BAR Fault Tolerance for Cooperative Services
Extended Technical Report TR-05-10

Jean-Philippe Martin, Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, , and Carl Porth

April 5, 2005

Abstract

This paper describes a general approach to constructing cooperative services that span multiple administrative domains. In such environments, protocols must tolerate both *rational behaviors* when nodes arbitrarily deviate from the protocol for their local benefit and *Byzantine behaviors* when a broken, misconfigured, or malicious node arbitrarily deviates from the protocol for any other reason. The paper examines this problem in the context of a cooperative backup system and makes three contributions. First, it introduces the BAR (Byzantine, Altruistic, Rational) model, which provides the foundation for reasoning about the properties of this class of services. Second, it presents a general three-tier architecture aimed at reducing the complexity of building services developed in the BAR model. Our realization of this architecture includes an asynchronous replicated state machine that provides the normal safety and liveness guarantees as long as at most than $\frac{n-2}{3}$ nodes are Byzantine; the rest of the nodes can be rational. The paper's third contribution is to describe an implementation of PIB, the first cooperative backup service to tolerate both Byzantine users and an unbounded number of rational users. We show that, under the BAR model, PIB provides provable safety and liveness guarantees. We also show that our approach is practical: our prototype of a BART state machine executes 20 requests per second and our PIB prototype can back up a gigabyte of data in 20 minutes.

1 Introduction

This paper describes a general approach to constructing cooperative services that span multiple administrative domains (MADs). In a cooperative service, nodes collaborate to provide some service that benefits each node, but there is no central authority that controls the actions of all nodes. Examples of such services include Internet routing [25, 61], wireless mesh routing [37], file distribution [18], archival storage [41], or cooperative backup [9, 20, 35]. As MAD distributed systems become more commonplace, developing a solid foundation for constructing this class of services becomes increasingly important.

There currently exists no satisfactory way to model MAD services. In these systems, the classical dichotomy between correct and faulty nodes becomes inadequate. Nodes in MAD systems may depart from protocols for two distinct reasons. First, as in traditional systems, nodes may either break—through component failure, misconfiguration, or corruption—or be genuinely malicious. Second, nodes may be selfish and alter the protocol in order to increase their utility [3, 31]. Byzantine Fault Tolerance (BFT) [15, 34, 39] handles the first class of deviations well. However, the Byzantine model classifies all deviations as faults and requires a bound on the number of faults in the system; these bounds are not tenable in MAD systems where all nodes may exhibit selfish behavior. Models that only account for selfish behavior [61] handle the second class of deviations, but the presence of a single node whose behavior deviates from the expected model may cause arbitrary disruptions.

Given the potential for nodes to develop arbitrarily subtle tactics, it is not sufficient to verify experimentally that a protocol tolerates a collection of attacks identified by the protocol's creator. Instead, just as for authentication systems [13] or Byzantine-tolerant protocols [34], it is necessary to design systems that *provably* meet their goals, no matter what strategies nodes may concoct within the scope of the adversary model.

To allow construction of such protocols, we define a system model that captures the essential aspects of MADs. The Byzantine-Altruistic-Rational (BAR) model accommodates three classes of nodes. *Rational* [61] nodes participate in the system to gain some net benefit and can depart from a proposed program in order to increase their net benefit. *Byzantine* [15, 34, 39] nodes can depart arbitrarily from a proposed program whether it benefits them or not. Finally, BAR accommodates the presence of *altruistic* [48] nodes that execute a proposed program even if the rational choice is to deviate. A protocol is BAR Tolerant (BART) if it provably provides to its non-Byzantine participants a set of desired safety and liveness properties. In this paper, we focus on BART protocols that do not depend on the existence of altruistic nodes in the system: we assume that at most $\frac{n-2}{3}$ of the nodes in the system are Byzantine and that every non-Byzantine node is rational.

A key question is whether useful systems can be built under the BAR model. To answer this question, we develop a general three-tier architecture for building BART services.

The bottom layer implements a small set of key BART abstractions (e.g., state machine replication and terminating reliable broadcast) that simplify implementing and reasoning about BART distributed services. The middle layer partitions and assigns work to either single nodes or state machines. Finally, the top layer implements the application-specific aspects of BART services (e.g., verifying that responses to requests conform to application semantics.)

We use this architecture to construct PIB, a BART cooperative backup service. PIB is targeted at environments–such as a group of students in a dorm, home machines for researchers in a group, or machines donated to non-profit organizations [36]—that, by supporting a notion of an identity that is “expensive” to obtain, avoid the Sybil attack [22]. We do not target open membership peer-to-peer systems.

We find that our architecture significantly simplifies and improves the design of PIB. Compared to previous peer to peer backup architectures [19, 20, 35], PIB has several advantages: it is unique in tolerating both rational and Byzantine peers; it provides deterministic retrieval guarantees; and it does not require peers to exchange storage symmetrically. Perhaps most importantly, we find that using a layered architecture significantly simplifies proving concrete safety and liveness properties.

We also show that our approach is practical: our prototype of a BART state machine executes 20 requests per second and our PIB prototype can back up a gigabyte of data to 21 nodes in 20 minutes, so that the data can be recovered despite the failure of 7 nodes.

In this paper we make three main contributions. First, we formalize a model for reasoning about systems in the presence of both Byzantine and rational behavior. Second, we introduce a general architecture and identify a set of design principles which, together, make it possible to build and reason about BART systems. Third, we describe the implementation of PIB, a cooperative backup system that provides provable safety and liveness properties within the BAR model. A key component of our system is a BART protocol for state machine replication that relies on synchrony assumptions only for liveness.

The rest of this paper is organized as follows. In Sections 2 and 3 we formally present the BAR model and our system model. In Section 4, we describe our overall 3-level architecture, and the next three sections present our implementation of each of the layers: our asynchronous BART Paxos protocol, our techniques for work assignment, and our PIB application. Section 8 evaluates the prototype and Section 9 discusses related work.

2 BAR Model

To model a MAD environment we must account for three important factors: (a) no node is guaranteed to follow the suggested protocol, (b) the actions of most nodes are guided by self interest [3, 31], and (c) some nodes may be fundamentally broken [34, 15, 39].

The Byzantine Altruistic Rational (BAR) model addresses these requirements by classifying nodes into three categories.

Altruistic nodes follow the suggested protocol exactly. Altruistic nodes may reflect the existence of Good Samaritans and “seed nodes” in real systems. Intuitively, altruistic nodes correspond to *correct nodes* in the fault-tolerance literature.

Rational nodes are self interested and seek to maximize their benefit according to a specified utility function. Rational nodes will deviate from the suggested protocol if and only if doing so increases their net utility from participating in the system. The utility function must account for the relevant costs (e.g., computation cycles, storage, network bandwidth, overhead associated with sending and receiving messages, power consumption, or threat of financial sanctions [35]) and benefits (e.g., access to remote storage [41, 9, 20, 35], network capacity [37], or computational cycles [59]) to a node for participating in a system.

Byzantine nodes may deviate arbitrarily from the suggested protocol for any reason. In some cases a node deviates because it is broken (e.g., misconfigured, compromised, malfunctioning, or misprogrammed). In other cases, the node is functioning properly from the point of view of an owner, but the owner’s utility function significantly differs from the utility function specified for rational nodes. Such a utility function may simply model costs in an unexpected manner or it may associate great value to inflicting harm on the system or its users for personal satisfaction or commercial considerations [50].

Useful protocols specify guarantees to their participants. Under BAR, the goal is to provide safety guarantees similar to those from Byzantine fault tolerance to “all rational and altruistic nodes” (as opposed to “all correct nodes”). We identify two classes of protocols that meet these goals.

- Incentive-Compatible Byzantine Fault Tolerant (IC-BFT) protocols. A protocol is IC-BFT if it guarantees the appropriate set of safety and liveness properties and it is in the best interest of all rational nodes to follow the protocol exactly.
- Byzantine Altruistic Rational Tolerant (BART) protocols: A protocol is BART if it guarantees the appropriate set of safety and liveness properties in the presence of rational deviations from the protocol.

An IC-BFT protocol thus must define the optimal strategy for a rational node. In a BART protocol a rational node may gain by deviating without violating the global safety guarantees.

3 System Model

Although we seek to develop a general framework for constructing a range of cooperative services, our approach is guided by a specific problem in a specific set of environments. In particular, we are building a cooperative backup system for three user communities: 30 co-workers who cooperatively back up their personal home machines, 500 students in a dormitory who cooperatively back up their personal machines, and 50 nonprofit organizations that receive free or low-cost refurbished PCs [36].

We assume that a trusted authority controls which nodes may enter the system, that each such member has a unique identity corresponding to a cryptographic public key, and that each member can determine whether a public key belongs to a specific member. These are reasonable assumptions for our target environments: a volunteer distributes a list of keys to coworkers, a university's electronic ID system maps identities to dormitory residents, and the refurbisher installs the relevant information in the non-profit scenario. The strong and limited identity assumption facilitates three important factors. First, it allows for a reasonable bound on the number of Byzantine nodes [22]. Second, it provides rational nodes with an incentive to consider the long-term consequences of their actions. Third, it allows us to tie identities to real world entities. This last point allows us to apply both internal sanctions (e.g. denial of service, data deletion) and external sanctions (e.g. monetary fines, suspension, social shunning) to nodes which misbehave. Support for external sanctions increases the flexibility of our protocols, but is not required for deployment.

We have different timing assumptions for PIB and for the underlying BART state machine replication. PIB is a synchronous protocol and relies on synchrony to guarantee both its liveness and safety properties—e.g. data trusted to PIB is guaranteed to be retrievable only until the lease associated with it expires.

The underlying BART state machine replication protocol instead relies on synchrony only for liveness. The protocol ensures safety despite message omission, message reordering, and message alterations that do not subvert the cryptographic assumptions associated with public key signatures [54] and secure hashing [49]. We guarantee liveness during periods of synchrony, as long as rational nodes consider the benefit of participating in the protocol to outweigh the costs. In periods of synchrony there is a known bound Δ on message delivery time.

In a system where costs may outweigh benefits, to ensure that rational nodes continue to participate in the protocol it is necessary to bound the cost that nodes will pay. This further requirement translates for us into a stronger liveness guarantee—we assume that if nodes a and b are non-Byzantine and a sends b a request at time t, b's response will reach a by time $t + max\_response$. This strengthening allows us to bound the state maintained by non-Byzantine in order to answer late requests; further, it allows us to improve the availability of our state machine by reducing the size of the quorum of responsive nodes required by our protocol (from $n-f-1$ to $\lceil (n+f+1)/2 \rceil$, where f is the number of Byzantine nodes). In order to complete our model, we must also make specific assumptions on the rational and Byzantine nodes in the system.

Rational nodes We make three technical assumptions about rational nodes. First, we assume that rational nodes receive a long-term benefit from participating in the protocol. Second, we assume that rational nodes are pessimistic when computing the impact of Byzantine nodes may have on their utility. Finally, we assume that Nash Equilibria are an appropriate solution target.

Rational nodes will only participate in a cooperative system if they receive a long term net benefit from participation. In practice, this requires that the long-term benefit (e.g. reliable backup) of participation is sufficient to offset the costs (e.g. storage, bandwidth, computation) of participating in the system. We consequently model our protocols as infinite horizon repeated games [7, 8].

Rational nodes want to reduce their work, if possible, without renouncing the benefits that come from participating in the protocol. We assume a fairly simple model, in which nodes' utilities are affected by the work that must be done do but not by the order in which work is performed, or who requests the work. These two variants can be handled by hiding the relevant factors (contents of the request or identity of the sender, respectively) until after nodes commit to executing the request. We assume that rational nodes deviate from the protocol only if they receive a net benefit from doing so—in a tie, they continue to follow the protocol. This appears reasonable, given that deviating from the protocol requires some effort. Furthermore, we assume that rational nodes abide by the promptness principle: if they gain no benefit from delaying the sending of a message, they send the message as soon as they have idle cycles available. This assumption recognizes that idle cycles are a perishable resource that may not be available at a later time.

Rational nodes are conservative when estimating the potential impact of Byzantine nodes on their utility. In particular, when computing the expected outcome of their actions, a rational node assumes that Byzantine nodes will act in the way that minimizes its utility. Additionally, since rational nodes are interested in continuing to benefit from the system, they conservatively assume that the maximum number of Byzantine nodes are in fact present in the system.

We focus on developing a protocol that achieves a Nash equilibrium [43]) in which no rational node has a unilateral incentive to deviate from the given protocol. [1] We assume that a rational node, if given a protocol that is a Nash equilibrium, will follow it. [38].

We assume that rational nodes do not have the computational power to subvert the standard cryptographic assumptions associated with public key signatures [54] and secure

[1]Because our "given protocol" can be regarded as coming from an external authority, Papadimitriou prefers to regard such an equilibrium as a *correlated equilibrium* [6], which is a generalization of Nash equilibrium. This view would not change our analysis.

hashing [49]. Additionally, we assume that rational nodes do not collude.

Byzantine nodes. We assume a Byzantine fault model for Byzantine nodes [15, 34, 39] and a strong adversary. Byzantine nodes can exhibit arbitrary behavior. For example, they can crash, lose data, alter data, and send incorrect protocol messages. Furthermore, we assume an adversary who can coordinate Byzantine nodes in arbitrary ways. However, we assume that Byzantine nodes do not have the computational power to subvert the standard cryptographic assumptions associated with public key signatures [54] and secure hashing [49]. We assume that at most $\frac{n-2}{3}$ of the nodes in the system are Byzantine.

4 System Architecture

We propose a three-layer architecture to support the development of BAR services. The *basic primitives* bottom layer provides IC-BFT versions of key abstractions (e.g. Terminating Reliable Broadcast (TRB) [34] and Replicated State Machine (RSM) [15, 33, 58]) for constructing reliable distributed services. Building on these abstractions, the middle *work assignment* layer implements mechanisms that address, in the BAR model, a basic design issue of many distributed services: how to partition work among the system's components. Finally, the top *application* layer performs application-specific actions, e.g. defining application-level requirements and guarantees, verifying that each component faithfully performs the work assigned to it, and taking appropriate action when one does not.

In our backup application, we use the bottom two application-independent layers to take a request r intended for node i and bind it to either (a) a well-formed response to r, signed by i (b) a provably ill-formed response (or set of responses) signed by i, or (c) a "no evidence" response, after $f + 1$ nodes unilaterally decide that an application-specific timeout has been exceeded for r. Given a response-request binding, the application layer is responsible for (a) judging whether a syntactically "well formed" response to a request constitutes a "legal" response to the request based on application semantics and (b) taking appropriate action in response to any proof of misbehavior against a node.

Accountability lies at the core of this approach to constructing BAR services: if nodes are accountable for their behavior, then rational peers have an incentive to behave correctly. Strong identities and restricted membership make it possible to enforce meaningful internal and external disincentives. But that is only part of the solution. How should a system detect and react to incorrect behavior?

One case occurs when a set of messages constitute a self-contained cryptographic Proof Of Misbehavior (POM) by a node. For example, if a node first signs a promise to store a file with a particular cryptographic hash and then responds to a request to read the file with a signed message that contains the wrong data, the two messages amount to a signed confession by the node that it is faulty and should be punished. This "aggressively Byzantine" behavior is actually the simplest kind of misbehavior to detect and punish, and a number of systems have done so [16, 44].

Two other "passive-aggressive" cases are more difficult. First, a node may decline to send a message that it should send. The receiver is in a position to accuse the node of wrongdoing, but it becomes a case of "he said/she said"—it is difficult for any third party to decide whether an accusation of inaction is legitimate or has been unjustly leveled by a self-interested or faulty node. Second, a node may exploit nondeterminism to provide incomplete information or make undesirable decisions that interfere with operation but are difficult to conclusively prove wrong. For example in an asynchronous replicated state machine [15], a node normally transmits a signed copy of the request, but for liveness it is permitted to unilaterally time out and transmit a signed timeout message instead. In such a system, it may be preferable for a self-interested node to send a timeout rather than transmit the request. This choice would inhibit progress, but it would be hard for another node to prove that a timeout message was inappropriate.

Our architecture addresses these passive-aggressive behaviors in two steps. First, at the lower, application-independent layers, we explicitly design the protocols and the incentive structure to ensure that it is not in the interest of a rational node (i) to be silent when the protocol calls for it to send a message, (ii) to send messages that are not *well-formed* (e.g., a well formed response message would have to be signed and include a hash of the corresponding request), or (iii) to substitute an undesired message for a desired one when it is free to make a nondeterministic choice. Conversely, the higher level protocols deal with these behaviors by (i) relying on lower levels to "force" applications to provide well-formed responses to requests and (ii) using application-level state and semantics to restrict what replies may legally be made to a request.

The purpose of this layered architecture is to simplify the design and analysis of BAR services. Abstractions with provable properties and layered architectures are powerful tools for managing the complexity involved in building and reasoning about fault-tolerant distributed services [10, 15, 39, 58]. We believe it is crucial to leverage these tools as we consider the added difficulties that a BAR model introduces – protocols that attempt to handle these challenges with monolithic, end-to-end solutions run the risk of being too prohibitively complex to reason about. On the other hand, a layered IC-BFT solution introduces its own set of challenges, as it needs to ensure a seamless binding between the incentive structure used in the lower layers and the overall end-to-end incentive structure of the application.

Our architecture essentially defines a contract between the application layer and two two lower, application-independent layers. The lower, application-independent layers provide important abstractions to the upper one, but they require from the application an incentive to drive nodes to participate in the lower-level work: the overall benefit of being in the service must exceed the cost for every rational node in the system.

5 Level 0: BART state machine

At the core of fault-tolerant distributed services are a few fundamental primitives. For instance, state machine replication is essential to most highly available replicated services [12], and quorum-based replication is the basis for fault-tolerant distributed storage systems such as Phalanx [40]. The purpose of the bottom layer of our architecture is to implement fundamental primitives so that they provide their familiar guarantees within the BAR model. In this section, we present a BART asynchronous replicated state machine (RSM). Our protocol is based on PBFT [15], with modifications motivated by the BAR model. These modifications are based on three high-level ideas.

The **benefit principle** states that nodes must gain long term utility for participating in the system. This long-term incentive is necessary to motivate self-interested nodes to participate faithfully. Our RSM rotates the leadership role to guarantee that every node has the opportunity to submit proposals to the system.

Predictable communication patterns encourage nodes to participate at every step of the protocol instead of just at the steps that bring them a direct benefit. Our protocol requires nodes to have participated in all past steps in order for them to be able to submit a proposal.

Limiting non-determinism ensures that the predictable communication patterns contain useful work. Non-determinism offers nodes the choice of multiple acceptable behaviors, each of which are "correct" in different circumstances. Given a specific state of the protocol, one of the behaviors is preferred by the protocol, and nodes must be given proper incentive to choose the preferred behavior. In our implementation of IC-BFT primitives we carefully limit the choices available to a node. For example, we base our state machine on TRB rather than consensus, because the former protocol, by allowing fewer valid outcomes, gives rational nodes fewer options to choose from when deciding which behavior maximizes their benefit.

When non-determinism is unavoidable, two low level techniques are often useful. *Cost balancing* is employed when a node has a choice between multiple actions. The costs of the actions involved are engineered so that the protocol-preferred choice is no more expensive than any other potentially legal choice. For instance, instead of sending a list of nodes that are up-to-date, an IC-BFT protocol would send n bits with an entries set to "1" for up-to-date nodes so that the sender saves no network bandwidth by sending incomplete information. Asynchrony is particularly challenging due to its inherent non-determinism; unfortunately, timeouts are required to ensure progress. *Encouraging timeliness* allows nodes to unilaterally judge whether other nodes' responses are "on time" and to inflict sanctions for untimely messages. Our techniques ensure that (a) nodes have incentives neither to mete out unwarranted sanctions nor to forbear deserved punishing and that (b) the costs imposed by Byzantine nodes through spurious sanctions are limited.

5.1 Protocol description

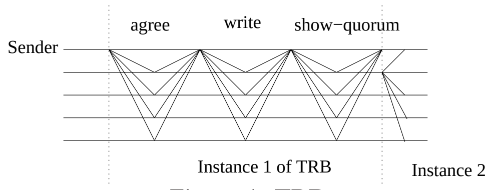

Figure 1: TRB

Our BAR replicated state machine protocol is based on PBFT [15]. When a node wants the state machine to execute a request, the node proposes the request in a TRB instance. Instances proceed in sequence, with instance i deciding the ith operation to be executed by the state machine. We differ from PBFT protocol in several key points.

1. We use TRB instead of consensus. This choice is a specific application of the principle of *limiting non-determinism*: as opposed to consensus, in TRB only the *sender* may propose a value during a particular instance. We initially attempted to use a consensus protocol as the engine of our state machine, but found the restriction on who can propose in each instance necessary in accounting for the behavior of rational nodes. Without this restriction, a new leader elected to terminate instance i after sufficiently many nodes have timed out on the sender may prevent progress by selfishly trying to make the state machine adopt its value, rather than the sender's (see Appendix C.4).
2. We enforce a round-robin leader election policy. This rotation gives every node a fair chance to propose commands to the RSM, and enforces a predictable exchange of the leadership position.
3. We require at least $3f + 2$ nodes (rather than $3f + 1$) to tolerate f Byzantine nodes. The reason is subtle and, once again, has to do with the need to account for rational nodes; in particular, in instance i liveness can be compromised if the sender s is slow, and, after a new leader has been elected to complete i, s is able to influence whether the new elected leader proposes s's original value, or a default value (see Appendix C.4).

   By using an extra node, we can prevent the slow sender from participating in the steps required to complete the instance without jeopardizing safety or liveness.

Our TRB protocol provides four guarantees in an eventually synchronous BAR environment. *Termination*: every non-Byzantine process eventually delivers exactly one message. *Agreement*: if a non-Byzantine process delivers a message m, then all non-Byzantine processes eventually deliver m. *Integrity*: if a non-Byzantine process delivers m, then the sender sent m. *Non-triviality*: In periods of synchrony where the benefit principle holds, if the sender is non-Byzantine and sends a message m, then the sender eventually delivers m.

The protocol provides safety under an asynchronous model, but is live only during periods of synchrony [28]. We assume that, during these periods, there exists a known bound Δ on message delivery time. Note, however, that practical implementations of our protocol are likely to further require a known but large (e.g., 1-week) bound on the time between when a request is made and a response is returned by a rational node. We use this stronger assumption to bound state, we discuss this issue in Section 5.1.5. Relaxing this timing assumption requires the inclusion of incentive-compatible garbage collection and checkpoint recovery; this does not appear to introduce additional fundamental difficulties and is the focus of ongoing work.

Figure 1 illustrates an execution of TRB occurring in a period of synchrony when no failures are present. Each TRB *instance* is organized in a series of *turns*. In each turn, some process is designated the *leader*. The pre-specified *sender* for instance i is the first leader for instance i. In the first turn, the *sender* attempts a three-phase-commit on a proposed value (the phases are labeled agree, write, and show-quorum). If the other nodes receive the messages on time then they accept the value and the broadcast is successful. If, on the other hand, nodes decide the message is late, they send a "set-turn" message to indicate that a new turn should start. Nodes other than the sender are selected round-robin for the leader role.

First, the newly selected leader performs a read: it queries all nodes for their current observed value and waits for a quorum of responses. If any node reports seeing the *sender*'s proposal, then the new leader attempts to broadcast that value. Otherwise, the new leader broadcasts $\perp$, indicating that the *sender* is suspected of having failed. Once a value is delivered, the $i + 1$th instance starts with the next *sender* in the sequence.

Appendix B includes detailed pseudo-code for the algorithm. Due to space constraints, we limit our discussion to the key differences between our protocol and traditional BFT implementations.

5.1.1 Message queue

Message queues are the low-level mechanism we use to enforce *predictable communication patterns*. All communication takes place through the message queue infrastructure.

Message queues implement a simple "you don't talk to me, I won't talk to you" policy: if node x next expects a message from node y, x will ignore any communication from and delay any communication to y until it receives the expected message. The message queue used by x to regulate its communication with y contains entries for the messages that x intends to send to y, interleaved with "bubbles" corresponding to messages that x expects from y. A bubble must be filled with an appropriate message from y before x can proceed to send the messages in the queue beyond the bubble. To ensure that y sends the appropriate message, and not just any message, a predicate is associated with each bubble: a message from y is allowed to fill some bubble only if it satisfies the corresponding predicate—otherwise, it is discarded. The message queue exports three operations : `send` and `expect`(*predicate*) insert in the queue, respectively, a message and a bubble; `deliver` removes the bubble closest to the head of the queue and lets x read the corresponding message.

Message queues, combined with quorums of size $n - f - 1$, provide the incentive for rational nodes to send messages expected in the protocol. If a given rational node r chooses not to send a message to some node s that follows the protocol, then s will ignore r in the future. In the worst case for r, the f Byzantine nodes in the system will not communicate with r, preventing it from gathering a quorum during its next turn as *sender*. This will prevent r from gathering the quorum of responses required in a later step of the protocol, stopping r from making progress and effectively excluding it from the state machine.

5.1.2 Rotating leadership

Traditional replicated state machines require the client to send a command to a sender, who proposes the command to the state machine. The client is missing from Figure 1 because we rotate the role of *sender* among the nodes in the system. This provides nodes with a periodic opportunity to propose values to the state machine, partially satisfying the *benefit principle*. Due to the self-interested nature of rational nodes, a node can only be certain that a specific request is proposed to the state machine if that nodes proposes the request itself.

5.1.3 Balanced messages

To apply the principle of *cost balancing* to the consensus protocol, we make sure that all messages have the same cost. This influences for example the behavior in the first phase of the protocol ("giveOldValue"), in which a newly elected leader asks nodes for the latest command they have seen. We require the answer to always be of the length of the largest possible command—even if in fact the node has received no command yet—so that lying would not allow a node to send a shorter message.

5.1.4 Penance

In the Byzantine model, correct nodes send all relevant protocol messages without fail. In the BAR model, rational nodes may skip messages that decrease their net utility. When communication patterns are predictable and a rational node knows that a specific message will be sent eventually, that message is sent immediately by the promptness principle. The *cost balancing* mechanism described in Section 5.1.3 provides incentives to send the protocol preferred message when a node must choose between two or more possibilities. Encouraging good behavior among rational nodes is more challenging when waiting may allow a node to avoid sending a specific message entirely. In our protocol this is especially relevant for the ”set-turn” message required as part of the new leader election phase.

We implement a “penance” mechanism to encourage timeliness in the state machine. Individual nodes maintain an *untimely* vector that tracks their perception of other nodes timeliness: a node is considered untimely if the node sends ”set-turn” messages earlier or later than they are expected by another node. Values proposed by a *sender* include the *sender*’s *untimely* vector. When a value is delivered, all nodes except the *sender* expect a penance message from each untimely node. The untimely nodes must send the penance message to all non-*sender* nodes in order to continue using the system. There are three important considerations to the penance message: (1) the size and form of the penance message must be chosen so that the expected benefit of sending late is less than the expected penance cost, (2) the *sender* is excluded from receiving penance messages to prevent the *sender* from incurring additional costs through truthfully reporting a penance, and (3) the spurious work introduced by Byzantine nodes through the penance mechanism is bounded by fn penance messages per node.

5.1.5 Timeouts and garbage collection

In specifying a *predictable communication pattern*, we require all nodes to send all protocol messages. In particular, if node a remains silent for an extended period of time it can force non-Byzantine node b to cache an arbitrarily large set of messages reflecting the history of the protocol. These messages must be cached so the two nodes can fulfill their message queue obligations once a becomes active again. In the absence of incentive-compatible checkpointing (to allow garbage collection [15]), the cost of participating in the system can grow without bound. Rational nodes will withdraw from the system when the costs become too large, eliminating liveness even in periods of synchrony.

We bound this state in two ways. First, we introduce an additional weak synchrony assumption: non-Byzantine nodes are guaranteed to respond to a request by $max\_response$ (e.g. 1 week) after the request is issued. Our state machine leverages this assumption to bound state through the per node maintenance of a $badlist$. Node a considers node b to be bad if b has not sent an expected message for longer than $max\_response$. Nodes publish their $badlist$s when submitting proposals to the state machine; any node listed in $f + 1$ lists is considered to be Byzantine. Nodes which consider another node Byzantine are able to unilaterally discard all messages corresponding to the presumed Byzantine node. In addition to helping to bound state, this mechanism enables the use of quorums of size $\lceil \frac{n+f}{2} \rceil$ rather than $n - f - 1$, improving availability of the state machine.

In addition to bounding the time that a node is required to store intermediate data to a coarse grained time out, we also limit the amount of data any node can insert in each timeout interval. The middle and upper layers implement a request-response communication pattern and it is appropriate to assign to a requestor the overheads imposed by both a request and the resulting response. We consequently address the details of limiting the rate of requests to the state machine as part of our work allocation primitive in Section 6.4.

5.2 Proving IC-BFT

To prove that a protocol is IC-BFT for a given model of rational nodes’ utility and beliefs, one must first prove that the protocol provides the desired safety and liveness properties under the assumption that all non-Byzantine nodes follow the protocol. Second, one must prove that it is in the best interest of all rational nodes to follow the protocol.

Our rationality model is described in Section 3. We assume that rational nodes will follow the protocol if they observe that it is a Nash equilibrium, so we must show that no node has a unilateral incentive to deviate. We show this by enumerating all possible deviations.

The simplest deviations are those that do not modify the messages that a node sends. In our state machine protocol, no such deviation increases the utility. We must then examine every message that the node sends and show that there is no incentive to either (i) not send the message, (ii) send the message with different contents, or (iii) send the message earlier or later than required. Also, we must show that nodes have no incentive to (iv) send any additional message.

Using these techniques, we arrive at the following (Appendix B).

Theorem 1. *The TRB protocol satisfies Termination, Agreement, Integrity and Non-Triviality.*

Theorem 2. *No node has a unilateral incentive to deviate from the protocol. (*Incentive compatibility*)*

To illustrate the methodology, we show some of the lemmas involved in verifying the incentive-compatibility of the sending of the “set-turn” timeout message. The incentive for sending the message at all and not sending it twice are discussed in more general lemmas, not shown here.

Lemma 1. *No rational node r benefits from delaying sending the “set-turn” message.*

Lemma 2. *No rational node r benefits from sending the “set-turn” message early.*

The proof sketch for the first lemma relies on the penance protocol described in the previous section. The second lemma deals with early time-outs. By construction, $\perp$ is at least as large as the sender’s command. Nodes other than the sender have no stake in which command is decided because they cannot prevent the sender’s command from executing but at most delay it. The sender itself could have an interest in manipulating the outcome by sending "set-turn" early or late, which is why in our protocol the sender is not allowed to send these messages.

Lemma 3. *No rational node r benefits from sending a malformed "set-turn" message.*

The set-turn message contains no information other than the turn number, so a malformed message reduces to either a nonsensical message, a resend or an early send.

6 Level 1: Partitioning Work

State machine replication requires each replica to process each command and maintain a full copy of the state required to do so. This can be impractical for many applications in cooperative services with 10's, 100's or more nodes. For example, a cooperative backup application that requires 100 GB of actual storage in order to provide the abstraction of 1 GB of stable storage is unacceptable. The purpose of the middle layer of our architecture is to support flexible and efficient strategies for partitioning work among the nodes in the system.

In particular, we support two general approaches. The first is to organize the nodes into multiple state machines [4, 53]. Intuitively, each replicated state machine can be thought of as providing the abstraction of a single, correct node. In the BAR model, this translates into saying that, if the incentives are in place for individual rational nodes to follow the assigned protocol, the replicated state machine provides the abstraction of a known, altruistic node. The work can then be partitioned among the replicated state machines, which can also be relied upon to route the work where appropriate. Using this approach, the replication of work and state grows with the size of the individual replicated state machines, rather than the size of the system.

A second, more aggressive approach is to assign work to specific nodes in the system. Of course, unlike replicated state machines, individual nodes may be faulty, but some classes of applications can cope with such failures. For example, in the context of backup, arithmetic coding [55] can be used to store different data on different nodes and thus tolerate faults while reducing the storage overhead.

Our middle layer supports a combination of these two approaches: as shown in Figure 2, we let an individual node assign work to the state machine to which it belongs (e.g., a to A); state machines can assign work to each other, and correctly route these work assignments (e.g., A to B to C); and finally, a state machine can assign work to any one of its nodes (e.g., B to b).

The implementation of work partitioning layer leverages three principles:

- State machines can function as unimpeachable witnesses of the interactions that lead to work assignments.

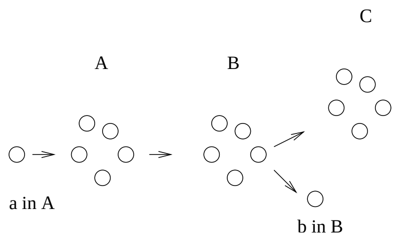

Figure 2: Partitioning work in the system

- The state machine "testimony", in the form of a Proof of Misbehavior (POM) can be used by higher levels of the protocol to hold rational nodes accountable for their (in)actions. In addition, the state machine can "take justice in its own hands" by refusing service to non-responsive nodes.
- Credible threats [21] can be used to reduce the load on the state machine. The credible threat of asking the state machine to witness a work assignment is enough, in the common case, to motivate rational nodes to honor work assigned directly to them by other nodes.

6.1 Assigning work to nodes

When node s wants to assign work w to node a, it submits to the state machine the command *assign*(*request*), where *request* is a tuple of the form (w, a). The state machine replies with an ack for *request*. When s receives the ack, it inserts an `expect` bubble in its message queues to each of the nodes in the state machine. The corresponding predicates indicate that s expects from each node i a well-formed response: i's response should either include a message signed by a with the result of executing w; or it should indicate that i received no evidence that a executed w. To avoid being shunned by s, each non-Byzantine node i has an incentive to send a well-formed response to s; furthermore, since by design the cost of sending the "no evidence" response is higher than the expected cost of forwarding a message from a, i has an incentive to try to respond with the latter. Hence, each non Byzantine i inserts in turn an *expect* bubble in its message queue to a, with a predicate that requires a to send it a well-formed message that includes the result of executing w.

Node s files to the application layer a POM against a if it either receives $f + 1$ "no evidence" responses, or it receives two or more different responses signed by a. In our extended technical report [1] we prove the following lemma:

Lemma 4. *If the state machine executes the command* assign(w, a) *and a is rational, then a will execute w.*

6.2 Assigning work to state machines

The primary difficulty in providing communication between state machines is in motivating state machine B to process requests which originated from another state machine A. We address this difficulty by overlapping membership of state machines. Figure 3 illustrates the architecture for a simple directed ring topology. Requests assigned to an adjoining state machine B are conceptually assigned to the $f+1$ nodes in the intersection. This intersection is guaranteed to contain at least 1 non-Byzantine node and all rational nodes in the intersection assume that the other f nodes are faulty and consequently submit the request to B at their next opportunity.

In our extended technical report [1] we prove the following lemma concerning state machines A and B which overlap in $f+1$ nodes:

Lemma 5. *If A executes the command* assign(w,B), *then B will execute w.*

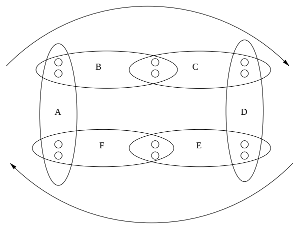

Figure 3: Linking state machines

In a system composed of state machines linked as we describe, the safety of the whole system relies on the assumption that there are no more than f Byzantine faults in any replicated state machines. As the number of state machines increases, it may be wise to be more conservative in one's choice of f.

6.3 Optimizing work assignment

In a backup service like PIB, requiring s to submit large backup requests through the state machine could incur large overhead when the system is under heavy load. To enable more efficient work assignment in these circumstances, we leverage the game-theoretic notion of credible threats [21]. In the game of chicken [2], a credible threat against rational players would be to visibly rip off the steering wheel and throw it out the window. In our case, a credible threat takes a somewhat less spectacular form.

In PIB, a node s issues a credible threat by submitting to the state machine the command *vow*(a,t). Through it s promises, by time t (i) to ask some work of a and (ii) to submit to the state machine the response it receives from a. When a state machine replica i executes *vow*(a,t), it inserts in its message queue to s an `expect` bubble. This makes the threat credible, as long as t is sufficiently far in the future, because a knows that s would be shunned by all replicas after time t if it were to break its vow. To fill the bubble, s either sends a receipt for the work performed by a, or a more expensive default message mandated by the protocol.

Intuitively, the presence of a credible threat gives an incentive to rational a to respond directly to the work assignment w that it receives from s without waiting for s to submit *assign*(w,a) through the state machine—by Lemma 4, a would in that case have to perform w anyway, and would furthermore have to send more messages, incurring higher cost. Counting on a's reasoning, s, having submitted a credible threat, would in turn have an incentive to contact a directly before issuing *assign*.

For technical reasons [1], the incentive compatibility of this approach is only guaranteed when a is the concurrent target of $f+1$ *open* credible threats, i.e. credible threats where a's action may determine whether the threat will come to pass.

6.4 Regulating work volume

As discussed in Section 5.1.5 we must regulate the volume of work submitted to the state machine. Since our fundamental communication pattern is that of request-response, it makes sense to attribute the work imposed by both the request and its subsequent response to the requester. We regulate the volume of work submitted to the state machine by granting each node a consumption quota that is reset after an appropriate time period, for our application this time period is on the order of one week. The size of every request the node issues through the state machine as well as the size of every resulting response counts against the node's quota. If the node exceeds its consumption quota, then the state machine treats the node as faulty as discussed in Section 5.1.5. It is the responsibility of the application layer to set the consumption quota to a reasonable value.

7 Level 2: The Application

In our architecture, BAR applications must discharge each of the following four responsibilities in order to take advantage of lower-level abstractions.

1. Provide rational nodes with a long-term benefit for participating in the system.
2. Assign work to nodes in a fault tolerant manner.
3. Determine if the contents of a request or response constitute a Proof of Misbehavior (POM) under the application semantics.
4. Sanction nodes that have provably misbehaved.

It is much simpler to design an application under these requirements than under the lower-level principles discussed in Sections 5 and 6. Because lower-level primitives handle reliable work assignment, the application focuses on defining the legal requests and responses over the system's data. As a result, the reader will notice that the following discussion is considerably simpler than that in earlier sections: it focuses on structuring the messages so that incorrect responses are also proofs of misbehavior and not on encouraging nodes to respond or on balancing costs.

To illustrate how an application addresses these issues, this section examines PIB, a MAD cooperative backup system.

7.1 PIB overview

PIB is a cooperative backup system in which nodes commit to contributing an amount of storage to the system (and to participate in the system's state machine) in exchange for an equal amount of space on other nodes. In normal operation, PIB consists of three fundamental operations: store, retrieve, and audit. When a group of files are marked for storage, the owner splits them into smaller pieces (chunks) which are sent to different nodes (storers) for storage on the system. The storers respond with signed receipts. The owner keeps the receipts and the storers keep the StoreInfos (part of the store request) as their "record of participation" in the system. When the owner needs to retrieve a file, it sends a retrieve request to each node holding a relevant chunk. Any node that refuses to return the chunk without valid reason (e.g. it holds a more recent StoreInfo that overwrote the chunk) is guilty of misbehaving and can be punished. Nodes periodically audit each other's records in order to verify that nodes are not using more space in the system than their quota allows. PIB relies on the work allocation primitive described in Section 6 to reliably distribute work in the system. The work allocation rate limit is set to prevent a node from triggering requests and responses totaling more than double the node's storage quota per coarse timeout interval; violation of this limit may lead to the formation of a POM against the offending node.

7.1.1 Arithmetic coding

We employ arithmetic coding for (a) fault tolerance and (b) reducing the cost of running the PIB system. Nodes erasure code [55] files with an $x - f$ out of x encoding and store the resulting chunks on different peers in order to tolerate f faults with less storage than required by full replication. For example, in a 10-node system with $f = 2$, a node must contribute 1.3GB of local storage to back up 1GB of data. Keeping this ratio reasonable is crucial to motivate self-interested node to participate faithfully.

7.1.2 Request-response pattern

The core of a BAR application in our system is carefully structuring messages so that an incorrect response to a request constitutes a POM against the sender of the response. The work assignment primitive in Section 6 provably binds responses to requests or to $\perp$ if the target fails to respond. Every message in the PIB protocol is stamped with a unique sequence number and signed by the sender.

Store. A PIB store request consists of two components. The first is a tuple *(name, owner, storer, hash, size)* called the *StoreInfo*. The second is the chunk being stored. The *hash* and *size* fields of the *StoreInfo* correspond to the hash and size, respectively, of the chunk being stored. The *StoreInfo* tuple is stamped and signed by the owner. There are three possible responses to a store request: (a) a *Receipt* containing the *StoreInfo* and stamped and signed by the storer, (b) a *StoreReject* containing the *StoreInfo* and a *Proof* that is stamped and signed by the storer, and (c) anything else. If a receipt is received then the receipt is added to the owner's record of consumption on the system, known as the *OwnList*. A *StoreReject* can validly contain proof that the storer is full: a list of *StoreInfo* records, each signed and stamped by their respective senders, where leases from their stamps have not expired and whose total size plus the request's *StoreInfo* size exceed the node's quota. Any other response constitutes a POM against the storer–either (a) the response itself is a POM generated by the work allocation layer (e.g., $\perp$) or (b) the response is inappropriate for the request and thus a signed confession.

Retrieve. A PIB retrieve request consists of the *Receipt* for the chunk to be returned. The three possible responses to a retrieve request are: (a) a *RetrieveConfirm* containing the *Receipt* and the corresponding chunk stamped and signed by the storer, (b) a *RetrieveDeny* containing the *Receipt* and a *Proof* stamped and signed by the storer, and (c) anything else. If the response is a *RetrieveDeny*, then the the *Proof* must show either (a) *Receipt* has expired (b) the *Receipt* has been superseded by a more recent *StoreRequest* from the same $owner$ to the same $name$, or (c) the storer is in the process of recovering its data (see below). Any other response constitutes a POM against the storer– either the response itself is a POM generated by the work allocation layer or the response is inappropriate for the request and thus a signed confession.

Audit. The audit mechanism takes place in three phases. First the auditing node selects a node to audit. The auditing node then requests both the *OwnList* and *StoreList* from the auditee. After retrieving the two lists, the auditing node requests the *OwnList* for every node the auditee claims to store files for and the *StoreList* for every node the auditee claims to own files on. The collection of lists are cross-checked for inconsistencies; any inconsistencies result in a POM against the offending node. An *OwnList* and *StoreList* are inconsistent if a *Receipt* indicated on one should be present but is not on the other. Audits are potentially very expensive operations, and nodes will avoid performing them if possible. We avoid this problem by requiring each node to submit the results from a recent audit – either a POM or a complete set of *OwnList*s and *StoreList*s to the state machine every 1000 proposals.

7.1.3 Time constraints

The primary purpose of a backup system is to provide convenient retrieval following a catastrophic disk or user failure. The utility of a backup program is greatly reduced if the retrieval guarantee is "eventual recovery" rather than "recovery within time t." In order to guarantee a concrete recovery window, PIB assumes that all non-Byzantine nodes will respond to a request within $max\_response$. Any node that fails to do so is considered faulty, a POM against such a node can be acquired by issuing a request through the work allocation primitive.

We utilize leases to bound the duration of store requests on the system. When messages are signed, the sequence number is actually a time stamp reflecting the local clock of the participating machine. In PIB, every *StoreInfo* expires 30 days after the request is signed by the owner. If the owner needs to keep the chunk in the system for more than 30 days, the owner must renew the chunk by sending a second *StoreRequest* before the original lease expires.

In order to support leases and allow nodes to consistently interpret the time stamps applied by other nodes, we assume that the clocks for all non-Byzantine nodes are synchronized to within one day of each other. If $f + 1$ nodes certify that a node's clock is outside this synchronization window, they are collectively capable of issuing a POM against the node.

The introduction of these timing assumptions and lease durations allows PIB to (a) provide stronger guarantees with respect to recovery time and (b) limit the amount of "dead" storage in the system. These two factors aid in increasing the overall utility of the system, making it more attractive for rational nodes.

7.1.4 Sanctions

Various components of the PIB system, from the primitives in Sections 5 and 6 to the mechanisms described earlier in this section, generate POMs against specific nodes. These POMs convict a node of misbehavior and require that the node be punished appropriately; without appropriate punishment nodes have no incentive not to misbehave.

For simplicity PIB handles all POMs in the same fashion: whenever a POM is submitted to the state machine, the POM is distributed to all nodes and each node evicts the guilty party. Note that the POM provides a basis for more sophisticated strategies including suspending a node's store and retrieve rights pending administrative intervention, increasing the storage a node must contribute (without increasing its quota) or emitting the POM "up a level" to an administrative entity for external disciplinary action.

7.2 Recovery

Since we are dealing with a backup system, nodes that lose their local state must still be able to make use of the system. Our approach (1) allows such a node to assume a new identity to access its old state and (2) restricts this ability to prevent rational nodes from shirking work and to limit damage by Byzantine nodes.

Initially we give each node a fixed series of *linked identities*, $i_0 \ldots i_{max}$. A node using identity i_{j-1} can begin using i_j at any time. Any node that receives a message from identity i_j (1) assigns all message queue bubble obligations of any preceding linked identity ($i_k (k < j)$) to i_j, (2) grants Retrieve rights to i_j for any data with a valid lease by i_k, (3) initiates a fixed grace period during which *RECOVERING* is considered a valid response by i_j to any retrieve request, and (4) evicts i_k from the system. i_j uses the grace period to first issue RECOVER requests to all nodes; nodes must reply with each non-expired chunk on their *StoreList* stored by i_k (they may have to Retrieve other chunks to construct this information, but the work assignment operation ensures that they perform this expensive action.) Then, i_j may retrieve its own data from the backup system. Identity i_k is technically removed from the system (and no longer counts against our limit of f simultaneous failures) at the moment that i_j finishes recovering other nodes' stored data.

Three factors prevent a rational node from deliberately exploiting linked identities to avoid punishment. First, each node has a small number of identities (e.g., 3 initially plus 1 every two years) and cannot recover its data after all have been used; using a linked identity thus reduces the future utility of the system. Second, a new linked identity is responsible for the messages of previous identities, so nodes cannot avoid work. Third, linked identity i_j must contribute 1.1^j times the storage of identity i_0 but receives no corresponding increase in quota; this ensures that a node cannot reduce its total disk storage costs by failing to store data and then switching to a new linked identity to hide this fact. Note that the first factor also limits the damage that can be done by a series of linked entities under a "persistently Byzantine" node's control.

7.3 Guarantees

The PIB system provides the following guarantees under PIB's coarse synchrony assumptions. (i) Data stored on PIB can be retrieved within the lease period. (ii) No POM can be gathered against a node that does not deviate from the protocol. (iii) No node can store more than its quota on PIB without risking being caught. (iv) If a node with at least one unused linked-identity crashes and loses its disk, it is guaranteed a window of time during which it can rejoin the system and recover all data it has stored.

8 Evaluation

In this section we evaluate our replicated state machine and PIB prototype. Our microbenchmarks show that our RSM prototype can perform about 15 operations a second for small groups of users, an adequate level of performance for our application's requirements. We then evaluate the performance of the PIB application by storing and retrieving large amounts of data. We find that our non-optimized PIB prototype can backup in 20 minutes a gigabyte of data to 21 nodes, which ensures that the data is recoverable despite the failure of 7 nodes.

8.1 Experimental setup

We ran all experiments on Pentium-IV machines with 2.4Ghz processors, 1 GB of memory, and Debian Linux 3.0. These are public machines, connected through 100Mbps ethernet.

Our prototypes are implemented using Java 1.4. We set the initial TRB network timeout to 10 seconds. The maximum response time and lease duration are set to a week and a month respectively, but our experiments did not rely on these values. The experiments assume no failures. We use the BouncyCastle cryptographic library and Onion Networks' FEC library for erasure coding.

8.2 Micro-benchmarks

We use micro-benchmarks to evaluate our replicated state machine prototype. The main questions we try to answer are (a) whether our RSM is practical, (b) whether our RSM scales to a reasonable number of peers, and (c) whether our RSM handles intentionally slow nodes well.

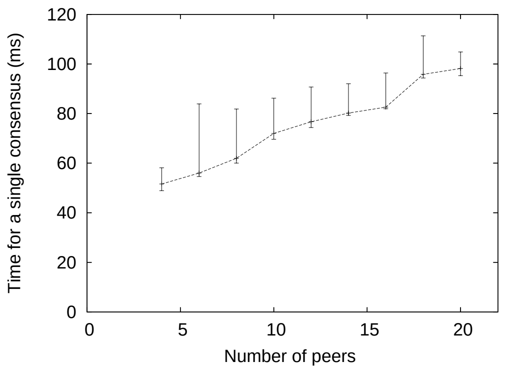

Figure 4: RSM performance as peers are added

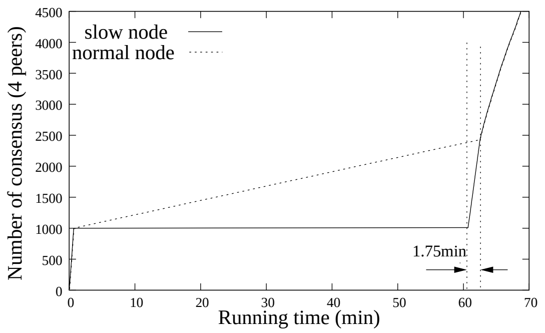

Figure 5: Time a node spends catching up for its absence

Figure 4 shows the average speed of consensus operations for systems of 4 to 20 nodes. Each trial measures the average duration over 50 consensus operations. We run each configuration 10 times and show the median value as well as the 10^{th} and 90^{th} percentiles. The chart shows consensus completes in less than 50ms for 4 peers or 100ms for 20 peers, a level of performance that is appropriate for our application. It also shows that performance scales almost linearly with the number of peers, hinting that performance would remain appropriate even with larger groups. Eventually, however a large cooperative service should be split into multiple state machines as described in Section 6.

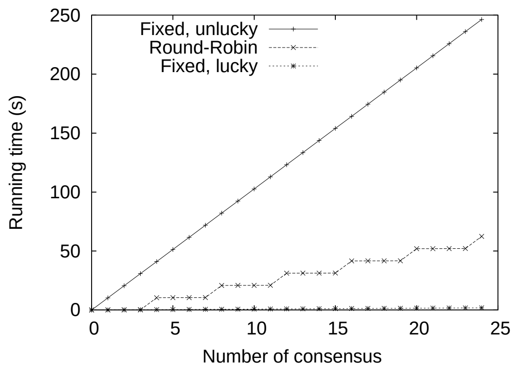

Figure 6: Impact of rotating leadership

Our performance is inferior to protocols that are not designed for the BAR model. PBFT [15] requires only 15 ms per consensus on less powerful hardware than ours. Part of the difference is explained by our language choice, but the main factor is the fact that our IC-BFT RSM requires the properties of digital signatures, so we cannot rely on the faster MAC primitives.

Figure 5 shows the effect of a single node (out of four) that is unreachable for an hour. Our protocol, for incentive compatibility reasons, does not allow that node to skip directly to later instances of TRB when it returns: a concern then is that it may take too long for nodes to catch up. The experiment shows that the unreachable node (solid line) was able to catch up in less than two minutes, so the impact of long periods of unreachability is minor.

Figure 6 shows the relative impact of two leader election policies in the presence of failures. Our protocol rotates the role of sender between each instance of TRB. A PBFT-like protocol instead rotates the sender only when the current sender is determined to be faulty or untimely. When the sender is timely and non-Byzantine, the state machine proceeds at full speed for either protocol, without timing out (cf. “Fixed, lucky”). However, a Byzantine sender can proceed slowly—just fast enough to avoid triggering a time-out (cf. “Fixed, unlucky”). Our sender rotation (cf. “Round-Robin”) limits the worst case damage imposed by a slow node and achieves a performance closer to the best case.

8.3 PIB

PIB performs adequately when storing and retrieving data. Figure 7 shows the time taken to store and retrieve 100MB of data using different encoding parameters. The experiment shows that (i) the use of error correcting codes instead of pure replication increases the performance in terms of both the storage time and overhead, and that (ii) PIB is able to transfer at a rate of 1-2 MBps, which is sufficient for a backup system. At this rate, it would take our non-optimized PIB prototype about 20 minutes to back up a gigabyte of data to 21 nodes so that the data can be recovered despite the failure of 7 nodes. The performance difference between the store and retrieve operations

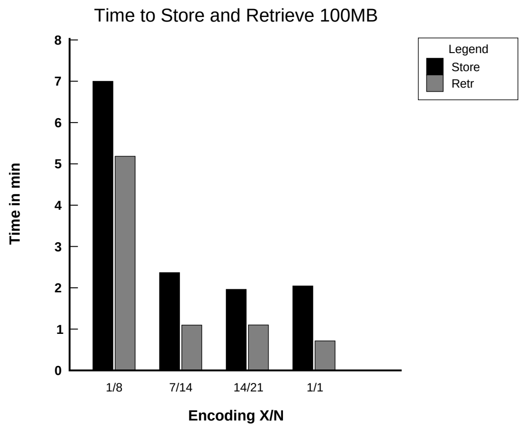

Figure 7: Access time for 100 MB with different encodings

comes from an inefficiency in the store code in our prototype: the data being stored is unnecessarily written to disk several times before being sent on the network. We intend to correct this problem in a later version.

9 Related Work

Our work brings together Byzantine fault-tolerance and game theory.

Byzantine agreement [34] and Byzantine fault tolerant state machine replication have been studied in both theoretical and practical settings [11, 14, 30, 52, 58]. Our work is clearly indebted to recent research [4, 15, 39, 56, 64] that has shown how BFT can be practical in distributed systems that fall under a single administrative domain—indeed, Castro and Liskov's BFT state machine [15] is the starting point for our IC-BFT state machine. Our work addresses the new challenges that arise in MAD distributed systems, where the BFT safety requirement that fewer than one third of the nodes deviate from the assigned protocol can be easily violated.

Game theory [29] has a long history in the economics literature [6, 43, 32] and has recently become of general interest in computer science [5, 26, 27, 24, 47, 51, 63]. Protocol and system designers have used game theoretic concepts to model behaviors in a variety of settings including routing [25, 61, 60], multicast [45], and wireless network [63]. Common across these works is the assumption that *all* nodes behave rationally—the presence of a single Byzantine node may lead to a violation of the guarantees that these system intend to provide.

Shneidman et al. [61, 62] recognize the need for a model that includes both Byzantine and rational nodes, but their protocols address only the latter. Nielson et al [46] identify different rational attacks and discuss high-level strategies that can be used to address them.

To our knowledge, Eliaz's notion of k Fault-Tolerant Nash Equilibrium (k-FTNE) [23] is the only previous attempt to formally model games that include both rational and Byzantine agents. Eliaz's model is more general than the one we assume—for our Nash equilibrium, a rational node that is considering deviating from the protocol assumes that Byzantine nodes will perform the actions that are most damaging to it; to achieve equilibrium, Eliaz requires that rational players have no incentive to deviate *regardless* of the actions of the Byzantine players. Eliaz's problem domain differs from ours: it targets auctions with human participants and provides no example of how k-FTNE may be used to build cooperative computer services with Byzantine and rational nodes.

Rigorous design for incentive compatible systems has largely been restricted to theoretical work. Practical systems for tolerating rational behavior [20, 17] commonly rely on informal reasoning. Bittorrent [17] uses a tit-for-tat strategy to build a Pareto efficient mechanism for content distribution. However Shneidman demonstrates that the algorithm is not actually incentive compatible [62]. Other systems use audits [44] or witnesses [42] to discourage rational nodes from deviating from their assigned task, but they do not specify an incentive compatible or Byzantine tolerant mechanism for implementing audits or witnessing. Using BART state machines to implement a reliable witness from self-interested or Byzantine nodes is one of the contributions of this paper.

Cooperative storage and backup systems have been studied extensively in the literature [4, 9, 19, 20, 35, 53, 57]. The backup systems proposed in [9, 19] rely on the assumption that all non-faulty nodes behave correctly. Samsara [20] and Lillibridge et al. [35] introduce a set of incentives to influence rational nodes, but they do not bound the damage Byzantine nodes can inflict to stored data. An additional limitation of Samsara is its reliance on random spot-checks to verify that a node is storing data it has promised under which if a node o fails such a spot check, the system probabilistically deletes o's data. This increases the likelihood that a node will be unable to retrieve its files precisely when they are needed most. Conversely, we guarantee that a node can recover its data for a period of time, even if it suffers a total disk failure. This property seems useful in a backup system.

10 Conclusions

This paper describes a general approach to constructing cooperative services spanning MADs in the context of a cooperative backup system. The three primary contributions of this paper are (1) the introduction of the BAR (Byzantine, Altruistic, and Rational) model, (2) a general architecture for building services in the BAR model, and (3) an application of this general architecture to build PIB, the first cooperative backup service to tolerate both Byzantine users and an unbounded number of rational users.

References

[1] Extended technical report. http://www.geocities.com/byzantine_rational/tr.pdf.

[2] The game of chicken. http://www.gametheory.net/Dictionary/Games/GameofChicken.html.

[3] E. Adar and B. Huberman. Free riding on gnutella. Technical report, Xerox PARC, Aug. 2000.

[4] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *5th OSDI*, Dec 2002.

[5] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. Selfish behavior and stability of the internet: a game-theoretic analysis of tcp. In *Proc. SIGCOMM*, pages 117–130. ACM Press, 2002.

[6] R. J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1(1):67–96, 1974.

[7] R. Axelrod. *The Evolution of Cooperation*. Basic Books, New York, 1984.

[8] R. Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 32–41. Morgan Kaufman, 1987.

[9] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[10] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. 10th SOSP*, pages 79–86. ACM Press, 1985.

[11] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[12] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[13] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM Trans. Comput. Syst.*, pages 18–36, Feb. 1990.

[14] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report 92-15, TR 92-15, Dept. of Computer Science, Hebrew University, 1992.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[16] J. Chase, B. Chun, Y. Fu, S. Schwab, and A. Vahdat. Sharp: An architecture for secure resource peering. In *SOSP*, 2003.

[17] B. Cohen. The bittorrent home page. http://bittorrent.com.

[18] B. Cohen. Incentives build robustness in bittorrent. In *Proc. 2nd IPTPS*, 2003.

[19] L. Cox and B. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th OSDI*, Dec 2002.

[20] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proc. 19th SOSP*, pages 120–132, 2003.

[21] A. K. Dixit and S. Skeath. *Games of Strategy*. W. W. Norton & Company, 1999.

[22] J. R. Douceur. The Sybil attack. In *Proc. 1st IPTPS*, pages 251–260. Springer-Verlag, 2002.

[23] K. Eliaz. Fault tolerant implementation. *Review of Economic Studies*, 69:589–610, Aug 2002.

[24] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *J. Comput. Syst. Sci.*, 63(1):21–41, 2001.

[25] J. Feigenbaum, R. Sami, and S. Shenker. Mechanism design for policy routing. In *Proc. 23rd PODC*, pages 11–20. ACM Press, 2004.

[26] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. 6th DIALM*, pages 1–13. ACM Press, New York, 2002.

[27] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Free-riding and whitewashing in peer-to-peer systems. In *Proc. PINS*, pages 228–236. ACM Press, 2004.

[28] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[29] D. Fudenberg and J. Tirole. *Game theory*. MIT Press, Aug 1991.

[30] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for $n{>}3t$ Processors in $t+1$ Rounds. *SIAM J. of Computing*, 27(1), 1998.

[31] K. P. Gummadi, R. J. Dunn, S. Saroio, S. D. Gribbl, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. 19th SOSP*, 2003.

[32] J. Harsanyi. A general theory of rational behavior in game situations. *Econometrica*, 34(3):613–634, Jul. 1966.

[33] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[34] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[35] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX ATC*, june 2003.

[36] M. Loney. Charity gives 40,000 pcs a fresh start. *CNET News.com*, February 4 2005. http://news.com.com/Charity+gives+403421.html.

[37] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *NSDI*, May 2005.

[38] G. J. Mailath. Do people play Nash equilibrium? lessons from evolutionary game theory. *Journal of Economic Literature, 36 (September 1998), 1347-1374*, 1998.

[39] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing 11/4*, pages 203–213, 1998.

[40] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th SRDS*, Oct 1998.

[41] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. 19th SOSP*, pages 44–59. ACM Press, 2003.

[42] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3):273–305, 2000.

[43] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, Sept 1951.

[44] T. W. Ngan, D. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. 2nd IPTPS*, 2003.

[45] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *2nd Workshop on Economics of Peer-to-Peer Systems*, 2004.

[46] S. J. Nielson, S. A. Crosby, and D. S. Wallach. A taxonomy of rational attacks. In *Proc. 4th IPTPS*, Feb. 2005.

[47] N. Nisanb and A. Ronenc. Algorithmic mechanism design. *Games and Economic Behavior*, 35:166–196, April 2001.

[48] N. Ntarmos and P. Triantafillou. Aesop: Altruism-endowed self organizing peers. In *Proc. 2nd DBISP2P*, August 2004.

[49] N. I. of Standards and Technology. Secure hash standard. Technical report, U.S. Department of Commerce, August 2002.

[50] Anti-piracy solutions. http://www.overpeer.com/antipiracy.asp, Mar. 2005.

[51] C. Papadimitriou. Algorithms, games, and the internet. In *Proc. 33rd STOC*, pages 749–753. ACM Press, 2001.

[52] M. Reiter. The Rampart toolkit for building high-integrity services. In *Dagstuhl Seminar on Dist. Sys.*, pages 99–110, 1994.

[53] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, 2003.

[54] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.

[55] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, 1997.

[56] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *Proc. 18th SOSP*, pages 15–28. ACM Press, Oct. 2001.

[57] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th SOSP*, pages 188–201. ACM Press, 2001.

[58] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[59] "seti@home". http://setiathome.ssl.berkeley.edu/.

[60] J. Shneidman and D. Parkes. Rationality and self-interest in peer to peer networks. In *Proc. 2nd IPTPS*, 2003.

[61] J. Shneidman and D. C. Parkes. Specification faithfulness in networks

with rational nodes. In *Proc. 23rd PODC*, pages 88–97. ACM Press, 2004.

[62] J. Shneidman, D. C. Parkes, and L. Massoulie. Faithfulness in internet algorithms. In *Proc. PINS*, Portland, USA, 2004.

[63] V. Srinivasan, P. Nuggehalli, C.-F. Chiasserini, and R. R. Rao. Cooperation in wireless ad hoc networks. In *INFOCOM*, 2003.

[64] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, pages 253–267. ACM Press, 2003.

```
1    send(message):
2      toSend.enqueue(message)
3
4    expect(predicat):
5      toSend.enqueue(predicate)
6
7    deliver():
8      block until received is not empty
9      return received.dequeue()
10
11   run():
12     while (not badlist[recipient]):
13       x := toSend.top()
14       if (x is a message):
15         sending.enqueue(x)
16         toSend.remove(x)
17
18   onReceive(message):
19     x := toSend.top()
20     if (x is not a predicate) then return
21     if (not x(message)) then return
22     toSend.remove(x)
23     received.enqueue(message)
```

Figure 8: The message queue

```
101  server-run():
102    while (not badlist[recipient]):
103      if sending is empty then
104        send "ping"
105      else
106        send all messages in sending
107      wait for send_delay
108      send_delay := min(2send_delay, 10 min)
109
110  onReceive(msg):
111    remove from sending all messages acknowledged by msg
112    send_delay := 2Γ
113    call higher−level onReceive(msg)
```

Figure 9: Message queue helper functions (server side)

```
201  onReceive(msg):
202    remove from sending all messages acknowledged by msg
203    send all messages in sending
204    if msg ≠ "ping" then call higher−level onReceive(msg)
```

Figure 10: Message queue helper functions (client side)

A Message Queue

A.1 Overview

The message queue implements, in an incentive-compatible manner, a reliable channel with a simple "you don't talk to me, I won't talk to you" policy. The pseudocode implementing this policy is shown in Figure 8.

The message queue is challenging because it is implemented on unreliable links, so nodes must resend messages—which is considered a cost. The implementation must make sure that no node can save cost by unilaterally deviating from the protocol.

A.2 Strawman protocol

Consider a TCP-like protocol in which messages are resent periodically until they are acknowledged. This protocol is reasonnable outside the BAR model, but it is not be incentive compatible: rational nodes can reduce their cost by unilaterally deviating from the protocol.

A rational node r may sometimes resend a message unnecessarily, because the previous send reached the recipient. To skip this cost, rational nodes can rely on the fact that their recipient resends its messages: The recipient's response will indicate whether r's message reached it or not. So r only sends its messages when it receives a message, instead of periodically. Clearly, if both correspondants deviate in this manner then no messages are exchanged.

The problem, at the root, is that some work is redundant: a rational node can shirk work because it knows that the other node's actions are enough to ensure progress.

A.3 Message queue protocol

Our protocol assigns different roles to the ends of the communication. Only one end, the *server*, is responsible for resending messages (the server could be the node with the lower I.P. address, or any other deterministic function). This solves the work shirking problem identified above by making all work necessary.

We assume that nodes believe that messages sent to non-Byzantine nodes either reach their destination by time Γ (e.g. the TCP timeout), or never do.

This assumption ensures that the server does not gain by waiting more than required in between sends, in the hope to receive a delayed answer from the other node.

Figure 8 presents the pseudocode for the message queue. The resending protocol is shown in Figures 9 (server) and 10 (client). As shown in lines 12 and 102, the message queue stops sending messages to a node b if it believes that b is Byzantine. One thing that is not shown explicitely in the pseudocode is the connection between sends and expects (lines 111 and 202): filling an *expect* bubble accounts as acknowledging some messages, but not necessarily the last message sent. The protocol itself must indicate which expects acknowledge which sends.

The protocol uses a $badlist$ array to indicate nodes that are believed to be Byzantine: there is nothing to be gained by sending to these nodes and therefore the message queue should not send to them (Lemma 10). There is no step in the message queue itself that puts nodes in the badlist, these come from higher levels of the protocol.

B Terminating Reliable Broadcast

The protocol is described in Section 5.1, and the pseudocode is in Figures 11 and 12. In terminating reliable broadcast, the sender goes through a three-phase commit to get all nodes to decide on its value. If nodes time out waiting for the last message for the sender they elect a new leader; the leader then continues where the sender left off. In our pseudocode the sender is process 0, although in fact the sender changes between instances of TRB.

The node that is the sender instantiates two threads, one that runs the *sender* function and the other that runs the *senderListen* function. Other nodes instantiate n threads for n turns. Each of these threads runs the *leader* function if that node is leader for that turn, or *nonleader* instead. Turn 0 is then started, which allows the code to proceed beyond line 47.

```
1    // sender protocol for processor i (i=0)
2    sender($i$):
3      updateBadlist($badlist$)
4      start senderListen($i$) in parallel
5      $prop$ := ($proposal$,$badlist$,$untimely$)
6      $nv$ := ($prop$, $i$)
7      $s$ := hash($nv$)
8      send (agree, $t$, $nv$, $\perp$) to all others
9      expect $\langle$agree-ack, $t$, $s\rangle_j$ in response
10     wait for a quorum $\vec{a}$ of answers
11     send (write, 0, $nv$, $\vec{a}$) to all others
12     expect $b_j = \langle$write-ack, $t$, $max\_pol\rangle_j$ in response
13     accumulate responses into the set $R$ until:
14       there exists $\vec{b} \subseteq R$ s.t. not failed($\vec{b}$, $t$),
15       or $|R| = n - f - 1$ (then pick any quorum in $R$ for $\vec{b}$)
16     send (show-quorum, $t$, $\vec{b}$) to all others
17     if not failed($\vec{b}$, 0) then decide($nv$, $\vec{a}$, $\vec{b}$)
18
19   // sender thread that listens to all turns>0
20   senderListen($i$):
21     expect (report-decision, $t$, $nv'$, $\vec{a}'$, $\vec{b}'$) from all others
22     wait for one
23     decide($nv'$, $\vec{a}'$, $\vec{b}'$)
24
25   // leader protocol for processor i on turn t (t>0, i=leaderForTurn(t))
26   leader($t$, $i$):
27     $pol_t$ := waitUntilElected($t$, $i$)
28     if $pol_t$ == "done" return
29     $\vec{r}$ := readOldValue($t$, $i$, $pol_t$)
30     $nv$ := latest($\vec{r}$,t)
31     $s$ := hash($nv$)
32     send (agree, $t$, $nv$, $\vec{r}$) to all except sender
33     expect $\langle$agree-ack, $t$, $s\rangle_j$ from all j
34     wait for a quorum $\vec{a}$ of answers
35     send (write, $t$, $\vec{a}$) to all except sender
36     expect $b_j = \langle$write-ack, $t$, $max\_pol\rangle_j$ from all j
37     accumulate responses into the set $R$ until:
38       there exists $\vec{b} \subseteq R$ s.t. not failed($\vec{b}$, $t$),
39       or $|R| = n - f - 1$ (then pick any quorum in $R$ for $\vec{b}$)
40     send (show-quorum, $t$, $\vec{b}$) to all except sender
41     if failed($\vec{b}$,t) then start next turn
42     else decide(nv,$\vec{a}$, $\vec{b}$)
43
44   // non—leader protocol for proc. i on turn t
45   nonleader($t$, $i$):
46     $l$ := leaderForTurn($t$)
47     wait until started
48     start timer $t$: if it fires, then start the next turn
49       time—out for $t$ is chooseTimeout($t$, $i$)
50     if (turn>0) then
51       send $\langle$set-turn, $(t + 1)\rangle_i$ to $l$
52       $x$ := giveOldValue($t$, $i$)
53       if $x$ == "done" then stop timer $t$; return
54     expect (agree, $t$, $nv$, $\vec{r}$) from l ; wait for it
55     send  $\langle$agree-ack, $t$, $hash(nv)\rangle_i$ to l
56     expect (write, $t$, $\vec{a}$) from l ; wait for it
57     // $\vec{a}$ is a vector of agree—ack for turn t with the same s we got
58     // $s ==$ hash($nv$)
59     // $\vec{r}$ contains a quorum of read—ack for turn t
60     // such that either nv is the latest value in $\vec{r}$,
61     // or $\vec{r}$ contains only $\perp$ values
62     if $t > m\_val\_t$ then
63        ($m\_val$, $m\_val\_t$, $\vec{m\_a}$) := ($nv$, $t$, $\vec{a}$)
64     send $\langle$write-ack, $t$, $max\_pol\rangle_i$ to l
65     expect (show-quorum, $t$, $\vec{b}$) from l; wait for it
66     // $\vec{b}$ contains a quorum of write—ack for turn t
67     stop timer $t$
68     if timer $t$ fired more than $avg\_latency + window$ ago,
69       then $untimely[l]$ := $untimely[l] + 1$
70     if failed($\vec{b}$, $t$) then start next turn
71     else decide($m\_val$, $\vec{a}$, $\vec{b}$)
```

Figure 11: IC-BFT TRB, high level functions

```
101  waitUntilElected($t$, $i$):
102     expect $\langle$set-turn, $t\rangle_j$ from all other nodes $j$
103     wait until started
104     start timer $t$: if it fires, then start the next turn
105       time—out for $t$ is chooseTimeout($t$, $i$)
106    $t_r$ := now() + $avg\_latency$
107     for every node $j$ from which we do not receive the expected
108     time—out message between $t_r - window$ and $t_r + window$:
109         $untimely[j]$ := $untimely[j] + 1$
110     wait until:
111       we receive a quorum $pol_t$ of these messages:
112         stop timer $t$
113         return $pol_t$,
114       or $i$ calls decide($nv$, $\vec{a}$, $\vec{b}$):
115         send (show-decision, $t$, $nv$, $\vec{a}$, $\vec{b}$) to all
116         stop timer $t$
117         return "done"
118
119  readOldValue($t$, $i$, $pol_t$):
120    send (read, $t$, $pol_t$) to all except the sender
121    expect $r_j = \langle$read-ack, $t$, $val$, $val\_t$, $\vec{a}\rangle_j$
122  from all j other than the sender
123    wait for a quorum $\vec{r}$ of answers
124    return $\vec{r}$
125
126  giveOldValue($t$, $i$):
127    expect (read, $t$, $pol_t$) or (show-decision, $t$, $nv$, $\vec{a}$, $\vec{b}$) from $l$
128    wait to receive it
129    if it's the latter then
130      $untimely[l]$ := $untimely[l] + decisionfee$
131      decide($nv$, $\vec{a}$, $\vec{b}$)
132      return "done"
133    if $i$ is the sender then return "skipping"
134    $max\_pol := \max(max\_pol, pol_t)$
135    send $\langle$read-ack, $t$, $m\_val$, $m\_val\_t$, $\vec{m\_a}\rangle_i$ to l
136    return "go on"
137
138  failed($\vec{b}$, $t$):
139    return true if $\vec{b}$ contains at least one
140    $\langle$write-ack, $t$, $mp\rangle_j$ with $mp.t \neq t$,
141    or false if all $q$ have $mp.t == t$
142
143  latest($\vec{r}$, $t$):
144    If all $r_j \in \vec{r}$ have $r\_j.val\_t == \perp$ then
145      return ($\perp$,$t$)
146    else
147      return $pad(r_j.val)$ for $r_j \in \vec{r}$ with the largest $r_j.val\_t$
148
149  updateBadlist($badlist$):
150    // this function is different for the "quasi—synchronous" version
151    if there is some node $j$ that sent us a malformed message then
152    $badlist[j]$ := true
153
154  leaderForTurn($t$):
155    if ($t$==0) return 0
156    return ($t$ mod ($n$—1)) + 1
157
158  computePenance($untimely$):
159    return a buffer of size $untimely * latefee$
160
161  chooseTimeout($t$, $i$):
162    if $i$ is leader next turn, then return $timeout * 2^t$
163    else pick the value that maximizes the likelihood that the
164    time—out message will reach the next leader in the center of its
165    window (in the absence of other information, pick $timeout * 2^t$).
166
167  decide(($v$, $t$), $\vec{a}$, $\vec{b}$):
168    $myPenance$ := computePenance($v.untimely[i]$)
169    for every node $x \neq 0$:
170      send ($myPenance$) to $x$
171      $penance$ := computePenance($v.untimely[x]$)
172      expect ($penance$) from $x$
173    if node $i$ was sender then
174      for all j: $untimely[j]$ := $untimely[j] - v.untimely[j]$
175    if some node $x$ is in $f + 1$ nodes' $badlist$ then $badlist[x]$ := true
176    if this process is not the sender and has not sent report—decision
177    to the sender yet then send (report-decision, $t$, $nv$, $\vec{a}$, $\vec{b}$) to the sender
178    let the application know that we decided on value $v$
179
180  // should "node" be prevented from participating in the next instance?
181  hasBubble($i$, $node$):
182    if decide(...) was not yet called then return true
183    if decide(($v$, $t$), $\vec{a}$, $\vec{b}$) was called then:
184      if $i$ has an outstanding expect for $node$ in any of the turns 0 through
185         $t$ (included), then return true
186      if there exists some turn in which
187         $i$ has sent a message to $node$, and
188         $i$ has an outstanding expect from $node$
189      then return true
190      else return false
```

Figure 12: IC-BFT TRB, low level functions

The send and expect calls in the pseudocode all refer to sending through a message queue. Each turn has one message queue per other node. Within one instance of TRB they are linked together so that if there is a bubble against node r in turn t, then sends on turn $t + 1$'s message queue to r are delayed until the bubble is filled.

Turns are created as needed, to make sure that no message will be discarded because the turn that would have inserted the expect for that message is missing. To prevent this situation it is enough to make sure that whenever some turn t finishes (because the last instruction of that turn's *leader* or *nonleader* method returns), turns $t + 1$ through $t + n$ are created. Turns that are created place expects on their message queue, and then wait to be started: this is done explicitely in the time outs (lines 48 and 104) or when a leader failed to gather a successful quorum for its write (lines 41 and 70). A write is successful in turn t if it includes a vector $\vec{r}$ such that $failed(\vec{r}, t)$ (line 138) returns false.

The asynchronous protocol uses quorums of size $n-f-1$, and quorums cannot include the protocol designated as sender for this instance of TRB.

C TRB Correctness

C.1 Proof technique

To prove that a protocol is IC-BFT for a given model of rational nodes' utility and beliefs, one must first prove that the protocol provides the desired safety and liveness properties under the assumption that all non-Byzantine nodes follow the protocol. Second, one must prove that it is in the best interest of all rational nodes to follow the protocol.

We start by proving correctness assuming that all non-Byzantine nodes follow the protocol.

C.2 Correctness assuming incentives

Here we assume that all non-Byzantine nodes follow the protocol.

Definition 1. *A value v is said to be* proposed *in turn t, if a leader sends a valid* `agree` *message in turn t with value v.*

Definition 2. *A value v is said to be* chosen *in turn t if there is a quorum Q such that all non-Byzantine nodes in Q answered the* `write` *message for v in turn t before receiving the* `read` *message from any later turn t'.*

Lemma 6. *If two non-Byzantine nodes satisfy the expect for a write message in turn t with values v and v' respectively, then $v == v'$.*

Proof. Each write message has the format $(\text{write}, t, nv, \vec{a}, \vec{r})$, where $\vec{a}$ consists of a quorum of answers of format $\langle\text{agree-ack}, t, s\rangle_j$, and s is the hash of the value v.

Since any two quorums intersect in a non-Byzantine node, and such a node sends only one *agree-ack* message in a particular turn, it follows that the same *agree-ack* message is used in the $\vec{a}$ value for the write of v and v'.

This requires that the hash for v and v' be the same. Under the secure hash assumption, it follows that $v == v'$. □

Lemma 7. *If a value has been chosen in turn t, then no other value can be proposed in turn t', $t' > t$.*

Proof. By contradiction. Let v be the value chosen in turn t, and $t' > t$ be the earliest turn after t in which some node proposed a different value v'.

If v has been chosen in turn t it follows that all non-Byzantine nodes in a quorum Q have received a write message for v, but have not received a read message from any later turn.

For a value to be proposed, it needs to contain agree-acks from a quorum of nodes. Non-Byzantine nodes will respond with an agree-ack only if the agree message is well formed, i.e. the value v' that is proposed is consistent with the vector $\vec{r}$ that has been sent (in particular, $latest(\vec{r}, t') == v'$ and every element of $\vec{r}$ is a valid message).

Vector $\vec{r}$ contains signed values from a quorum of nodes Q' and cannot be modified. Since Q and Q' intersect in at least one non-Byzantine node, and the non-Byzantine node will send the value v, it follows that there is at least one entry in $\vec{r}$ stating that value v was written in turn t.

Since entries in $\vec{r}$ include $\vec{a}$ in addition to the value and turn, all non-$\perp$ values in $\vec{r}$, even if they are from a Byzantine node, must have been proposed earlier.

Moreover, t' is the earliest turn after t to propose a value other than v. So there cannot be any proposed value $v" \neq v$ with a turn number $t" > t$ in $\vec{r}$ received in turn t'.

Value v from turn t is therefore the value in $\vec{r}$ with the highest turn number, and $latest(\vec{r}, t')$ will return v. Therefore the leader in turn t' must propose value v. □

Lemma 8. *A value v is chosen in turn t only if v was proposed in turn t.*

Proof. A non-Byzantine node accepts a `write` message only after it accepted the corresponding `agree` message. Since all quorums contain at least one non-Byzantine node, it follows that for v to be chosen at turn t it must have been proposed at turn t. □

Theorem 3 (Safety). *If some non-Byzantine node decides on a value v in turn t then no non-Byzantine node will decide on a value other than v.*

Proof. A node decides on a value v only after either seeing evidence that the value was chosen, either through a show-quorum message (lines 42 or 73), a show-decision message (line 129) or report-decision (line 22). The previous two lemmas indicate that at most one value may ever be chosen. □

Theorem 4 (Liveness). *Eventually every non-Byzantine node decides.*

Proof. Since the time-out delays increase exponentially, during the synchronous period there will be some turn after which every leader is guaranteed to have enough time to complete without being interrupted by another leader election. Consider the first such leader who is non-Byzantine. That leader will be able to write a consensus value without interference, and it will have gathered a quorum of acknowledgments ($\vec{b}$) that show that no other leader was elected before the end of the write. That information allows nodes to decide. Since the leader is non-Byzantine he sends it to all and all non-Byzantine nodes decide, and report to the sender if necessary. □

Theorem 5. *The protocol satisfies the conditions for TRB.*

Proof.
- Termination is guaranteed by Theorem 4.
- Agreement follows from Theorem 3 and Theorem 4

- Integrity is assured because a leader cannot propose any arbitrary value. The expect in line 56 is satisfied only if the proposed value has been written earlier, or is $\perp$. The fact that a leader cannot propose an arbitrary value hence follows by induction on the turn number t.
- In a period of synchrony, if the sender is non-Byzantine then no non-Byzantine node will time out on the sender because the time out values are larger than the known guaranteed delivery time Δ. It follows that the sender will be able to complete the turn and get all non-Byzantine nodes to deliver the message.

□

C.3 Equilibrium and incentive compatibility

Background We now show that the protocol represents an equilibrium point. More specifically, it represents a *Nash equilibrium*. We start by introducing this concept and relating it to our domain.

Nash Equilibira are a game theory concept. Game theory studies "games" among rational players. In one-shot games, for example, every player i (we shall call them nodes from here on) simultaneously picks some *strategy* σ_i. The rules of the game determine a *utility* u for each node, as a function of its strategy and the strategy of the other $n-1$ nodes. The utility for node i can be written as the function $u_i(\sigma_0, \ldots, \sigma_{n-1})$, which we abbreviate $u_i(\sigma_i, \sigma_{-i})$.

The Nash equilibrium is defined as follows [29]:

$$u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(s_i, \sigma_{-i}^*) \text{ for all } s_i \in S_i$$

Where σ_i^* is the strategy proposed to node i, and S_i is the set of all deterministic strategies i can choose from.

To link these concepts to our domain, we observe that the strategy represents which actions the node will take in response to events it can observe. In other words, the strategy is the protocol that the node follows. A game-theoretic "game" is determined by a function that takes every node's strategy as input and outputs a resulting utility for each node. In our case, the input is which protocol each node follows and node's utilities are determined by the costs and benefits that the node experiences from running the protocol. We define the cost precisely later in this section. The two differences between our setting and the traditional phrasing of the Nash equilibrium is that, first, the utility can be influenced by network delays so that rational nodes must reason based on their expected utility. Second, Byzantine nodes may deviate arbitrarily from the protocol.

In a way similar to how an assignment of strategies to nodes can be said to be a Nash equilibrium for a given game if no player can improve its utility by unilaterally deviating from the assigned strategy, we say that a given protocol is a Nash equilibrium if no rational node can improve its expected utility by unilaterally deviating from the assigned protocol.

Proof technique To prove that the protocol is a Nash equilibrium, we show that it is in every node's best interest not to deviate from the proposed protocol under the assumption that all other non-Byzantine nodes follow the protocol.

Showing that something is in the best interest of a rational node is dependent on what the node considers in its interest, but also of the node's beliefs and knowledge. For example, a node that knows that a given node x is Byzantine will see no incentive to send messages to x, whereas one that does not know who is Byzantine must instead consider the expected utility of sending a message to x.

A rational node r evaluates its utility u for a strategy σ by computing its worst-case expected outcome. The worst case is computed over the choices of which nodes are Byzantine, and what Byzantine nodes do. The expectation is over network performance. The outcome then includes the costs: sending and receiving messages and computing signatures, and the benefits are: having their own proposal accepted. Node r also includes future effects of its actions, for example whether some node(s) now consider r to be Byzantine (by setting the corresponding entry in $badlist$ to $true$) or whether nodes will ignore r in the future (because the $hasBubble$ function returns $true$). A change that would prevent r from participating in future instances of TRB is considered to have infinite cost since it robs r from an infinite number of beneficial instances of TRB.

Our assumptions are presented in the System model, Section 3. In short, we assume that rational nodes gain a long-term benefit in participating, we assume that they consider the worst-case outcome of their actions, and we assume that if they observe that the protocol is a Nash equilibrium then they will follow the protocol.

The simplest deviations are those that do not modify the messages that a node sends. In our state machine protocol, no such deviation increases the utility. We must then examine every message that the node sends and show that there is no incentive to either (i) not send the message (ii) send the message with different contents, or (iii) send the message earlier or later than required. Also, we must show that nodes have no incentive to (iv) send any additional message.

Our protocol also imposes two requirements that must be met in an implementation: (a) The penance is larger than the benefit of sending a time-out message late, and (b) The $\perp$ answer (in read-ack) is at least as large as the largest allowed proposal value. (that size does not need to be constant, it may grow as the sender fails to propose values)

Theorem 6 (Incentive Compatibility). *No node has any unilateral incentive to deviate from the protocol.*

In order to show that no deviation is beneficial, we systematically explore all deviations. Table 1 maps each deviation to the lemma that shows that it is not beneficial. The concern of nodes sending additional messages is covered by Lemmas 9 and 10.

	not send	send different	diff. time
set-turn	Lemma 11	Lemma 16	Lemma 14
read	Lemma 11	Lemma 17	Lemma 26
read-ack	Lemma 11	Lemma 18	Lemma 30
agree	Lemma 11	Lemma 21	Lemma 27
agree-ack	Lemma 11	Lemma 19	Lemma 30
write	Lemma 11	Lemma 22	Lemma 28
write-ack	Lemma 11	Lemma 20	Lemma 30
show-quorum	Lemma 11	Lemma 23	Lemma 29
show-decision	Lemma 11	Lemma 24	Lemma 31
untimely	Lemma 11	Lemma 15	Lemma 15

Table 1: Map of deviation to lemma

Lemma 9. *Rational nodes only send a message m to node j if j expects that message.*

Proof. The queue protocol discards messages that are not expected. Therefore no rational node i would send an unexpected message to a non-Byzantine node because it has no benefit, but some costs (cost of sending the message, plus any signature in the message). Sending an unexpected messages to Byzantine nodes cannot improve their worst-case behavior (if anything, it may help them drive the system to an even less pleasant state). Therefore, no rational node sends an unexpected message to anyone, Byzantine or not. $\square$

Lemma 10. *Once a rational node i knows that some other node j is Byzantine, i will not send any further message to j.*

Proof. If j is known to be Byzantine (for example because it was observed deviating from an incentive-compatible protocol), then sending messages to it does not affect the worst-case outcome. In particular, node j can always opt to ignore any message from i. Therefore, there is nothing to be gained from the expense of sending messages to j. $\square$

Lemma 10 is a natural consequence from the fact that nodes are rational and that they believe that some nodes may be Byzantine. Naturally, in the worst case Byzantine nodes will not do something so foolish as letting themselves be identified.

Lemma 11. *If a rational node r knows that not sending some expected message m to non-Byzantine node s would cause the $hasBubble$ function in s to return $true$, then r has incentive to send the message.*

Proof. If $hasBubble(s, r)$ returns $true$ (indicating that s believes that r has not fulfilled all its obligations toward s), then s will not answer r's messages in futures instances of TRB. In the worst case (for r), all f Byzantine nodes will ignore r. In that case, when r is sender in a later instance i, it will not be able to gather the required $n - f - 1$ answers to its $agree$ message (since $f + 2$ nodes will not be included: the sender itself, s, and the f faulty nodes). As a result $hasBubble(x, r)$ will be true for all non-Byzantine nodes in instance i. From then on, node r will not be able to send its proposals to anyone: it is effectively excluded from the state machine. Node r would forgo participation in an infinite number of future beneficial instances of TRB: no finite benefit from not sending the message m may be worth this cost. Node r will therefore make sure to send all expected messages whose absence would cause $hasBubble$ to return true. $\square$

Lemma 12. *No rational node r (r is not the sender) can ensure with certainty that $\perp$ will be delivered in a given instance of TRB.*

Proof. Nodes can influence the delivered value through their actions. However, if Byzantine nodes were to follow the protocol, then in a period of synchrony the sender will be able to communicate with a quorum of nodes and get its value delivered regardless of the actions of r (in particular if r does not send any message).

Therefore, rational node r cannot ensure with certainty that value $\perp$ will be delivered as the result of r's actions. $\square$

Lemma 13. *Rational nodes other than the sender have to do the same amount of total work if in a given instance of TRB the decision is $\perp$ instead of the sender's value.*

Proof. If a sender's proposal is not accepted, then the sender will propose it again next time. Lemma 12 indicates that if a sender tries forever, the proposed value will be eventually delivered. The total amount of work, therefore, is the same (of course, the utility may be different because a different number of messages may be exchanged). $\square$

Lemma 14. *There is nothing to be gained by sending the timeout message earlier or later than the protocol calls for.*

Proof. The protocol requires non-leader nodes to send the timeout message for turn t ("set-turn(t)") as soon as they believe that turn t started: either because timer $t - 1$ fired (i.e. a time-out) or because show-quorum for turn $t - 1$ failed. The leader in turn t never sends "set-turn(t)", and the sender never sends set-turn messages either.

Starting the next turn earlier (or later, as the case may be) may influence the outcome of TRB (toward either $\perp$ or the sender's value), but that has no effect on the amount of work that node r has to perform (Lemma 13).

All the messages for the current turn must be sent, so there is no other benefit from starting a turn earlier.

Delaying the start of turn t may save a node some effort, because it is possible that the delay allows turn $t - 1$ to receive (or compose, if the node is the leader) a successful show-quorum message, so that there is no need to send the set-turn message anymore.

However, the recipient of set-turn expects that message at a given time (and follows the protocol by hypothesis), so if the node sends "set-turn" late it increases its chance of missing the window, thus raising the expected cost through the penance mechanism. By requirement (a), this expected cost is larger than the expected benefit from potentially not having to send "set-turn" and going through an extra turn (potentially with value $\perp$). □

Lemma 15. *Rational nodes have no incentive to omit or modify the untimely message.*

Proof. The untimely message (computed in lines 109 and 130, sent in line 5) is intended to inflict additional cost onto nodes that are believed to be untimely. If a rational node r omits this message, then its agree message is malformed (see Lemma 21). Modifying the contents of the message does not change its size, and the untimely message sent by node r does not impact node r (it impacts everyone else, as lines 169–172 show). Therefore, node r has no incentive to modify the untimely message. □

Lemma 16. *There is nothing to be gained by sending a set-turn message with the wrong contents.*

Proof. Since set-turn only contains a turn number and a signature, wrong contents would be equivalent to either sending twice to the message queue or sending a malformed message (Lemma 9), or sending set-turn early (Lemma 14). □

Lemma 17. *There is no incentive to lie in the read request.*

Proof. The format of the read request is entirely determined (line 127), the only freedom being in the specific choice of which quorum of entries in the POL are filled. Since all POL entries have the same size, all choices result in a POL of the same total size and hence the same cost. Since using a different valid POL has no impact on the protocol and does not reduce cost, there is no reason why a rational node would choose one quorum over another. □

Lemma 18. *There is no incentive to lie in the response to a read message.*

Proof. There are only two different possible answers to a read message: either the sender's value, or $\perp$. Since the sender's value is signed and nodes cannot forge signatures, the only possible lie is to answer $\perp$ when, in fact, one has received a value.

This lie increases the likelihood of $\perp$ being delivered instead of the sender's value, which has two consequences. First, it changes the amount of work that must be done in this turn. However, as we argue in Lemma 13, nodes other than the sender expect to have to do the same amount of work even if they try to increase the likelihood of $\perp$ being delivered. Second, it increases the size of the messages that must be sent because the $\perp$ answer has the same size (in bytes) as the longest allowed proposal (requirement b). Therefore there is no benefit to lying in response to a read message. □

Lemma 19. *There is no incentive to lie in the response to an agree message.*

Proof. The answer to agree is entirely determined by the agree message itself, so any deviation would be equivalent to not sending a message that the leader expects. Lemma 11 shows that there is no incentive to do that. □

Lemma 20. *There is no incentive for a rational node r to lie in the response to a write message.*

Proof. The only choice in the response is max_pol, the latest leader that the node has received a message from. Since these messages are signed, the only possible lie for a rational node is to reply with some POL it has received.

Since the size of the POL is constant, the only benefit of replying with an older POL is to influence the protocol. As we argued before (Lemma 13), only the sender has a stake in influencing the decision and the sender does not receive write messages.

Remains the possibility that answering with a different POL will influence the number of turns that the protocol takes to complete (that's a cost). Answering with the requester's POL instead of a later one (if we received one) means that there is some chance that the requester now thinks its proposal succeeded when, in fact, it failed. The potential benefit would be that if the requester succeeds, then there is no need to send a time-out message to the next leader. However, the fact that node r heard from the later leader means that it has already sent a time-out message to the later leader, therefore there is no incentive to lie in the response to the write message. □

Lemma 21. *There is no incentive to send incorrect data in the agree message.*

Proof. The agree message (sent in lines 8 and 32) include the turn number, proposal, and $\vec{r}$. Changing the turn number would be equivalent to not sending the agree message, which would result in $hasBubble$ returning $true$ (Lemma 11). The protocol does not restrict which proposal the sender can send, other than the condition that it must include the untimely vector. Lemma 15 argues that there is no incentive to send an incorrect untimely vector. Leaders that are not the sender have no choice in the proposal, as it is entirely determined by the contents of $\vec{r}$ and the $latest$ function. The vector $\vec{r}$ itself contains signed answers from other nodes, so it cannot be tampered with, other than choosing which answers to include in $\vec{r}$. These deviations are covered by Lemma 27. □

Lemma 22. *There is no incentive for r to send incorrect data in the write message.*

Proof. Sending a value that does not match the agreed-up hash would cause everyone to consider r Byzantine. The vectors $\vec{r}$ and $\vec{a}$ are both constant-size and cannot influence the protocol other than marking r as Byzantine, so there is no incentive to change them either. □

Lemma 23. *There is no incentive for r to send incorrect data in the show-quorum message.*

Proof. That message contains information signed by others, so it cannot be faked by r. ☐

Lemma 24. *There is no incentive for r to send incorrect data in the show-decision (or report-decision) message.*

Proof. Both messages have the same content. Their size is fixed, and nodes cannot lie about the decided value because they cannot forge signatures. The only deviation would be to use a different quorum for $\vec{r}$, but there is no benefit to that. ☐

Lemma 25. *There is no incentive for a rational leader r to send a message in its leader turn t before the protocol indicates turn t should start.*

Proof. It may prevent the previous leader from succeeding. Leaders have no stake in the outcome, so all that preventing the other from succeeding achieves is potentially cause more set-turn messages to be sent.

The sender cannot start early because the protocol says it should start immediately. ☐

Lemma 26. *There is no incentive for a rational leader r to wait for more than a quorum of time-out messages before starting its leader duty.*

Proof. That would allow the leader to go the show-decision route instead of the normal three phase commit. We use the penance mechanism to balance the costs ($decisionfee$, line 130). ☐

Lemma 27. *There is no incentive for a rational leader r to wait for more than a quorum of answers to its read message.*

Proof. Waiting for more answers may allow the leader to go from a situation in which it must propose $\perp$ (because none of the answers so far have seen the sender's value) to one in which it can propose the sender's value (because one of the answers includes it, cf. the $latest$ function in line 143)—or the other way around.

These two situations do not modify the expected number of turns for this instance of consensus. They are also identical in term of message size, because the leader must pad the proposal to maximum size, the same size as $\perp$. The difference between the two is which value is decided in the end, which may change how much work the leader must go through in this instance. However, as we argue in Lemma 13, this does not change the total amount of work. There is therefore no incentive for r to deviate from the protocol by waiting for more answers. ☐

Lemma 28. *There is no incentive for a rational leader r to wait for more than a quorum of answers to its agree message.*

Proof. Getting more answers cannot influence the outcome, so there is no incentive to wait for more. ☐

Lemma 29. *There is no incentive to wait for more answers to* `write`.

Proof. The protocol lets you wait for $n - f - 1$ answers, waiting for more may get you stuck. ☐

Lemma 30. *There is no incentive to answer late to either a read, agree or write message.*

Proof. The effect of a late reply to these requests is to potentially slow down the leader (or sender), increasing the risk that this instance of TRB lasts one more turn and potentially influencing the outcome.

Only the sender has a stake in the outcome, and it does not answer to these messages. Remains the possibility of adding a turn, which would cause the rational node to send more messages and therefore increase its cost. Rational nodes therefore have an incentive to respond to these queries immediately. ☐

Lemma 31. *There is no incentive to send the show-decision message late.*

Proof. Once a leader r knows that it must respond with the show-decision message, then further waiting has no impact on its cost: nothing can remove the requirement on r to send that message. The leader therefore has nothing to gain by delaying the answer. ☐

C.4 Enlightening examples

The protocol is Figure 11 distinguishes between the sender and the leader: the sender proposes a value and, if it is not timely, a new leader is elected. This distinction may seem unnecessary, but in fact it is important that the sender not be involved in steps where it may influence whether its value gets decided. This can occur in two places.

First, the sending of the "set-turn" messages. Suppose an execution in which the sender receives a POL from a later leader, and then a write for the value $\perp$, indicating that the new leader did not see any of the messages sent by the sender. The sender may then have an incentive to send its "set-turn" message early to elect a new leader, in the hope that the new leader may see one of the written values and will attempt to write the sender's value instead of $\perp$.

Second, the answer to "read" requests. In the same scenario as described above, if the sender receives a write for $\perp$ by leader 1 and then a read request from leader 2, then the sender would have an incentive to deviate from the protocol and send its own value instead, pretending it hasn't received the message from leader 1.

In order to avoid both scenarios, we allow the sender to try to write its value only once: it cannot be elected leader in later turns, and read messages are not sent to it. Since the read and write quorums must still intersect in at least one correct node and there must be a quorum of correct nodes among all nodes but the sender, it follows that $n \geq 3f + 2$.

D Replicated State Machine

The “hasBubble” function (Figure 12) is used to determine whether a given node should be allowed to participate in the next instance of TRB.

The replicated state machine provides the following guarantee under our liveness assumption that all non-Byzantine nodes get some overall benefit from participating in the state machine.

Theorem 7. *If non-Byzantine node a submits some command c to the state machine then eventually every non-Byzantine node n in the state machine will deliver c.*

Proof. Eventual synchrony guarantees that eventually a gets its turn as sender in the state machine. TRB’s non-triviality condition then guarantees that a will successfully deliver its proposal. Once a is done with earlier submissions it will submit c, which it will deliver. The agreement condition guarantees that all non-Byzantine nodes will deliver c as well. □

E Work Assignment

This section addresses issues related to work assignment and relevant efficiency optimizations. In general, work assignment is used to reduce replication factors associated with running a protocol and to increase communication efficiency and reliability. The work assignment protocol leverages the state machine to replicate the assignment of work to a specific node or set of nodes. The work itself is then performed on the specific nodes. In general, the messages and execution of allocation are orders of magnitude less expensive than the execution of the work itself.

The work assignment protocol proceeds in 5 basic steps: (1) submit request to state machine, (2) state machine delivers request, (3) subset of state machine performs request, (4) result is sent back to all nodes in state machine, (5) all nodes in state machine forward request to the requester and done.

For all proofs in this section, we make the “sufficient benefit” assumption, that is rational node a gains sufficient benefit from membership to outweigh the cost of participating in the system if no more than f nodes deviate from the protocol.

Let w be work instructions, a, b be nodes in state machine A. Let u be the result of performing w. We also assume that all liveness conditions are met.

E.1 Work assignment to individual nodes

Here we address issues related to assigning work to an individual node. Let A be a state machine, a and b be nodes in the state machine, and w be work assigned by b to a. We assume that the result of performing w can only be acquired through actually performing w (i.e. that w is an unforgeable operation). The most relevant of this is the following lemma:

```
1   // protocol for execution of $assign(w, a, b)$ on $p \in A$
2   execute($assign(w, a, b)$):
3     if ($p = a$)
4        $r$ := perform($w$);
5        $\forall c \in A$ c.send($r$);
6
7     start $w.timer$;
8     a.expect $< m : m.reqHash = hash(w) >$;
9     m := a.deliver();
10    cancel $w.timer$;
11    b.send ($witness(m, a)$);
12    if ($p = b$)
13       responses := {};
14       $\forall c \in A$ start$w_c.timer$;
15       $\forall c \in A$
16         c.expect $< m : m = witness(u : u.reqHash = hash(w)) >$;
17         responses := responses $\cup$ c.deliver();
18         cancel $w_c.timer$;
19         if $|$responses$| \geq f + 1$
20            process(responses);
21
22  \\ protocol for handling $w.timer$
23  on timeout($w.timer$):
24    b.send($witness$(‘‘no evidence’’));
25    $badlist[a] := true$;
26
27
28  \\ protocol for handling $w_c : timer$
29  on timeout($w_c : timer$):
30    $badlist[c] := true$;
31
32  \\ protocol for processing responses
33  process(responses):
34    hand responses up to the caller of $b$
35
36
37  \\ protocol for execution of $assign(w, B, b)$ on $p \in A$
38  execute($assign(w, B, b)$):
39    if ($p \in A \cap B$)
40       if ($r$ not already submitted to $B$)
41         $r$ = $B$.submit($w$);
42       else
43         $r$ = $B$.result($w$);
44       $\forall c \in A$ c.send($r$);
45    $\forall a \in A \cap B$
46       start $w_a.timer$;
47       a.expect $< m : m.reqHash = hash(w) >$;
48       m := a.deliver();
49       cancel $w_a.timer$;
50       b.send ($witness(m, a)$);
51    if ($p = b$)
52       responses = {};
53       $\forall c \in A$ start$w_c.timer$;
54       $\forall c \in A$
55         c.expect $< m : m = witness(u : u.reqHash = hash(w)) >$;
56         responses := responses $\cup$ c.deliver();
57         cancel $w_c.timer$;
58         if $|$responses$| \geq f + 1$
59            process(responses);
```

Figure 13: Work assignment protocol

Lemma 32. *If state machine A executes the command* assign(w, a, b) *and a is rational, then a will perform w.*

Proof. Line 8 of Figure 13 introduces an expect to the message queue. If the expect is not filled, then a will be added to the $badlist$ in line 25, resulting in a’s loss of access to the state machine and subesequent loss of benefit. Since w is a non-forgeable operation, a must perform w to have a result which will be accepted by the message queue. □

It is important to note that the exact semantics of “performing w” must be suitably defined by the application. In the context of PIB, the result of performing a store request is returning a Receipt or StoreReject for that request. There may be additional side effects (such as storing the actual chunk if the receipt is returned) which must be enforced at the application level.

E.2 Work assignment to another state machine

We can also partition the nodes of a system into multiple state machines. When this is done, it becomes necessary to assign work between state machines. The following lemma provides certain guarantees when our state machine liveness criteria are met:

Let A and B be state machines which share $f+1$ nodes in their intersection.

Lemma 33. *If A executes* assign(w, B, a) *then B will execute w.*

Proof. By Theorem 7, if rational $b \in B$ submits w to B then B will execute w. It remains to show show that $\exists b$ that will submit the request to B. Line 47 inserts an expect into the message queue. By Lemma 11 rational b will send the expected message which is the result of w being performed by B. Since $|A \cap B| = f+1$, $\exists b \in A \cap B$ such that b is non-Byzantine. If b is altruistic it submits w to B trivially. Let b be rational. Since b is rational, b assumes that the other f nodes in the intersection are Byzantine and will not submit w to B (thus possibly preventing b from returning the actual result of w to A). So in order to get a valid result of w, b must submit w to B. ☐

E.3 Credible threats

Ordinarily, all work is assigned through the state machine. A node submits an *assign* request to the state machine, the request is replicated on all nodes, and finally some subset of the nodes performs the request and the result is returned to the state machine. This can lead to problems if the requests themselves are large or costly to store and transport. A performance optimization is to instead submit a promise to assign work to the state machine, a direct message assigning the work to an individual node (submitting the request through the state machine only if the initiating node does not get a response back directly), and then report the result of the work assignment back to the state machine. The actual protocol is presented in Figure 14. There are two technical difficulties in implementing such a scheme: (1) should the requester actually submit the direct request and (2) should the requestee submit the direct request.

When requests are considered in isolation, the answer to both (1) and (2) is "no." Due to our modeling assumptions, in a pairwise communication rational nodes assume that the other node is Byzantine and plan for the worst action the other node could do to them. For both cases, the "worst" thing would be to require the state machine to be used regardless. Since this is a possibility, neither node will use the "fast" path solution. We address this by introducing *open* threats.

Definition 3 (Open Credible Threat). *A credible threat is* open *iff the requester can expect a response from a fast path request, based on the current time, before the requester must submit the request to the state machine in order to guarantee being able to fulfill his vow.*

Since rational nodes assume the maximum number of Byzantine nodes in the worst configuration, rational nodes will assume that they receive requests from Byzantine nodes, and that Byzantine nodes will send the request through the state machine regardless of whether or not the rational node responds directly. If, however, there are at least $f+1$ nodes who have issued concurrent *open* credible threats against a rational node, at least one of them must be non-Byzantine so will in fact not go through the state machine if the node responds directly, and the rational node has reason to respond directly upon receiving the direct request. Similarly the sender of the request would expect to be ignored unless it currently has $f+1$ *open* credible threats issued. In order to quantify the necessary conditions for the fast path to be used, we must introduce the concept of *sufficient open threats*.

Definition 4 (Sufficient Open Threats). *There are sufficient open threats for a to follow the fast path iff $a \in A$ such that $|A| \geq f+1$ and $\forall b \in A$, b is the requester or recipient of a credible threat from $B \subset A$ such that $|B| \geq f+1$.*

Lemma 34. *Rational a will follow the fast path if there are* sufficient open threats*.*

Proof. If there are sufficient open threats, then other non-Byzantine nodes are assumed to use the fast path appropriately. By the *sufficient open threats* definition, there are open threats involving a from at least $f+1$ distinct nodes. Let k be the number of distinct nodes involved in threats with a. Since $k \geq f+1$, at least one of these nodes is non-Byzantine. The cost of following the fast path is the cost associated with sending a single message m. When following the slow path, the cost is at least nm (the cost of sending a single message to all other nodes in the state machine). So the expected cost of ignoring the fast path for all nodes is $(k)nm$ while the cost of following the fast path for all nodes i at most $(k)m + (f)(nm)$. If the node follows a fast path for only some of the nodes, then by our modeling assumption the ignored nodes will be assumed to be Byzantine, so no node should be ignored if the $k \geq f+1$ threshhold is met. So the expected cost is less if a follows the fast path and a will follow the fast path as suggested. ☐

While the *sufficient open threat* requirement is rather heavy handed, it is instructive to note that the requirements are structured so that the fast path will be used when the system is under heavy load – precisely when it using the fast path will have the most noticeable affect.

```
// protocol for submitting a vow to the state machine
// $w$ is the request, $a$ is the target, $b$ is the requester
threaten($w, a, b$):
  RSM.submit($vow(hash(w), a, b)$);
  delivered := false;
  start ($vow.timer1$);
  if (sufficient open threats)
     a.send(w);
     a.expect < m : m.reqHash = hash(w) >;
     $r$ := a.deliver();
     delivered := true;
     RSM.submit($r$);
     cancel($vow.timer1$);


// process to execute a vow request
execute($vow(u, a, b)$):
  if ($p = a$ and sufficient open threats)
     b.expect $< m : hash(m) = u >$
     $r$ := b.deliver();
     b.send($r$);
  vowdelivered := false;
  start ($vow.timer2$);
  b.expect $< m : m.reqHash = u >$;
  $r$ := b.deliver();
  vowdelivered := true;
  cancel ($vow.timer2$);



on timeout($vow.timer1$):
 if (delivered = false)
   $r$ := RSM.submit($assign(w, a, b)$);
   RSM.submit($r$);


on timeout($vow.timer2$):
  if (vowdelievered = false)
    $badlist[b]$ := true;
```
Figure 14: Credible threat protocol