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
University of Texas at Austin - Dept. of Computer Science

ABSTRACT

This paper describes a general approach to constructing cooperative services that span multiple administrative domains. In such environments, protocols must tolerate both *Byzantine behaviors* when broken, misconfigured, or malicious nodes arbitrarily deviate from their specification and *rational behaviors* when selfish nodes deviate from their specification to increase their local benefit. The paper makes three contributions: (1) It introduces the BAR (Byzantine, Altruistic, Rational) model as a foundation for reasoning about cooperative services; (2) It proposes a general three-level architecture to reduce the complexity of building services under the BAR model; and (3) It describes an implementation of BAR-B, the first cooperative backup service to tolerate both Byzantine users and an unbounded number of rational users. At the core of BAR-B is an asynchronous replicated state machine that provides the customary safety and liveness guarantees despite nodes exhibiting both Byzantine and rational behaviors. Our prototype provides acceptable performance for our application: our BAR-tolerant state machine executes 15 requests per second, and our BAR-B backup service can back up 100 MB of data in under 4 minutes.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems

General Terms

ALGORITHMS, RELIABILITY

Keywords

Game theory, Byzantine fault tolerance, Reliable systems, Peer to Peer

1. INTRODUCTION

This work was supported in part by NSF award CNS 0509338 and NSF CyberTrust award 0430510. Lorenzo Alvisi was also supported by an Alfred P. Sloan Fellowhip.

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 nodes' actions. Examples of such services include Internet routing [21, 59], wireless mesh routing [33], file distribution [14], archival storage [38], or cooperative backup [5, 16, 31]. 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 [56] becomes inadequate. Nodes in MAD systems may depart from protocols for two distinct reasons. First, as in traditional systems, nodes may be *broken* and arbitrarily deviate from a protocol because of component failure, misconfiguration, security compromise, or malicious intent. Second, nodes may be *selfish* and alter the protocol in order to increase their utility [1, 27]. Byzantine Fault Tolerance (BFT) [10, 30, 36] 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; this bound is not tenable in MAD systems where *all* nodes may benefit from selfish behavior and be motivated to deviate from the protocol. Models that only account for selfish behavior [59] handle the second class of deviations, but may be vulnerable to arbitrary disruptions if even a single node is broken and deviates from expected rational behavior.

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 [8] or Byzantine-tolerant protocols [30], it is necessary to design protocols 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 model that captures the essential aspects of MADs. The Byzantine-Altruistic-Rational (BAR) model accommodates three classes of nodes. *Rational* [59] 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* [10, 30, 36] nodes can depart arbitrarily from a proposed program whether it benefits them or not. Finally, BAR accommodates the presence of *altruistic* [45] 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-level architecture for BART services. The bottom level implements a small set of key abstractions (e.g., state machine replication and terminating reliable broadcast) that simplify implementing and reasoning about BART distributed services. The middle level partitions and assigns work to individual nodes. Finally, the top level 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 BAR-B, a BART cooperative backup service. BAR-B 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 [32]—that, by supporting a notion of identity that is "expensive" to obtain, avoid the Sybil attack [18]. We do not target open membership peer-to-peer systems.

We find that our architecture makes the design of BAR-B easier to derive, implement, and comprehend. Compared to previous peer to peer backup architectures [15, 16, 31], BAR-B 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 layereded architecture simplifies the task of proving safety and liveness properties.

We also show that our approach is practical: our prototype BART state machine executes batches of 15 requests per second and our BAR-B prototype can back up 100 MB of data to 10 nodes in under 4 minutes while guaranteeing data recovery despite the failure of 3 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 BAR-B, a cooperative backup system 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 levels: our asynchronous BART state machine, our techniques for work assignment, and our BAR-B 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 [1, 27], and (c) some nodes may be categorically broken [10, 30, 36].

The Byzantine Altruistic Rational (BAR) model addresses these considerations 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 known 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 a node's costs (e.g., computation cycles, storage, network bandwidth, overhead associated with sending and receiving messages, power consumption, or threat of financial sanctions [31]) and benefits (e.g., access to remote storage [38, 5, 16, 31], network capacity [33], or computational cycles [57]) for participating in a system.

Byzantine nodes may deviate arbitrarily from the suggested protocol for any reason. They may be broken (e.g., misconfigured, compromised, malfunctioning, or misprogrammed) or may just be optimizing for an unknown utility function that differs from the utility function used by rational nodes—for instance, ascribing value to harm inflicted on the system or its users.

Under BAR, the goal is to provide guarantees similar to those from Byzantine fault tolerance to "all rational and altruistic nodes" as opposed to "all correct nodes." We distinguish two classes of protocols that meet this goal.

- Incentive-Compatible Byzantine Fault Tolerant (IC-BFT) protocols: A protocol is IC-BFT if it guarantees the specified set of safety and liveness properties and if 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 specified set of safety and liveness properties in the presence of all 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 exploit local optimizations not specified in the protocol without endangering the global guarantees. Note that IC-BFT protocols are a subset of the BART protocols.

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 [32].

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, that each member can determine whether a public key belongs to a specific member, and that no set of nodes has the computational power to subvert the standard cryptographic assumptions associated with public key signatures [50] and secure hashing [46]. These assumptions are reasonable for our target environments—a volunteer distributes a list of keys to coworkers; a university's electronic ID system maps identities to dormitory residents; the refurbisher installs the key information on machines before they are distributed to non-profits—and facilitate the design of BART systems in three ways. First, they provide justification for our assumption that the number of Byzantine nodes in the system can be bounded. Second, they give rational nodes an incentive to consider the long-term consequences of their actions, making it easier to apply internal sanctions (e.g. denial of service or data deletion) against misbehaving nodes. Third, they allow us to tie system identities to real world entities, so that external sanctions (e.g. social disgrace, monetary fines, or contractual penalty) may be applied against the owners of nodes that misbehave. Support for external sanctions increases the flexibility of our protocols, but our protocols do not require the use of external sanctions for safety or liveness.

We have different timing assumptions for BAR-B and for the underlying BART replicated state machine. BAR-B relies on synchrony to guarantee both its liveness and safety properties—data trusted to BAR-B is guaranteed to be retrievable only until the lease associated with it expires. Conversely, the underlying BART state machine is safe even in an asynchronous system, though liveness is only guaranteed during periods of synchrony.

To ensure liveness under the BAR model, we make two additional timing assumptions. First, we give nodes an incentive to stay as synchronized as possible through a "penance" mechanism (discussed in Section 5.1.3) that penalizes untimely nodes. For this mechanism to be acceptable, nodes' clocks must be sufficiently synchronized that these penalties do not outweigh the benefits of participating in the system. Second, 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 $t + max\_response\_time$. This assumption allows us to bound the state that non-Byzantine nodes maintain in order to answer late requests and thereby allows rational nodes to ensure that the benefits of participation outweigh the costs.

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 four 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 conservative when computing the impact of Byzantine nodes on their utility. Third, we assume that if a protocol provides a Nash equilibrium, then all rational nodes will follow it [34] [1] .Finally, we assume that rational nodes do not collude—colluding nodes are classified as Byzantine. Relaxing these assumptions is future work.

Rational nodes will only participate in a cooperative system if they receive a net benefit from their 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; otherwise rational nodes will refuse to participate, compromising liveness.

Rational nodes want to reduce their cost without relinquishing the benefits that come from participating in the protocol. We assume a simple model in which nodes' utilities are affected by the work that must be done but not by the order in which work is performed or by 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 assumption 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 and bandwidth available. This assumption recognizes that idle resources are perishable.

Rational nodes are conservative when estimating the potential impact of Byzantine nodes on their utility: we assume that for each rational node, the benefits of the service greatly outweighs the costs, and therefore any increase in the risk of service failure is unacceptable. So, when computing the expected outcome of its actions, a rational node r assumes that the maximum number (f) of Byzantine nodes are present in the system and that they will act in the way that minimizes r's utility.

[1]Because the protocol can be regarded as coming from an external authority, some prefer to regard such an equilibrium as a *correlated equilibrium* [4], which is a generalization of Nash equilibrium. This view would not change our analysis.

Byzantine nodes. We assume a Byzantine fault model for Byzantine nodes [10, 30, 36] 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. Finally, we assume that at most $\frac{n-2}{3}$ of the nodes in the system are Byzantine.

4. SYSTEM ARCHITECTURE

This section provides an overview of our design. The sections that follow describe each level of our design in more detail.

4.1 3-Level Architecture

We propose a three-level architecture for building BART services (Figure 1). The layered design simplifies the analysis and construction of systems by isolating and addressing classes of misbehavior at appropriate levels of abstraction.

Architecture	**Prototype**		
Level 3: Application	BAR−B Backup		
Level 2: Work Assignment	Guaranteed Response	Periodic Work	Authoritative Time
Level 1: Primitives	Replicated State Machine		
	Message Queue		

Figure 1: System architecture

Level 1, the *basic primitives* level, provides IC-BFT versions of key abstractions (e.g. Terminating Reliable Broadcast (TRB) [30] and Replicated State Machine (RSM) [10, 29, 55]) for constructing reliable distributed services. The BART RSM gives us the abstraction of a correct (e.g., reliable and altruistic) node.

Level 2, *work assignment*, allows us to build a system in which work can be assigned to specific nodes instead of executed by all replicas in the RSM. The assignment is done through a the *Guaranteed Response* protocol that generates either a verifiable match between a request and the corresponding response or a verifiable proof that a node failed to respond to a request. The assignment protocol enables efficient replication for our backup application, and the protocol itself is optimized to use the RSM as little as possible.

Level 3, the *application* level, implements a desired service using the levels underneath. Our architecture defines a contract between the application and the two lower, application-independent levels. The lower levels provide reliable communication and authoritative request-response bindings, while the application is responsible for providing a net benefit and defining legal request-response pairs.

4.2 Principles of Operation

Accountability lies at the heart of our approach to constructing BART 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?

The simplest kind of misbehavior to detect and punish 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 easy to address, and a number of systems have done so [11, 41].

Two other "passive-aggressive" behaviors are more problematic. 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 it has been unjustly leveled by a self-interested or faulty node. Second, a node may exploit non-determinism to provide incomplete information or take undesirable steps that interfere with the protocol's operation but are difficult to conclusively prove wrong. For example, in one step of an asynchronous replicated state machine protocol [10], a node normally transmits a signed copy of the request, but for liveness it is permitted to transmit a signed timeout message instead. In such a protocol, self-interested nodes may choose to send the timeout message 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.

The implementation of Level 1 primitives addresses such challenges in three ways. First, nodes *unilaterally deny service* to nodes that fail to send expected messages. This low-level, local tit-for-tat technique provides incentives for cooperation without requiring a third party to judge which node is to blame. Second, the protocol *balances costs* so that when nodes have a choice between two messages, there is no incentive to choose the "wrong" one. Third, nodes can *unilaterally impose extra work* (called *penance*) when they judge that another node's response is not timely. The penance mechanism safeguards liveness by discouraging rational nodes from improperly exploiting timing-based non-determinism.

Addressing the challenges of non-responsiveness and non-determinism in the two higher levels is much simpler. For Level 2 (work assignment), if a node fails to reply to a request issued via the underlying state machine, then a quorum of nodes in the state machine generates a proof of misbehavior against the node. And because applications at Level 3 make use of reliable work assignment, each request is bound to a reply or timeout. As a result of this binding, the application protocol must merely be designed so that requests and responses include sufficient information for any node to judge the validity of a request/response pair.

5. LEVEL 1: BART STATE MACHINE

At the core of fault-tolerant distributed services are a few fundamental primitives. For instance, state machine replication is an essential building block for a range of highly available replicated services [7] and quorum-based replication is the basis for fault-tolerant distributed storage systems [37]. The purpose of the first level of our architecture is to implement fundamental primitives so that they continue to provide their customary guarantees within the BAR model. In this section, we present a BART asynchronous replicated state machine (RSM). Our protocol is based on PBFT [10], with modifications motivated by the BAR model. These modifications are based on four guiding principles.

Ensure long-term benefit to participants. Self-interested nodes must gain long-term utility for participating in the system to be motivated to participate faithfully. Ultimately, these benefits must stem from the higher level service, but as a hook for providing such benefits to all participants our RSM rotates the leadership role to guarantee that every node has the opportunity to submit proposals to the system.

Limit non-determinism. Non-determinism offers nodes the choice of multiple behaviors. Although each of these behaviors is legal under different circumstances, given the specific state of each node one of the behaviors is preferred by the protocol. Self-interested nodes can hide behind non-determinism to shirk work: they can disregard the preferred behavior and adopt a less costly one that other nodes cannot definitively identify as illegal. In our implementation of BAR primitives we carefully limit the choices available to a node. For example, we base our state machine on terminating reliable broadcast (TRB) rather than consensus [24], because the former protocol, by allowing fewer valid outcomes, gives rational nodes fewer options from which to choose when deciding which behavior maximizes their benefit.

Mitigate the effects of residual non-determinism. When non-determinism is unavoidable, two low-level techniques are often useful. First, we employ *cost balancing* when a node has a choice between multiple actions. The costs of the actions 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 entries set to "1" for up-to-date nodes so that the sender saves no network bandwidth by sending incomplete information. Second, *encouraging timeliness* addresses the non-determinism inherent in an asynchronous system by allowing nodes to judge unilaterally whether other nodes' responses are early, on time, or late 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 punishments and that (b) the costs imposed by Byzantine nodes through spurious unilateral sanctions are limited.

Enforce predictable communication patterns. We 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 to be able to propose a command.

5.1 Protocol Description

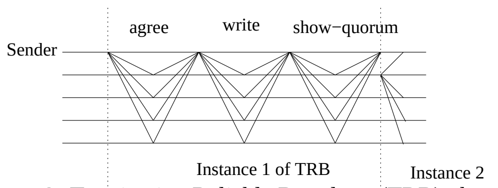

Figure 2: Terminating Reliable Broadcast (TRB) phases.

In this subsection, we first examine the high-level structure of the protocol. We then detail the low-level mechanisms used to enforce periodic communication and limit the effects of non-determinism. Due to space constraints, we limit our discussion to the key differences between our protocol and traditional PBFT implementations. Appendix B

Our BART replicated state machine protocol is based on PBFT [10]. When a node wants the state machine to execute a command, the node proposes the command in a TRB *instance*. Instances proceed in sequence, with instance i deciding the ith command to be executed by the state machine. We differ from the PBFT protocol in several key ways.

1. We use TRB instead of consensus. This choice is an application of the principle of limiting non-determinism. In TRB, only the initial *sender* may propose a value during a particular instance and an instance can terminate only in two ways:

all non-Byzantine nodes either adopt the value proposed by the sender, or, if the sender is faulty or slow, a default value. Conversely, in consensus a timeout caused by a slow or faulty sender may allow a new leader to propose a different value for that 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 useful for limiting the choices available to rational nodes. Without this restriction, a new leader elected to terminate instance i may prevent progress by selfishly trying to make the state machine adopt its value rather than the sender's (see Appendix C.4). By limiting the possible outcomes of instance i, TRB avoids this conflict of interest.

2. We use a round-robin leader selection policy to ensure that all nodes can benefit from their participation in the state machine. Traditional replicated state machines require a client to send a command to a sender, who proposes the command to the state machine. But, a rational sender would have no incentive to act on a remote client's wishes. So, for each TRB instance we rotate the role of *sender* to the next node in the system. Each participant thus has a periodic opportunity to propose values to the state machine.

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 desire to avoid a conflict of interest involving the sender node in a TRB instance. Suppose the sender s of instance i is slow, and, after sufficiently many nodes time out on s, a new leader is elected to bring instance i to conclusion. Every node but s is interested in a timely conclusion of instance i to ensure *its* turn to propose a value; s, however, is interested in ensuring that i terminates with the adoption of s's original value—rather than the default value—and to this end can take steps that compromise liveness (see Appendix C.4). By using an extra node, we prevent s, after it has proposed its value, from participating in the steps required to complete instance i, eliminating this potential conflict.

Our TRB protocol provides four guarantees in an eventually synchronous BAR environment in which the higher-level service provides net benefits to all participants. *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, if the sender is non-Byzantine and sends a message m, then the sender eventually delivers m.

The protocol provides safety (Agreement and Integrity) under an asynchronous model, but guarantees liveness (Termination and Non-Triviality) only during periods of synchrony [24] when there exists a known bound Δ on message delivery time. The requirement that all rational participants realize a net benefit from the service is needed only to achieve liveness; rational nodes that do not benefit from the service will not take any action on behalf of the protocol, but they will not compromise the safety properties.

Figure 2 illustrates an execution of TRB 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 *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: in this case, the instance consists of a single turn. 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.

If a collection of set-turn messages selects a new leader, the newly selected leader first performs a read: it queries all nodes for their 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 the null value *senderTO*, indicating that the *sender* is suspected of having failed. Once a value is delivered, the $i+1$st instance starts with the next *sender* in the sequence.

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 local retaliation policy: if node x next expects a message from node y, x will ignore any communication from—and delay any communication to—node 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, a predicate is associated with each bubble: a message from y is allowed to fill a 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 returns the corresponding message.

Message queues, combined with quorums of size $n-f-1$, provide the incentive for rational nodes to send all messages expected in the protocol. If a given rational node r chooses not to send a message to some node s, then s will ignore r in the future. In the worst case for r, an additional f Byzantine nodes in the system will not communicate with r, preventing it from gathering a quorum during its next turn as *sender*. This situation would 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. Because we assume that the value of the service greatly exceeds the cost of communication, a rational r prefers to send all expected messages to avoid any risk of losing access to the state machine.

5.1.2 *Balanced messages*

To apply the principle of *cost balancing* to the state machine protocol, we ensure that whenever the protocol provides a node with the opportunity to choose which message to send next, the intended message is never more expensive to send than the alternatives. For example, after a timeout a node should send either the command issued by the sender for the instance or *senderTO* if no such command was received. We construct the *senderTO* message to always be of the length of the largest possible command so that lying would not allow a node to save bandwidth.

5.1.3 *Penance*

We implement a "penance" mechanism to encourage timeliness in the state machine. In particular, although the *promptness principle* (Section 3) encourages nodes to promptly send any messages they are deterministically bound to send, we use penances to encourage good behavior when waiting may allow a rational node to avoid sending a specific message. To implement the penance mechanism, each node maintains an *untimely vector* that tracks their perception of other nodes timeliness: a node is considered untimely if any timeout message electing a new leader arrives significantly earlier or later than expected according to the receiver's local clock. When a node x becomes the sender, it includes its untimely vector with the value it proposes. After agreeing on the proposal, all nodes except the sender *expect* a *penance message* from each node indicted in the untimely vector. Because of the way message queues handle expects, 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 are chosen so that the expected benefit of sending late is less than the expected penance cost, (2) the sender is excused from receiving penance messages to prevent the sender from incurring additional costs for truthfully reporting a penance, and (3) the spurious work introduced by Byzantine nodes through the penance mechanism is bounded.

5.1.4 Timeouts and garbage collection

The system makes use of two timeouts for liveness: (1) a "set-turn" timeout to transfer leadership away from a slow leader and (2) a $max\_response\_time$ timeout to garbage collect messages queued for extremely slow nodes.

A sufficient number of set-turn timeout messages transfers leadership of an instance to the node lexicographically after the current turn's leader. The first turn of an instance uses a pre-specified timeout, and this timeout is increased for each subsequent turn of that instance until the instance completes. Note that in every TRB instance only the initial sender (the first leader of the instance) can propose a non-trivial value, so it is important that the initial timeout be significantly larger than common-case network delays. Our prototype uses 10 seconds for its initial set-turn timeout.

The timeout after $max\_response\_time$ bounds local state in the presence of extremely slow nodes. In order to ensure a *predictable communication pattern*, we require all nodes to send all protocol messages. If node a remains silent for an extended period of time, it can force non-Byzantine node b to retain an arbitrarily large set of pending messages to a. If this state becomes too large, the cost of participating in the protocol will exceed the benefit, and rational nodes will withdraw from the system, endangering liveness even in periods of synchrony. The timeout allows a node to bound this state so that its benefits from the system exceed its costs[2].

In particular, if a has been holding pending messages for b for more than $max\_response\_time$, then a (i) records b as faulty by adding b to its $badlist$, (ii) garbage collects all state associated with b, and (iii) refuses further communication with b.

It is undesirable for a non-Byzantine node to declare incorrectly a slow node to be faulty: doing so jeopardizes liveness and thus puts at risk the net utility that nodes expect to gain from participating in the system. Nodes therefore use an extremely long $max\_response\_time$ (e.g., 1 week in our prototype) that significantly exceeds the expected worst-case network disconnection time between any pair of nodes.

5.1.5 Global punishment

The state machine includes a mechanism to transform local suspicion against other nodes (as recorded in each node's $badlist$) into POMs. The POMs allow nodes to agree that someone misbehaved so that an appropriate global punishment may be applied. This mechanism also enables the use of quorums of smaller size ($\lceil \frac{n+f}{2} \rceil$ rather than $n - f - 1$), improving the availability of the state machine.

When node a is the sender of an instance, it includes its *badlist* as a bit vector with the value it proposes. Nodes monitor the *badlist*s they receive from others: if over time node b appears on at least $f+1$ different senders' *badlist*s, then the receivers of these *badlist*s also begin to consider b faulty: they add b to their own $badlist$, discard the state associated with b, and refuse to communicate with b in the future.

In addition to helping punish misbehaving nodes, the *badlist* mechanism enables us to reduce the size of quorums from $n-f-1$ to $\lceil \frac{n+f}{2} \rceil$. Without *badlist*s, quorums of size $n - f - 1$ are required to provide an incentive for a non-Byzantine node to send all required messages to all recipients that expect the message: by failing to send messages to even one node, the sender jeopardizes its ability to propose new commands to the state machine because the skipped node and f Byzantine nodes could together prevent a quorum from forming. With quorums of size $\lceil \frac{n+f}{2} \rceil$, a sender that skips a node does not risk losing the ability to form quorums for its proposed values; however, the badlist mechanism ensures that the sender faces the equally severe risk of being included on $f + 1$ badlists from the skipped node and f Byzantine nodes.

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.

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*)*

The full proofs appear in (Appendix B). 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.*

[2]An alternative for bounding state that we are exploring as future work is to provide an incentive compatible variation of the garbage collection and checkpoint recovery protocol described by Castro and Liskov [10].

LEMMA 2. *No rational node r benefits from sending the "set-turn" message early.*

The proof for the first lemma relies on the penance protocol described in the previous section. The second lemma deals with early time-outs. This deviation may cause the sender's proposal to be ignored, and *senderTO* to be decided instead. By construction, *senderTO* is at least as large as a resend of the sender's command, so no bandwidth is saved. Nodes other than the sender have no stake in which command is decided because they cannot unilaterally prevent the sender's command from executing—at most, they can delay it. The sender itself could have an interest in manipulating the outcome by sending "set-turn" messages 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 2: PARTITIONING WORK

Our second level partitions work to reduce the replication overhead required by cooperative applications. Even though state machine replication technically suffices to support a backup service directly, the overhead of such an approach would be unreasonable: each replica would have to process each command and maintain a full copy of the program state. In a cooperative backup service with 100 participants, 100 MB of data backed up would consume 10 GB of disk space. Conversely, by assigning work to individual nodes, we can make use of arithmetic codes to provide low-overhead fault-tolerant storage.

We introduce three protocols for work assignment. The *Guaranteed Response* protocol ensures that every request is answered, possibly with a message indicating that the work was not done in in a timely fashion. The *Periodic Work* protocol ensures that clients periodically answer implicit requests required by an application. The *Message Binding* protocol binds messages to an authoritative time.

We first describe these protocols using the abstraction of a trusted altruistic node, which we call the *witness node*. Then we show how the witness node can be implemented on our replicated state machine in an incentive-compatible manner.

For large systems, using a single replicated state machine to implement the witness node becomes impractical. To allow our current system to span more than a few dozen nodes, large systems should be partioned into disjoint state machines of 10-30 nodes each. For applications in which nodes must all be able to work together, these state machines should be able to communicate with each other [2, 49]. There are BAR-specific challenges related to communication between state machines. We believe these challenges are surmountable, but leave them for future work.

6.1 Guaranteed Response

The Guaranteed Response protocol gives rational nodes an incentive to respond to requests. The protocol is necessary because direct communication does not suffice when nodes can behave rationally. Consider an example where some node a sends a request to another node b and gets no answer. Node a may well complain about b, but because a cannot prove that b received and ignored its request, it would be unwise to punish b based on a's complaint. The Guaranteed Response protocol eliminates all ambiguity: in the above situation, it ensures that a lack of response to a can only be the result of uncooperative behavior by b, who can then be safely punished.

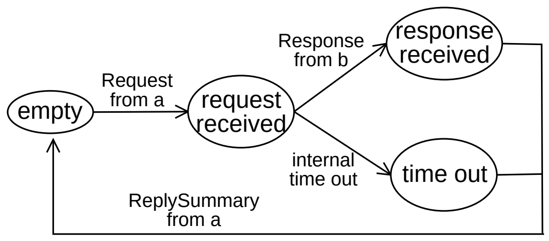

Figure 3: Basic Guaranteed Response protocol

Figure 3 shows the state transition diagram for a correct witness node running the Guaranteed Response protocol. The basic idea is that node a never sends work requests directly to b, but instead goes through the witness node. The witness is then in a position to answer with *NoResponse* if necessary.

More precisely, client a starts by sending *Request* to the witness, who is initially in the *empty* state. *Request* contains the name of the intended recipient, b, as well as the work w that must be performed by it, and causes the witness to transition to state *request received*. The witness stores *Request* and forwards a copy of it to b. If b is correct, it will send a signed *Response* to the witness, causing it to enter state *response received*. *Response* contains the answer to a's request together with a summary of the request to which b is responding. The witness then discards *Request*, forwards *Response* to a, and keeps a copy of *Response* until it receives from a a *ReplySummary* containing a summary of *Response*. If node b does not answer *Request* within a predetermined $max\_response\_time$, then the witness transitions to state *time out* and sends *NoResponse* to a. This message is signed and contains a summary of the request. Again, node a must send *ReplySummary* (this time with a summary of *NoResponse*), returning the witness to the *empty* state.

The state of the witness node includes a copy of the last message sent. This state allows the witness to resend messages that were lost, which in turn allows our protocol to handle nodes that come and go. Nodes cannot stay away for too long, however, because our backup application requires that nodes answer within $max\_response\_time$.

6.1.1 *Implementing the witness node*

The incentive-compatible replicated state machine allows us to implement the abstraction of a correct witness node on top of a collection of BAR nodes. We must be careful to maintain incentive compatibility: our state machine only provides incentive for communication with members of the state machine, not outsiders, so nodes a and b must be part of the replicated state machine. Therefore communication with the witness node is not by actual message sending: when the Guaranteed Response protocol talks of a node sending to the witness, this translates to the node submitting a command to the RSM. Whenever the protocol talks of the witness sending to a node, no actual sending is necessary: since every node in the RSM has a copy of the witness state, the RSM replica running on the destination node passes the message to the local code that handles it.

The *NoResponse* message is a special case for two reasons. First, the "timeout" decision must be made deterministically. We accomplish this by having the state machine maintain a deterministic RSM time that is a function of recent values of the local clocks of all nodes (see Appendix E.4) for details). The RSM replica running on node a is responsible for submitting a "timeout" command to the RSM when the deterministic RSM time indicates that the response is late. Second, the abstraction of a single, signed *NoResponse* message from the witness node to a is actually implemented by having a receive a signed message from $f + 1$ RSM replicas. After nodes transition to the *time out* state, these signatures are gathered by replica a, which uses the message queue primitive described in Section 5.1.1 to `expect` a signature from every other replica. The other replicas therefore know, when entering *time out*, that a is ready for their signature message and the message queue mechanism gives them an incentive to send the signature. Once a has enough signatures to form a *NoResponse* message, it passes the message to the local code at a that handles it.

Provided that the application provides sufficient sanctions for nodes that cause a *NoResponse*, the following theorem holds (see Appendix E.

THEOREM 3. *If the witness node enters the* $request\ received$ *state for some work* w *to rational node* b*, then* b *will execute* w.

6.1.2 State limiting

The witness node, naturally, can communicate with more than one node at a time. It runs several instances of the protocol highlighted above, and each instance (which we call a *slot*) is reserved for a particular node.

We limit the overhead associated with Guaranteed Response by limiting the number of slots available to a node. Limiting the number of slots accomplishes three purposes: (1) it applies a limit to the memory overhead of running the Guaranteed Response protocol, (2) it limits the rate at which requests are inserted into the system, and (3) it forces nodes to acknowledge responses to requests.

6.2 Optimization through Credible Threats

The Guaranteed Response protocols allows data to be replicated only where necessary. However, requests and responses are still sent to every node that is part of the replicated state machine and, because backup requests contain the data being backed up, they can be large. We therefore optimize our protocol so that in the common case nodes can communicate directly.

To get the benefits of the Guaranteed Response protocol without requiring all requests and replies to go through the RSM, we leverage the game-theory notion of credible threats [17]. In the game of chicken [12], a credible threat against rational players would be to visibly rip off the steering wheel and throw it out the window [35]. In our case, the credible threat takes a somewhat less spectacular form.

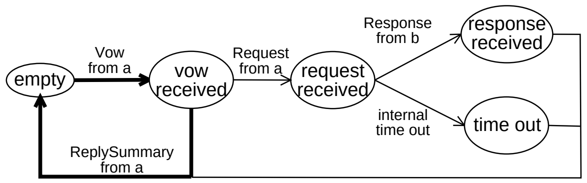

Figure 4: Guaranteed Response protocol with fast path

We optimize the Guaranteed Response protocol by adding a *fast path*. The new protocol is shown in Figure 4 with the fast path in bold. Instead of sending its request to the witness node, node a now only sends it a *Vow* with a summary of its request. The witness supplies the vow to the target (b in our example). Target b sends an ack to node a, and a sends the full request directly to b. Target b then replies to a, who forwards summary of the response to the witness. If the target does not answer then the protocol proceeds as in the unoptimized case, with a sending the full request to the witness.

The threat in this case is the vow: if b does not answer a's request directly, then a will ask b to answer to the witness node, a costlier operation for b. Key to the threat is the fact that it is credible. By sending its vow a has forfeited its right to utilize that slot until it sends the full request or supplies a reply from b that matches the vow. A rational target b knows that if it does not answer a directly, then a *will* send the request through the witness node—and this is enough to motivate b to answer a's direct request.

6.3 The Periodic Work Protocol

Cooperative systems may include maintenance tasks that need to be performed periodically, for example auditing nodes' storage records. However, there may be no incentive for any individual node to initiate such maintenance work. Under the Periodic Work protocol, the witness node checks that this periodic work is done. The existence of this check, in turn, means that rational nodes will perform these tasks.

In the case of the RSM witness, the Periodic Work protocol initializes the system with the expectation that, with a certain frequency, each node will provide the witness with an application-specified response type indicating its completion of a periodic task. If a node does not supply the expected *ReplySummary*, the witness node can either unilaterally deny its services to the offending node or generate a POM to be handled by the application.

6.4 Authoritative Time Service

In applications where time has meaning, authoritatively binding messages to time is a potentially important action. The *Authoritative Time Service* serves two purposes. First, it maintains an authoritative time that is recent, nondecreasing, and identical at all state machine nodes. Second, it binds messages to times according to that time. In particular, our Guaranteed Response protocol relies on this time when generating *NoResponse*s for non-participating nodes, and BAR-B relies on the message-time binding to identify certain classes of misbehavior.

In order to maintain the time, each proposal to the state machine is required to contain a local timestamp generated by the proposer. The authoritative time is computed by taking the maximum of the median of the timestamps of the $2f + 1$ most recent decisions and the previous authoritative time; when "no decision" is decided, then the time for that decision is defined to be the previous authoritative time. In order to bind a message to a time, node a submits the message to the *Message Binding Protocol* and proposes a *BindingRequest* to the RSM. The nodes in the RSM then send a a signature binding the message to the current authoritative time.

7. LEVEL 3: THE APPLICATION

In our architecture, BART 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 concerns discussed in Sections 5 and 6. The replicated state machine of the first level provides the abstraction of a correct node, which is useful in implementing sanctions. Reliable work assignment is taken care of by level two primitives, so the application can focus 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 act as 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 BAR-B, a MAD cooperative backup system.

7.1 BAR-B Overview

BAR-B is a cooperative backup system in which nodes commit to participating in the system's state machine and contributing an amount of storage to the system in exchange for an equal amount of space on other nodes. Under normal circumstances (see Section 7.2 for recovery), nodes interact with BAR-B through three operations: store, retrieve, and audit. To store a backup file, the owner compresses it, splits it into smaller pieces (chunks), encrypts the chunks, and then sends them 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. The retrieve request contains the receipt, so the storer has three options: (i) return the chunk, (ii) show that the chunk's storage lease duration has expired, or (iii) show a more recent StoreInfo for the same chunk. Any other response would indicate that the storer prematurely discarded data entrusted to it and should be punished.

These receipts constitute audit records. Nodes periodically exchange audit records in order to verify that some node is not using more space in the system than its quota allows (both in terms of total storage and number of chunks). BAR-B allows each node to store a limited number of chunks, thereby binding both the state maintained in the system and the cost of performing audits.

7.1.1 Arithmetic coding

To tolerate f faults using less storage than required by full replication, nodes erasure code [51] files with an $x-f$ out of x encoding and store the resulting chunks on different peers. 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 nodes to participate.

In practice, many files are small. Since there is both a limit on the number of chunks and some per-chunk overhead, it is beneficial to keep chunks reasonably large. The BAR-B user interface therefore aggregates small files together before uploading the aggregate to BAR-B as a single backup file.

7.1.2 Request-response pattern

The responsibility of Level 2 is to structure messages carefully 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 requests either to responses or to NoResponse if the target fails to respond. Every message in the BAR-B protocol is stamped with a unique sequence number and signed by the sender.

Store. A BAR-B store request consists of two components, the chunk being stored and a tuple *(chunkId, owner, storer, hash, size, prevSize, time)* called the *StoreInfo*. The *hash* and *size* fields of the *StoreInfo* correspond to the hash and size, respectively, of the chunk being stored. The *prevSize* field is the size of the file being replaced—the owner's quota is charged for max(*size, prevSize*) until $max\_response\_time$ expires. The *time* is a real-time stamp used to calculate when the storage lease will expire; if it is too far into the future, a storer can generate a POM via the time service described in the previous section. There are three possible responses to a store request: (a) a *Receipt* containing the *StoreInfo* and time-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. A *StoreReject* can return proof that the storer is full in the form of a list of *StoreInfo* records, each signed and stamped by its respective sender and holding an active lease, and such that the total size of the *StoreInfo* records plus the request's *StoreInfo* size exceed the node's quota. When the owner issues a *StoreInfo* request or receives a response, the owner adds it to its record of utilization of the system, known as the *OwnList*. Any other response constitutes a POM against the storer—either (a) the response itself is a POM generated by the work allocation level (e.g., NoResponse) or (b) the response is inappropriate for the request and thus a signed confession.

Retrieve. A BAR-B 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 $chunkId$, or (c) the storer is in the process of recovering its data (see below). Any other response constitutes a POM against the storer—either (a) the response itself is a POM generated by the work allocation level (e.g., NoResponse) or (b) the response is inappropriate for the request and thus a signed confession.

Audit. An audit 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* and *StoreList* for f nodes chosen at random in the system. 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 expensive operations, and rational nodes would avoid performing them if possible. We avoid this problem by leveraging the Periodic Work protocol described in Section 6.1. The RSM-implemented *witness node* periodically expects the results from a recent audit—either a POM or a complete set of *OwnList*s and *StoreList*s.

7.1.3 Time constraints

The primary purpose of a backup system is to provide 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, BAR-B assumes that all non-Byzantine nodes will respond to a request within $max\_response\_time$. 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. In BAR-B, every *StoreInfo* expires 30 days after the request is issued 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 an additional *StoreRequest* before the current lease expires. Otherwise, the storer is free to discard the data. A lease expires when the *Authoritative Time Service* described in Section 6.4 indicates a date 30 days after the *time* field of a *StoreInfo*.

The introduction of these timing assumptions and lease durations allows BAR-B 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 BAR-B 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 may find it in their interest to misbehave.

We leverage the Periodic Work protocol to force each node to submit periodically to the state machine either a POM it has generated, or a special NoPOM (which, to obey the cost balancing principle, is no cheaper than a POM).

For simplicity, BAR-B 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 releasing the POM 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.

A node only needs a few things to be able to recover: the list of its peers, its membership certificate, and its key pairs. The user saves this information to a safe place when installing the program. Initially we give each node a fixed series of *linked identities*, $i_0 \ldots i_{max}$. A key pair is associated with each identity. After a node using identity i_{j-1} crashes and loses data, it uses a new identity i_j and sends a RECOVER message to every other node. In response, these node send the list of i_j's chunks that they are storing. From this list, the recovering node can then retrieve its backup data as needed.

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. The node also notes that the data it entrusted to i_k is gone, and therefore starts refreshing its backup data. The erasure coding scheme ensures that the full backup is recoverable despite the loss of the chunks stored on i_k.

Two factors prevent a rational node from 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. The first factor also limits the damage that can be done by a series of linked entities under a "persistently Byzantine" node's control. A possible addition to the two factors would be to require identity i_j to contribute 1.1^j times the storage of identity i_0 but give it no corresponding increase in quota; this measure is not strictly necessary, but it would further discourage nodes from needlessly changing identities.

A natural concern in our model is to ensure that nodes respond truthfully to the RECOVER message. A node might send only a subset of the list of chunks it is storing, in the hope of deleting the unlisted chunks to save disk space. This would be against the best interest of a rational node, however, because the recovering node might not have lost all of its receipts. In that case, a signed incomplete answer along with the receipts that should have been listed form a POM against the node.

7.3 Guarantees

The BAR-B system provides the following guarantees under BAR-B's coarse synchrony assumptions. (i) Data stored on BAR-B 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 BAR-B 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 BAR-B prototype. Our microbenchmarks show that our RSM prototype can perform about 15 operations a second, an adequate level of performance for our application's requirements. We then evaluate the performance of the BAR-B application. We find that our non-optimized BAR-B prototype can back up 100 MB of data to 10 nodes in under 4 minutes and guarantee that the data are recoverable despite the failure of 3 nodes.

8.1 Experimental Setup

Except where noted, experiments run on Pentium-IV machines with 2.4 Ghz processors, 1 GB of memory, and Debian Linux 3.0. These are shared machines, connected through 100 Mbps ethernet. The Emulab [63] experiments were run on Pentium-III machines with 850 Mhz processors, 256 Mb of RAM, and Red Hat Linux 9.

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 do not rely on these values. Each node is allocated 40 slots in the Guaranteed Response Protocol. Unless otherwise noted, we do not introduce failures in our experiments. 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 nodes, and (c) whether our RSM handles intentionally slow nodes well.

Figure 5 shows the average speed of TRB operations for systems of 5 to 23 nodes. Each trial measures the average duration over 30 TRB operations with 4 KB proposals. We run each configuration 10 times and show the median value as well as the 10^{th} and 90^{th} percentiles. We run the experiment twice: once with $f = 1$ and once with the maximal f tolerated given the number of nodes. The chart shows that TRB completes in less than 60 ms for 5 nodes

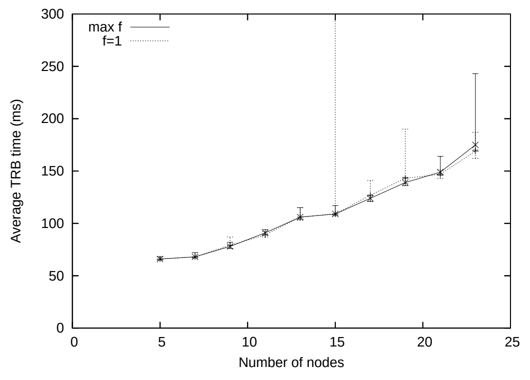

Figure 5: RSM performance as nodes are added

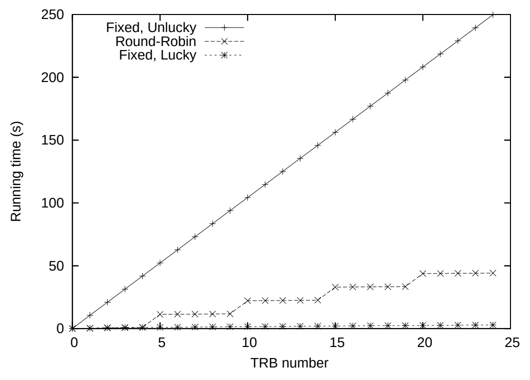

Figure 6: Impact of rotating leadership

or 175 ms for 23 nodes, a level of performance that is sufficient for our application due to (a) the relatively modest response time demands of backup and (b) the ability to batch multiple application commands in each TRB instance to improve throughput [10]. The graph also shows that performance is hardly affected by the choice of f and is reasonable for the range of sizes we chose. Eventually, however a large cooperative service should be split into multiple state machines as proposed in Section 6.

Our performance is inferior to protocols that are not designed for the BAR model. PBFT [10] 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. Note that (just as in PBFT) to maximize application throughput, nodes can submit multiple commands in a batch for each TRB operation.

Figure 6 shows the relative impact of two leader election policies in the presence of failures. Our protocol rotates the role of sender between instances 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.

8.3 BAR-B

Our BAR-B experiments are designed to determine the following: (a) whether the performance of BAR-B is adequate, (b) the value of the fast path optimization, and (c) the cost of performing system audits.

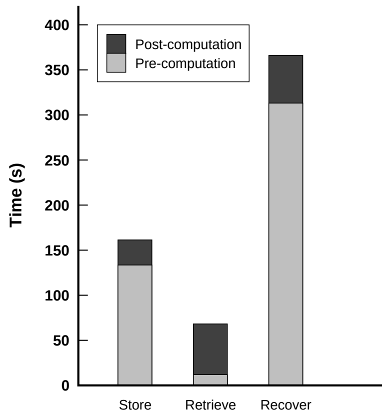

Figure 7: Operation time for 100 MB

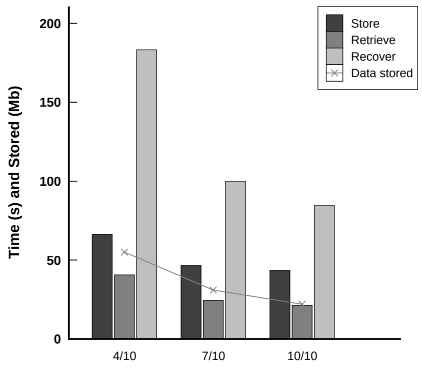

Figure 8: Operation time for 20MB at various encodings

Figure 7 shows the time required to perform our basic system operations for 100 MB of data on a system with 11 nodes. A node can place data on the system at the rate of 100 MB in 219 seconds, and retrieve it in 90 seconds. Recovery of the data after a local disk failure completes in just under six minutes. Recover is slower than the basic retrieve operation because it performs additional tasks—fetching *StoreList*s from all nodes and reconstructing and storing local BAR-B metadata.

Figure 8 shows the performance of our system under a range of encoding parameters when storing a 20 MB file on a system with 11 nodes. Each group of bars represents a choice of encoding parameters. As the ratio gets closer to one, the total amount of stored data stored on the system (indicated by the line) diminishes and the performance of the system increases. Storing a 20 MB file when encoded at 7 out of 10 (7/10) transmits approximately 31 MB of encoded data at 0.67 MB/s. The corresponding retrieve operation operates at 1.2 MB/s. Overall, the additional cost required for utilizing 7/10 encoding as compared to 10/10 is modest.

Figure 9 shows the effects of loading the system with multiple nodes storing or retrieving at the same time. The experiment itself records the time required when the specified number of nodes each store or retrieve of a single 20 MB file using the 7/10 encoding. When all nodes are active each node sees a modest reduction in throughput (from 0.67 MB/s to 0.54 MB/s) but the aggregate sys-

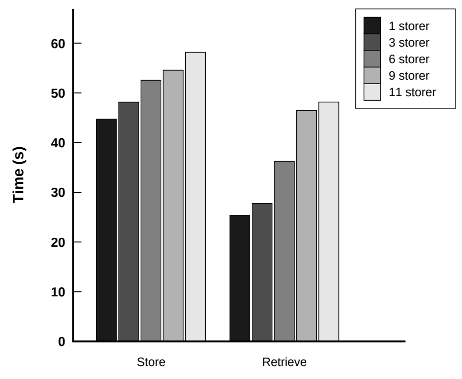

Figure 9: Concurrent operations

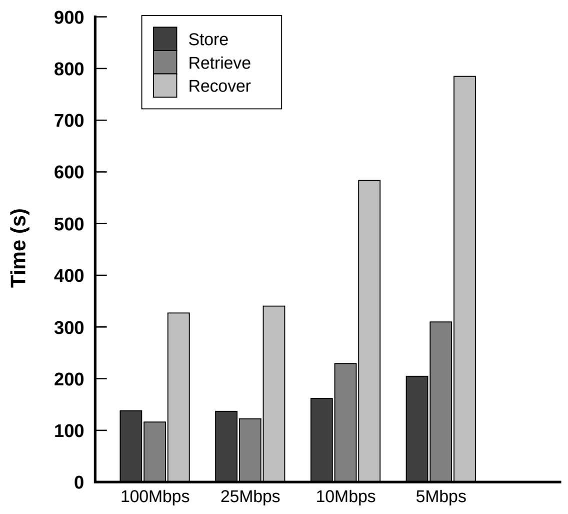

Figure 10: Operation under different network conditions

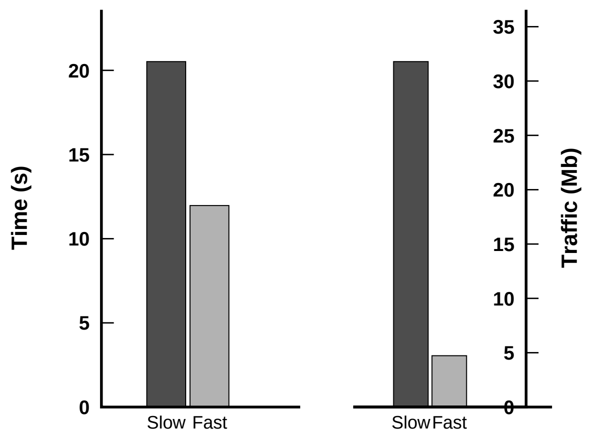

Figure 11: Impact of the fast path optimization

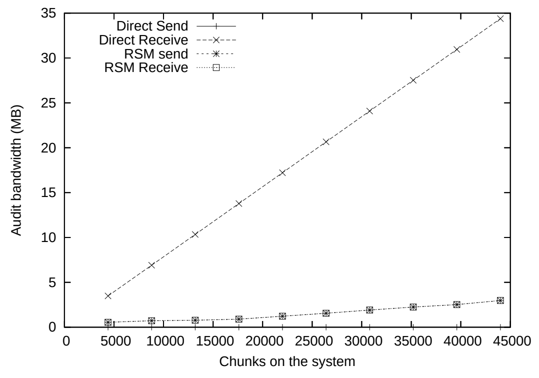

Figure 12: Cost of audit as capacity grows

tem throughput grows to 5.86 MB/s.

To this point, we have shown experiments run on a 100 Mbps LAN. Figure 10 shows the time required to store, recover, and retrieve a 20 MB file on Emulab machines with connections of 100 Mbps, 25 Mbps, 10 Mbps, and 5 Mbps, and round-trip latency of 26ms. At bandwidths of 10 Mbps and 5 Mbps, the network becomes a limiting factor and performance falls with network bandwidth. The baseline 100 Mbps store is slower than that found in Figure 8 due to the higher latency and slower machines on the Emulab site.

Figure 11 illustrates the effect of the "fast path" (Section 6.2) optimization on time and bandwidth. In each pair, the first bar shows a measurement of the unoptimized system, and the second bar shows the version with the fast path. For a 2 MB file encoded at 7/10, the fast path cut the duration of the store operation by 40% and reduced the traffic by a factor of five. Larger files would see even greater relative improvement.

Figure 12 shows the bandwidth required to perform an audit of an 11 node system. The bandwidth required is plotted against the number of chunks stored on the system. The "Direct Send/Receive" bandwidth lines correspond to the exchange of *OwnList*s and *StoreList*s. The number and size of requests and replies sent through the RSM is constant, but the RSM bandwidth consumed during an audit increases as the duration of the audit increases because the RSM completes additional (empty) instances. 38 MB of traffic is necessary for audit when the system stores 44000 chunks, which corresponds to up to 40 GB of storage or about 28 GB of backup files encoded at 7/10.

9. RELATED WORK

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

Byzantine agreement [30] and Byzantine fault tolerant state machine replication have been studied in both theoretical and practical settings [6, 9, 26, 48, 55]. Our work is clearly indebted to recent research [2, 10, 36, 52, 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 [10] 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 [25] has a long history in the economics literature [4, 28, 40] and has recently become of general interest in computer science [3, 20, 22, 23, 44, 47, 61]. Protocol and system designers have used game theoretic concepts to model behaviors in a variety of settings including routing [21, 58, 59], multicast [42], and wireless network [61]. 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. [59, 60] recognize the need for a model that includes both Byzantine and rational nodes, but their protocols address only the latter. Nielson et al [43] 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) [19] 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 [13, 16] commonly rely on informal reasoning. Bittorrent [13] 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 [60]. Other systems use audits [41] or witnesses [39] 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 [2, 5, 15, 16, 31, 49, 53]. The backup systems proposed in [5, 15] rely on the assumption that all non-faulty nodes behave correctly. Samsara [16] and Lillibridge et al. [31] 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 BAR-B, the first cooperative backup service to tolerate both Byzantine users and an unbounded number of rational users.

11. ACKNOWLEDGMENTS

The authors would like to thank Alfredo Di Tillio and Peter Stone for help with the subtleties of game theory and the whole LASR group for interesting conversations and help with early drafts.

12. REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

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

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

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

[27] 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.

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

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

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

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

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

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

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

[35] D. Malhotra. Making threats credible. *Negotiation*, 8(3), Mar. 2005.

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

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

[38] 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.

[39] 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.

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

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

[42] 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.

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

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

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

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

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

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

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

[50] 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.

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

[52] 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.

[53] 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.

[54] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. Tcp congestion control with a misbehaving receiver. *SIGCOMM Comput. Commun. Rev.*, 29(5):71–78, 1999.

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

[56] F. B. Schneider. *Distributed Computing* (Editor: Sape Mullender), chapter 2, *"What Good are Models and What Models are Good?"*, pages 17–26. ACM Press, second edition, 1993.

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

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

[59] J. Shneidman and D. C. Parkes. Specification faithfulness in networks with rational nodes. In *Proc. 23rd PODC*, pages 88–97. ACM Press, 2004.

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

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

[62] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: A mechanism for background transfers. In *Proceedings of the 2002 USENIX Operating Systems Design and Implementation (OSDI) conference*, Dec 2002.

[63] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th OSDI*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[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, Oct. 2003.

APPENDIX

A. COMMUNICATION PRELIMINARIES

Communication in the BAR model is handled through a collection of special mechanisms. At the core of these mechanisms is a BART channel which handles message resends on unreliable links in an incentive compatible manner. The second component of these mechanisms is the message queue infrastructure. The message queue infrastructure enforces a tit-for-tat communication policy between nodes. Under this policy node a will send messages to node b only as long as b has previously sent appropriate messages to a. This tit-for-tat policy is important in implementing predictable communication patterns in our protocols.

A.1 BART channel

We instrument a bidirectional BART channel between two nodes as a pair of modified one way TCP connections [3]. The underlying network is unreliable – messages may be dropped or reordered. We assume that the higher level protocols using the channel provide some benefit to rational node a when a successfully delivers a message to non-Byzantine node b.

Unreliable networks are subject to message loss and reordering. We address the problem of message loss by constructing a variation of a one way TCP connection. We address issues associated with reordering through consideration of well formed messages.

A.1.1 One way channel

Unreliable networks are subject to message loss and reordering. In this section we address the issue of message loss and a TCP based resend policy.

TCP employs a resend policy based on local timeouts [62]. The policy itself is straightforward and easy to understand. Each message m sent by node a to node b must be explicitly acknowledged.

In a bidirectional TCP connection, the acknowledgement can be included in the next message sent from b to a. As long as a has not received some message m_{i+1} from b, a resends m_i every *round-trip-time* time period. Similarly, b resends m_{i+1} every *round-trip-time* time period until receiving m_{i+2} from a. Since m_{i+1} serves as the acknowledgement to m_i and m_{i+2} acknowledges m_{i+1}, rational node b can save work by not resending m_{i+1} and instead relying on node a to resend messages as appropriate. Unfortunately a follows similar reasoning and does not resend m_i, resulting in the resend mechanism failing entirely.

We address the above issue by restricting our attention to one way TCP connections. This restriction allows us to make a clear demarcation between the sender and acker on the connection and we can state by fiat that the sender is responsible for resending messages. Since the sender receives benefit when the message is received by a non-Byzantine node, the sender has sufficient incentive to continue sending as long as he believes the acker is non-Byzantine.

The TCP Daytona optimistic ack attack [54] exposes a flaw in allowing ackers to send ack messages which are not bound to the message being acknowledged. In the original optimistic TCP Daytona attack, a client aggressively acks packets in order to increase the TCP window and induce the sender into devoting an unfair amount of bandwidth to that client. In the context of a BAR connection, a rational client could proactively generate ack messages

[3]In a one way TCP connection, exactly one node sends data and the other node only sends acknowledgement for data received.

```
run on node $a$ to send messages to $b$:
  mcount := 0;  // message sequence number
  queue := $\emptyset$;  // FIFO set of pending messages
  $curr$ := null; // message currently being sent
  enabled := true; //flag indicating whether messages should be sent

  on timeout:
     send($curr$);

  on rcv($ack_m$):
    if $curr$ = $m$ then
       disable timer;
       $curr$ := nextMessage();
       send($curr$);

  send($msg$):
     if enabled then
       $curr$ := $msg$;
       send $msg$ over network;
       start timer;

  reliable−send($msg$):
    $m$ := $\langle msg, mcount++\rangle$;
    queue.append($msg$);
    if $curr$ = null then
       $curr$ := nextMessage();
       send($curr$);

  nextMessage():
    if queue.empty() then
       wait for queue.hasElement();
    return queue.removeFirst();

  disable():
    enabled := false;

  enable():
    enabled := true;

run on node $b$ to receive messages from $a$:
  $next$ := 0; // sequence number of next message to receive
  enabled := true;   // flag indicating whether acks should be sent

  on rcv($\langle msg, i\rangle$):
     $m$ := $\langle msg, i\rangle$;
     if $next > i$ then
        discard $m$;
        return;
     if $next = i \vee next = i - 1 \wedge$ enabled then
        send $ack_m$ over network;
     if $next = i$ then
        $next$ := $i + 1$;
     if $next < i$ then
        disable()
     process($msg$);

  disable():
    enabled := false;

  enable():
    enabled := true;

  process($msg$):
     // pass message to higher level protocol
     // call rcv of higher level
```

Figure 13: BAR one way channel

to prevent the sender from resending dropped messages. We address this issue in the same fashion as [54], by introducing a nonce. Our nonce consists of a hash of the message being acknowledged.

In summary, we construct a one way connection between two nodes in which one node is designated the sender. Messages sent over the connection receive a unique non-decreasing sequence number. The sender is responsible for inserting new messages onto the connection and resending the message until it receives an appropriate ack from the acker; message m_i should be sent only after the receipt of $ack_{m_{i-1}}$. Acks are uniquely tied to a specific message through a nonce consisting of the hash of the message being acknowledged. This mechanism is sufficient to provide the following guarantees if neither node behaves in a Byzantine fashion:

- m_i will never be sent before $ack_{m_{i-1}}$
- ack_{m_i} will never be sent before m_i
- m_i will never be sent after m_{i+1}
- ack_{m_i} will never be sent after $ack_{m_{i+1}}$

The resend policy is successful as long as both nodes are non-Byzantine and the sender receives sufficient benefit from delivering message m to a non-Byzantine node.

Messages received out of order are considered to be malformed and are immediately dropped. The pseudocode for one way channels can be found in Figure 13. As long as non-Byzantine node a believes node b to be non-Byzantine, the higher level protocols provide sufficient benefit to a for having its message received by b, and rational nodes believe that if no ack has arrived by *round-trip-time* from a non-Byzantine node then a message has been dropped, the following theorems hold:

THEOREM 4. *If a is the sender, then a will resend m_i if a does not receive ack_{m_i} within round-trip-time.*

PROOF. It is assumed that a believes b to be non-Byzantine, implying that a believes b will follow the protocol directly. It is also assumed that a believes the lack of response by b indicates a message has been dropped. Since a receives sufficient benefit when b receives the message, a will resend. □

THEOREM 5. *If a is the acker and receives m_i from b for the first time or at least round-trip-time after the last time m_i was received, then a will send ack_{m_i} to b.*

PROOF. By assumption a considers b to be non-Byzantine, so b will stop sending m_i only upon receiving ack_{m_i} from a. Since a incurs cost from processing m_i, it is in a's best interest for b to stop sending m_i. If the receipt of m_i marks the first time a received m_i, then a must send ack_{m_i} in order to stop b from resending. If the receipt of m_i is at least *round-trip-time* after the previous receipt of m_i, then a considers it likely that a message has been dropped and again must resend ack_{m_i} to stop b from resending m_i indefinitely. In both cases, a incurs less cost by sending ack_{m_i} and stopping b from resending m_i. □

In addition to providing the above properties, the one way channel can be unilaterally disabled to save on future costs when the node at the other end is believed to be Byzantine.

A.1.2 Two way channel

In order to allow nodes a and b to both send and receive messages to each other, we must construct a bidirectional channel. We build our BART bidirectional channels from two one way channels. Pseudocode for the bidirectional channel is shown in Figure 14. The bidirectional channel provides the same guarantees as

```
run on node $a$ to coordinate communication with $b$:
  sendChannel   // one way channel for sends
  rcvChannel    // one way channel for receives
  queues        // collection of message queues

  send($msg$):
    sendChannel.reliable—send($msg$)

  rcv($msg$):
    mq := msg queue expecting $msg$
    if mq = null then
       disableSend()
    else
       mq.process($msg$)
       if mq.completed() then
          queues.remove(mq)

  register($mq$):
    queues.add($mq$)

  disable():
    sendChannel.disable()
    rcvChannel.disable()

  enable()
    sendChannel.enable()
    rcvChannel.enable()

  disableSend():
    sendChannel.disable()
```

Figure 14: Bidirectional BAR channel

the underlying one way channels, while facilitating the coordination between messages sent by a and b.

In addition to the *send* and *receive* methods, the two way channel exports operations to associate message queues with the channel (*register(), remove()*) and disable communication to a node that is considered faulty by higher levels of the protocol (*enable(), disable(), disableSend()*).

A.2 Message queue infrastructure

The message queue implements, in an incentive-compatible manner, a reliable channel with a simple local retaliation policy of "If you don't talk to me, then I won't talk to you". The pseudocode implementing this policy is shown in Figure 15.

Our description of the message queue is built over the BART *two-way-channel* described in Figure 14 and focuses on the implementation of a tit-for-tat policy.

A.3 Message queue tit-for-tat

If we say that two nodes, a and b, communicate through a *message queue* that means that both nodes have an instance of the message queue object (it's not a distributed object). The most important methods of this object follow.

- send(message)
- expect(predicate)
- receive(predicate)

The first two calls return immediately. The message queue will send the messages passed to the send() method, but only after it has received messages that are expected (indicated through the expect() function). The receive(predicate) function extracts the message which was previously expected and received. Consider for example some hypothetical protocol in which two nodes answer each others queries. The protocol for node a, using message queue mq, could be similar to the pseudocode below.

```
mq.send("QUERY: 1+1=?")
mq.expect(msg that starts with REPLY)
msg := mq.receive(msg that starts with REPLY)
mq.expect(msg that starts with QUERY)
qry := mq.receive(msg that starts with QUERY)
answer := compute_answer(qry)
mq.send("REPLY: "+answer)
```

```
201  class message—queue:
202
203    public message—queue(destination):
204      ready=1
205      dest=destination
206      readySet = new HashSet()
207      successors = new Set();
208      queue = new Queue();
209      channel[destination].register(this)
210
211    public send(msg):
212      queue.enqueue(msg)
213
214    public expect(predicate):
215      queue.enqueue(predicate)
216
217    public msg receive(predicate):
218     msg = ⊥
219     while (msg == ⊥)
220       msg = readySet.get( predicate )
221     readySet.remove(msg);
222     return msg
223
224    public link(nextQueue):
225      successors.add( nextQueue )
226      if (!is—really—done): nextQueue.ready——
227
228    public done():
229      queue.enqueue(”done”)
230      check—send()
231
232    joinGroup(agroup):
233      assert(agroup ==null)
234      group := agroup
235      agroup.onestarted()
236
237    public leaveGroup(): // needs to be atomic
238      if( !is—really—done && group != null )
239          group.onedone()
240      group := null
241
242    public is—enabled():
243      return true iff ((ready >0)
244        and (group==null or group.previousGroup == null or
245             group.previousGroup.pending==0))
246
247    public is—done():
248      return is—really—done
249
250    public process(msg)
251      wait until is—enabled()
252      predicate := queue.getFirstMatchingExpect(msg)
253      if ( predicate == null ) return
254      if (predicate != queue.top() || isInconsistent(msg) ):
255        channel[msg.sender].disableSend()
256      queue.remove( predicate )
257      readySet.add( predicate , msg )
258      check—send()
259
260    protected check—send():
261      if( !is—enabled() ) return
262      while (queue.top() instanceof message
263      or queue.top()==”done”):
264         x := queue.dequeue()
265         if (x==”done”):
266           really—done()
267         else:
268           channel[x.receiver].send(x)
269
270    protected really—done():
271      is—really—done := true
272      foreach successor in successors
273         successor.ready++
274      if (group!=null):
275         group.onedone()
276
277    public unlink(nextQueue):
278      if( successors.contains(nextQueue) && !is—really—done )
279        nextQueue.ready++
280        successors.remove(nextQueue)
281
282    protected static boolean isInconsistent( msg ):
283      // check to see if the current message along with some previously
284      // received messages reveal that the sender has
285      // deviated from the protocol.
286      // return true if the received messages cannot be sent
287      // by a node following the protocol;
288      // return false otherwise
289
290  class group:
291
292    public link(nextGroup):
293      assert(nextGroup.previousGroup==null)
294      nextGroup.previousGroup := this
295
296    public onedone(peer):
297      pending——
298
299    public onestarted(peer):
300      pending++
```

Figure 15: Message queues

The message queue mechanism ensures that if node b does not answer the first query, then it will not get an answer from a either.

This mechanism is implemented by putting both the messages to be sent and the predicates to check incoming messages on a single queue (Figure 15, lines 212 and 215). Predicates are removed when the corresponding message is received (line 256), and outgoing messages are only sent when there is no predicate ahead of them in the queue (function *check-send()*, lines 260-268).

A.4 Message queue linking

The message queue offers the ability to link several message queues together so they behave like a single, longer message queue. Our protocol uses this function so that different threads communicating with the same peer can have their own message queue (thus imposing a known order of these messages even though the thread execution may be interleaved).

To link message queues a and b, one uses the following call: a.link(b) (line 224). The messages on queue b will not be sent until the last message in a was sent. The function done() is used to indicate when message queue a ends. The message queue will report that it is done as soon as it has sent all the messages and satisfied all the expects before “done”. Consider the following example:

```
mq1.link(mq2)
mq1.send("QUERY: 1+1=?")
mq1.expect(msg that starts with REPLY)
msg := mq1.receive()
mq1.done()
mq2.expect(msg that starts with QUERY)
qry := mq2.receive()
answer := compute_answer(qry)
mq2.send("REPLY: "+answer)
```

In the example above, the two message queues $mq1$ and $mq2$ are linked to provide the same functionality as the previous example. In addition, the function mq1.is-done() can be called to determine whether the answer to the “1+1” query was received.

A.5 Message queue groups

Linking message queues does not fundamentally change the way they work. But the grouping we describe here does, because it allows a message queue to be blocked until *several* other message queues are done.

We introduce a new object, the Group. It has a single method, link(), that links groups together. Message queues can join a group by calling joinGroup(group). If groups g and h are linked by calling $g.link(h)$, then the message queues that are part of group h will not send messages to the network until all message queues that are part of group g are done (done is defined exactly the same as for linking between message queues).

This functionality is implemented through the Group class (lines 290-300), the function joinGroup (line 232) and the function is-enabled (line 242). Intuitively, the purpose of group linking is to provide a higher-level tit-for-tat policy, where we can think of each message queue as a transaction or a remote procedure call: group linking ensures that other nodes cannot participate in new transactions unless they have completed all the transactions that were initiated in the previous group.

A.6 Message queue sets

Fault-tolerant protocols often use quorum-based communication, in which nodes communicate with more than one node at a time. In order to provide quorum-style communication primitives, we introduce *message queue sets*: these objects contain a message queue for

```
instance-ctor(ins):
  for every peer p:
    create messge queue group instance[ins][p]
    if (ins>0):
      instance[ins-1][p].link(instance[ins][p])
  for t in 0 .. (n-1):
    turn-ctor(t)
  notification-ctor()

instance-run():
  call instance-ctor() for the next instance
  call turn-run(0) on a new thread
  notification-run()

notification-ctor():
  create message queue set MQS to all
  create message queue set MQE to all
  MQS.joinGroup( instance[ins] )
  MQE.joinGroup( instance[ins] )
  MQE.expect(decided, $nv$, $\vec{a}$, $\vec{b}$) from all
  MQE.receive(decided, $nv$, $\vec{a}$, $\vec{b}$) from all
  on receiving (decided, $nv$, $\vec{a_j}$, $\vec{b_j}$) from j:
    for each message-queue mq to $j$ in MQ[t]:
           $t > min\{turn(a_j), turn(decision)\}$
      if mq has not sent or received a message:
        mq.leaveGroup()

notification-run():
  // ensure that everyone forwards the decision messge
  wait until
  we receive (decided, $nv$, $\vec{a}$, $\vec{b}$) through MQE:
     then decide($nv$, $\vec{a}$, $\vec{b}$)
  if( !sender ):
    MQE.expect( untimely-penance ) of size
          untimely[j]*penance from each j != sender
    MQE.receive( untimely-penance ) of size
          untimely[j]*penance from each j != sender
  MQE.done()

decide($nv$, $\vec{a}$, $\vec{b}$):
  if (decided != ⊥):
    if( turn(decided) < turn( $\vec{a}$ ) ):
      MQ[turn( $\vec{a}$ )].link( MQS )
      MQ[turn(decided)].unlink( MQS )
      decided := ($nv$, $\vec{a}$, $\vec{b}$)
    return
  MQ[turn( $\vec{a}$ )].link( MQS )
  decided := ($nv$, $\vec{a}$, $\vec{b}$)
  MQS.send(decided, $nv$, $\vec{a}$, $\vec{b}$) to all
  if( (nv != ⊥) and (not sender) ): // we decided on the sender's value
      MQS.send(untimely-penance ) of size
          nv.untimely[i]*penance to all but the sender
  if( sender and proposed value was decided ):
    untimely[j] = 0 for all j
  MQS.done()
  call instance-run() for next instance
```

Figure 16: IC-BFT TRB, instance level functions

each other node in the system, and allow grouped communication using the following commands.

- send($message$) to $nodes$
- expect($predicate$) from $nodes$
- receiveQuorum($predicate$)

The first two functions simply delegate to the underlying message queues. The third function calls receive($predicate$) on all message queues and returns as soon as a quorum of message queues have delivered an answer.

For convenience, we also define

- link(MessageQueueSet)
- unlink (MessageQueueSet)

These functions link and unlink the corresponding message queues (to same peer) with each other.

B. TERMINATING RELIABLE BROADCAST

Each node in the RSM participates in a series of TRB instances, to decide on the operations to be executed on the RSM.

Each TRB instance has a distinguished sender node, which proposes a command to be executed. Nodes are chosen to be the sender in a round-robin fashion, so that each node can benefit from having its operations executed by the RSM.

```
turn-ctor(t):
  if myLeaderTurn(t)
   message queue set MQ[t] = set of message queues to all but the sender
  else
   message queue set MQ[t] = set consisting of message queue to
                       the leader (to sender if t==0)
  MQ[t].joinGroup(instance[ins])
  if (t>0):
    MQ[t-1].link(MQ[t])
    if myLeaderTurn(t):
      expect-election(t)

turn-run(t):
  // sender participates only in the first turn
  if( sender and t > 0 ):
    return
  if (t>0) and (decided==⊥): turn-ctor(t+(n-1))
  if (myLeaderTurn(t)): turn-leader(t)
  else:
     if (decided == ⊥ or decided.turn >= t ): turn-follower(t)
  MQ[t].done()

turn-leader(t):
  if( not sender )
    start timer timeout: if it fires then
      call turn-run(t+1) on a new thread
  if (t>0):
    pol := receive-election(t)
  else
    pol := ⊥
    proposal := (value,untimely[])
  if (pol=="done"): return
  ($nv$, $\vec{a}$, $\vec{b}$) := leader-three-phase-commit(t,proposal)
  MQ[t].send(ack) to all
  stop timer timeout
  if (showsChosen($\vec{b}$)): decide($nv$, $\vec{a}$, $\vec{b}$)

turn-follower(t):
  start timer timeout: if it fires then
    call turn-run(t+1) on a new thread
    send-election(t)
  follower-three-phase-commit(t)
  MT[t].expect(ack) from leader
  ack := MQ[t].receive(ack) from leader
  stop timer timeout
  if timer timeout fired more than $avg\_latency + window$ ago:
     untimely[ leader-of(t) ]++

leader-of(t):
  // determine which node is leader for a given turn
  if (t==0): return 0
  else: return ((t-1)%(n-1))+1

latest($\vec{r}$, $t$):
  If all $r_j \in \vec{r}$ have $r\_j.\vec{a} == \perp$ then
    return $SF$
  else
    return $r_j.val$ for $r_j \in \vec{r}$ with the latest $r_j.\vec{a}$

showsChosen($\vec{b}$):
  return true if there's a quorum who accepted the value

myLeaderTurn(t):
  return (id == leader-of(t))
```

Figure 17: IC-BFT TRB, turn level functions

```
send—election(t):
  // send a message to elect the new leader
  MQ[t].send( $\langle$set-turn, $t\rangle_i$ ) to leader—of(t)

expect—election(t):
  MQ[t].expect( $\langle$set-turn, $t\rangle_j$ ) from every $j \neq 0$

receive—election(t):
    $t_r$ := now()
    for every node $j$ from which we do not receive the expected
    set—turn message between $t_r - window$ and $t_r + window$:
        $untimely[j]$ := $untimely[j] + 1$
    wait until:
      $pol$ = MQ[t].receiveQuorum( $\langle$set-turn, $t\rangle_j$ )
      we receive a quorum $pol$ of messages from MQ[t]:
        return $pol$,
      or decide($nv, \vec{a}, \vec{b}$) is called and received at least one set—turn
                message from others:
        MQ[t].send (junk), worth 3 messages, to all who have sent the
                set—turn message.
        MQ[t].expect (junk—ack), worth 3 messages, from all who have sent
                the set—turn message.
        MQ[t].receive (junk—ack), worth 3 messages, from all who have
                sent the set—turn message.
        return "done"

leader—three—phase—commit(t, proposal):
  // 1. read the old values
  if( t $>$ 0 )
    MQ[t].send (read, $t, pol_t$) to all except the sender
    MQ[t].expect $r_j = \langle$read-ack, $t, val, \vec{a}, padding\rangle_j$ from all j other than
            the sender
  wait until:
    $\vec{r}$ := MQ[t].receiveQuorum( $\langle$read-ack, $t, val, \vec{a}, padding\rangle_j$ )
      we receive a quorum $\vec{r}$ of messages from MQ[t]:
    or decide($nv, \vec{a}, \vec{b}$) is called:
     MQ[t].send (junk), worth 2 messages, to all
     MQ[t].expect (junk—ack), worth 2 messages, from all
     MQ[t].receive (junk—ack), worth 2 messages, from all
     return "done"
  // 2. get the agreement vector
  if( t $>$ 0 )
    $nv$ := pad(latest($\vec{r}$))
  else
    $nv$ := pad($proposal$))
  $s$ := hash($nv$)
  MQ[t].send (agree, $t, nv, \vec{r}$) to all except sender
  MQ[t].expect $\langle$agree-ack, $t, s\rangle_j$ from all j
  wait until:
    $\vec{a}$ := MQ[t].receiveQuorum($\langle$agree-ack, $t, s\rangle_j$ )
      we receive a quorum $\vec{a}$ of messages from MQ[t]:
    or decide($nv, \vec{a}, \vec{b}$) is called:
     MQ[t].send (junk) , worth 1 message, to all
     MQ[t].expect (junk—ack) , worth 1 message, from all
     MQ[t].receive (junk—ack) , worth 1 message, from all
     return "done"
  // 3. update the state of the acceptors
  MQ[t].send (write, $t, nv, \vec{a}$) to all except sender
  MQ[t].expect $b_j = \langle$write-ack, $t, max\_pol_j\rangle_j$ from all j
  wait until:
    $\vec{b}$ := MQ[t].receiveQuorum( $\langle$write-ack, $t, max\_pol_j\rangle_j$ )
      we receive a quorum $\vec{b}$ of messages from MQ[t]:
    or decide($nv, \vec{a}, \vec{b}$) is called:
      return "done"
  while failed($\vec{b}, t$) and waiting won't cause a penance:
    wait for more answers from MQ[t], put them in $\vec{b}$.
  return ($nv, \vec{a}, \vec{b}$)

follower—three—phase—commit(t):
  l := leader—of(t)
  if (t>0):
     MQ[t].expect((read, $t, pol_t$) or junk) from l
     m := MQ[t].receive((read, $t, pol_t$) or junk)) // start atomic1
      if (m is junk):
         // verify the size of the junk is worth 3 messages
         // wait for receiving the decided message on MQE from l
        MQ[t].send (junk—ack), worth 3 messages.
         return
     $max\_pol := \max(max\_pol, pol_t)$
     MQ[t].send $\langle$read-ack, $t, m\_val, \vec{m\_a}, padding\rangle_i$ to l
  MQ[t].expect (agree, $t, nv, \vec{r}$) or junk from l // end atomic1
  m := MQ[t].receive((agree, $t, nv, \vec{r}$) or junk ) // start atomic2
  if (m is junk):
    // verify the size of the junk is worth 2 messages
    // wait for receiving the decided message on MQE from l
     MQ[t].send (junk—ack), worth 2 messages.
     return
  MQ[t].send $\langle$agree-ack, $t, hash(nv)\rangle_i$ to l
  MQ[t].expect (write, $t, \vec{a}$) or junk from l // end atomic2
  m := MQ[t].receive((write, $t, \vec{a}$) or junk ) // start atomic3
  if (m is junk):
    // verify the size of the junk is worth 1 message
    // wait for receiving the decided message on MQE from l
     MQ[t].send (junk—ack)
     return
  $mp$ := $max\_pol$
  if $t >= mp$:
     $(m\_val, \vec{m\_a}) := (nv, \vec{a})$
  MQ[t].send $\langle$write-ack, $t, mp\rangle_i$ to l // end atomic3
```

Figure 18: IC-BFT TRB, low level functions

Figures 16, 17 and 18 show the protocol for executing a single instance of TRB. The TRB protocol, similar to the Paxos protocol, proceeds turn-by-turn. In the first turn, turn 0, the sender tries to propose a value and write it to a quorum of nodes. If the sender is unable to write the values by the timeout then the remaining nodes start the next turn. For turns $t > 0$, a leader (chosen in a round-robin manner) tries to do a three-phase-write to complete the TRB instance. Again, if the leader for turn t fails to complete the turn by timeout, turn $t+1$ starts off and the leader for turn $t+1$ tries to complete the instance.

The timeouts for successive turns in a instance increase exponentially. Eventually, during a period of synchrony, a leader will be able to complete the instance in the turn before the timeout fires and will inform all other nodes about the decision.

Each node, on hearing about the decision, will (after checking the proof) decide on the value and inform all other nodes about the decision, if it has not already done so.

To ensure incentive-compatibility, we require incentives for nodes to send the messages according to the protocol. The incentives for sending and receiving the messages correctly is addressed by the message queues.

For each instance, there is a message-queue between each pair of nodes, for each turn. For every turn in an instance, there is a message queue between each pair of nodes[4]. A message queue to node j (from say node i) for turn t is represented (by node i) as MQ[t][j]. The set of message queues (from say node i) to all nodes, in turn t, forms a message queue set and is represented (by node i) by MQ[t].

In addition to having a message queue to every node per turn, there are two additional message queues, between each pair of nodes, used to send and expect the decision values. Every node expects to hear the decision value from every other node j on the message queue MQE[j], and sends the decision value to all the other nodes on MQS[j].

If the sender's value is decided upon, then nodes also send and expect appropriate amount of penance-penalty on these message queues.

Linking of message queues provides a mechanism to ensure that a node can only make progress w.r.t another node, if it satisfies all the expectations in the existing message queues. Specifically, the message queues to nodes in a turn t are linked to message queues in turns $> t$, so that if a node has not fulfilled its obligation in turn t, it will not receive messages for turns $> t$.

The message queues MQE and MQS are not linked to any other message queues in the instance, so a node can always hear about the decision, for a particular instance, from other nodes.

Linking of message queues groups, across instances, provides a way for long-term incentives. All message queues in a particular instance, to a node, form a message-queue-group. Message queue groups are linked across instances, this ensures that if there are any unfulfilled expects for a node in any message queue, that has communicated in instance l, then the node will not receive any messages for instances $> l$ unless the expect is fulfilled.

B.1 Deviation detection

Once a rational node r knows that a node a is Byzantine it no longer needs to send any message to a (method $disableSend()$). Nodes that visibly deviate from the protocol are considered to be Byzantine.

[4]This is actually a overkill; due to the communication pattern followed by the protocol, only the message queues communicating with the leader will be used

Deviations such as sending out-of-order messages can be handled by the message queue itself (line 254). However, there can be some application level detection that can be done. The *isInconsistent()* method (in line 282) looks for any visible inconsistencies in the messages that are received. If the inconsistencies reveal a node to deviating from the protocol, then the node is considered Byzantine and ignored thereafter.

Example:

Let R^a_t and W^a_t denote the events when node a receives the read and write messages for turn t. Also, let RA^a_t and WA^a_t denote the events when node a sends the read-ack and write-ack for turn t. We assume that the follower nodes immediately (atomically) send responses to the read/agree/write messages. Therefore, R^a_t and RA^a_t can be imagined to be the same events. Similarly, W^a_t and WA^a_t can also be considered the same events.

Note that a read-ack response, for turn t, contains the vector $\vec{a}$ corresponding to the latest written value at time RA^a_t (line 575).

B.1.1 Deductions from a write-ack

We say a write in turn t is successful at node a, if node a updates its m_val with the value of nv in response to the write message in line 593. A write for turn t is successful at node a if and only if a has not responded to a read with a higher turn number, before responding to the write (lines 574 and 592).

$$write\_succeded(a,t) \iff \forall t' : (t < t' \Rightarrow WA^a_t \rightarrow RA^a_{t'})$$

Any node that sees a write-ack message of the form $\langle write-ack, t, pol_t\rangle_a$ can infer that $t < t' \Rightarrow WA^a_t \rightarrow RA^a_{t'}$.

B.1.2 Deductions from a read-ack

Now consider a node's response to a read message. The protocol specifies that the read-ack message sent by a must include the vector $\vec{a}$ from a successful write at a with the highest turn seen so far. If node a responds with a read-ack for turn t with a vector $\vec{a}$ which is from turn t_0 then, it must be that:

$$\forall t' : write\_succeded(a,t') \wedge WA^a_{t'} \rightarrow RA^a_t \Rightarrow t' \leq t_0$$

B.1.3 Detecting a possible misbehavior

Now we demonstrate how these semantic checks can help detect additional deviations.

Suppose node a responds to a read for turn t_2 before it receives a write for turn t_1. The protocol specifies that the write for turn t_1 fail. If node a deviates from the protocol and makes the write for t_1 succeed at a then the deviation can be detected.

Without loss of generality, assume that t_2 is the earliest turn $> t_1$ to which a has responded before sending the write-ack message for turn t_1. Therefore the vector $\vec{a}$ included in the read-ack message for turn t_2 must be from a turn t_0 such that $t_0 < t_1$[5].

Thus, on receiving the read-ack message, a rational node can deduce that

$$\forall t' : write\_succeded(a,t') \wedge WA^a_{t'} \rightarrow RA^a_{t_2} \Rightarrow t' \leq t_0$$

Specifically, for $t' = t_1$, $\neg write\_succeded(a,t_1) \vee RA^a_{t_2} \rightarrow WA^a_{t_1}$.

However, on receiving the (successful write at a) write-ack message for turn t_1, nodes can deduce that

$$t_1 < t' \Rightarrow WA^a_{t_1} \rightarrow RA^a_{t'}$$

[5]It cannot be from t_1 because a has not yet received the write for turn t_1. It cannot be from any turn $> t_1$ because t_2 is the earliest turn $> t_1$ to which a responds

Specifically, for $t' = t_2$, $WA^a_{t_1} \rightarrow RA^a_{t_2}$, which contradicts the previous deduction based on the read-ack message, thus exposing that node a's misbehavior.

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 4. *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 $v == v'$. $\square$

LEMMA 5. *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 in any turn $t' > 0$, it needs to contain read-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 6. *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 6 (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. The previous two lemmas indicate that at most one value may ever be chosen. □

THEOREM 7 (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 8. *The protocol satisfies the conditions for TRB.*

PROOF.

- Termination is guaranteed by Theorem 7.
- Agreement follows from Theorem 6 and Theorem 7
- Integrity is assured because a leader cannot propose any arbitrary value. The expect in line 576 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.

□

LEMMA 7. *A non-Byzantine node will be eventually able to satisfy the expect of every other non-Byzantine node, which could cause the other node to ignore this node.*

PROOF. A non-Byzantine node only expects messages listed in Table 1. For each of the messages listed, we argue that a non-Byzantine node will eventually be able to send the message to fulfil the expect of the other non-Byzantine nodes.

- *decided:* Every non-Byzantine node will eventually decide, then it will send the decided message to all the other non-Byzantine nodes.
- *set-turn:* If a decision has not been reached earlier, then eventually the time out for the turn will fire and the non-Byzantine node will send the set-turn message.

  If a decision has been reached in an earlier turn, then it does not matter if the node does not send the set-turn message because message queues after the decision turn, which have not communicated, do not cause nodes to ignore the other node.
- *read/agree/write-acks:* These messages are sent in response to the read/agree/write messages and can be generated from the corresponding messages locally. Therefore a non-Byzantine will not get stuck in sending these messages.
- *junk/junk-ack:* This can be generated locally, without waiting for other nodes, therefore no correct node will get stuck because of these messages.
- *untimely:* Once a decision has been reached (which will happen eventually) this message can also be generated locally.
- *ack:* This message can be generated locally, therefore no node will get stuck not being able to send this message.
- *read:* If the decision has been reached in an earlier turn, then the node will eventually hear about the decision on MQE, and can bail out by sending a junk message.

  If the decision has not yet been reached, then eventually all the non-Byzantine nodes will send the set-turn message to this node, which will be sufficient to form a quorum and generate the appropriate read message.
- *agree:* If the decision has been reached in an earlier turn, then the node will eventually hear about the decision on MQE, and can then bail out by sending a junk message.

  If the decision has not yet been reached, then eventually all the non-Byzantine nodes will send the read-ack messages to this node, in response to the read message that it sent, which will be sufficient to form a quorum and generate the appropriate agree message.
- *write:* If the decision has been reached in an earlier turn, then the node will eventually hear about the decision on MQE, and can then bail out by sending a junk message.

  If the decision has not yet been reached, then eventually all the non-Byzantine nodes will send the agree-ack messages to this node, in response to the agree message that it sent, which will be sufficient to form a quorum and generate the appropriate write message.

□

THEOREM 9. *Liveness across instances, assuming Incentive Compatibility: No non-Byzantine node will be ignored by another non-Byzantine node indefinitely.*

PROOF. A non-Byzantine node i, following the protocol, will only ignore another node j

1. *permanently:* if j has revealed itself to be Byzantine by visibly deviating from the protocol (e.g. by sending messages out-of-order or sending messages that do not satisfy the structure of the expect).
2. *temporarily:* as long as an expect is yet to be fulfilled.

Firstly, no non-Byzantine node will be considered Byzantine by another non-Byzantine node, because the protocol only sends messages when the message is expected and the messages will be of the required format.

Moreover, by Lemma 7 a non-Byzantine node will eventually be able to satisfy the expect of all the other non-Byzantine nodes.

Hence a non-Byzantine node will not indefinitely ignore another non-Byzantine node for not fulfilling an expect. □

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 [25]:

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

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.

	not send	send different	diff. time
set-turn	Lemma 10	Lemma 16	Lemma 14
read	Lemma 10 and 36	Lemma 17	Lemma 31
read-ack	Lemma 10 and 36	Lemma 18	Lemma 34
agree	Lemma 10 and 36	Lemma 21	Lemma 32
agree-ack	Lemma 10 and 36	Lemma 19	Lemma 34
write	Lemma 10 and 36	Lemma 22	Lemma 33
write-ack	Lemma 10 and 36	Lemma 20	Lemma 34
decided	Lemma 10	Lemma 23	Lemma 24 and 25
junk	Lemma 10 and 35	Lemma 28	Lemma 29
ack	Lemma 10	Lemma 26	Lemma 27
untimely	Lemma 10	Lemma 15	Lemma 15

Table 1: Map of deviation to lemma

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 the requirement that must be met in an implementation: (a) The penance is larger than the benefit of sending a time-out message late

THEOREM 10 (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 11 and 8.

LEMMA 8. *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. □

Lemma 8 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 9. *If a rational node r is being ignored by a non-Byzantine node a, then r may not be able to write its value.*

PROOF. Sending a valid $write$ message requires the message to contain a quorum of *agree-ack* messages in $\vec{a}$. If the f Byzantine nodes, along with the non-Byzantine node a do not send the *agree-ack* message to r, then r can never gather a quorum of *agree-ack*s to generate a write message. Hence cannot send any write message.

Moreover, if the sender r does not send a valid $write$ message, then no further leader will be able to read the value and the TRB will decide $\perp$.

LEMMA 10. *If a rational node r knows that not sending some expected message m to non-Byzantine node s would cause s to ignore r, then r has incentive to send the message.*

PROOF. If s ignores r, then r will not receive any messages from s. In the worst case (for r), all f Byzantine nodes will also ignore r. Then, for instances where r is the sender, 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 r cannot gather enough messages to form a $write$ message, it will then be ignored by all non-Byzantine nodes. 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 other non-Byzantine to ignore it. ☐

LEMMA 11. *Rational nodes only send a message m to node j if j expects that message.*

PROOF. The queue protocol regards any node that sends a message which is not the current expected message as a Byzantine node. Therefore a rational node will only send a message m to a non-Byzantine node, if it is currently expected.

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. ☐

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. ☐

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. Moreover, the size of the messages are padded so a rational node will incur the same cost in the instance, irrespective of what is being decided. The total amount of work, therefore, is the same. ☐

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. 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 the decided message to arrive, if the decision has been reached, 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 446 and 512, sent in line 350) is intended to inflict additional cost onto nodes that are believed to be untimely. Each node, other than the sender, expects this message every instance for the appropriate size. If the message is not received, or is not of the appropriate size, then other nodes will ignore this node in future instances. Therefore rational nodes will not omit or 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 11), 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 567), 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 10 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. But, doing so can expose r to be deviating from the protocol and get ignored by other nodes, so a rational node will not send an older POL. □

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

PROOF. The agree message (sent in line 542) 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 10). 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 32. □

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 decided 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 delay sending the decided message.*

PROOF. Every other node expects a decided message from r and will not talk to r in future instances unless it receives the decided message. Further, since there is no hope of finding a cheaper decided message by waiting, a rational r will not wait in order to hear from a

Once a decision has been reached, it has been reached. Waiting longer is not going to change the decision, or the cost of sending the decision messsage.

However, by waiting longer, one could possibly hear that a decision had been reached earlier, which could potentially help in avoiding sending some extra messages for turns that have not yet communicated.

This, we argue, will not be the case because MQS is linked to MQ[decision.turn]. This ensures that the decision message will be sent only after all expects upto MQ[decision.turn] are satisfied. At this point, however, both the nodes have already communicated upto turns decision.turn and therefore will not be able to remove the expects for those turns even if they find out that a decision has been reached earlier. □

LEMMA 25. *There is no incentive for r to send the decided message early.*

PROOF. r is supposed to send the decided message to other nodes, as soon as they have fulfilled the obligations up to decided.turn. If r sends the decided message early, it gains nothing. On the contrary, it may stand to lose potential saving that it could have got if r were to hear about an earlier decision. □

LEMMA 26. *There is no incentive for r to send a different ack message.*

PROOF. The format of the ack message is fixed. Sending a ack message with different content is equivalent to not sending the ack message. From Lemma 10, it follows that a rational node r has no motivation to skip sending the ack messages. □

LEMMA 27. *There is no incentive for r to delay sending the ack message.*

PROOF. The format and size of the ack message is fixed and will not change by waiting longer. Therefore a rational leader, r, has no motivation to delay sending the ack message. □

LEMMA 28. *There is no incentive for r to send a different junk message.*

PROOF. The junk message is allowed to be anything. Only the size and number of signatures for the junk message matters. Moreover, if the size and number of signatures are wrong, then the message will not help to fulfil the expect and r will get ignored. □

LEMMA 29. *There is no incentive for r to delay sending the junk message.*

PROOF. The format and size of the junk message is purely dependent on whether it replaces which message and will not change by waiting longer. Therefore a rational node, r, has no motivation to delay sending the junk message. □

LEMMA 30. *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 31. *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 $junk$ message route instead of the normal three phase commit. We use the penance mechanism to balance the costs. $\square$

LEMMA 32. *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 453)—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. $\square$

LEMMA 33. *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. $\square$

LEMMA 34. *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. $\square$

LEMMA 35. *There is no reason why a rational node r should try to send read/agree/write messages instead of junk.*

PROOF. The junk message has the same cost as sending the remaining read/agree/write messages so there is no reason why a rational node should prefer to send these messages instead of the junk message. $\square$

LEMMA 36. *There is no reason why a rational node r should try to send junk instead of read/agree/write.*

PROOF. The junk message has the same cost as sending the remaining read/agree/write messages so there is no reason why a rational node should prefer to send junk messages instead of the read-/agree/write messages.

Also, sending a junk message has the semantics that a decision has already been reached in a previous turn. If this is not the case, then it is possible that the recipient may consider the node to be a Byzantine node. $\square$

C.4 Enlightening examples

The protocol in Figure 17 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 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 11. *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. $\square$

E. WORK ASSIGNMENT

Work assignment is introduced in Section 6. Here we add to that discussion by providing pseudocode and proof sketches.

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.

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.

```
// user calls this to submit a request
submit(query, target):
  _queries.enqueue((query, target))

// this is always running
run():
  while true:
     // ready for query
     (q,t) := _queries.dequeue(); // blocks if no query
     client_to_witness((q,t))
     // waiting for reply
     r := receive_from_witness()
     // handle reply
     if (r is a response from t):
        // deliver informs the user that a reply has arrived
        deliver(r)
        client_to_witness( summary(r) )
     else:
        // r is a time—out, witness has a POM
        POM := receive_signed_from_witness()
        deliver(POM)
        client_to_witness( summary(POM) )
```

Figure 19: Guaranteed response client protocol

```
client_to_witness( m )
    propose( m )


witness_to_client( msg, c )
    if( this == c ): deliver( msg )


witness_to_client_signed( msg, c )
// execute this code while processing the message in the decide
// function, just before line no 354
    MQS.send( sign(m) ) to c
    if ( this == c ): MQE.expect( sign(m) ) from all
```

Figure 20: Communication with the witness

E.1 Guaranteed-Response Protocol

We show the Guaranteed Response pseudocode in Figures 19, 21 and 22. The code matches the state diagrams in Figure 3. We show the code for a single slot, but in practice we use several slots, so there are several instances of the code running in parallel.

Provided that the application provides sufficient sanctions for nodes that cause *NoResponse*, the following theorem holds:

THEOREM 12. *If the witness node enters the request received state for some work w to rational node b then b will execute w.*

PROOF. If b does not execute w then b cannot send the (correct) response to the witness. Since the witness is guaranteed to behave correctly, the witness will generate a verifiable $NoResponse$ message when the timeout fires. Since, by assumption, rational nodes will take whatever actions necessary to avoid a verifiable $NoResponse$ message on their behalf b will execute w. □

Each node is allocated a constant number of slots. The strict transitions permitted within each individual slot requires that a previous request is completed before the next request using that slot may start. Our model of rationality specifies that nodes respond within $max\_response\_time$ or they are considered Byzantine.

```
// slot for client c
run():
  isByzantine[c]=false
  while (not isByzantine[c]):
    // empty state
    (query, target) := receive_from_client(c)
    // request received
    witness_to_client(query, target)
    wait until:
      either: reply := receive_from_client(target)
      or:     internal time out
    if (received a reply):
       // response received
       witness_to_client(reply, c)
    else:
       // time out
       isByzantine[target]=true
       witness_to_client("timeout",c)
       reply := sign("POM: NoResponse")
       witness_to_client_signed(reply, c)
    ack := receive_from_client(c)
    if (ack != summary(reply)):
       isByzantine[c]=true
```

Figure 21: Guaranteed response witness protocol

```
onReceive(qry):
  response := process(qry);
  client_to_witness(response);
```

Figure 22: Guaranteed response target protocol

E.2 Credible threats

The fast path optimization is presented in Section 6.2. We now argue that it is in the best interest of the rational nodes to follow the fast path.

LEMMA 37. *Rational clients will partake in the fast path optimization.*

PROOF. The fast path requires sending the request to 1 node. The slow path requires sending the same message through the state machine (requiring sending the message at least n times). The target in the request is faulty with probability f/n, so the expected cost of taking the fast path is $f + 1$ times the cost of sending the message while the expected cost of the slow path is n times the cost of sending the message. The fast path has lower expected cost so will be followed. □

LEMMA 38. *Rational targets will partake in the fast path optimization.*

PROOF. The fast path requires sending the response to 1 node. The slow path requires sending the same message through the state machine and results in sending the message at least n times. The requesting node is faulty with probability f/n, so the expected cost fo the fast path is $f+1$ times the cost of sending the message while the cost of the slow path is n times the cost of sending the message. The fast path has lower expected cost so is preferred by rational nodes. □

THEOREM 13. *Rational nodes will follow the fast path.*

PROOF. By Lemma 37 the client will follow the fast path and by Lemma 38 the target node will follow the fast path. Thus all rational nodes follow the fast path. □

E.3 Periodic Work Protocol

The periodic work protocol is a mechanism used by the application designer to specify general maintenance work that must be performed by individual nodes. The periodic work protocol functions by requiring each node to periodically submit the result of a work request to the witness. If the result of the work request is received before the period expires, the period is restarted and the request is repeated. When the witness is implemented using a RSM, the period can be described by passage of time or successful proposals to the RSM. Figure 23 shows the state transitions for the periodic work protocol. Figure 24 shows pseudocode for the protocol where work is expected every k successful proposals by the participating node.

THEOREM 14. *Rational nodes will follow the periodic work protocol.*

PROOF. Failure to send an expected message results in being ignored by the shunned node. If the shunned node is non-Byzantine, then the shunned node plus the f Byzantine nodes are sufficient to deny access to the underlying RSM. By assumption, the benefit of access to the RSM is larger than the costs associated with following the protocol faithfully. □

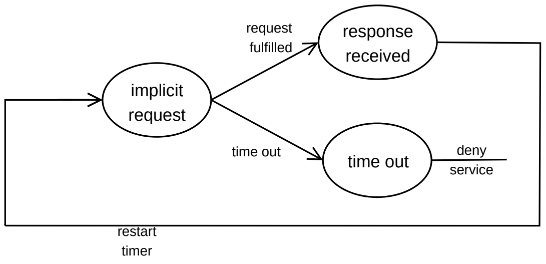

Figure 23: Periodic work protocol

```
run on replicated node $p$:
 Work work;
 long counter;
 long period;
 String target;

setup(Work w, long k, String peer):
  work := w;
  counter := k;
  period := k;
  target := peer;


on decision $d$ from target:
  counter := counter − 1;
  if $d$ is result of work then
    counter := period;
  if counter = 0 then
    isByzantine[target] = true
```

Figure 24: Periodic work protocol

E.4 Deterministic RSM Clock

The deterministic RSM clock (DRC) is at the core of the authoritative time service discussed in Section 6.4. The objective of the DRC is not to synchronize the local clocks of the nodes, but to provide a consistent global clock which can be used to order operations and define when events (as defined by state machine decisions) take place.

DEFINITION 3 (NON-DECREASING). *A clock is* non-decreasing *iff for time t_i returned before time t_j, $t_i \leq t_j$.*

DEFINITION 4 (RECENT). *A replicated clock is* recent *iff in periods of synchrony, the time t_i returned by the clock is no smaller than the value time proposed by a non-Byzantine sender in the last $2f + 1$ instances.*

DEFINITION 5 (IDENTICAL). *A replicated clock is* identical *iff for all local calls of the getTime() function on a replica between processing decision k and decision $k+1$ getTime() returns the same time t on all replicas.*

The protocol itself is very simple. The current time is computed by taking the maximum of the median of the timestamps of the $2f + 1$ most recent decisions and the previous deterministic time. Decisions of the underlying state machine (Section B) include a timestamp field which is set to the local time of the sender when the proposal is first made. When “no decision” is decided, then the time for that decision is defined to be the previous determinstic time. Pseudocode for the protocol is shown in Figure 25.

The following theorems provide the correctness argument for the clock protocol.

THEOREM 15. *The time returned by the DRC is* non-decreasing.

PROOF. At each decision the value of the deterministic clock is potentially updated. Line 13 guards the change to the persistent clock value, insuring that the clock is only changed if the new value is strictly larger than the old value. □

```
1001  run on replicated node $p$:
1002
1003   time $m$ // variable holding the current time
1004   decisions $S$ // variable holding set of decisions
1005
1006  getTime():
1007    return $m$;
1008
1009  on decision $d_k$:
1010      $S := S \setminus d_{k-(2f+3)}.time$
1011      $S := S \cup \{d_k.time\}$
1012      $m' := median(S)$
1013      if $m' > m$ then
1014          $m := m'$
```

Figure 25: Deterministic RSM clock protocol

```
bind(msg):
   rsm.propose(bind(msg));

on delivering bind(msg) from a:
   if (this == a):
       MQE.expect((sig,t)) from all
   t := DRC.getTime();
   sig := sign(t, hash(msg));
   MQS.send((sig,t)) to a;
```

Figure 26: Message binding protocol

THEOREM 16. *The time returned by the DRC is* recent.

PROOF. Lines 10 and 11 ensure that the time is computed over the $2f + 1$ most recent decisions. □

THEOREM 17. *The time returned by the DRC is* identical.

PROOF. The time is updated only when a decision is reached. Any call to getTime() between successive decisions returns the same value. □

THEOREM 18. *In periods of synchrony, rational nodes include the correct time in their proposals.*

PROOF. Rational nodes must always include some time in their proposals, otherwise they will be considered Byzantine by their peers. If they get a benefit speeding up or slowing down the DRC then they may consider sending an incorrect time value. However, the worst the Byzantine nodes can then do is be honest, and so the other times proposed in the $2f + 1$ proposals window will be correct, and the median value chosen for the DRC will in fact not be influenced by the actions of the rational node. Since rational nodes only deviate from the protocol if there is some benefit in it, the rational nodes will choose to follow the protocol. □

E.4.1 Message Binding Protocol

In order to bind a message to a time, node a submits the message to the *Message Binding Protocol* and proposes a *BindingRequest* to the RSM. The nodes in the RSM then send a a signature binding the message to the current authoritative time through the message queue architecture. Figure 26 shows pseudocode for the Message Binding Protocol.

Provided the overall application provides sufficient benefit for maintaining use of the state machine, the following theorem holds:

THEOREM 19. *The message binding protocol is incentive compatible.*

PROOF. When $bind(msg)$ is decided, a non-Byzantine proposing node expects a message of the form (sig, t) from all members of the state machine. If the expect is not fulfilled by node b then the proposing node will stop sending messages to b. Furthermore, if the time t is not correct for the decision, then the message is malformed and b will be ignored. These events both potentially result in $f + 1$ total nodes (the shunned non-Byzantine node and f Byzantine nodes) ignoring b, denying b the expected benefit from membership in the state machine. □