FullReview: Practical Accountability in Presence of Selfish Nodes

Amadou Diarra
Grenoble University
Grenoble

Sonia Ben Mokhtar
LIRIS CNRS
Lyon

Pierre-Louis Aublin
LIRIS CNRS
Lyon

Vivien Quema
Grenoble INP
Grenoble

***Abstract*—Accountability is becoming increasingly required in today's distributed systems. Indeed, accountability allows not only to detect faults but also to build provable evidence about the misbehaving participants of a distributed system. There exists a number of solutions to enforce accountability in distributed systems, among which PeerReview is the only solution that is not specific to a given application and that does not rely on any special hardware. However, this protocol is not resilient to selfish nodes, i.e., nodes that aim at maximising their benefit without contributing their fair share to the system. Our objective in this paper is to provide a *software solution* to enforce *accountability* on any underlying application in presence of *selfish nodes*. To tackle this problem, we propose the *FullReview* protocol. *FullReview* relies on game theory by embedding incentives that force nodes to stick to the protocol. We theoretically prove that our protocol is a Nash equilibrium, i.e., that nodes do not have any interest in deviating from it. Furthermore, we practically evaluate *FullReview* by deploying it for enforcing accountability in two applications: (1) SplitStream, an efficient multicast protocol, and (2) Onion routing, the most widely used anonymous communication protocol. Performance evaluation shows that *FullReview* effectively detects faults in presence of selfish nodes while incurring a small overhead compared to PeerReview and scaling as PeerReview.**

I. Introduction

Distributed systems have always been the scene of various software and hardware failures. These failures can have diverse sources such as the crash of machines, bugs, misconfigurations, as well as malicious attacks and users that deliberately tamper with their software to gain some benefit. These failures are especially difficult to deal with when the distributed system spans over multiple administrative domains (also referred to as MAD distributed systems) [2]. Examples of such systems include peer-to-peer systems, computer grids, network services (e.g., DNS), federated information systems and inter-domain routing.

Accountability, which refers to the ability to detect and expose node faults, is a promising paradigm to deal with these types of failures. In the last decade various solutions have been proposed to enforce accountability for specific applications (e.g., anonymous communication [8], online games [25], network storage [26], randomised systems [3], inter domain routing [15], virtualised systems [14]). While these solutions offer strong accountability guarantees, their usability is limited to the specific application domain for which they have been devised. Hence, generic solutions that are not tailored to a specific application have been proposed, some of which rely on trusted hardware (e.g., Trinc [20], A2M [7], Pasture [18]) while others are generic software solutions. Our work targets this second category of systems as they do not require users (worldwide) to acquire specific hardware. To the best of our knowledge, PeerReview [16] is the only protocol that falls into this category of systems. In this protocol, nodes log their interactions with other nodes in a *secure log*. This log is then periodically audited by a set of other nodes assigned by the system, i.e., the node's monitors. During their audit, the monitors verify that the monitored node did not tamper with its log and that the latter corresponds to a correct execution of the monitored protocol. An attractive result of PeerReview in addition to its wide applicability is that it provides two theoretical guarantees: completeness and accuracy. Informally, completeness refers to the ability to detect (eventually) all the observable faults, while accuracy refers to the ability to never accuse correct nodes of misbehaviour.

PeerReview works under the Byzantine failure model, i.e., a model where a majority of nodes are correct and where a fixed (known) proportion of nodes in the system can behave arbitrarily. While dealing with Byzantine nodes is important, it has been demonstrated that in open collaborative environments selfish nodes, also called free riders, constitute a real threat [1], [19], [11], [10]. Selfish nodes are nodes that tamper with their software (or download a tampered software developed by others) in order to benefit from the system without contributing their fair share to it.

In PeerReview, nodes are not encouraged to participate to the monitoring of other nodes, which makes it vulnerable to selfish nodes. Specifically, in presence of a proportion of selfish nodes, some nodes in the system can be unsupervised if all their monitors behave selfishly. As a result, these nodes can harm the system without being detected, breaking the completeness property of PeerReview. To measure the impact of this threat in practice, we deployed PeerReview for enforcing accountability in the following two protocols: SplitStream [6], an efficient multicast protocol and Onion routing [12], the most used anonymous communication protocol. Experiments show that in presence of 30% of selfish nodes, 54% and 85% of messages are lost using the first and the second protocols, respectively.

In this paper, we embrace the challenge of designing a selfish-resilient protocol for enforcing accountability in distributed systems and present the *FullReview* protocol. The

objective of *FullReview* is to force selfish nodes to participate in the monitoring of other nodes while they are executing a given protocol. To reach this objective, the first idea that one may have is to make monitors themselves accountable for their actions by applying PeerReview. We show in this paper that this is not possible because using PeerReview to monitor itself would require that each node's log contains the log of all the other nodes in the system, which is not scalable.

To overcome this problem, *FullReview* relies on a game theoretic approach to force selfish nodes to stick to the monitoring protocol. Specifically, *FullReview* is a complete redesign of the PeerReview protocol, in which we have embedded incentives in such a way that it is not in the interest of any node to deviate from the protocol, i.e., we prove that *FullReview* is a Nash equilibrium [23] (Section VII-D).

We implemented *FullReview* and used it to monitor the two protocols SplitStream and Onion routing. Performance evaluation performed on a cluster of 50 machines shows that *FullReview* is resilient to selfish nodes and that it incurs a reasonable overhead compared to PeerReview. Complementary simulations show that *FullReview* scales up to 1000 nodes.

The remaining of this paper is structured as follows. First, we present the related works in Section II. Then, we show the impact of selfish nodes in PeerReview and present our system model in Section III. Further, we present an overview of *FullReview* and its detailed description in sections VI and VII, respectively. Finally, we present the performance evaluation of *FullReview* in Section VIII and concluding remarks in Section IX.

II. Related works

Building robust distributed systems has been at the heart of many research efforts in the last decade. In this context, a new model called the Byzantine, Altruistic, Rational (BAR) model has been proposed [2]. This model considers three types of nodes: *Byzantine* nodes are nodes that can deviate arbitrarily from the protocol; *rational* nodes are nodes that deviate from the protocol if the performed deviation allows them to increase their own benefit according to a known utility function; *altruistic* nodes are nodes that always stick to the protocol. In this context, a protocol is said to be BAR-resilient if it tolerates a fixed amount of Byzantine nodes and an unlimited proportion of rational nodes. BAR-resilient protocols often combine game theory by adding incentives that encourage rational nodes to stick to the protocol and accountability techniques that expose Byzantine nodes in case of deviation. In the last years, various collaborative systems have been designed according to this model including protocols for spam resilient content dissemination [5], distributed file systems [2], video live streaming [22], [21], [13], anonymous communication [4] and N-party data transfer [24]. The process by which a new BAR-resilient protocol is designed usually involves the following steps: (1) define the utility function of rational nodes in the considered protocol; (2) list all the possible rational deviations according to the defined utility function; (3) for each identified deviation, propose incentives for rational nodes such that any deviation would engender a loss in the utility perceived by the deviating node and mechanisms that would catch the considered Byzantine deviation; (4) prove that the proposed protocol is a Nash equilibrium. The major limitation of this approach is that it has to be performed manually by a system expert, which is complex and possibly error prone. Furthermore, any modification in the original system requires to rethink the system as a whole, as the latter may introduce new rational or Byzantine deviations. Rational nodes in the BAR-model correspond to selfish nodes in our work.

A grail that security managers may dream of having is a way of automatically transforming a given protocol into a BAR-resilient protocol. Two solutions that go towards this direction have been proposed in the literature. First, Nysiad [17] allows the automatic transformation of a given protocol to a Byzantine resilient system. Nysiad reaches this objective by replicating each host using a variant of replicated state machines (RSMs). However, the resulting system does not deal with selfish nodes. Contrarily to Nysiad, PeerReview [16] allows to automatically detect all sorts of observable deviations, including both selfish and Byzantine deviations, that a node would perform in a given monitored protocol. PeerReview reaches this objective by using tamper evident logs and assigning monitors to nodes, which periodically assess the correctness of a node by comparing its log with a correct execution of the protocol obtained using a reference implementation. However, while PeerReview allows to deter faults in the underlying protocol to which it is applied, it does not detect deviations performed by nodes on its own protocol steps.

Our objective in this paper is to design the first generic protocol that deals with both selfish and Byzantine nodes on any underlying protocol.

III. Problem statement and system model

We present in this section an evidence that the PeerReview protocol fails to enforce accountability in presence of selfish nodes in Section III-A. We then present our system model in Section III-B.

A. Problem statement

Let us consider a system where nodes can be correct, selfish or Byzantine. As introduced in the previous section, correct nodes follow the protocol, Byzantine nodes can behave arbitrarily and selfish nodes aim at maximizing their benefit with respect to a known utility function. The PeerReview protocol has been designed under the assumption that every node is monitored by a set of monitors and that each monitor set contains at least one correct node that executes all the monitoring steps. In this work, we raise this assumption and consider that any node in the system can behave selfishly if it has an interest in doing so. We show that nodes executing PeerReview can skip some steps of the monitoring protocol without being detected and that such behaviour can have a dramatic impact on the performance of the monitored protocol. We provide in Section IV a complete analysis of all the protocol steps of PeerReview and list all the selfish deviations that

they are subject to. Due to the lack of space, we present here our practical results only. Specifically, to assess the impact of selfish nodes in PeerReview, we performed the following two experiments. In the first experiment, we deployed on one hundred nodes the SplitStream protocol [6], an efficient tree based multicast protocol, monitored by PeerReview. In the second experiment, we deployed one hundred nodes running the Onion routing protocol [12] monitored by PeerReview. In both cases, we used the same experimental settings as the ones described in Section VIII. In both experiments, if a selfish node notices that its monitors are selfish (e.g., because they never ask to audit its log), it also behaves selfishly with respect to the SplitStream and Onion routing protocols by dropping messages it receives and that are not intended to him.

We measure the percentage of lost messages with respect to the proportion of selfish nodes in the system. Results, depicted in Figure 1 show that in presence of up to 30% of selfish nodes, correct nodes running the SplitStream protocol observe 54% of message loss. Similarly, in the Onion routing application, correct nodes experience a loss in their onions that can reach 85% with 30% selfish nodes in a configuration with five relays. This proportion increases and reaches 100% when the number of relays increases. This is due to the fact that the probability of having a selfish relay in a path increases proportionally with the number of relays constituting this path.

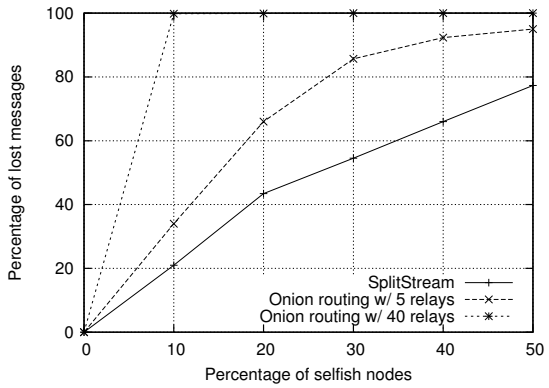


Fig. 1: Impact of selfish nodes in PeerReviewed SplitStream and Onion routing protocols.

The question we raise in this paper is thus how to enforce accountability in any underlying protocol in presence of selfish nodes? We answer this question in the remaining of the paper.

B. System model

Our target system is composed of two protocols: the monitored protocol to which we will refer as P and the monitoring protocol to which we will refer as M.

Fault model. We consider a fixed proportion of *Byzantine* nodes that can take arbitrary decisions. They can deviate from either P or M protocols for any reason (e.g., a failure, a bug, a threat). Furthermore, we consider any number of *selfish* nodes. These nodes aim at maximising their benefit according to a known utility function. Selfish nodes will deviate from M if they gain some benefit in doing so. Specifically, this benefit can be represented along the following axes:

1) (*Communication*) Sending/receiving as little as possible monitoring messages to/from other nodes.
2) (*Computation*) Performing as little as possible monitoring-related computations for other nodes.

Moreover, we assume that selfish nodes are *risk averse*. This means that before performing any deviation, a selfish node estimates the probability to be detected in the future. If this probability is greater than zero, a selfish node sticks to the protocol. This assumption is commonly used in BAR systems [2]. This assumption makes particularly sense in accountable systems because the detection of a deviation in these systems directly leads to the eviction of the faulty node from the system. Instead, in systems where the penalty is weaker, e.g., a decrease in a reputation value, it appears more appropriate to consider different selfishness models (e.g., risk affine). This is not the case of our system.

The BAR model also supposes that selfish nodes join and remain in the system for a long time and seek a long-term benefit. Moreover, selfish nodes do not collude and assume that other nodes are correct.

System assumptions. As in PeerReview, we assume a cryptographic identification of nodes. Specifically, each message sent in the network is signed using the sender's cryptographic key. Moreover, we assume that messages sent by a sender to a given receiver are always received if retransmitted infinitely often. We assume that cryptographic primitives can not be forged and that hash functions are collusion resistant. We assume that nodes have a deterministic reference implementation of P that can be initialised with checkpoints and to which we can inject inputs in order to get the corresponding outputs.

IV. Accountability: the PeerReview Approach

Consider a system composed of N nodes executing a protocol P. We assume that P can be represented as a set of deterministic state machines S_i and that every node has access to a reference implementation of each state machine. Furthermore, we assume that each node uses a collusion resistant hash function H and holds a pair of public/private key. We note by $(m)_{\sigma_i}$ a message signed using i's private key.

By relying on the architecture described in Figure 6 and on the secure logs described above, nodes engage in being accountable for their actions by executing the following protocols:

Commitment Protocol. This protocol ensures that the sender (resp. the receiver) of a message obtains verifiable evidence that the receiver (resp. the sender) has logged the transmission as illustrated in the left part of Figure 2. Specifically, when node i sends a message m to node j, it adds an entry e_k to its log and generates the corresponding authenticator $\alpha^k_{\sigma_i}$. It then sends this authenticator along with m. When j receives the messages, it checks the authenticator, adds an entry corresponding to the reception of m and generates the corresponding authenticator. Then, j sends an ack message to

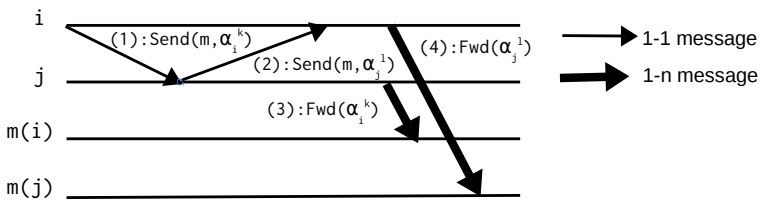


Fig. 2: Commitment and Consistency Protocols

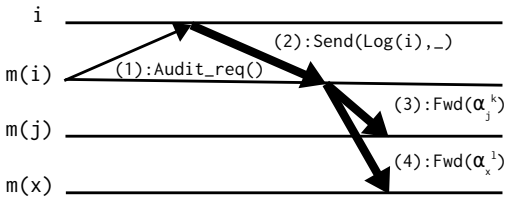


Fig. 3: Audit Protocol

i, with evidence that it has added the reception of m to its log.

Consistency Protocol. This protocol ensures that each node either maintains a single linear log that is consistent with all authenticators the node has issued, or it is exposed by at least one correct monitor. This protocol works as depicted in right part of Figure 2. Specifically, each node receiving an authenticator from another node (both i and j receive an authenticator in the figure), forwards the authenticator to the node's monitors. In the figure, i (reap. j) forwards $\alpha^l_{\sigma_j}$ (reap. $\alpha^k_{\sigma_i}$) to j's monitors (resp. to i's monitors). This helps monitors to collect verifiable evidence of all the messages the monitored node has sent or received.

Audit Protocol. Periodically each monitor challenges its monitored nodes to return all log entries in a given range of sequence numbers as shown in Figure 9 (nodes in $m(i)$ send an audit message to i). Using the log entries sent by i, each monitor extracts all authenticators that i has received from other nodes and that appear in its log and forward them to the nodes's monitors (in the figure authenticators signed by j and others signed by k are forwarded to j's monitors and k's monitors respectively). Then, each monitor ensures that node's log corresponds to a correct execution of the protocol.

Challenge-Response Protocol. This protocol works as described in Figure 4. Specifically, if a node i waits for a given message from a given node j for too long, i suspects j. It then creates a challenge for j and sends this challenge to j's monitors, who forward the challenge to j.

Evidence Transfer Protocol. This protocol ensures that eventually, every correct node collects the same evidence about other nodes in the system. Specifically, as depicted in Figure 5, node i periodically fetches the challenges collected by the monitors of every other node j it is interested in (e.g., its direct partners). It then replays these challenges and eventually outputs the same indication as j's monitors about j.

V. Selfish deviations in PeerReview

We analyse in the following each PeerReview subprotocol and show that they are all subject to selfish deviations as nodes have no interest in performing some their steps.

Indeed, in the *Commitment Protocol*, a selfish node can avoid sending (a subset of) authenticators to its partners along with the messages of the protocol P that it is executing. Furthermore, in the *Consistency Protocol*, a selfish node that receives an authenticator from another node, can avoid forwarding this authenticator to the node's monitors to save bandwidth (e.g., node i (resp. j) may skip step (4) (resp. step (3)) in Figure 2). In the *Audit Protocol*, a selfish monitor can (sometimes) avoid sending audit requests to its monitored nodes to save bandwidth and computational resources (e.g., a node in $m(i)$ can skip step (1) in Figure 9). Additionally, it could send audit requests and claim that the audited node is correct without effectively performing the various checks described in Section VII-A. Finally, upon performing an audit, a selfish monitor can avoid forwarding authenticators extracted from the monitored node's log to these nodes' monitors (e.g., a node in $m(i)$ can skip steps (3) and (4) in Figure 9). In the *Challenge-Response Protocol*, a selfish node may avoid challenging other nodes if they are missing some messages from them (e.g., node i may skip step (2) in Figure 4). Selfish monitors can also skip the forwarding of a challenge to the suspected node (e.g., a node in $m(i)$ may skip step (3) in Figure 4). In the *Evidence Transfer Protocol*, selfish node can avoid fetching the challenges about other nodes and avoid executing the fetched challenges hoping that other nodes will take care of expelling misbehaving nodes from the system (e.g., node i may skip steps (1) and (3) in Figure 5).

VI. *FullReview* protocol overview

Let us consider a set of N nodes executing a protocol P defined as a set of deterministic state machines. In *FullReview*, nodes take part in a classical accountability architecture as depicted in Figure 6. Specifically, each node i in our system interacts with a set of nodes referred to as i's partners and appearing on its right side in the figure. In addition to its set of partners, node i is assigned a set of monitors that periodically verify whether i sticks to the specification of the protocol P or not. This set of nodes is referred to as $m(i)$ and appears above i in the figure. Symmetrically, i monitors a set of nodes: the set of nodes referred to as $m^{-1}(i)$ and appearing below i in the figure. To perform this monitoring, each node maintains a *secure log* that is tamper evident and append only, in which it writes all its interactions with its partners (details on secure logs are given in Section VII-A). This log is periodically audited by i's monitors. Each monitor runs a monitoring protocol M also described as a set of deterministic state machines.

The objective of *FullReview* is to force selfish nodes to execute all the steps of both protocols P and M and to detect when Byzantine nodes deviate from either protocols P or M. To reach this objective, each node i logs in its secure log all its actions related to both protocols P and M. Then, i's

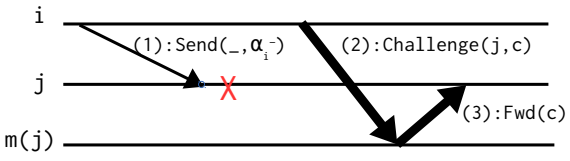


Fig. 4: Challenge-Response Protocol

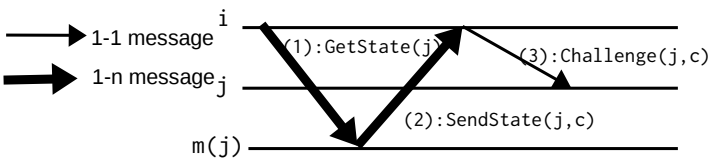


Fig. 5: Evidence Transfer Protocol

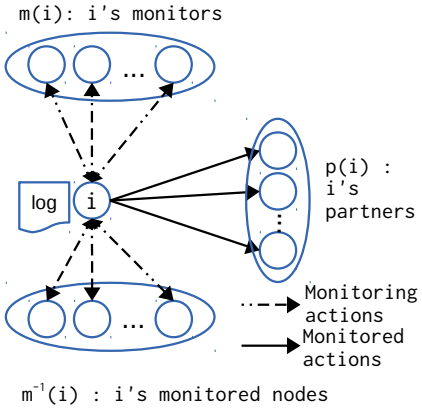


Fig. 6: Simple accountability architecture.

monitors, i.e., nodes in the set $m(i)$, periodically perform a set of verifications on this log. These verifications, which are depicted in the diagram of Figure 7, allow each monitor to reach evidence about the correctness of i. Specifically, each node in $m(i)$ starts by verifying that i did not tamper with its log (e.g., that the node did not delete previously inserted entries). We call this verification, which appears on the top of the diagram, log *coherence check*. We explain how this verification is performed in Section VII-B1.

Further, each node in $m(i)$ verifies that i holds a unique log for all its partners. We call this verification, which appears second in the diagram, log *consistency check*. The above two verifications are critical for the accountability system to be effective. Indeed, if a node manages to add/delete log entries or to have multiple versions of a log, it could deviate from the protocol without being detected. We explain how this verification is performed in Section VII-B2.

Moreover, each node in $m(i)$ verifies that the *communication patterns* appearing in i's log are coherent with M and P's state machines (third verification in the diagram). This verification ensures that i's log contains a sequencing of messages that reflect a correct behaviour. For instance, a correct log should contain periodic requests from i to the set of nodes it monitors, i.e., the nodes in $m^{-1}(i)$. The absence of such periodic messages reflects a faulty behaviour. We explain how these verifications are performed in *FullReview* in Section VII-B3.

However, a log that exhibits a correct sequencing of messages is not sufficient to guarantee a correct behaviour. Hence, the last verification that is performed by i's monitors is to assess whether i's log corresponds to a *correct execution* of the protocols P and M or not. Verifying the conformance of i's log with a correct execution of P is performed as in the PeerReview protocol, i.e., by re-executing the code of the protocol P using a reference implementation. Specifically, the inputs present in i's log are injected in the reference implementation of P and the produced outputs are compared with the outputs present in i's log. Mismatching outputs would constitute an evidence that i did not correctly execute P.

Doing the same verification for the protocol M is not possible. Indeed, as further discussed in Section VII-B4 re-executing the monitoring code is a recursive task and requires that a node's log contains the log of all the other nodes that are linked to him in the monitoring graph (which may possibly be all the nodes in the system). To avoid such an overkill, we identify all the computations performed in the protocol M and ensure that these computations are performed by a set of nodes in parallel. The outcome of each computation is then collected from the various participating nodes and sent to the nodes' monitors. The latter compare the outcome of the computation performed by their monitored node with respect to what other nodes have computed. As selfish nodes do not want to be exposed by correct nodes, they will always perform the computation correctly. In the diagram of Figure 7, this last verification is performed before the re-execution of P's code because the latter is more costly. Details of how *FullReview* verifies that nodes correctly executed the computations appearing in both protocols M and P are described in Sections VII-B4.

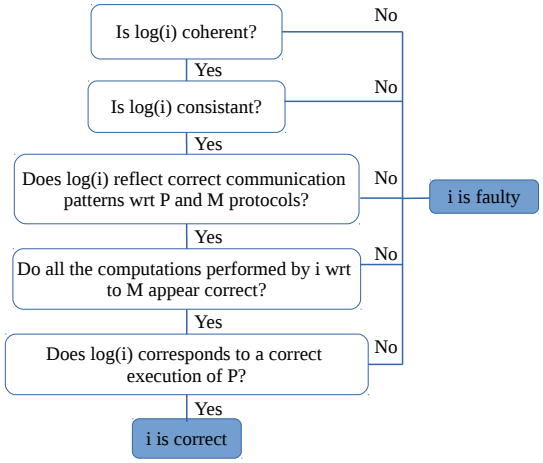


Fig. 7: *FullReview* monitors decision diagram.

VII. *FullReview* DETAILED DESCRIPTION

We start this section by introducing secure logs, a central component for enforcing accountability (Section VII-A). We then present the two major parts of our protocol, i.e., the audit protocol (Section VII-B) and the omission failure protocol (Section VII-C). Finally, we give some information about how we carried out the Nash equilibrium proof for our protocol (Section VII-D).

A. Accountability tools: Tamper Evident Log

Secure logs are often used to enforce accountability in distributed systems. A secure log is generally used to store the messages exchanged by a node with its partners. According to the requirements of the accountable system, log entries labelled $e_0, ..., e_k$ can contain various information among which an identifier of the logged message, whether the message was sent or received by the node as well as its parameters.

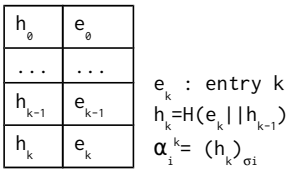


Fig. 8: Example of a secure log.

To each log entry e_k corresponds a recursive value h_k, computed as a hash of e_k concatenated with the value of h_{k-1} (where h_{-1} is a fixed value), and an authenticator α_i^k, which is a message containing the value of h_k signed with i's private key, i.e., $\alpha_i^k = (h_k)_{\sigma_i}$. Authenticators allow verifying that a node log has not been tampered with. For instance, consider a node j, among node's i monitors. If j gets a pair of authenticators α_i^0 and α_i^k corresponding to the entries e_0 and e_k of i's log respectively, it can ask i of its log entries $e_0, ..., e_k$ and recompute $h_0, ..., h_k$. If the computed h_k differs from the one held by j, the latter can accuse i of tampering with its log. Further, j can convince any other correct node of the misbehaviour of i by sending to it the signed authenticators α_i^0 and α_i^k along with the log entries sent by i.

B. FullReview selfish-resilient audit protocol

Using the secure log described above, a node j monitoring the behaviour of a node i performs a set of verifications to assess the correctness of i following the diagram of Figure 7. However, selfish monitors might be tempted not to perform these verifications. In order to force monitors to perform them, we make audits proactive. Specifically, we divide time in rounds and give the responsibility for each node to periodically (e.g., at the end of each round) ask its monitors to audit its log following the diagram depicted in Figure 9 (the Audit_req message sent from i to its monitors $m(i)$). Then, each monitor performs the required verifications and produces a certificate of correctness if the node passes all of them. In the opposite case, i's monitors send a proof of misbehaviour to i including the evidence of i's misbehaviour, which any correct node can recompute. This certificate is then used by i at the beginning of the following round in order to communicate with its partners. Without such a certificate, i's partners will refuse to interact with i. Note that some of i's monitors might be unresponsive (either because of a failure or to avoid auditing i's log). We describe how we deal with this situation in Section VII-C. Finally, after collecting the outcome of the audit produced by its monitors (Audit_resp message), i forwards the aggregated outcome to the monitors of each of its monitor (Fwd_outcome message). This last step is useful for the monitors of i's monitors (i.e., $m(m(i))$) in order to verify whether the nodes they monitor correctly performed their monitoring tasks or not. Further details on this verification are given in Section VII-B4.

In the following we describe in detail the set of verifications performed by the monitors of each node to assess its correctness.

*1) **Log coherence check**:* Allows verifying that a node's log has not been tampered with. Consider a node j that monitors a node i. If j gets a pair of authenticators α_i^0 and α_i^k corresponding to the entries e_0 and e_k of i's log respectively, it can ask i for its log entries $e_0, ..., e_k$ and recompute $h_0, ..., h_k$. If the computed h_k differs from the one held by j, the latter can accuse i of tampering with its log. Further, j can convince any other correct node of the misbehaviour of i by sending to it the signed authenticators α_i^0 and α_i^k along with the log entries sent by i. To perform this type of verification each node shall log each message it sends as part of the protocols P and M and send the corresponding authenticator to its partner. Furthermore, each node shall forward the received authenticators to its partners' monitors. However, selfish nodes might be tempted not to follow these steps, i.e., avoid attaching authenticators with messages they send and/or avoid forwarding received authenticators to the partner's monitors. We show how we deal with this issue in Section VII-B3.

*2) **Log consistency check**:* Node i might be tempted to maintain many correct logs (e.g., one log for each node with whom it interacts). To detect this type of misbehaviour, a monitor j that holds a set of authenticators sent by i to other nodes verifies that these authenticators belong to the same log. Similarly to the log coherence check, this verification requires that nodes attach authenticators to all messages they sent and forward received authenticators to their partners' monitors, and that monitors perform the consistency check. We show how we encourage selfish nodes to perform these steps in Section VII-B3.

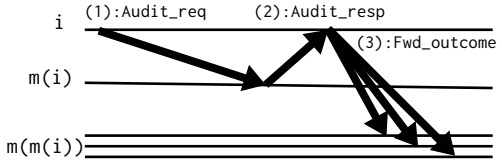


Fig. 9: *FullReview* audit protocol.

*3) **Verifying communication patterns**:* In this part of the protocol, a node in the monitor set of a node i is responsible for assessing whether the log of i reflects correct communication patterns with respect to the state machines of P and M. However, it is not possible to consider the state machines of these two protocols separately as in some situations steps of M need to be interleaved with steps of P. For instance, as seen in the log coherence and consistency checks described above, nodes need to send authenticators along with messages related to P and need to forward authenticators received along with messages related to P. To reach this objective, the state machine of the protocol P is automatically augmented with a set of mandatory transitions as depicted in Figure 10. In this figure, and in all the figures depicting automata in the paper, transitions are labelled as follows: (P|M:IN|OUT:message_type) where the first part refers to whether the message belongs to the protocol P or M; the second part indicates respectively whether the message is received or sent and the third part is the message type. This figure shows that each time a node is expecting a message as part of the protocol P, it should: (1) upon receiving the message, forward the included authenticator to the sender's monitors (transition labelled (M:OUT:fwd_auth)); or (2) accuse the sender if the message did not contain an authenticator by sending an accusation message to the sender's monitors (transition labelled (M:OUT:accuse)); or (3) suspect its partner if the latter did not send the expected message (transition labelled (M:IN:timeout)). The transitions following this latter transition are further described in Section VII-C.

Augmenting all the transitions of P related to the reception of messages as shown in Figure 10 forces selfish nodes to attach authenticators to the messages they send (otherwise, nodes that receive theses messages might accuse them). Furthermore, it forces selfish nodes to forward the received authenticators to their partner's monitors (otherwise, their monitors might accuse them of behaving selfishly).

In addition to verifying that a monitored node's log is coherent with the state machine of the P augmented automaton, monitors verify that the log is coherent with M state machines related to the audit protocol (described earlier in this section) and with M state machines related to the handling of omission failures. The former state machines are depicted in Figures 11 and 12 while the latter are described in the following section. Specifically, the automaton of Figure 11 shows the correct communication patterns of a node i asking one of its monitors for an audit (transition labelled (M:OUT:audit_req)). After sending his audit request, node i either receives a response from its monitor containing the outcome of the audit (transition labelled (M:IN:audit_resp)) or it does not receive a reply (the transition labelled (M:IN:timeout)). In the former case, node i forwards the outcome of the audit to the monitors of all of its monitors, which allows them to verify that their monitored node reached the same outcome about the correctness of i as the other monitors of i. In the latter case, i considers that its monitor has failed and handles this failure as described in the following section.

The automaton of Figure 12 shows the correct communication patterns of a monitor j that receives an audit request from a node i that it is monitoring (the transition labelled (M:IN:audit_req)). After the reception of this request, node j performs the audit of the i's log and sends back the outcome of the audit to i (the transition labelled (M:OUT:audit_resp)).

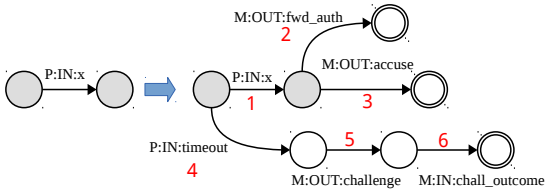


Fig. 10: Augmenting the P protocol.

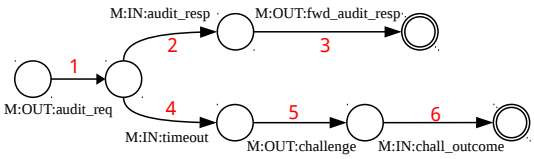


Fig. 11: Sending audit requests.

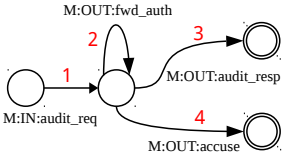


Fig. 12: Dealing with audit requests.

*4) **Verifying computations**:* In this part of the protocol, each monitor j in the monitor set of a node i verifies that the computations performed by i as part of the protocols P and M are correct. For the computations performed by i and that are related to P, j use checkpoints stored in i's log and initializes the reference implementation it has with the oldest non-verified checkpoint. Further, j replays all the inputs available in the portion of i's log it is auditing and verifies that the outputs produced by the reference implementation match with the outputs stored in the log. If the computed outputs do not match with the logged ones, j accuses i of misbehaviour. Whether i passes this verification or not, j stores the outcome of the audit along with the authenticators corresponding to the portion of the log of i that it has audited and sends the outcome of the audit to i as prescribed by the audit protocol (described earlier in this section).

Contrarily to the computations related to the protocol P, verifying those related to the monitoring protocol M can not be done by re-executing the steps of the protocol M. To intuitively understand why, let us consider the following example, where node i monitors node $i-1$ (among other nodes) and is monitored by node $i+1$ (among other nodes). At a given execution time, the monitor of node $i+1$, say node $i+2$ would like to audit node $i+1$'s log to verify that

it is correctly performing its monitoring actions regarding the behaviour of i. To do so, node $i+2$ needs to get access to node i's log, which is available in $i+1$'s log. Hence, to check whether $i+1$ has correctly done his monitoring actions, it needs to verify whether $i+1$ correctly audited i's by replaying the audit verifications itself. However, to verify whether i is effectively correct, $i+2$ must verify whether i correctly executed its monitoring steps with respect to $i-1$. To do this last verification, $i+2$ must verify whether the outcome of i's audit over $i-1$'s log is correct and is thus obliged to audit itself $i-1$'s log. This process clearly leads each node to recursively obtain and audit the logs of all the other nodes that it is connected to in the monitoring graph, which is not practical.

To avoid such an overkill, we use incentives to force selfish nodes to correctly perform the computations taking part of the protocol M instead of recomputing them. Specifically, as described earlier, after receiving the outcomes of the audit sent by its monitors, a node aggregates these results and forwards them to the monitors of its monitors. These nodes receive an information of the type: (audited node ID, authenticators, monitor ID, outcome) for each of i's monitors that took part in the audit. If a majority of monitors detects a misbehaviour in i's log and one of them, say node j, did not, then j is accused of misbehaviour. In this situation, j is selfish if it claimed that i is correct without performing the verification or Byzantine if it replied arbitrarily. As selfish nodes do not want to be excluded from the system, they always perform the computations related to M correctly. Instead, if a majority of monitors but j considers that i is correct, j is considered Byzantine, as a selfish node do not have any interest in accusing a correct node of misbehaviour.

C. Handling omission failures

The handling of omission failures is done in *FullReview* as depicted in Figure 13. Specifically, if a node i waits for a given message from a given node j for too long, i suspects j (after step (1) in the figure). To do so, i creates a challenge for j and sends this challenge to j's monitors (step (2) in the figure), who forward the challenge to j (step (3) in the figure). If j is still alive in the system then it replies to the challenge (step (4)). Whether j replied on not to the challenge, after a given amount of time j's monitors send an outcome of the challenge to i summarizing the situation (step (5)).

A selfish node may be tempted not to suspect a node even if it has waited for too long to receive a message assuming that other nodes will take care of that. Similarly, a selfish monitor might be tempted not to forward a challenge send by i to j assuming that the other monitors will do so. These two deviations are not possible in *FullReview* because of the verification of communication patterns performed by monitors on their monitored node's log. Specifically, the automata of Figures 14 and 15 show the correct communication patterns that should be present in the log of a node when, as a monitor, it receives an omission failure complaint about one of its monitored nodes and when, as a suspected node, it receives a challenge from its monitor. The log of a selfish node should be conform to these automata, otherwise it is accused by its monitors.

In addition, a selfish node might be tempted to suspect a node instead of performing a costly interaction with him. To avoid this deviation, we make the cost of suspecting a node higher than the cost of interacting with him. To avoid to overload the system, we adapt this cost to each step of the protocols P and M. For instance, if sending a message m costs xB of bandwidth to node i, we make the cost of suspecting a node j to whom i was supposed to send m equal to $x+\delta B$. As such, a selfish node i will always prefer to send m instead of suspecting j.

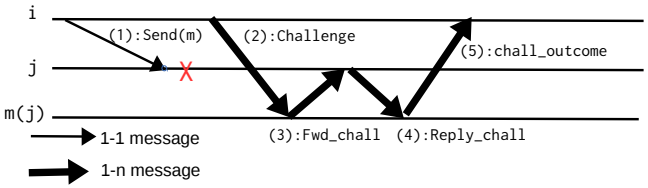


Fig. 13: *FullReview* handling of omission failures

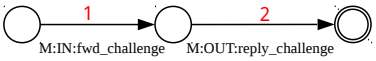


Fig. 14: Dealing with omission failures.

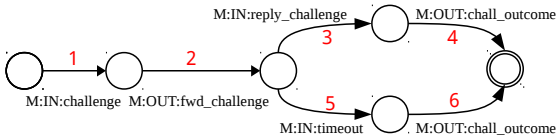


Fig. 15: Dealing with omission suspicions.

D. Resilience to selfish nodes

We carried out a detailed analysis of all the protocol steps of *FullReview*. For each of these steps we listed all the selfish deviations and the corresponding incentives that prove that selfish nodes do not have any interest in performing them. The incentives corresponding to each part of the protocol are depicted in tables I, II III, IV, V.

VIII. Performance Evaluation

In this section we evaluate the performance of *PeerReview* and *FullReview* with two distributed applications: SplitStream and Onion routing. We start by introducing the two applications and our experimental settings in Section VIII-A and VIII-B, respectively. We then present the performance of *FullReview* in presence of selfish nodes (Section VIII-C) and in the fault-free case (Section VIII-D). Finally, we assess the scalability of *FullReview* (Section VIII-E).

Overall, our evaluation draws the following conclusions. First, we show using real experiments that *FullReview* can effectively detect faults in presence of selfish nodes. Second,

Step	Description	Selfish deviation	Incentive
Fig 9 step 2	After the reception of Audit_req, each $m(i)$ performs a required verifications and produces a ceriticate of correctness if i passes all of them	Some selfish nodes in $m(i)$ set can refuse to perform verifications of correctness	At the reception of Audit_resp, the node i forwards the aggregated outcome to the monitors of each $m(i)$. These latter can verify if a node in $m(i)'s$ set performed all steps of verifications.

TABLE I: Incentives for audit protocol corresponding to the Figure 9

Transition	Label	Selfish deviation	Incentive
1	P:IN:x	Selfish nodes can refuse to forward received authenticators to their partner's monitors.	Selfish nodes always stick to the commitment protocol as they will assume that the node they are interacting with is correct and would hold evidence of their deviation if any.
5	M:OUT:challenge	A selfish node never suspects a node even if it has waited for too long to receive a message. Instead, it assumes that the node is Byzantine and that some other node will eventually suspect him, create the challenge, contact the node's monitors and propagate the suspicion.	A selfish node has no interest to do this action which appears in its log and then it risks eviction by its monitors.

TABLE II: Incentives for augmenting the P protocol corresponding to the automaton of Figure 10

Transition	Label	Selfish deviation	Incentive
1	M:OUT:audit_req	A selfish node never requests its monitors for an audit.	If a node does not present a certificate of correctness at the following round, its partners will refuse interacting with him. Hence, a selfish node always ask its monitors to periodically audit its log.
2	M:IN:audit_resp	A selfish monitor never replies to an audit request.	If it does not do so, the requesting node will suspect him. As dealing with a suspicion is more costly than replying to a message and may further lead to the eviction of the node, a selfish monitor always replies to an audit request.
3	M:OUT:fwd_audit_resp	A selfish node never forwards the outcome of its audit to the monitors of its monitors.	This deviation would be detected by the verification of communication patterns performed by the nodes monitors. Specifically, if a node's log contains the reception of an audit request, and this entry is not followed by a forwarding of the received message, the node is accused of misbehaviour and risks eviction. A selfish node always forwards a received audit response.
4	M:IN:timeout		
5	M:OUT:challenge	A selfish node never suspects a monitor that did not reply to an audit request.	After sending an audit request, if the log of a node does neither contain the reception of an audit response nor the sending of a challenge message to the monitors of the non-responsive monitor, it will be accused for misbehaviour by its monitors at the next audit. As a selfish node does not want to be accused, it always challenges unresponsive nodes.
6	M:IN:chall_outcome	A selfish monitor never replies to a challenge	A selfish monitor that does not reply to a challenge risks imminent eviction. As a selfish node does not want to be evicted, it always replies to a challenge.

TABLE III: Incentives for the sending of audit requests corresponding to the automaton of Figure 11

Transition	Label	Selfish deviation	Incentive
1	M:IN:audit_req	A selfish node can ignore the reception of an audit request.	If a node ignores the reception of an audit request it is suspected by the audited node. As a selfish node does not want to be suspected, it will always consider an audit request.
2	M:perform_audit	A selfish node can avoid performing the audit and claim that the audited node is correct in order to save CPU.	If a selfish node lies about the outcome of an audit, it risks eviction if correct nodes arrive to a different outcome (thanks to the replication of computations). As a selfish node does not want to be evicted, it effectively performs the verifications as prescribed by the protocol.
3	M:OUT:audit_resp	A selfish node can avoid sending an audit response to save bandwidth.	If a selfish node avoids sending the response of an audit it will be suspected by the requesting node. As a selfish node does not want to be suspected, it will always consider an audit request.

TABLE IV: Incentives for the handling of audit requests corresponding to the automaton of Figure 12

Transition	Label	Selfish deviation	Incentive
1	M:IN:fwd_challenge	A selfish node can ignore a challenge sent from its monitors.	A selfish node that ignores the reception of a challenge risks eviction as it will be considered as dead from the other nodes in the system.
2	M:OUT:reply_challenge	A selfish node does not reply to a challenge.	The same incentive as above holds.

TABLE V: Incentives for the handling of omission failures corresponding to the automaton of Figure 14

FullReview adds a small overhead compared to *PeerReview* both in terms of traffic generated and storage. Finally, using complementary simulations, we show that *FullReview* is scalable up to at least 1000 nodes.

A. Applications

1) Accountable Efficient Multicast: SplitStream [6] is a protocol that organises nodes in a tree structure where each node receives multicast messages from its parent node and forwards them to its child nodes. The specificity of SplitStream is that it aims at balancing the forwarding load between nodes. It reaches this objective by splitting the multicast stream into stripes and using different multicast trees to distribute each stripe. For our experiments, the source node generated a video stream of 300kb/s, which is a common rate for video-streaming applications. Each packet emitted by the source was sent through a different multicast tree where each node had two children.

In SplitStream, selfish nodes deviate by not forwarding updates to their child nodes. As a result they can get the video stream while saving bandwidth. However, in presence of selfish nodes, correct nodes may experience frame loss and consequently receive a degraded version of the video stream.

2) Accountable Anonymous Communication: Onion routing [12] is a protocol designed for anonymous communications. It is the protocol used in the TOR project [9], which is widely used by thousands users daily. In this protocol, when a node S wants to send a message to a node D, it chooses R other nodes, called relays, that will forward the message up to its destination. Node S encrypts successively the message using the public key of each of these relays, which constitutes the *onion* and then sends it to the first relay. Each relay decrypts one layer of the onion (i.e., removes one layer of encryption) and forwards it to the next one until it reaches its final destination. For Onion routing experiments, each node periodically emitted a packet to a randomly chosen node through a parametric number of relays. In all our experiments, messages have a fixed size of 10kB; smaller messages are padded with additional bytes in order to meet this requirement. Fixing message size is usually done in onion routing as it avoids an attacker to follow the progression of an onion in the system by comparing the size of forwarded messages.

In this protocol, a selfish node can choose not to forward an onion that is not intended to him. As a result, the destination will never receive the original message. The objective with designing a selfish-resilient version of Onion routing is to ensure that nodes will forward the onions they receive while providing anonymity guarantees.

B. Experimental settings

We have measured the performance of SplitStream and Onion routing in two configurations: (i) with *PeerReview* and (ii) with *FullReview*. Our experiments have been performed in two different settings. First, we performed experiments in real conditions using the public Grid5000 cluster[1]. In this cluster we used 50 quad-core physical machines clocked at 2.6GHz with 4GB of RAM that are interconnected via a Gigabit switch. These experiments have been run by deploying one logical node per physical machine and corresponding curves are annotated with [G5K] in their labels. To complement our experiments, we performed simulations using the *PeerReview* simulator that has been developed by *PeerReview* authors[2]. We performed simulations with up to 1000 nodes, in order to assess the scalability of *FullReview*. Results of these experiments are annotated with [SIM] in their labels.

[1]Grid 5000: https://www.grid5000.fr

[2]PeerReview code: http://peerreview.mpi-sws.mpg.de/.

C. Performance in presence of selfish nodes

In this section we show that *FullReview* tolerates selfish nodes. To this end, we performed three experiments. In the first experiment, selfish nodes follow the model presented in Section III-B. Specifically, they deviate only if they have an interest to do so and if there is no risk to be caught. In the second experiment instead, we consider that selfish nodes deviate if they have an interest to do so without considering the risk of exclusion. This latter experiment shows that if they decide to do so, selfish nodes are quickly detected by their monitors and excluded from the system.

For all but the second experiment we used the two applications monitored by *PeerReview* and *FullReview*. We used only *PeerReview* and Onion routing in the second experiment. In all the cases, the number of monitors per node is fixed to 2 and the audit period is set to 10s.

The results of the first experiment are presented in Figure 16. This figure shows the percentage of received messages as a function of the percentage of selfish nodes. SplitStream and *FullReview* are deployed with 50 nodes on G5K. In this experiment Onion routing was configured with 5 relays, chosen at random. We first observe in this figure that, using *PeerReview*, SplitStream and Onion routing do not tolerate selfish nodes. Indeed, in presence of only 10% of selfish nodes, only 79% and 66% of messages are received in the SplitStream and Onion routing applications, respectively. This represents a loss of 21% and 34% messages, respectively, which is not acceptable. This percentage decreases when the proportion of selfish nodes increases reaching 23% in SplitStream and 5% if Onion routing, in presence of 50% of selfish nodes. Instead, using *FullReview*, we observe that all messages are received in both applications as selfish nodes have no interest in deviating.

In the second experiment we evaluate the impact of the number of relays of Onion routing on the percentage of received message. Results, presented in Figure 17, show that increasing the number of relays leads to worst results for *PeerReview* as the probability to choose a selfish node in an Onion routing path becomes higher. For instance, with 10% of selfish 67% of messages are received when using 5 relays, while this number is as low as 9% when using 40 relays (in simulations). Moreover, we can observe that the percentage of received messages in the experiments on G5K is lower. For instance, with 10% of selfish and 40 relays only 1 onion has been received over the whole experiment. Note that Onion routing-*FullReview* does not experience message loss whatever the number of relays is. This is again due to the fact that selfish nodes have no interest in deviating.

The results of the third experiment are presented in Figure 18. In this experiment, we measure the percentage of received messages in SplitStream with *PeerReview* and *FullReview* where selfish nodes start to deviate from the protocol after 20s. This experiment has been launched with 50 nodes using simulations. As explained above, in this experiment, selfish nodes behave selfishly without reasoning on the risk of being detected. Using *PeerReview*, we observe that selfish nodes impact the system as soon as they behave selfishly, without ever being detected. Using *FullReview*, we observe that selfish nodes impact the system during a small time frame, corresponding to the audit frequency, after which they are detected and evicted from the system. As a result, all the messages are received for the rest of the experiment. Note that choosing a smaller audit period allows the system to detect selfish nodes more rapidly, but at the expense of some additional overhead, as we show in the next section.

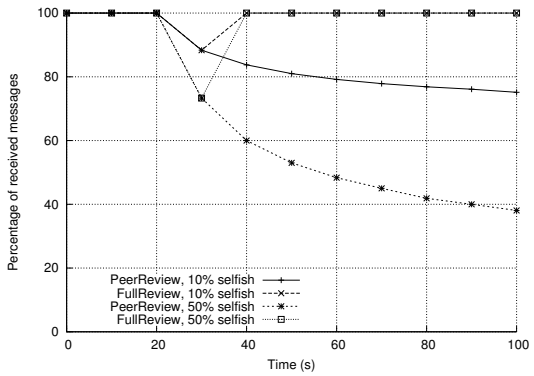


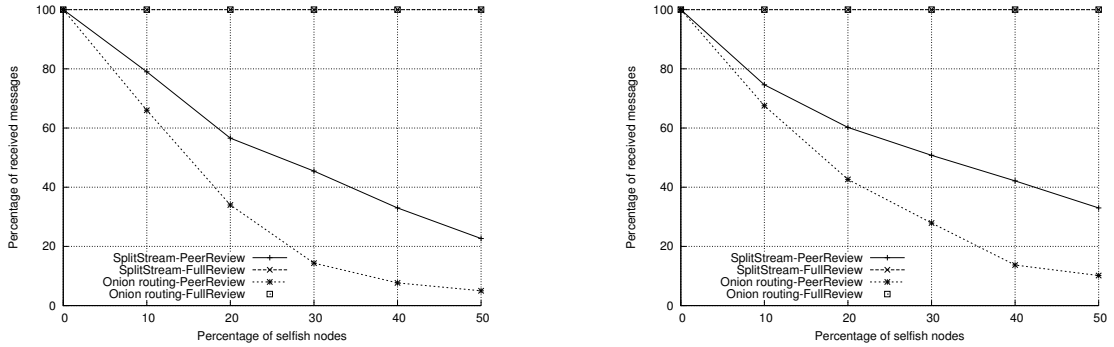
Fig. 18: [SIM] SplitStream percentage of received messages during an experiment in which between 10% and 50% of nodes start to act selfishly after 20s.

D. Performance in the fault-free case

In this section we assess the performance and the overhead of *FullReview*, compared to *PeerReview*, in the fault-free case. To this end, we performed three experiments. We launch each of the experiments of this section both using simulations and G5K. As we show in the following of this section, the results using simulations are consistent.

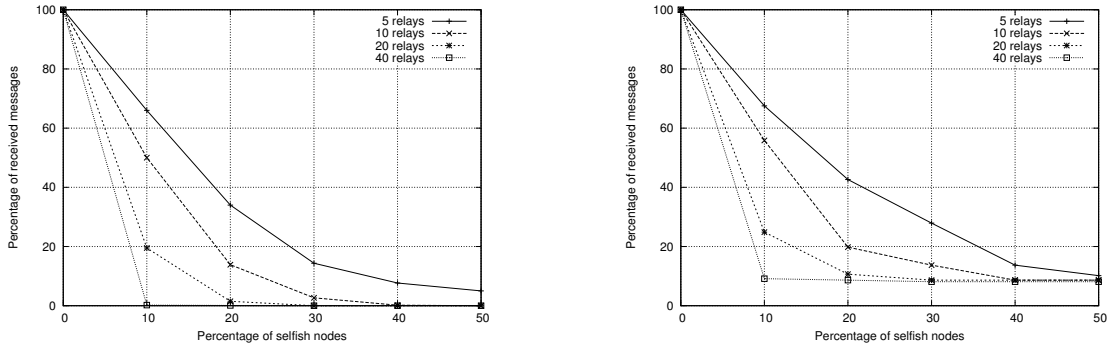
In the first two experiments, we measure the network traffic and the rate at which the logs grow w.r.t. the number of monitors, in *PeerReview* and *FullReview* respectively. In the case of Onion routing, an onion path was composed of 5 relays. Figure 19 presents the results for both SplitStream and Onion routing on G5K, while Figure 20 presents the results using simulations. Each value has been obtained by running the system with 50 nodes during 5 minutes. We can observe that the results on G5K are consistent with the results using simulations. As a result, we detail the results on G5K only, but the same observations can be made for the simulations.

In the left figure, each bar represents the traffic due to the payload of the application. On top of this payload is the traffic due to *PeerReview*, on top of which is the overhead of *FullReview* in addition to the one of *PeerReview*. In this figure, we observe that the average traffic per node increases wrt to the number of monitors for both *PeerReview* and *FullReview* in the two applications. This is due to all the messages that need to be exchanged between nodes and their monitors. Further, we observe that the overhead due to accountability in the SplitStream application has an overall cost of 14% in *PeerReview* with two monitors and an extra cost of 7% in



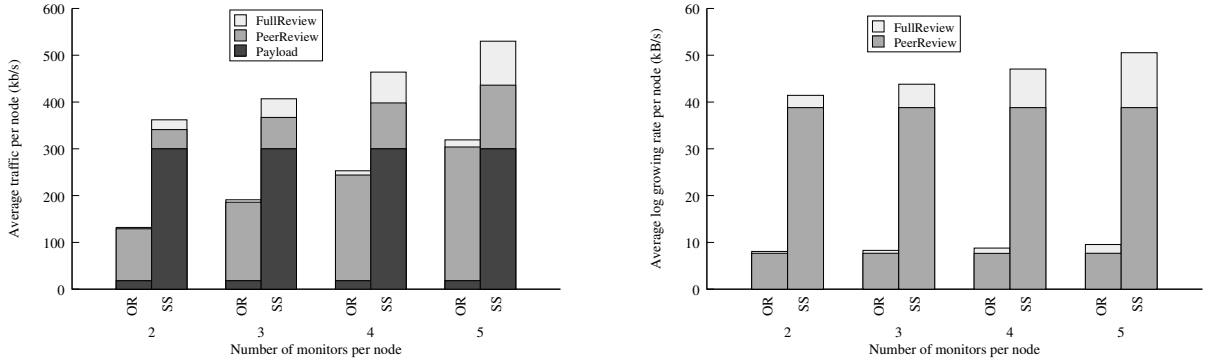
(a) [G5K] (b) [SIM]

Fig. 16: Percentage of received messages in SplitStream and Onion routing as a function of the percentage of selfish nodes.



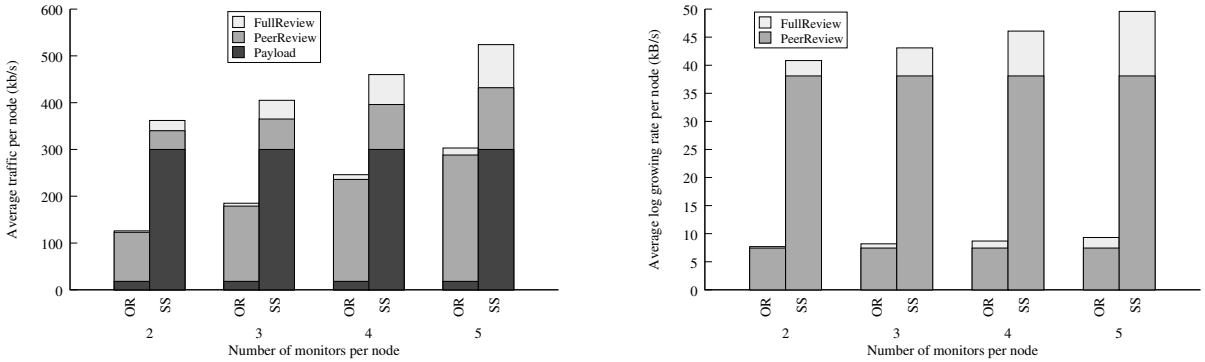
(a) [G5K] (b) [SIM]

Fig. 17: Impact of the number of relays on the percentage of received messages in Onion routing-*PeerReview*.



(a) Network traffic. (b) Log growing rate.

Fig. 19: [G5K] Average network traffic and log growing rate per node of SplitStream (SS) and Onion routing (OR) w.r.t. the number of monitors.



(a) Network traffic.

(b) Log growing rate.

Fig. 20: [SIM] Average network traffic and log growing rate per node of SplitStream (SS) and Onion routing (OR) w.r.t. the number of monitors.

FullReview. This overhead grows up to 45% for *PeerReview* and an additional 31% for *FullReview* when 5 monitors are used. These costs are much higher if compared to the payload of the Onion routing application. For instance, enforcing accountability in Onion routing using *PeerReview* generates a traffic of 129kb/s per node while the application itself generates a payload of only 18kb/s per node. However, put into context this result is not bad, as enforcing accountability in anonymous communication protocols is a very challenging task for which existing solutions often require the heavy use of broadcast primitives (e.g., RAC [4], Dissent [8]). Further, assuming that nodes are connected using Gigabit links (in the case of a LAN) or even using few Megabit links (in the case of a WAN), 129kb/s seems a reasonable overhead. The good news is that if the developer accepts to pay the cost of accountability using *PeerReview* in a system with a small payload, using a selfish resilient accountability system, i.e., *FullReview* would cost him an extra 3kb/s (i.e., 2% more traffic) with two monitors and an extra 15kb/s (i.e., 5% more traffic) with five monitors. Note that, overall, enforcing accountability using *PeerReview* is more expensive in the Onion routing application than in the SplitStream application because in the former application the full onions are stored in the log while in the latter instead of storing the video chucks received by nodes in the log, we store only their identifier. Indeed, storing onions was the only way we found to enable monitors to verify that a node has correctly decrypted and forwarded an onion it received.

In the right figure, each bar represents the average growing rate of the log of nodes. Similarly to the previous figure, the cost of *FullReview* is shown as a delta in addition to the cost of *PeerReview*. Note that logs do not grow forever. Indeed, as in *PeerReview*, logs are truncated after a given amount of time and audits are performed only for the new parts of the log. Obviously, the longer the logging period chosen by the designer, the higher the probability to deter faults.

Results depicted in this figure show that the log growing rate of the SplitStream application is higher than log growing rate of the Onion routing application, which is due to the fact that the SplitStream application generates more messages to send the video stream than Onion routing, and thus more interactions are added to the log. Further we observe that the higher the number of monitors per node the higher the log growing rate. On the Onion routing application, the overhead in terms of log growing rate is equal to 4.9% when using *FullReview* with two monitors and increases up to 24% when using five monitors. On the SplitStream application, this overhead is higher as it spans from 6.8% to 30% when using respectively two and five monitors. Yet, we consider this overhead as reasonable. Indeed, in the worst of our experiments (i.e, in the SplitStream application using five monitors), for 24 hours logging, nodes need to devote 4.4GB of storage for enforcing accountability in presence of selfish nodes, which is reasonable.

In the third experiment, we measure the impact of the audit period on the overhead of *FullReview* compared to *PeerReview*. The audit period was ranging from 1s to 30s. We set the number of nodes to 50, with 2 monitors per node and 5 relays for the Onion routing application. Each experiment last 5 minutes. Results, presented in Table VI, show that even with a high frequency of audit (i.e., every second), *FullReview* generates only 6.7% more traffic and logs are 8.2% larger than *PeerReview* in the worst case. Note that these results are consistent with the results of the simulations, presented in Table VII.

	Audit period	1s	5s	10s	30s
SS	Log size	+7.4%	+6.8%	+6.7%	+6.4%
	Network traffic	+6.7%	+6.2%	+6.1%	+5.9%
OR	Log size	+8.2%	+4.9%	+4.8%	+3.3%
	Network traffic	+2.9%	+2.6%	+2.3%	+1.9%

TABLE VI: [G5K] Overhead of *FullReview* compared to *PeerReview*, for both SplitStream (SS) and Onion routing (OR), with an audit period ranging from 1s to 30s.

Finally, we analytically measure the number of additional operations performed by *PeerReview* and *FullReview* for each

	Audit period	1s	5s	10s	30s
SS	Log size	+10.5%	+7.5%	+7.2%	+6.2%
	Network traffic	+9.0%	+6.7%	+6.5%	+5.7%
OR	Log size	+13.4%	+6.7%	+3.4%	+3.3%
	Network traffic	+3%	+2.6%	+1.6%	+1.2%

TABLE VII: [SIM] Overhead of *FullReview* compared to *PeerReview*, for both SplitStream (SS) and Onion routing (OR), with an audit period ranging from 1s to 30s.

of their sub-protocols, independently of the considered application (see Table VIII). This Table allows us to quantitatively show why *FullReview* overhead is as it is. Note that we do not present the cost of the Commitment, Challenge/response and Evidence transfer sub-protocols as *FullReview* does not modify them.

	Sub-protocol	Consistency	Audit
Exchanged messages	*PeerReview*	$\psi + \frac{\psi^2}{P}$	$\frac{\psi}{P}$
	FullReview	$2(\psi + \frac{\psi^2}{P})$	$3\frac{\psi}{P}$
Cryptographic operations	*PeerReview*	ψ	ψ
	FullReview	$\psi + \frac{\psi}{P}$	$\frac{4\psi}{P}$
New log entries	*PeerReview*	0	0
	FullReview	$\psi + \frac{\psi}{P}$	$\frac{2\psi}{P}$

TABLE VIII: Overhead of *FullReview* compared to *PeerReview* for one message exchange, independently of the considered application. The audit period is P messages exchanges.

To summarize, *FullReview* adds a small overhead to *PeerReview* in terms of generated traffic and log size. This overhead is mainly due to the new log entries inserted by *FullReview* to detect selfish nodes. Similarly to *PeerReview*, the cost of *FullReview* increases with the number of monitors per node and with the frequency of the audits. Overall, accounting for the increasing resources (storage and network bandwidth) at the disposal of a large public (Terabytes of storage and Megabits of network bandwidth), the cost of enforcing accountability in presence of selfish nodes becomes a realistic option.

E. Scalability of FullReview

In this section we show that SplitStream-*FullReview* and Onion routing-*FullReview* scale up to at least 1000 nodes.

Figure 21 presents the network traffic and the log growing rate of SplitStream and Onion routing, for both *PeerReview* and *FullReview*, as a function of the number of nodes in the system. Each value has been measured via a simulation that lasts 100s. Moreover, the system has been configured with 5 monitors per nodes. As one could expect from the results of Figure 19, using less monitors provides better performance. In addition, the audit period was set to 10s. Finally, Onion routing was configured with 40 relays and was sending onions at a rate of 16kb/s.

From this figure we can draw the following conclusions. First of all, for both SplitStream and Onion routing, the network traffic and log growing rate of *FullReview* is within a constant factor of *PeerReview*. For instance, with SplitStream, the log growing rate (resp. network traffic) of *FullReview* is equal to 1.4x (resp. 1.3x) the one of *PeerReview*. This is due to the fact that *FullReview* adds a constant number of operations on the ones performed by *PeerReview*. Second, we can observe that *FullReview* scales up to 1000 nodes, as the network traffic and log size remain fairly stable despite the increase of the number of nodes. The reason is that each node always interacts with the same number of nodes on average, whatever the overall number of nodes in the system (i.e., its partners wrt to the application and a fixed number of monitors).

IX. Conclusion

This paper addresses the problem of accountable distributed systems in presence of selfish nodes. We have shown that *PeerReview*, the only software generic solution to enforce accountability, does not tolerate selfish nodes. To tackle this problem we propose the *FullReview* protocol. This protocol uses game theory techniques by embedding incentives that force selfish nodes to stick to the protocol. We have evaluated *FullReview* on a cluster of physical machines and using simulation with two applications: SplitStream, an efficient multicast protocol, and Onion routing, the most widely used anonymous communication protocol. Our evaluation makes the following points. First, contrarily to *PeerReview*, *FullReview* effectively tolerates selfish nodes. Second, *FullReview* has a low additional overhead compared to *PeerReview*. Finally, *FullReview* scales up to 1000 nodes.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Eytan Adar and Bernardo A Huberman. Free riding on gnutella. *First Monday*, 5(10), 2000.

[2] Amitanand Aiyer *et al.* Bar fault tolerance for cooperative services. In *Proceedings of SOSP*, 2005.

[3] Michael Backes, Peter Druschel, Andreas Haeberlen, and Dominique Unruh. A practical and provable technique to make randomized systems accountable. 2007.

[4] S. Ben Mokhtar, G. Berthou, A. Diarra, V. Quma, and A. Shoker. RAC: a freerider-resilient, scalable, anonymous communication protocol. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[5] Sonia Ben Mokhtar *et al.* Firespam: Spam resilient gossiping in the bar model. In *Proceedings of SRDS*, 2010.

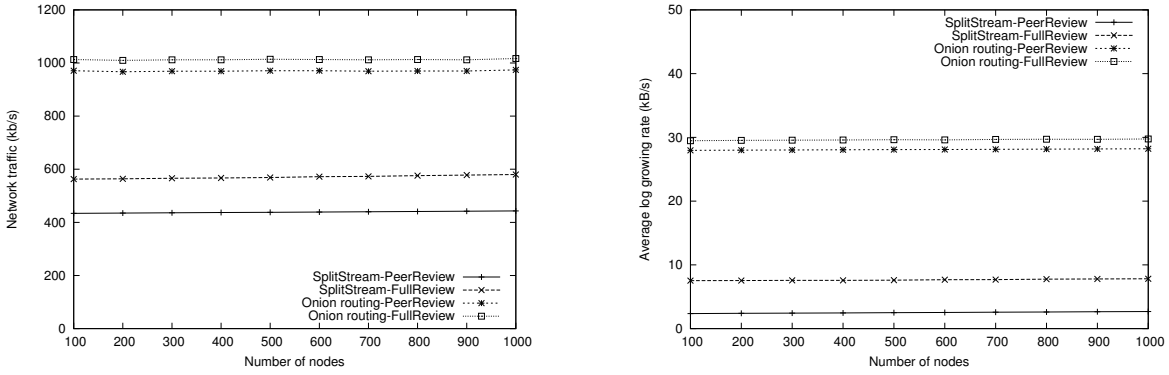
[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 298–313. ACM, 2003.

[7] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[8] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 340–350. ACM, 2010.

[9] Roger Dingledine *et al.* Tor: the second-generation onion router. In *Proceedings of USENIX Security Symposium*, 2004.

[10] Joao FA e Oliveira, Ítalo Cunha, Eliseu C Miguel, Marcus VM Rocha, Alex B Vieira, and Sérgio VA Campos. Can peer-to-peer live streaming systems coexist with free riders? In *Peer-to-Peer Computing, 2013. P2P'13. IEEE 13th International Conference on*, 2013.



(a) Network traffic.

(b) Log size.

Fig. 21: [SIM] Average network traffic and log growing rate of SplitStream and Onion routing w.r.t. the number of nodes in the system.

[11] Michal Feldman, Christos Papadimitriou, John Chuang, and Ion Stoica. Free-riding and whitewashing in peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 24(5):1010–1019, 2006.

[12] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Commun. ACM*, 42(2), 1999.

[13] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. Lifting: lightweight freerider-tracking in gossip. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pages 313–333. Springer-Verlag, 2010.

[14] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *OSDI*, pages 119–134, 2010.

[15] Andreas Haeberlen, Ioannis C Avramopoulos, Jennifer Rexford, and Peter Druschel. Netreview: Detecting when interdomain routing goes wrong. In *NSDI*, pages 437–452, 2009.

[16] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):175–188, 2007.

[17] Chi Ho, Robbert Van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *NSDI*, volume 8, pages 175–188, 2008.

[18] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 321–334. USENIX Association, 2012.

[19] Ramayya Krishnan, Michael D Smith, Zhulei Tang, and Rahul Telang. The impact of free-riding on peer-to-peer networks. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.

[20] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

[21] Harry Li *et al.* Bar gossip. In *Proceedings of OSDI*, 2006.

[22] Jacob Jan-David Mol, Johan A Pouwelse, Michel Meulpolder, Dick HJ Epema, and Henk J Sips. Give-to-get: free-riding resilient video-on-demand in p2p systems. In *Electronic Imaging 2008*, pages 681804–681804. International Society for Optics and Photonics, 2008.

[23] John Nash. Non-Cooperative Games. *Annals of Mathematics*, 54(2), 1951.

[24] Xavier Vilaça, Joao Leitao, Miguel Correia, and Luís Rodrigues. N-party bar transfer. In *Principles of Distributed Systems*, pages 392–408. Springer, 2011.

[25] Amir Yahyavi, Kévin Huguenin, Julien Gascon-Samson, Jörg Kienzle, Bettina Kemme, et al. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 1–10, 2013.

[26] Aydan R Yumerefendi and Jeffrey S Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, 2007.