure of indulgent consensus," in *Proc. of 23rd IEEE International Conference on Distributed Computing Systems*, 2003.
5. R. Guerraoui and M. Raynel, "The alpha of indulgent consensus," *The Computer Journal*, 2006.
6. L. Lamport, "The part-time parliament," *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

7. T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.
8. M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*. Montreal, Quebec: ACM SIGOPS-SIGACT, Aug. 1983, pp. 27–30.
9. L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *Trans. on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.
10. D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, pp. 203–213, Jun. 1998.
11. M. Naor and A. Wool, "The load, capacity, and availabiliity of quorum systems," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 423–447, Apr. 1998.
12. Y. J. Song, R. van Renesse, F. B. Schneider, and D. Dolev, "Evolution vs. intelligent design in consensus protocols," Cornell University, Tech. Rep. CUL.CIS/TR2007-2082, May 2007.
13. A. Mostéfaoui and M. Raynal, "Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach," in *Proc. of the International Symposium on Distributed Computing*, 1999, pp. 49–63.
14. P. Urbán and A. Schiper, "Comparing distributed consensus algorithms," in *Proc. of International Conference on Applied Simulation and Modelling*, 2004, pp. 474–480.
15. N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama, "Performance comparison between the Paxos and Chandra-Toueg consensus algorithms," in *Proc. of International Arab Conference on Information Technology*, Doha, Qatar, Dec. 2002, pp. 526–533.

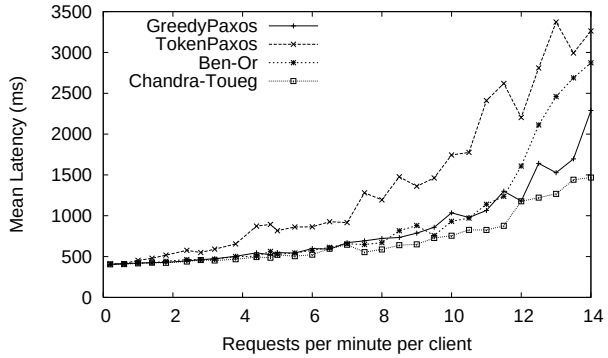

Fig. 2. Mean time to decide under varying request rates

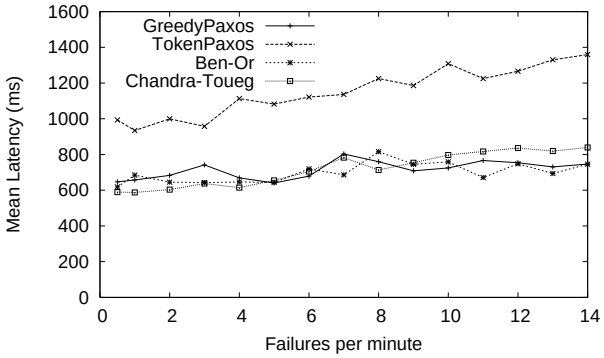

Fig. 5. Mean time to decide under varying failure rates

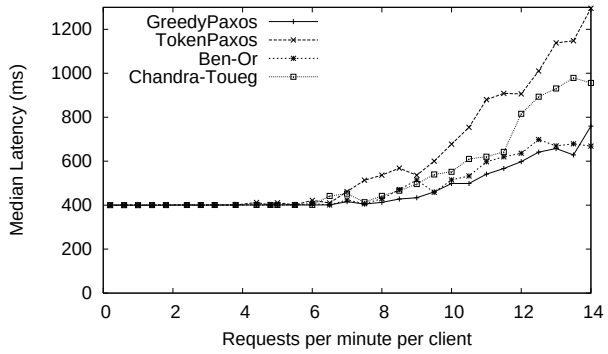

Fig. 3. Median time to decide under varying request rates

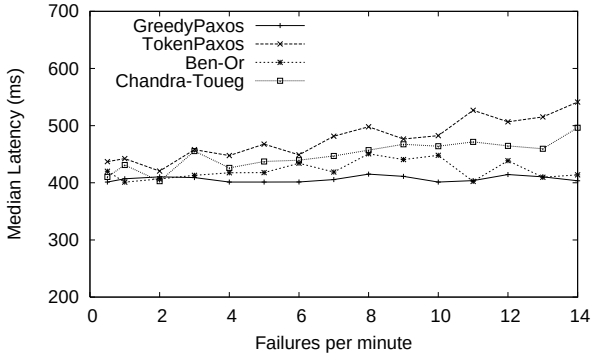

Fig. 6. Median time to decide under varying failure rates

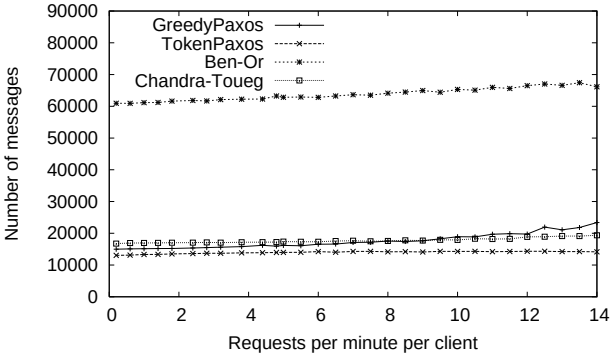

Fig. 4. Communication overhead under varying request rates

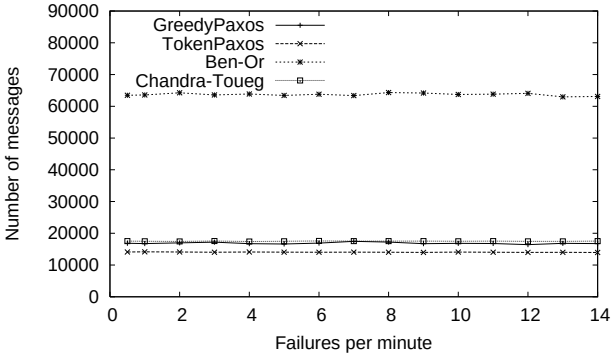

Fig. 7. Communication overhead under varying failure rates