AcTinG: Accurate Freerider Tracking in Gossip Technical Report

Sonia Ben Mokhtar
CNRS - LIRIS
sonia.benmokhtar@insa-lyon.fr

Jérémie Decouchant
Grenoble University
jeremie.decouchant@imag.fr

Vivien Quéma
Grenoble INP
vivien.quema@imag.fr

***Abstract*—Gossip-based content dissemination protocols are a scalable and cheap alternative to centralised content sharing systems. However, it is well known that these protocols suffer from rational nodes, i.e., nodes that aim at downloading the content without contributing their fair share to the system. While the problem of rational nodes that act individually has been well addressed in the literature, *colluding* rational nodes is still an open issue. Indeed, LiFTinG, the only existing gossip protocol addressing this issue, yields a high ratio of false positive accusations of correct nodes. In this paper, we propose AcTinG, a protocol that prevents rational collusions in gossip-based content dissemination protocols, while guaranteeing zero false positive accusations. We assess the performance of AcTinG on a testbed comprising 400 nodes running on 100 physical machines, and compare its behaviour in the presence of colluders against two state-of-the-art protocols: BAR Gossip that is the most robust protocol handling *non*-colluding rational nodes, and LiFTinG, the only existing gossip protocol that handles colluding nodes. The performance evaluation shows that AcTinG is able to deliver all messages despite the presence of colluders, whereas LiFTinG and BAR Gossip, both suffer heavy message losses. It also shows that AcTinG is resilient to massive churn. Finally, using simulations involving up to a million nodes, we show that AcTinG exhibits similar scalability properties as standard gossip-based dissemination protocols.**

I. Introduction

It is well known that content sharing applications account for a large proportion of traffic over the Internet. The most popular of these applications include collaborative downloading (e.g., BitTorrent) and peer-to-peer live streaming (e.g., P2PLive). Relying on the P2P paradigm offers robustness to failures, scalability up to hundreds of thousands of nodes, and adaptability. Indeed, P2P systems can handle massive node arrival/departure and are highly resilient to churn. From the point of view of content providers, relying on a P2P system allows shifting cost (e.g., bandwidth) to clients, and avoids the need for maintaining dedicated servers.

A major problem that face large scale P2P systems deployed on the public domain is the existence of rational nodes, i.e., nodes that aim at receiving content without contributing their fair share, by forwarding it to others. Existing studies have shown that the presence of even a small portion of rational nodes significantly degrades the system performance [1]–[5]. This is why a number of protocols have been devised in the last decade to deal with the problem of rational nodes in collaborative systems, (e.g., rational resilient live streaming [6]–[8], spam filtering content dissemination [9] and N-party transfer [10]). All these protocols provide incentives that encourage/force rational nodes to participate in the system. However, apart from the protocol presented in [8], all the existing solutions work under the assumption that rational nodes do not collude. This problem though has been demonstrated to be a reality in existing file sharing applications [11]. Handling colluding nodes is a difficult problem provided that colluders generally perform unobservable actions from the point of view of the collaborative protocol [12], which makes their deviations difficult to deter. For example, a group of colluders could be a group of nodes that collaborate to exchange content between each other "off the record" (e.g., using the silent broadcast protocol described in [12]). Such colluders do not share with nodes not belonging to the group the content they receive off the record, thus harming the protocol.

To the best of our knowledge, the only gossip-based content dissemination protocol trying to prevent collusions is the LiFTinG protocol [8]. In this protocol, nodes log their interactions with other nodes and perform distributed audits of each others logs. In order to be cost effective, this protocol relies on cryptography-free procedures, and statistical analysis of these logs. For instance, a node is suspected of colluding with another node if the frequency of its interactions with the latter is greater than an expected average. Unfortunately, as analysed by the authors themselves, due to their statistical nature and to message losses, the mechanisms implemented in LiFTinG do not allow to catch all rational collusions (false negatives), and may even lead to wrong exclusions of correct nodes (false positives). Experiments that we performed and that are described in Section VI-B, confirm this result and further show that, for instance, when 30% of nodes collude (either in a large group or in small collusion groups), correct nodes observe 25% of message losses.

The challenge we embrace in this paper is the design of a rational-resilient content dissemination protocol that prevents collusions to occur and that does not wrongfully exclude correct nodes. An observation one can start with is: a colluding behaviour can be considered as a Byzantine behaviour [13]. A legitimate question is thus to know whether it is possible to rely on existing techniques for Byzantine fault tolerance and Byzantine fault detection, such as Nysiad [14], PeerReview [15], Accountable Virtual Machines [16], Trinc [17], or A2M [18]? The answer is No. The reason is that these generic solutions for Byzantine fault tolerance and detection either assume a limited proportion of faulty nodes, or the existence of trusted nodes or hardware. Instead, we assume in this paper that all nodes can be rational, and we do not rely on any trusted entity, whether software or hardware.

We present AcTinG, a content dissemination protocol that tolerates an unlimited number of (possibly colluding) rational nodes, while guaranteeing that no correct node is ever expelled, and that all rational deviations are eventually detected. To reach this objective, we adopt a different approach than the one used in the LiFTinG protocol: rather than trying to detect collusions a-posteriori, we built AcTinG in such a way that it is *not* in the interest of nodes to collude. We analyse each step of the protocol and describe the incentives that force rational (possibly colluding) nodes to stick to the protocol. We perform a performance evaluation of AcTinG. We compare its performance to two protocols: BAR Gossip, the state-of-the-art gossip protocol that is able to handle *non*-colluding rational nodes and LiFTinG, the state-of-the-art gossip protocol that is able to handle colluding nodes. We implement a streaming application that we deploy on top of the three protocols. We deploy 400 nodes on one hundred physical machines and show that AcTinG is able to deliver the entire stream despite the presence of colluders, whereas LiFTinG and BAR Gossip, both suffer heavy message losses. We also show that AcTinG is resilient to churn, and using complementary simulations involving up to a million nodes, that it is scalable: it yields a logarithmic growth of memory and bandwidth consumption, comparable to standard gossip based protocols [19].

The rest of the paper is structured as follows. Section II describes our system model. Section III introduces the core ideas of AcTinG. Section IV provides a detailed presentation of AcTinG. Section V discusses its resilience to (colluding) rational nodes. Section VI presents a detailed performance evaluation. Section VII reviews the related works. Section IX concludes the paper.

II. System Model

We consider a system with N nodes, which are uniquely identified, e.g., using a hash value of their IP address. We assume that nodes can join and leave the system (gently or by crashing) at any time. We consider two classes of nodes: *correct* nodes and *rational* nodes. Correct nodes follow the protocol. Rational nodes are defined as in [7] extended with the notion of collusion: they aim at getting the content (i.e., missing the lowest possible number of updates) at the lowest possible overhead in terms of bandwidth consumption. This means that rational nodes would deviate in any sort from the protocol, possibly by *colluding* with each other, as long as the deviation saves their resources while not impacting the quality of the content they are getting.

Specifically, the benefit of colluding rational nodes can be represented along the following axes:

1) (*Stream Quality*) Receiving as much as possible (possibly, all) stream updates,
2) (*Communication*) Sending as little as possible (possibly, none) stream updates or protocol messages to nodes not belonging to their coalition,
3) (*Computation*) Performing as little as possible computations for other nodes.

Colluding rational nodes would typically exchange updates off the record, and, in order to save bandwidth, would not share the updates they obtained secretly with nodes outside their group. It is important to note that rational nodes are *risk averse*, i.e., they never deviate from the protocol if there is any risk of being evicted from the system. This assumption is commonly used in BAR systems [20]. Furthermore, this assumption is particularly relevant in our context as we use accountability techniques to deter faults and accuse nodes (as described in the following section). In this context, when a fault is detected, a proof of misbehaviour is produced, which can convince any correct node in the system of the necessity of evicting the misbehaving node. As eviction corresponds to an infinite penalty, no benefit is worth taking such risk. We also suppose that rational nodes join and remain in the system for a long time and seek a long-term benefit.

We refer to the source as the node that is disseminating a given content. We assume that each content is disseminated from a single source at a time but our principles can be easily applied to systems where the content is disseminated from multiple sources at the same time. We assume that all nodes but the source may be rational, or experience failures, and may organise themselves in colluding groups of arbitrary sizes. We assume that the source does not fail, but this assumption can easily be relaxed using classical fault-tolerance techniques (e.g., [21])

We assume that the network allows every pair of nodes to exchange messages, and that they are eventually received if retransmitted sufficiently often. We also assume that hash functions are collision resistant and that cryptographic primitives cannot be forged. We denote a message m signed by a node i as $(m)_{\sigma(i)}$.

As in [7] and [22], we assume that nodes maintain clocks synchronised within δ seconds, and we structure time as a sequence of rounds in which nodes exchange updates. We assume that nodes have a secure log that is used to check their correctness through its analysis. A secure log is a log that is tamper evident and append only. Many systems recently defined variants of secure logs among which [15]–[18]. We build on the secure log presented in [15].

III. Protocol Overview

We present AcTinG, a gossip-based dissemination protocol that guarantees the following two properties: (i) a correct node is never expelled, and (ii) a rational node that deviates from the protocol in a way that impacts the performance of correct nodes is eventually suspected by all correct nodes. In the remainder of this section, we describe the principles of AcTinG that allow us to guarantee the above two properties. Protocol details are then presented in Section IV.

Figure 1 shows an overview of our protocol. In this figure, the source node s, which is the node from which the dissemination originates, cuts the content into chunks that we

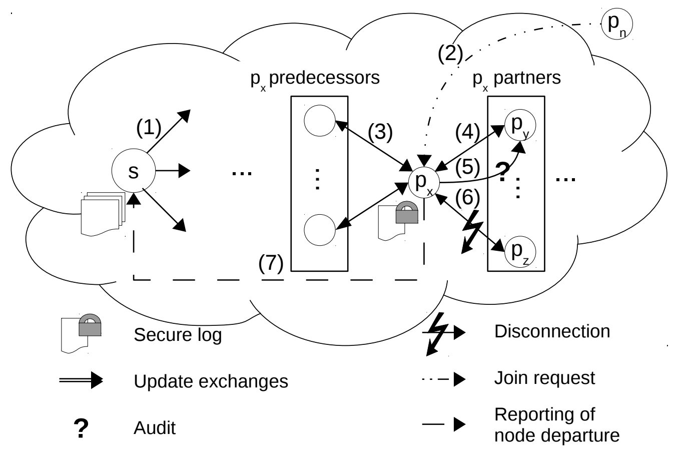


Fig. 1. Overview of AcTinG.

call *updates*. It then periodically disseminates these updates to a set of nodes (arrows 1 in the figure). To join this content dissemination session, a new node (p_n in the figure) needs to know a node that is already part of it, as described in Section IV-A (arrow 2 in the figure). In the middle of the figure, a node p_x, which characterises any node in the system except the source, has a set of nodes that it has selected as *partners* (depicted on its right side in the figure). Further, p_x has a set of nodes that selected it as a partner (depicted on its left side), to which we refer as p_x's *predecessors*. Periodically, p_x has to share with its partners (arrow 4 in the figure), and with its predecessors (arrow 3 in the figure) the updates it received. In order to maximise the quality of the content it receives, p_x may be tempted to (1) act rationally by receiving updates and not sharing them with its partners, or predecessors, and (2) collude with other nodes in the system (not necessarily its partners or predecessors) to get updates off the record without sharing them with anyone else. To avoid these temptations, the core idea underlying AcTinG is to make nodes accountable for their actions. Specifically, each node in AcTinG logs in a *secure log* its interactions with other nodes in the system, including the identifiers of the updates it received. Because any node can verify the information in the log of a node it is interacting with, the latter will be obliged to send to its partners the updates it has, and to receive the updates it is missing. Consequently, no node will have an interest in behaving rationally or forming collusions. Indeed, assume that node p_x colludes with another node to receive an update *u* off the record. Node p_x will not be able to record update *u* in its log (because the exchange was unofficial; we explain later how it is done). The good news for node p_x is that it does not have to forward *u* to other nodes because *u* does not appear in its log. The problem is that the next time a correct node having *u* in its log will interact with node p_x, it will send update *u* to p_x. Consequently, p_x will eventually have to forward *u*, and thus will have wasted its bandwidth, because it will have received *u* twice (off the record and from a correct node).

This core idea raises several questions and challenges that we answer in the remainder of this section.

"What if p_x chooses only colluders as partners with which it will interact with in the near future?". This way, p_x could accept updates and arrange with its future partners so as they do not audit its log, or so they do not send it updates it already received unofficially. Our protocol deals with this issue by forcing nodes to (periodically) establish random, yet *deterministically verifiable* partnerships as presented in Section IV-B. Specifically, each time a node p_x has to change its partners, it computes their identifier using a pseudo random generation function seeded with a deterministically computed seed. As such, nodes that will audit its log will be able to verify the legitimacy of the partners that it has selected.

"What if a node, p_x, maintains many (correct) logs?". For instance, p_x could have a log in which an update *u* appears, which it will show to nodes who already have *u* (to avoid sending it to them), and another log in which the same update does not appear, which will be presented to nodes that do not have *u* (to avoid having to send it to them). This problem is known as *equivocation*, i.e., the ability to make conflicting statements to different participants [17]. We deal with this issue by forcing nodes to audit their partners' logs at the beginning of each new partnership (arrow 5 in the figure). This audit verifies the consistency of the log of a node as a whole as presented in Section IV-C.

"Is not this periodic exchange of logs a performance overkill?". It is not necessary to audit the logs of nodes each time two nodes exchange updates. Indeed, we build on the assumption that colluders, and rational nodes in general, are risk averse. Hence, it is enough to ensure that for each step of the protocol, a deviation has a high probability to be detected in the near future, in order to make sure that rational nodes will not deviate. Consequently, instead of performing audits each time nodes communicate, audits are triggered in a *random yet verifiable* manner. Indeed, audits (from the point of view of audited nodes) must not be predictable, because rational nodes would seize an opportunity to deviate undetected if they could predict them. Yet they must be verifiable (from the point of view of nodes performing them), because rational nodes have to be forced to trigger this procedure. To reach this objective, a node that starts a new partnership with a node, performs a deterministic computation that results in a boolean telling it whether it should audit its partner or not.

"What if rational nodes decide not to answer to correct nodes to avoid trading updates, or being audited?". There are many reasons why a rational node may be tempted not to answer to a request from a correct node. This could, for instance, preserve it from sending its log and being audited as a result (arrow 6 in the figure). This type of misbehaviour is known as *omission failures*. We deal with this problem using a mechanism where unresponsive nodes are eventually suspected by all correct nodes, which stop interacting with them (as described in Section IV-A). As it is not in the interest of rational nodes to be isolated in the system, a rational node in AcTinG will answer all correct node requests. To avoid correct nodes to be expelled from the system because one of their message has been lost or delayed, we allow suspicions to be released, e.g., if the missing message eventually arrives.

Similarly, rational nodes may be tempted to wrongly suspect correct nodes of omission failure, by claiming that they did not send a given message to them, as it is the only reason why a node can skip mandatory interactions. We avoid this deviation by overcharging the sending of suspicion messages in such a way that it is more costly to suspect a node of omission failure than to effectively interact with it. As such, nodes would suspect other nodes of omission failures only if they are really missing a given message. Instead, if a node effectively left the system (assume node p_z in the figure), its predecessors (among which, node p_x in the figure) contact p_z's partners to collect evidence about the effective unresponsiveness of p_z (as described in Section IV-A). Then, p_x sends this evidence to the source node (arrow 7 in the figure), which eventually updates the membership list, and will also inform its partners during future exchanges.

Summarising, our protocol builds on accountability techniques, and on a set of mechanisms to provide incentives to rational, possibly colluding, nodes to stick to the protocol. Specifically, to avoid nodes from selecting their partners, our protocol relies on *random yet verifiable partnerships*. To be efficient it relies on *random yet verifiable audits*. To discourage rational nodes from being falsely unresponsive, our protocol handles *omission failures*. Finally, to discourage nodes from wrongly suspecting their partners our protocol *associates an extra cost with suspicion messages*.

IV. Protocol Details

We have presented the principles of AcTinG in the previous section. In this section, we detail the steps of the protocol.

In a nutshell, AcTinG divides time in rounds. At each round the source disseminates new updates, which come to expiration after *RTE* rounds, to a small set of randomly chosen nodes. To get updates, each node initiates, and maintains partnerships with f other nodes with whom it exchanges updates at each round. The partners are selected using a pseudo-random number generator function, i.e., PRNG, seeded deterministically (e.g., with the node's public key concatenated with the round number). At the beginning of a round, each node contacts all of its partners in order to propose updates to them and to request updates from them. Every *Period* rounds, each node updates its set of partners. Each time a node starts a new partnership with a node, the two nodes audit each others' log with a given probability. The membership is managed in a distributed manner by nodes who periodically inform the source of the arrival and the departure of nodes. Yet, it is the responsibility of the source to disseminate an updated list of alive nodes every *Epoch* rounds.

The remainder of this section describes the sub protocols constituting AcTinG in detail, as follows. First, we present the membership protocol (Section IV-A), which allows dealing with new nodes joining the system, nodes leaving it and unresponsive nodes. Then, we present the partnership management (Section IV-B), the audit (Section IV-C) and the update exchange protocols (Section IV-D), which allow handling the partnerships between nodes, auditing their logs and exchanging updates between partners, respectively.

A. Membership protocol

The membership protocol handles the arrival and the departure of nodes as well as the management of the membership list. Our membership protocol is fully distributed, rational resilient, and handles massive nodes arrival and departure.

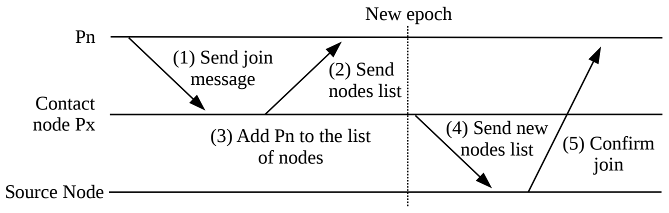


Fig. 2. Arrival of a new node.

Node arrival: The arrival of a new node follows the sequence of messages depicted in figure 2. In this figure, we assume that node p_n, which would like to join a given content dissemination session has installed the AcTinG software. This means that p_n has an empty secure log with the related security primitives. We also assume that p_n knows an entry point in the system, say p_x, which we call the *contact node* of p_n. To join a content dissemination session, p_n sends a join request to p_x (step (1) in the diagram). The latter replies with the list of active nodes of the current epoch (step (2) in the diagram). Using this list, p_n computes its list of new partners using the PRNG function as described in Section IV-B and contacts them to start receiving the content. At the beginning of a new round, each node, including node p_x informs the source of the arrival of new members that have contacted it (step (4)). Using theses messages, the source confirms to the new members their integration in the system and updates the membership list (step (5)).

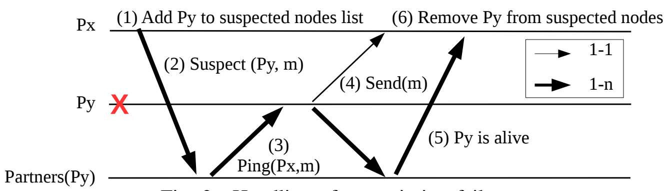


Fig. 3. Handling of an omission failure.

Node departure and omission failures: If a node p_x is expecting a message from one of its partners p_y for too long[1], it suspects p_y of omission failure as depicted in the diagram of Figure 3. Specifically, p_x adds p_y in its local list of suspected nodes (step (1) in the figure) and sends a suspicion message to the other partners of p_y (step (2)). This message includes the type of message that p_x is expecting from p_y. Then, each of p_y's partners pings p_y (step (3)). If p_y is alive, it replies to both its partners and p_x with the missing message (step (4)). After a given time slot, each of p_y's partners replies to p_x with a signed message certifying whether p_y responded to the ping message or not (step (5)). Using this message, p_x either removes p_y from its list of suspected nodes if p_y replied (step (6)) or sends an eviction message to the source including the messages received from p_y's partners.

[1]Delays for node suspicion are configured in an implementation dependent manner

If p_y responded to the ping message, it would be informed that p_x is expecting a message from it, and it will have to send the missing message to p_x before engaging further communications. Jeremie ►*If node p_x leaves the system, node p_y can still lift the suspicion if it sends the missing message to its partners.*◄ If p_y did not respond to the ping messages sent by its partners, p_x will hold evidence that p_y has left the system. Jeremie ►*Obtaining the evidence is illustrated in figure 4.*◄ In order to make sure that a rational node will never suspect a correct node in order to avoid initiating or accepting an interaction, we make the cost of sending a suspicion message higher than the cost of a normal interaction. Hence, unless it is a real suspicion, a node will never suspect another node.

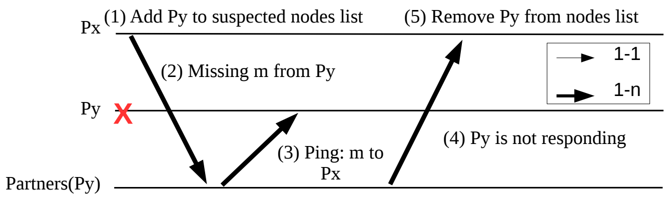


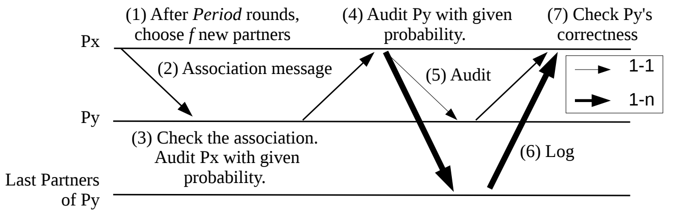
Fig. 4. Handling of a node departure.

Membership list update: Periodically, nodes that served as contact nodes for others send their list of new nodes to the source. Furthermore, nodes that hold an evidence of the departure of one partner send it to the source. The latter updates the membership list and sends it, at the beginning of each epoch, to the nodes along with the content. In order to fasten the removal of dead nodes from the membership list, an optimisation consists in letting the source disseminate the list of dead nodes at the beginning of each round along with the stream, instead of waiting the following epoch. As soon as a node receives these incremental updates from the source, it removes the corresponding nodes from its list of alive nodes, which avoids selecting them when new partnerships have to be established before the new epoch. In order to preserve nodes from the massive arrival of new nodes, which may consume their bandwidth, we adopt the optimisation defined in [22], which allows splitting the load between the older nodes and the new ones. Specifically, this optimisation prevents new nodes from establishing too many partnerships with older nodes.

B. Partnership management

Each node p_x maintains partnerships with f other nodes, which are selected with the PRNG function seeded with a deterministically computed seed (e.g., p_x's public key concatenated with the round number) among the non-suspected nodes of the last membership list. This process is depicted in the diagram of Figure 5. If a selected node is not responding, node p_x has to propagate a suspicion, and once it is confirmed, p_x is allowed by the source to find a new partner. Every *Period* rounds, a node p_x breaks the f partnerships it initiated, without informing its partners which know when the partnerships are supposed to end. A node having an identifier *id* will change its partnerships during round *r* if $(id + r) \mod Period = 0$. To initiate a new partnership with a node p_y, node p_x sends an association request to p_y (step (2) in the diagram).

At the beginning of a partnership, a node p_x may trigger an in-depth audit of its new partner p_y (step (4) in the diagram), by contacting the partners p_y had in the *RTE* previous rounds, and asking them to return their own log of the last *RTE* rounds including the current round (step (5) in the diagram). To reduce the cost of the protocol, nodes perform these audits in a random manner, i.e., each time they are in a position to perform an audit, they flip a coin and decide whether they should audit their partner or not. Nevertheless, to avoid that rational nodes hide behind this randomness to avoid auditing their partners, we make this randomness verifiable. Towards this purpose, we use the secure log *authenticators*, which are signed messages computed from the node's log as detailed in Section IV-C. These values are unpredictable as they depend on the current state of a node's log. Specifically, each time a node p_x is in a position to perform an audit of a new partner p_y, it computes the hash of its public key concatenated with the public key of p_y and the round number. The value of this hash modulo 100 gives a number that p_x uses to decide whether it should audit its new partner. For instance, if the probability of auditing a node fixed by the protocol is 30%, p_x audits p_y if the result of the modulo function is between 0 and 29. Node p_x further logs the authenticators it used to compute the value of this boolean, in order to justify, in future audits, the reason why it performed or did not perform the audit of p_y. If the audit must take place, p_x contacts p_y's partners, and ask for their logs.


(1) After *Period* rounds, choose *f* new partners
(4) Audit Py with given probability.
(7) Check Py's correctness
Px
(2) Association message
(5) Audit
1-1
1-n
Py
(3) Check the association. Audit Px with given probability.
(6) Log
Last Partners of Py
Fig. 5. Establishment of new associations between nodes, which may imply audits.

C. Audit protocol

In our protocol, the secure log is used to keep track of the communication a node had with other nodes in the system. Specifically, each entry in the log of a node A corresponds to a message sent (resp. received) by A to (resp. from) another node B. A log entry e_i is of the form $e_i = (seqno_i, h_i, c_i)$ where $seqno_i$ is a monotonically increasing sequence number, h_i is a hash value linked with the previous entries in the log and c_i is a type-specific content, which may include the message sent (resp. received) by A as well as other information such as authenticators (as defined below). The value of h_i is computed as follows: $h_i = H(h_{i-1}||seqno_i||H(c_i))$, where $h_0 = 0$, H is a hash function and $||$ stands for concatenation.

Each time a log entry e_i is added to the log of a node A, an authenticator α_i is generated. This authenticator, which is a signed message $\alpha_i = (seqno_i, h_i)_{\sigma(A)}$, states that A has a log entry e_i with a corresponding hash h_i. By sending the authenticator α_i to a node B, A commits to having logged the entry e_i and to the content of its log before e_i. Any node that receives α_i can use it to inspect e_i and all the entries preceding

e_i in the log of A. Upon reception of a log, any node is able to recompute the hash values it contains, according to the log entries, and thus to check their validity. In addition, a log entry for a received message must include a matching authenticator, implying that a node cannot invent an entry for a message it did not receive. These two properties make the secure logs tamper-evident and append only.

As described in the partnership management protocol, when node p_x must audit node p_y, it asks p_y's partners to return their logs. Upon reception of these logs, node p_x verifies:
(i) the consistency of the logs, by recomputing the recursive hash values associated to log entries,
(ii) the presence of the exchanges p_y was supposed to initiate,
(iii) that p_y declared the updates it was supposed to receive from the source, if p_y was supposed to interact with the source,
(iv) that the exchanges correspond to a correct execution of the protocol, i.e., that p_y proposed to all its partners all the updates that appear in its log, that p_y requested from its partners all the updates it was missing, that p_y served to its partner all the updates they were requesting and that p_y logged all the identifiers of the updates it received,
(v) that p_y suspected all its partners that did not follow a given step of the protocol as prescribed by the omission failure protocol,
(vi) that p_y audited all the partners it was supposed to audit.
As any other node, the source also maintains partnerships and regularly changes its partners, i.e., the nodes it serves. The source follows the partnership management and the updates exchange protocols, except that it does not send any log and it is not audited by nodes[2]. This forces the nodes to log the identifiers of the updates they received from the source, as they are deterministically chosen among the epoch membership list, which is known by all nodes. Hence, any node can check that the received updates were correctly declared. As the serving rate of the source is constant, the identifier of the updates that are released at each round are also known.

D. Update exchanges

At the beginning of each round and for the duration of their partnership, two partners, p_x and p_y exchange updates as depicted in Figure 6. Specifically, node p_x (resp. p_y) starts the exchange by generating a proposition message containing the identifiers of all the updates that appear in its log and that did not expire yet. Node p_x (resp. p_y) logs this proposition message in its log and generates the corresponding authenticator. Then, p_x (resp. p_y) sends the proposition message along with the corresponding authenticator to p_y. Upon reception of the proposition message, which it logs, node p_y (resp. p_x) selects the updates it is missing and replies to p_x (resp. p_y) with an update request. The update request is logged at the two parties. Finally, p_x (resp. p_y) serves the missing updates, and logs the serve message. Each partner then terminates the exchange by logging the identifiers of the updates it received, in its log. The nodes will then propagate the received updates during the following rounds, because we cannot ensure that nodes will immediately share them.

[2]We recall that the source is assumed to be a correct node.

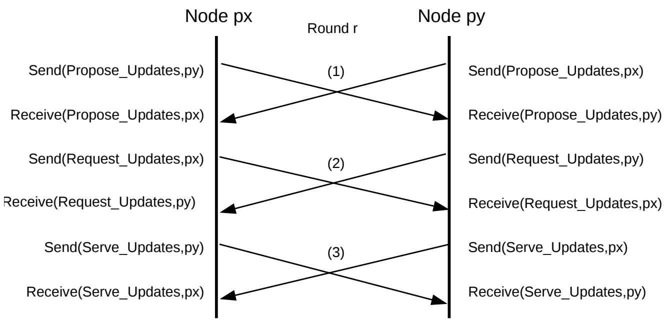


Fig. 6. Update exchanges between nodes.

V. Risk versus gain analysis

The aim of this section is to demonstrate that rational nodes will not collude with their partners, because audits will detect deviations with a high probability, and because the estimated gain of collective deviations is low. Complementary to this analysis is a proof that the protocol is a Nash equilibrium. This proof lists all the protocol steps and possible rational deviations and proves that rational nodes do not have any interest in deviating from the protocol whether individually or as a group. Due to the lack of space, this proof is available in the companion technical report [23].

We first evaluate the risk that two colluding partners take by deviating, for example when interacting as prescribed by the protocol, but without logging the updates they exchange. Specifically, consider two partners p_x and p_y that decide to collude. Assume p_x holds update *u*. To help p_y saving bandwidth in future rounds, p_y sends a proposition message to p_x that does not contain *u*, but logs that it has proposed *u*. As such, the logs of p_x and p_y appear correct if audited separately as p_x can not be blamed of not requesting *u* (as the official proposition sent by p_y does not contain *u*) and p_y can not be blamed of not proposing *u* as it appears in his log that he has done so. We define the risk as the probability that such a deviation is deterred by an audit.

Let us compute this risk. If any of the two colluding nodes is audited during the time where the exchange is contained in their log, they will be discovered. Let us consider a system of N nodes, where C nodes are part of a single colluding group. A node's log contains the entries of the last RTE rounds. A participating node initiates f partnerships with other nodes, which are changed after $Period$ rounds. Let P_{audit} the probability that a node audits each of its new partners. When establishing a new partnership, a rational node is not audited if its new partner is colluding with it (which happens with probability $\frac{C}{N}$), or if the new partner is correct but the protocol prescribes not to perform the audit. On average, each of the two nodes interacts with $\frac{2 \times f \times RTE}{Period}$ partners during the time the deviation is visible. Finally, we obtain that the risk a deviation is detected is equal to: $\left\{1 - \left(\frac{C}{N} + \left(1 - \frac{C}{N}\right) \times (1 - P_{audit})\right)^{\frac{2.f.RTE}{Period}}\right\}^2$

Let us now compute the gain of performing the above deviation. To do this we need to compute the number of interactions that a rational node may have with correct nodes that do not hold the update u after receiving it from its colluding partner, i.e., correct nodes to which the rational node would have had to send u if it has received it officially from its colluding partner. To do so, we use the algorithm of figure 7. The principle of this algorithm is that during each of the RTE rounds that follow the round at which the deviation occurred, $2 * f$ interactions happen. Each of these interactions, has a probability $\frac{C}{N}$ to involve another colluding node. When it is not the case, this other node owns the missing update with a probability that depends on the number of rounds elapsed since its release by the source. When the rational node receives the update from a correct node, it will have to share it with its future partners.

```
saved_sends_nb = 0;
received_update = **false**;
**for** round_id in 1..RTE **do**
  **for** association_id in 1..2*f **do**
    **if** $random() > \frac{C}{N}$ **then**
      **if** $random() <$ probability[round_id] **then**
        received_update = **true**;
      **else**
        saved_sends_nb = saved_sends_nb + 1;
      **end if**
    **end if**
  **end for**
  **if** received_update **then**
    **break**;
  **end if**
**end for**
**return** saved_sends_nb
```

Fig. 7. Pseudocode of the algorithm used to estimate the number of times a colluding node avoids to send an update.

In figure 8, we present a part of the algorithm that we used to determine the probability that a node receives an update r rounds after it is released by the source. Using the values of the parameters we used in our evaluations, we obtained the probabilities that are contained in table I.

Round	1	2	3	4	5	6	7	8	9	10
Probability (%)	0.12	0.48	1.9	7.3	25.6	65.5	87.4	95.4	98.3	99.4

TABLE I
PROBABILITY THAT A NODE RECEIVES AN UPDATE IN FUNCTION OF THE NUMBER OF ROUNDS ELAPSED SINCE ITS RELEASE BY THE SOURCE.

Using the average of the outputs of this algorithm, we can compute the proportion of interactions in which an update will not be sent by rational nodes. To obtain the long term gain, we multiply this proportion by the probability that a rational node has to meet an accomplice to be able to execute this deviation, which is $\frac{C}{N}$.

```
infected_nb = f;
non_infected_nb = N;
**for** round in 1..RTE **do**
  new_nodes = $\emptyset$;
  **for** node in 1..infected_nb*f **do**
    node_id = random(1, N);
    **if** node_id $>$ infected_nb **and not** new_nodes.contains(node_id) **then**
      new_nodes.add(node_id)
    **end if**
  **end for**
  **Print**(infected_nb)
**end for**
```

Fig. 8. Pseudocode of a part of the algorithm used to estimate the probability that a node received an update depending on the round number.

Computing the risk, and the gain, with the values of the parameters used in the protocol and further described in the following section, we obtain that the risk two colluding nodes take is equal to 60%, and the long term gain of the associated deviation is equal to 3%. Thus, rational nodes are exposed with a high risk each time they execute the deviation, and can only hope for a very small benefit. As a result, we conclude that rational nodes will not collude with their partners to exchange updates off the record. As said above, a complete analysis of the incentives provided by the protocol can be found in [23]. Note that nodes can still collude silently with nodes that are not their partners. Yet, it they do so, they are still obliged to execute the protocol correctly, i.e., request updates they do not officially hold and propose updates they officially hold to correct nodes. Hence, their collusion will not have any impact on the quality of the stream perceived by correct nodes.

VI. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of the AcTinG protocol. We start by introducing our methodology (in Section VI-A). Then, we compare the impact of colluders on AcTinG, BAR Gossip, and LiFTinG. We choose BAR Gossip as it is the most robust rational resilient content dissemination protocol that has been proposed so far and LiFTinG as it is the only state-of-the-art content dissemination protocol that handles colluders. We then assess the bandwidth consumption of AcTinG, its performance in the case of massive node departure (Section VI-D) and its scalability in terms of memory and bandwidth consumption using simulations involving up to a million nodes (Section VI-E).

Overall, our evaluation draws the following conclusions: In a real deployment involving 400 nodes and in presence of colluders, correct nodes using AcTinG do not experience any degradation in the quality of the content they receive while those using BAR Gossip and LiFTinG experience heavy message loss in presence of colluders independently from their organisation (whether in small or larger groups). On the other hand, we show that nodes that decide to collude in AcTinG, experience a heavy overhead, which discourages them from staying in the coalition. Moreover, we show that AcTinG bandwidth consumption is reasonable and that AcTinG is resilient to massive node departure. Finally, we show that AcTinG is scalable as simulations involving up to a million nodes exhibit that both the bandwidth and memory consumptions of AcTinG follow a logarithmic growth in the number of nodes. However, we acknowledge that the source may become a bottleneck as the number of nodes increase, as it periodically receive notifications. Solving this issue, is classically done by using a tracker, i.e., a centralised server that handles membership, as in the FlightPath protocol [22], which could easily be integrated in our system. The tracker could even be replicated using classical fault-tolerance techniques (e.g., [21]).

A. Methodology and Parameter Setting

To assess the performance of AcTinG, BAR Gossip and LiFTinG, we used them to implement three video live streaming applications. In these applications, a source node, selected randomly, diffuses a video stream at a rate of 300 kbps, during 5 minutes, and proposes each update to 5 random nodes. Updates are then disseminated using either AcTinG, BAR Gossip or LiFTinG, respectively. In order to provide a fair comparison, we implemented the three streaming applications in Java using the same code base. We deployed the three applications in 400 nodes running in one hundred physical machines of the Grid5000 cluster[3], interconnected with a 1Gb/s network that we limited to 1Mb/s. Each machine is composed of an Intel Xeon L5420 processor clocked at 2.5GHz with 32GB of RAM. In the three applications, to provide further tolerance to message loss (combined with retransmissions), the source groups packets in windows of 40 packets, including 4 FEC [4] coded packets.

The duration of one round is set to one second, and updates are released 10 seconds before being consumed by the nodes media player. Note that nodes dynamically adapt the number of their partners according to the size of the membership list: each node establishes $\left\lceil \frac{\ln(NbNodes)}{2} \right\rceil$ partnerships that it maintains for a duration of five rounds. For instance, in the fault free case, with $NbNodes = 400$, each node has 3 partners. At the beginning of each partnership, nodes performed audits with a probability of 5%, which, as we show in Section V, allows the system to detect deviations with a probability of 60% when up to 10% of the audience colludes in a single group. The cryptographic primitives consisted in a 1024-bit RSA signature and a SHA-1 hash.

B. Impact of Colluders

In this section, we experimentally study the impact of colluders on the BAR Gossip, LiFTinG, and AcTinG protocols. We implemented colluders from the code base of correct nodes in each protocol as follows. Colluders exchange unofficially among each other all the stream updates they received from correct nodes. Furthermore, colluders execute all the possible undetectable rational deviations that exist in the underlying protocol. For instance, in BAR Gossip, colluders never take part of the optimistic push protocol, which allows nodes to altruistically push updates to other nodes. Similarly, in LiFTinG, colluders do not audit the logs of other nodes and do not reply to messages sent by other nodes asking them to assess the behaviour of their previous partners unless the considered partner is among the group. As a result, correct nodes will be blamed by their correct auditors. In this situation the system administrator has two choices: (1) adjust the detection threshold to avoid false positives (by decreasing its value), which opens the doors to colluders for freeriding or (2) adjust the detection threshold to detect colluders (by increasing its value), which results in very high values of false positive accusations. In this experiment, we considered the first situation. A complementary experiment showed that in the second situation, adjusting the threshold to exclude 20% of colluders incurred the exclusion of 43% of correct nodes in the system. Finally, in AcTinG, colluders do not forward updates they received unofficially to their correct partners unless they also received them officially.

We varied the number of colluders, as well as the size of colluding groups. We measure the percentage of missed updates observed by correct nodes in presence of a proportion of colluders. We first studied the case in which all colluders belong to the same group. Results are depicted in Figure 9. The X axis presents the proportion of nodes that collude, while the Y axis presents the percentage of missed updates experienced by correct nodes in presence of colluders. We notice that correct nodes miss up to 98% of updates with BAR Gossip and 72% of updates with LiFTinG, whereas they do not miss any update with AcTinG.

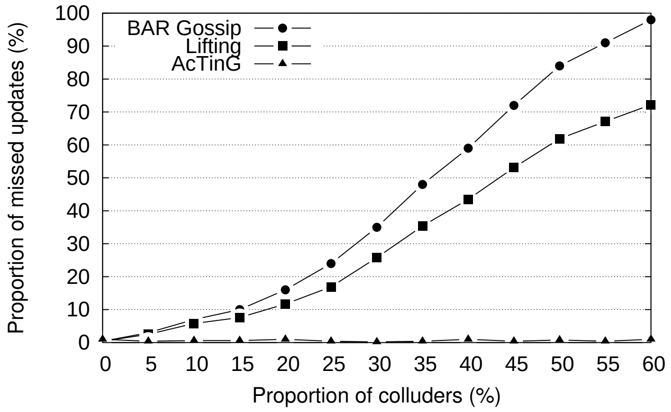


Fig. 9. Proportion of missed updates by correct nodes when a given proportion of the audience collude as a single group.

We then studied the impact of spreading colluders in multiple independent groups. More specifically, we made several experiments in which we distributed 30% of all the nodes in colluding groups of identical size. We depict the results in Figure 10. The X axis presents the size of colluding groups, while the Y axis presents the percentage of missed updates observed by correct nodes. We observe that spreading colluders in different groups has the same impact on the

[3]Grid5000: https://www.grid5000.fr/

[4]FEC stands for Forward Error Correction.

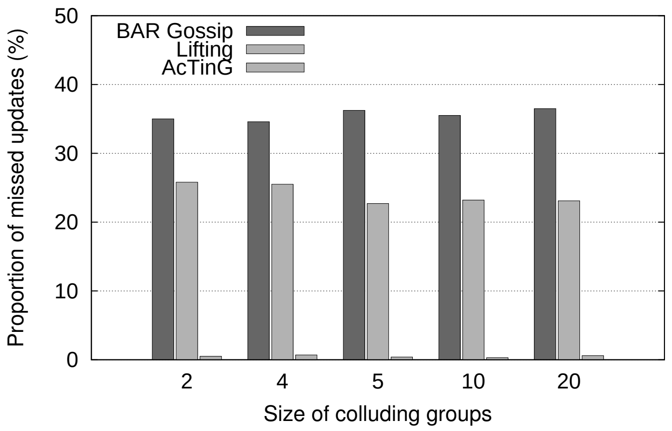


Fig. 10. Proportion of missed updates by correct nodes when 30% of the audience is rational, and collude in independent groups of equal sizes.

quality of the content downloaded by correct nodes.

The reason why correct nodes do not observe missed updates when using AcTinG, is that we designed AcTinG in such a way that colluders will eventually receive all the updates officially from their correct partners and will thus be obliged to forward them officially to their correct partners. Hence, engaging in a colluding group only yields an extra overhead due to the unofficial dissemination of updates among the group. We have measured this overhead and results are depicted in Table II. From this table we observe that the overhead due to collusion is at least of 34% of the size of the stream (case of a group containing only two colluders). In addition, as seen in section V, in a scenario where 10% of nodes collude, and where audits are performed 5% of the time, each deviation will be detected with a probability of 60%. Moreover, exchanging updates without declaring them will provide at most a gain equal to 3%. Consequently, nodes in AcTinG have no interest in colluding as they would not observe any increase in the quality of the stream they get, take a very high risk of being evicted, experience very low benefit, while suffering a useless waste of bandwidth.

Group size	2	4	8	10	50
Overhead (%)	34.35	51.53	60.12	61.84	67.33

TABLE II
OVERHEAD OF COLLUDERS IN ACTING.

C. Bandwidth consumption

To assess the overhead of AcTinG, we plot in Figure 11 the cumulative distribution of the average bandwidth consumption of nodes. Recall that AcTinG is used to broadcast a 300kbps stream. Figure 11 shows that AcTinG induces a reasonable overhead (that is mostly due to the transmission of logs). We also measured the memory consumption of AcTinG, which is due to the storage of secure logs and authenticators. Our measures have showed that a node consumes 3MB of memory for each partnership, in the worst case.

D. Resilience to massive node departure

In the case of a massive node departure, the remaining nodes need to quickly replace their left partners with alive

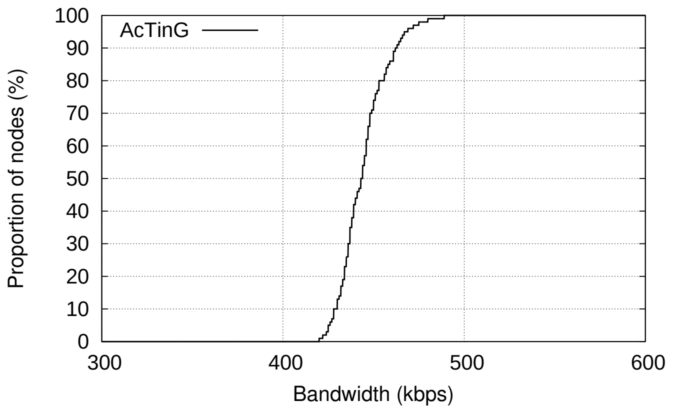


Fig. 11. Fault-free case: Cumulative distribution of average bandwidths.

nodes in order not to miss updates. In this experiment, we measure the bandwidth consumption and the percentage of missed updates when 60% and 70% of nodes suddenly leave the streaming session. Results are depicted in Figures 12 and 13 respectively. Specifically, we observe in Figure 12 that the massive node departure, i.e., which happens 500 seconds after the beginning of the experiment, immediately causes a decrease in the average bandwidth consumed by the remaining nodes, as they stop exchanging messages with their left partners. This decrease (62% and 75% in the case of the departure of 60% and 70% of nodes, respectively) is followed by an increase (of up to 18% and 27% in the former two cases), which corresponds to the messages exchanged by nodes to establish new partnerships (including a given proportion of audits). Finally, we observe that 30 seconds later, the average bandwidth consumption stabilises around 430 kbs (13% less than the original value), which is due to the decrease of the necessary number of partners per node.

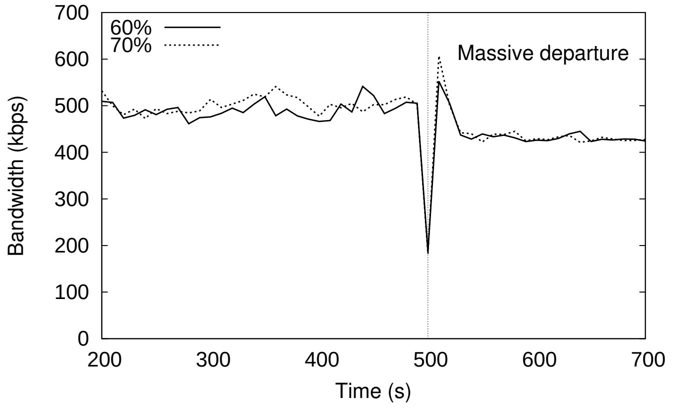


Fig. 12. Nodes average bandwidth after a massive departure.

We also compute the percentage of nodes that do not receive a viewable stream[5]. We observe in Figure 13 that only 2,5% nodes do not receive a viewable stream during the first second when 60% nodes leave the system, and between 5% and 15% nodes do not receive a viewable during at most five seconds when 70% nodes leave the system.

[5]The stream is not viewable when more than 5% of the streaming windows cannot be displayed because of missed updates [7]

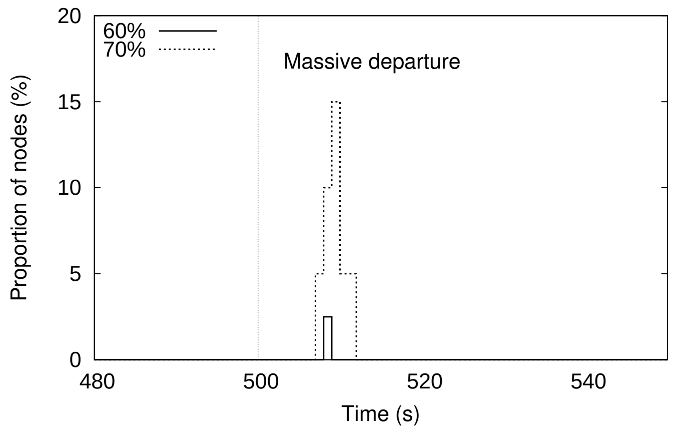


Fig. 13. Percentage of nodes that do not receive a viewable stream after a massive departure.

E. Scalability

We performed simulations to evaluate the bandwidth, and the memory consumption, of AcTinG when the number of nodes increases in the system.

Results depicted in Table III show that both the bandwidth consumption, and the memory consumption, of AcTinG grow logarithmically with respect to the number of nodes in the system. Indeed, these values depend linearly on the number of partners a node has, which grows logarithmically with the system size.

System size	Bandwidth consumption (Kbps)	Memory usage (Mb)
100	380.0	6.4
500	436.6	9.5
3,000	511.1	12.7
22,000	603.4	15.9
160,000	713.5	19.1
1,200,000	841.4	22.3

TABLE III
AVERAGE BANDWIDTH AND MEMORY USAGE OF ACTING IN FUNCTION OF THE SYSTEM SIZE.

VII. RELATED WORKS

In this section, we focus on peer-to-peer content dissemination protocols that handle rational nodes. These protocols can be classified into two categories, according to the way file chunks (called *updates* in the following) are exchanged between nodes. The first category of protocols is composed of *symmetric* protocols. These protocols force nodes to collaborate, as the number of updates they get from a node is proportional to the number of updates they have to offer (this principle is often referred to as tit-for-tat). BAR Gossip [7] and FlightPath [22] are symmetric protocols relying on game theory. Both provide incentives to ensure that rational nodes respectively have no, or a limited, interest in deviating from the protocol. In terms of robustness to rational nodes, the BAR Gossip protocol exhibits stronger properties than the FlightPath protocol. Indeed, nodes in FlightPath are assumed to deviate only if the benefit they get is higher than a threshold, which is not the case in BAR Gossip. While the authors of these two protocols point out the problem of colluding rational nodes in [7], none of these two protocols address it.

The second category of protocols is composed of *asymmetric* protocols. These protocols require nodes to altruistically push update identifiers to other nodes, which subsequently pull updates of interest. A first protocol in that category is the one presented in [24]. This protocol aims at adapting the contribution of nodes to the systems, according to their available resources. This protocol assumes the existence of trusted auditors that run in dedicated external nodes and does not deal with colluders. A second protocol in that category is LiFTinG [8]. To the best of our knowledge, LiFTinG is the only existing peer-to-peer content dissemination protocol that tackles the problem of colluding rational nodes. Specifically, LiFTinG sporadically verifies the distribution of the interactions a given node performed with other nodes in the system. Nodes that collude with other nodes break the uniform distribution of partner selection, which may result in their detection. In order to be cost effective, LiFTinG only performs sporadic audits, and relies on non-secure logs that can contain wrong information, be incomplete, be tampered with and, as a consequence, be inconsistent the ones with respect to the others. As a result, LiFTinG suffers from two major limitations: correct nodes can be wrongly evicted from the system (false positives), and a proportion of colluding rational nodes can harm the system without being detected (false negatives).

VIII. RESILIENCE TO (COLLUDING) RATIONAL NODES

In this section, we analyse the major steps of the protocol and show that rational, possibly colluding nodes do not have any interest in deviating from these steps. For each step, we consider all the possible deviations, and provide the incentives that make rational nodes follow the protocol.

We present in table IV the rational deviations associated to the integration of a new node in the system, and the incentives that make nodes stick to the protocol.

In table V, we then proceed similarly to prove that the omission failure, and node departure, handling protocols will be observed.

In table VI, we show why nodes execute the partnership protocol.

nodes among those that will be contacted by p_x or one of the correct nodes that will contact p_x during following rounds, say p_k, is correct and finds out that p_x behaved rationally through an audit, p_k could use the attestations sent by p_y and which are in p_x's log to prove that p_y behaved rationally. This will result in the eviction of p_y. As rational nodes do not want to be evicted, they do not attest for the correctness of a node without performing the corresponding verifications, and colluders do not protect themselves.

Finally, in table VII the deviations and the associated incentives concerning the exchange of updates between nodes are exposed.

Step	Description	Rational deviation	Incentive
Fig 2 Step 2	To join a content dissemination session, a new node p_n sends a join request to a contact node p_x (Step 1), which then adds p_n to its list of new nodes.	Node p_x does not add p_n to its local list of nodes, thus avoiding to inform other nodes about the arrival of a new node, saving resources and increasing the probability for its group of colluders to be served by other nodes.	During the future rounds, any node auditing the node p_n will see the identity of p_x, its contact node, in its log, and ask for the log of p_x and check that it informed its partners about the arrival of a new node in the system. If it is not the case, the node will be evicted.
Fig 2 Step 3	Node p_n then replies with the list of active nodes of the current epoch.	Node p_x ignores the request that p_n sent, and does not reply with the list of active nodes to node p_x saving its bandwidth.	If the new node does not receive a reply, it will continue to send the request periodically, and more and more frequently. As a rational node wants to preserve its bandwidth, it will always consider join requests immediately.
Fig 2 Step 3		Node p_x does not send the correct list of alive nodes in the system (e.g., it could send a truncated list, to hide the nodes it colludes with).	If the new node is audited in the RTE rounds that follow its arrival, the log of its contact node will be verified, resulting in the eviction of the latter if the list of active nodes it sent was not correct.
Fig 2 Step 4	At the beginning of the next epoch, node p_x sends to the source of the stream the list of new nodes that contacted it. It then logs the acknowledgement of the source to prove that it realized this step.	Node p_x does not send the correct list of new nodes to the source of the stream, or does not send it at all.	The node p_n that is willing to join the session is expecting to receive a message from the source, confirming its integration in the list of nodes. If it does not receive this message after some time, it will send join messages to the contact node more and more frequently, consuming its resources. Thus a rational contact node will immediately transfer the source node about the arrival of a node in the system.

TABLE IV
INTEGRATION OF A NEW NODE STEPS, AND THEIR ASSOCIATED POSSIBLE DEVIATIONS AND INCENTIVES.

IX. Conclusion

A number of gossip-based content dissemination protocols tolerating rational behaviours have been proposed. A limitation of these protocols is that they do not handle rational nodes that collude, i.e. that act as a group in order to improve their benefit. The only exception is the LiFTinG protocol that performs sporadic checks on insecure logs to try to detect colluding nodes. We have shown in this paper that LiFTinG is not effectively resilient to colluders neither is BAR Gossip the most robust rational resilient content dissemination protocol. In this paper, we have presented AcTinG, the first content dissemination protocol that tolerates rational nodes acting both individually and in collusions, and that guarantees zero false positive accusations. Performance evaluation combining both a real deployment and simulations has demonstrated that nodes running AcTinG are able to deliver the entire content despite the presence of colluders. We have also shown that AcTinG is resilient to churn, and exhibits very desirable scalability properties with a logarithmic growth of memory and bandwidth consumption, comparable to standard gossip based protocols. Our future works includes the study of the applicability of the AcTinG principles to other types of collaborative applications for the accurate detection of rational (possibly colluding) nodes.

References

[1] E. Adar and B. A. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, no. 10, 2000.

[2] R. Krishnan, M. D. Smith, Z. Tang, and R. Telang, "The impact of free-riding on peer-to-peer networks," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. IEEE, 2004, pp. 10–pp.

[3] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica, "Free-riding and whitewashing in peer-to-peer systems," *Selected Areas in Communications, IEEE Journal on*, vol. 24, no. 5, pp. 1010–1019, 2006.

[4] J. F. e Oliveira, Í. Cunha, E. C. Miguel, M. V. Rocha, A. B. Vieira, and S. V. 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.

[5] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer, "Free riding in bittorrent is cheap," in *Proc. Workshop on Hot Topics in Networks (HotNets)*. Citeseer, 2006, pp. 85–90.

[6] J. J.-D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. Epema, and H. J. Sips, "Give-to-get: free-riding resilient video-on-demand in p2p systems," in *Electronic Imaging 2008*, 2008.

[7] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "Bar gossip," in *Proceedings of OSDI'06*.

[8] R. Guerraoui, K. Huguenin, A.-M. Kermarrec, M. Monod, and S. Prusty, "Lifting: lightweight freerider-tracking in gossip," in *Proceedings of Middleware'10*.

[9] S. Ben Mokhtar *et al.*, "Firespam: Spam resilient gossiping in the bar model," in *Proceedings of SRDS*, 2010.

[10] X. Vilaça, J. Leitao, M. Correia, and L. Rodrigues, "N-party bar transfer," in *Principles of Distributed Systems*. Springer, 2011, pp. 392–408.

[11] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li, "An empirical study of collusion behavior in the maze p2p file-sharing system," in *ICDCS'07*.

[12] R. Eidenbenz, T. Locher, and R. Wattenhofer, "Hidden communication in p2p networks steganographic handshake and broadcast," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 954–962.

[13] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, 1982.

[14] C. Ho, R. van Renesse, M. Bickford, and D. Dolev, "Nysiad: practical protocol transformation to tolerate byzantine failures," in *Proceedings of NSDI'08*.

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007.

[16] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines," in *Proceedings of OSDI'10*.

Step	Description	Rational deviation	Incentive
Fig 4 Step 1	If a node p_x is expecting a message from one of its partners p_y for too long, it adds it to its list of suspected nodes.	Node p_x does not add p_y to the list of suspected nodes.	Audits check that a node sent, and received, all the messages an association implies, and, if it is not the case, that during the following rounds the node informed the other nodes about the suspected nodes. Otherwise, the audited node will be declared incorrect. Thus, as p_x can not predict whether it will be audited or not in futures rounds, it will emit the necessary suspicion messages regarding p_y.
Fig 4 Step 2	Then, it sends a suspicion message to the other partners of p_y.	Nodes p_x does not send a suspicion message, for example to protect an accomplice.	Audits check that a node sent, and received, all the messages an association implies, and, if it is not the case, that suspicion messages were sent. Otherwise, the audited node will be declared incorrect. Thus, as p_x can not predict whether it will be audited or not in futures rounds, it will emit the necessary suspicion messages regarding p_y.
Fig 4 Step 2		Node p_x, and the partners of node p_y, collectively decide to exclude node p_x from the system, even though it is correct.	Suspecting a node is made more costly than a normal interaction, and the expected benefit of excluding a correct node is low, because correct nodes are a source of updates for other nodes. In addition, the probability for such a situation to occur is extremely small (1 out of 10 millions, when 10% of the nodes collude).
Fig 4 Step 3	Then, each of p_y's partners pings p_y.	The partners of node p_y does not ping it, as the node p_x asked them	If node p_x does not receive a message from a partner p_z of node p_y, it will contact the partners of node p_z and ask them to obtain the missing answer from p_z. Thus node p_z will have to answer, if he wants to avoid being suspected, and to consume more bandwidth than if it answered directly to node p_y. Thus, a rational node will always immediately execute the ping procedure.
Fig 4 Step 4	The partners of p_y then reply to p_x with a signed message certifying whether p_y replied to the ping message or not.	The partners of node p_y lie saying that it is responding, either to save bandwidth or to protect it.	Nodes cannot lie because ping messages, and their answers, contain log entries which testify the time at which the messages were sent by nodes. Thus, to say that a node is responding, another node needs to communicate with it, and rational nodes have no way to protect their partners.
Fig 4 Step 5	After having received the confirmation that node p_y is not responding, node p_x removes p_y from its list of nodes.	The node p_x does not remove p_y from the list of nodes in the system, thus avoiding it to be evicted from the system.	The partners of node p_y sent messages that can not be forged, and if all of them indicate that the node is not responding, then p_x would be evicted if an audit detected that it did not removed p_y from the list of nodes in the system.
Fig 3 Step 4	Node p_y is informed that node p_x is expecting a given message m. It then sends this message to node p_x and to its partners.	The node p_y does not send the message m that was not received to the partners of node p_y and to node p_x.	If the node p_y does not send the expected message, the suspicion will not be released, and it will eventually be evicted from the system, as other nodes will refuse to interact with it.
Fig 3 Step 5	The partners of node p_y confirm to node p_x that they received the message from node p_x, joining to the message the autenticator that node p_y sent along with message m.	The partners of node p_y lie to node p_x, saying that it is not responding, to evict it from the system.	Is it not clear if this deviation is rational, however, the probability that such a deviation occurs is small. For example, if nodes maintain 3 partners, and 10% of the audience colludes, then such a deviation can occur with probability 10^{-6}. If one node replies that node p_y is responding, and proves it with the authenticator, then node p_y will not be evicted from the system.
Fig 3 Step 6	Node p_x receives the message m it was expecting, and remove p_y from its list of suspected nodes.	The node p_x does not remove p_y from the list of suspected nodes, aiming at isolating it from other nodes.	Evicting correct nodes does not bring a clear benefit, because correct nodes are those that propagate updates. In addition, if an audit occurs, it will appear that the node p_x is suspecting a node even though it received messages that proved that the node sent the message it was supposed to send. If this node does not appear in the node's log, for example because it does not want to log it, then the costly suspicion procedure has to be executed once again, which clearly will degrade the resources consumption of node p_x.

TABLE V
HANDLING OF AN OMISSION FAILURE, OR A NODE DEPARTURE, AND THE ASSOCIATED POSSIBLE DEVIATIONS AND INCENTIVES.

Step	Description	Rational deviation	Incentive
Fig 5 Step 1	Every $Period$ rounds, node p_x stops exchanging with its f partners and deterministically selects f new partners using a pseudo-random number generator seeded with a deterministically computed seed (e.g., the round number concatenated with p_x's public key).	Node p_x tries to establish a new partnership with nodes whose IDs are other than those computed using the PRNG function, e.g., to interact with colluders.	A rational node p_x will never select such nodes as it risks eviction during the next audit. Indeed, node p_x can be selected by a correct node, say p_0, in the future, which will verify whether p_x effectively selected the nodes it was supposed to interact with by examining its log. If node p_0 detects such a deviation, it will expose p_x.
Fig 5 Step 1		Node p_x tries to establish less than f partnerships to save bandwidth	The same incentive as above holds.
Fig 5 Step 3	When node p_x proposes to start a new partnership to node p_y, p_y checks that p_x had to contact it by rerunning the PRNG. If the check succeeds, p_x and p_y will exchange during the next round.	Node p_y does not reply to the proposition message sent by p_x.	If p_x does not receive a reply after it sent its proposition, it will suspect p_y. Not doing so would expose it during the next $Period$ rounds, and prevent it to interact with any correct node. In order not to be evicted, p_y will answer to the partnership request.
Fig 5 Step 3		Node p_y replies without verifying the legitimacy of p_x's request, which could happen if nodes p_y and p_x are colluders, and try to protect themselves.	We distinguish two cases: (a) p_x is correct, and (b) p_x is rational and the verification p_y has to perform should not pass. If p_x is correct and p_y attests that p_x has passed the verification without effectively performing it there is no way to detect that p_y is behaving rationally. However, as p_y does not effectively know whether p_x is correct or not, p_y risks eviction as well as p_x. Hence, p_y will prefer to verify whether p_x was supposed to contact it or not. Instead, if p_x is rational and p_y attests that p_x passed the verification without performing it, p_y risks eviction. Indeed, if one of the following nodes among those that will be contacted by p_x or one of the correct nodes that will contact p_x during following rounds, say p_k, is correct and finds out that p_x behaved rationally through an audit, p_k could use the attestations sent by p_y and which are in p_x's log to prove that p_y behaved rationally. This will result in the eviction of p_y. As rational nodes do not want to be evicted, they do not attest for the correctness of a node without performing the corresponding verifications, and colluders do not protect themselves.
Fig 5 Step 3, 4	Node p_x deterministically decides whether to audit p_y's log or not.	A node does not audit its partner when it should.	When this node will start new partnerships in future rounds, its log will possibly be audited. During this audit its future partner recomputes the boolean that indicates whether the node should have audited its previous partners or not. If the node did not perform the audits while it was supposed to do so or did not perform them correctly, it will be exposed by its partner. As the node cannot predict the occurrence of its future audits, it will audit its current partners, and will contact all their own partners to get their logs, following the results of the deterministic computation it has to perform.
Fig 5 Step 5	If the result of the computation implies that p_x must audit p_y, the former contacts all p_y's partners and ask for their log.	Node p_x does not contact the right set of nodes to obtain their logs	The same incentive as above holds.
Fig 5 Step 6	The partners of p_y reply with their logs.	Nodes do not reply with their logs, either to save their bandwidth, or to protect node p_x from being declared faulty.	A node that does not send its log will be suspected of omission failure, and will have to handle a suspicion procedure, which is costly, and will eventually have to send the missing log to delete this suspicion. A rational node has no interest in refusing to send its log.
Fig 5 Step 7	Upon receiving the logs of p_j and p_i, one of p_y's partners, p_x checks that the two nodes interacted correctly.	Upon receiving the logs of p_j and p_i, p_x claim that they are correct whithout effectively performing the necessary verifications to save resources.	If p_x skips some or all the verifications described in the audit protocol, its risks to be exposed by p_y or p_j future partners that will detect their misbehaviour. As p_x does not want to take such risk, it will correctly perform the audit of p_y and p_j's logs.
Fig 5 Step 7		Node p_x does not denounce node p_y when it is discovered faulty to protect it from being evicted from the system.	The same incentive as above holds.

Step	**Description**	**Rational deviation**	**Incentive**
Fig 6 Step 1	Consider node p_y among the set of partners selected by p_x. Node p_x contacts p_y and sends it a proposition message containing the updates it owns.	Node p_x does not send a proposition message to node p_y	Each correct partner of p_x expects to receive its proposition at the beginning of each round, and will thus suspect p_x if the latter does not send it. Hence, to avoid being suspected, p_x always sends a proposition to its partners.
Fig 6 Step 1		Node p_x sends an invalid proposition message to node p_y (e.g., including less updates than what it holds).	Furthermore, the proposition sent by p_x is necessarily correct (i.e., includes all the updates that appear in p_x's log), otherwise it risks eviction by its future partners if one of them audits its log.
Fig 6 Step 2	Then, p_y replies with a request message containing the updates it is missing.	Node p_y formulates an invalid request.	Regarding the formulation of requests by p_y, the latter risks immediate eviction if the request is incorrect. Indeed, p_x would hold a proof of misbehaviour of p_y, which it would immediately send to the source of the content dissemination session.
Fig 6 Step 3	Finally, p_x serves the updates requested by p_y.	Node p_x serves less updates than what node p_y requested to save resources.	A similar ending would happen to p_x if it does not serve the updates expected by p_y.
Fig 6 Step 1, 2, 3		Node p_x colludes with node p_y and exchange updates, but temper with their logs to make other nodes believe they did not, thus avoiding to exchange updates, and saving their future bandwidth.	If instead, p_x colludes with its partners in order to exchange updates off the record, the group risks to receive most of the updates a second time from correct nodes in future exchanges waisting their bandwidth.

TABLE VII
UPDATE EXCHANGE STEPS, AND THEIR ASSOCIATED POSSIBLE DEVIATIONS AND INCENTIVES.

[17] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: small trusted hardware for large distributed systems," in *Proceedings of NSDI'09*.

[18] B. gon Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *Proceedings of SOSP'07*.

[19] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *IEEE Computer*, vol. 37, no. 5, 2004.

[20] A. Aiyer *et al.*, "Bar fault tolerance for cooperative services," in *Proceedings of SOSP*, 2005.

[21] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[22] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin, "Flightpath: obedience vs. choice in cooperative services," in *Proceedings of OSDI'08*.

[23] S. Ben Mokhtar, J. Decouchant, and V. Quéma, "Acting: Accurate freerider tracking in gossip," University of Grenoble, Tech. Rep., 2014, https://sites.google.com/site/soniabm/.

[24] R. van Renesse, M. Haridasan, and I. Jansch-Porto, "Enforcing fairness in a live-streaming system," in *Proceedings of MMCN'08*.