Accountability in Distributed Systems

Nimit Singhania
University of Pennsylvania

September 8, 2014

Abstract

Nodes in distributed systems can become faulty due to multiple reasons: software or hardware bug, node misconfiguration or node being compromised by an attacker. Often such faulty nodes can be difficult to detect and localize. An approach to solve this problem is to hold each node accountable for its actions i.e. each node is required to provide verifiable evidence for its actions and detect a node as faulty if it fails to provide correct evidence. Thus, accountability of nodes can help detect faulty nodes and address faults quickly.

In this report, we present a general framework for accountability in distributed systems and use this to review three accountability protocols: PeerReview, which provides a generic accountability protocol for distributed systems where each node follows a deterministic protocol; CATS, which is an accountability protocol for network storage; and Network Professional, which provides a protocol for accountable measurement of domain performance in inter-domain network routes. Finally, we compare and contrast these protocols and derive conclusions which can be useful for future work.

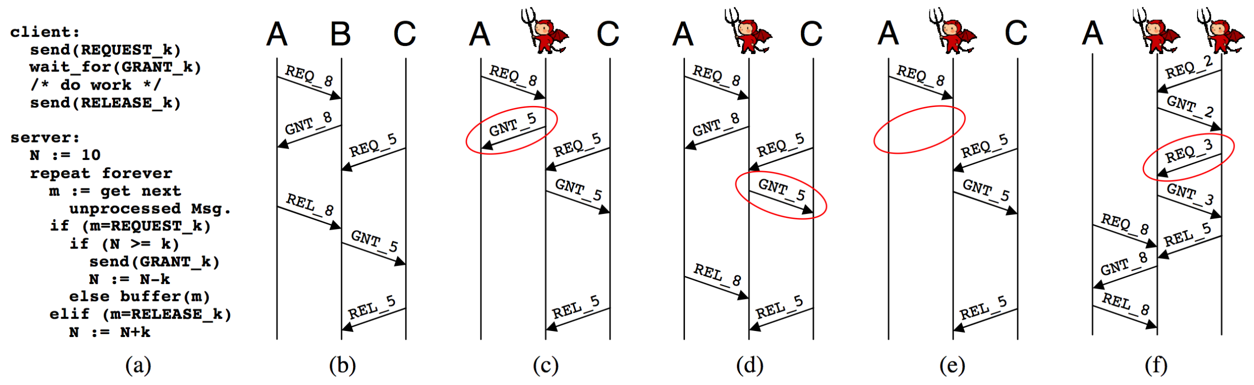
1 Introduction

A distributed system is a set of computer nodes that interact with each other by sending messages and work together towards a common goal. Each node in the system follows a specific protocol. It is analogous to a team or a corporation where every person does a specific job to achieve the goal collectively. If a person in the team does not perform his required role, the whole team can be affected. Similarly, if a single node in the distributed system malfunctions, the output of the distributed system can be adversely affected. Thus, such faulty nodes need to be detected in time. However, often, these nodes can get away without being caught because there is not enough evidence to put blame on them.

Accountability in distributed systems ensures that each node is held accountable for its actions. To do this, the distributed system is augmented with an accountability protocol. In the protocol first, a notion of correctness of a node is established and this is used to verify if a node is correct or not. Next, evidence is collected by each node in the system that can be used to substantiate its claim of correctness. Finally, a mechanism is established to inspect the evidence provided by nodes and check if a node is correct or faulty. Note that, the overhead of evidence collection and evidence inspection can be huge and this is often reduced at the cost of having only probabilistic guarantees, i.e. allowing a small probability of faulty nodes not being detected. Further, not all faults can be detected even if the accountability protocol works correctly, because the faulty nodes might collude such that they appear correct to the correct nodes.

We use an example from [3] on distributed storage to elaborate this further as shown in Figure 1. Here, the client and the server are required to follow the protocol as given in Figure 1(a) and any deviation from the protocol corresponds to a violation. Any node (client) requesting resources sends a `REQUEST_k` message to another node (server) and waits for the server to allocate the resources and reply with a `GRANT_k` message. When it is done using the resources, client sends back a `RELEASE_k` message. Note that, a client needs to release its resources before requesting any more resources. Similarly, a server can allocate at most 10 resources and thus, processes requests only when it has the required resources available. Figure 1(b) represents a correct execution of the system while, in Figure 1(c)-(f) either the server or the client deviate from the protocol.

Any accountability protocol broadly consists of the following aspects:



```
client:
  send(REQUEST_k)
  wait_for(GRANT_k)
  /* do work */
  send(RELEASE_k)

server:
  N := 10
  repeat forever
    m := get next
       unprocessed Msg.
    if (m=REQUEST_k)
      if (N >= k)
        send(GRANT_k)
        N := N-k
      else buffer(m)
    elif (m=RELEASE_k)
      N := N+k
```

Figure 1: Example from [3]

1. **Notion of Correctness**: The accountability protocol associates with each node, a notion of correctness such that the node is *correct* if it satisfies the correctness properties and *faulty* otherwise. In the example described above, a node is correct if it follows the protocol and faulty if it deviates from the protocol.

2. **Evidence Collection**: Each node needs to collect evidence that can be used to substantiate its claim of correctness. It needs to be such that anyone inspecting the evidence can be convinced about the correctness of the node. Each node needs to collect two kinds of evidence.

   (a) **Evidence for self-correctness** - This is the evidence that can be used to show that the node is correct assuming that its interaction with other nodes is known correctly. In the above example, this corresponds to the evidence that a node followed the protocol correctly assuming that the messages received and sent by the node are known correctly. So, in Figure 1(d), while A and C should be able to give evidence for correctly following the protocol, B should not be able to give the correct evidence if the messages it sent and received are known correctly.

   (b) **Evidence for mutual-correctness** - This is the evidence about a node's interactions with other nodes and can be used to check consistency of the evidence. A node might deny sending or receiving a message or claim to have sent/received a different message in a different order. This evidence can be useful in such scenarios. For example, in Figure 1(c), B might falsely claim that it received `REQUEST_5` from A. Thus, A needs to have evidence that it sent `REQUEST_8` and B received it, to defend against this claim.

3. **Evidence Inspection**: The protocol needs to provide a mechanism to inspect the evidence provided by the nodes. The mechanism consists of the following.

   (a) **Consistency** - First, the protocol needs to check if the evidence provided by a node is consistent. So, it needs to check if the node provides the same evidence to all the other nodes and if not, it is considered faulty. Further, it needs to check if all the messages that the node sent or received are accounted for in the evidence and no additional messages are used to construct the evidence. If both these conditions are satisfied then the evidence is consistent. This is important because, a node might get away by providing different version of evidence to different nodes or disregarding certain messages that it received or sent. For example, in Figure 1(d), server B is correct with respect to A and C individually. Thus, B might claim that it never interacted with C and might not be detected if the messages it sent or received from C are not accounted for. Similarly, it might present different evidence to A and C. It might claim to A that it only interacted with A and similarly it might claim to C that it only interacted with C. However, the evidence is different in both these cases and B can be caught if A and C exchange the evidence provided by B.

   (b) **Audit** - After the evidence provided by a node is found to be consistent, the protocol needs to check if the evidence is correct. Evidence provided by a node is correct, if it proves that the node satisfies all the required correctness properties. In the above example, the evidence is correct if it proves that the node followed the protocol correctly. To do this, the execution of a node is replayed using the inputs and messages that it received and is compared against the evidence provided. If the evidence and the replayed execution do not match then, the node is considered faulty. For example, if the evidence provided by B in Figure 1(d) is consistent, then it would not match the replayed execution as server can not grant more than 10 resources at a time. Hence, B would get caught during audit, if it is not caught during consistency check.

(c) **Challenge/Response** - Finally, some nodes might hesitate to provide evidence of their correctness. Such nodes need to be challenged to provide evidence and if they refuse, appropriate action needs to be taken. For example, in Figure 1(e), B refuses to respond to A and provide evidence of receiving A's request. The protocol should provide a mechanism to handle such scenarios as well.

4. **Probabilistic Guarantees**: There is a significant overhead in both evidence collection and evidence inspection. Thus, the accountability protocol might give means to reduce the overhead at the expense of providing only probabilistic guarantees of detecting faulty nodes. Thus, there might be a small chance that a faulty node gets away without being detected.

5. **Fault Detection Power**: The accountability protocol can only detect faults where either a node refuses to provide evidence or the evidence is inconsistent or incorrect. If none of the above is true, then faulty nodes can get away without being detected. For example, in Figure 1(f), nodes B and C might collude and give fake evidence that C sent `REQUEST_5` to B, to which B responded with `GRANT_5` and finally, C sent `RELEASE_5` to B. Thus, B might hide the faults of C by providing fake evidence and C might get away. Hence, accountability protocol is limited by this and can not detect faults when faulty nodes collude to give fake evidence that is consistent and correct with respect to the evidence provided by other nodes.

An accountability protocol needs to ensure that a correct node is never falsely implicated and a detectably faulty node is eventually detected. Evidence for self-correctness helps check the correctness of a node and evidence for mutual-correctness helps defend correct nodes against false implications. Thus, if a protocol ensures that each node collects both kinds of evidence and it is checked for consistency and correctness then the above required property is satisfied.

Now, we describe three accountability protocols and how they handle different aspects of an accountability protocol as described above. First, we describe PeerReview [3], which is a general accountability protocol and can be applied to any distributed system in which each node follows a deterministic protocol. Then, we describe CATS [5], which is an accountability protocol for network storage. It holds the storage server accountable for all its actions and the server in turn holds clients accountable for their requests. Finally, we describe Network Professional [1], which provides a protocol for accountable measurement of domain performance in inter-domain routes or paths.

2 PeerReview

PeerReview [3] provides a general accountability protocol that can be applied to a distributed system where each node follows a deterministic protocol i.e. given a fixed sequence of inputs and received messages, the node executes the same sequence of steps. PeerReview assumes that each node has a private/public key pair and can sign messages and another node can not forge its signature. This is necessary to irrefutably associate a message with a node. If a node A receives a message signed by another node B, then A has evidence that B signed the content of the message and thus, B can not deny producing the message.

To apply PeerReview to a node in the distributed system, the node is split into two components, State Machine and Application. State Machine is the component that needs to be observed for correctness by PeerReview and Application handles the remaining functionality of the node. State Machine is assumed to be deterministic. Further, a new component Detector is introduced in the node that can observe the input and output of the State Machine and also the messages sent and received by it on the network. Detector can also communicate with the Detector component in other nodes of the distributed system. Now, we describe how PeerReview implements different aspects of an accountability protocol.

1. **Notion of correctness**: A node is considered correct, if the State Machine follows the protocol which is given by a reference implementation and faulty if it deviates from the protocol.

2. **Evidence Collection**:

   (a) **Evidence for self-correctness** - The evidence for self-correctness consists of a secure *log* that records the inputs and outputs and messages sent or received by it in a chronological order. The log also stores periodic state snapshots and annotations from the detector. A log entry $e_k = (s_k, t_k, c_k)$ consists of a monotonically increasing sequence number s_k, type of entry t_k and type specific content c_k. Further, it contains a recursively defined hash-value $h_k = H(h_{k-1}||s_k||t_k||H(c_k))$, where H is a collision resistant hash function. Note that, the hash values are such that a specific value, h_k represents a unique sequence of log entries upto e_k. An *authenticator* $\alpha_k^j = \sigma_j(s_k, h_k)$ is a signed statement by a node j that its log entry e_k has a hash value h_k, where $\sigma(m)$ represents a message m signed with the private key of node j. Authenticators are used as evidence of mutual-correctness as we describe next.

   (b) **Evidence of mutual-correctness** - When a node i sends a message m to j, j needs evidence of the fact that i sent the message m, so that i can not later deny sending m to j. It also needs evidence about the step in i's execution when it sent m, so that i can not claim to have sent m in a different step. Similarly, i needs evidence of j receiving m and the step in which it receives m. Thus, before sending m to j, i creates a log entry $(s_k, \texttt{SEND}, \{j, m\})$ and sends h_{k-1}, s_k, and $\sigma_i(s_k, h_k)$ along with m to j. Now, j uses the information to check if the signature α_k^i is valid from the information and discards the message if not. Otherwise, α_k^i is the required evidence for j. Similarly, j creates a log entry $(s_l, \texttt{RECV}, \{i, m\})$ and sends h_{l-1}, s_l and $\sigma_i(s_l, h_l)$ to i, which gives the evidence α_l^j to i.

3. **Evidence Inspection**:

   (a) **Consistency** - With each node j, we associate a set of nodes, known as *witness set*, $w(j)$ that is given the responsibility of checking consistency of the evidence provided by j. It is assumed that at least one of the nodes in $w(j)$ is not faulty and can expose j, if the evidence provided by j is found to be inconsistent. Whenever a node i receives an authenticator α_k^j from node j, it forwards it to the witness set $w(j)$. Thus, $w(j)$ receives evidence regarding all the messages j has sent or received. Each witness in $w(j)$ picks the authenticators with smallest and largest sequence numbers and challenges j to send log entries in between these sequence numbers. If hash chain obtained from the log entries returned by j matches the authenticators obtained earlier, then, the evidence is considered consistent. Otherwise j is detected as faulty.

   Further, each witness in $w(j)$ extracts the authenticators received by j in the inspected log entries and forwards them to witnesses of corresponding nodes. This is necessary because, j might act as a faulty accomplice of a node k and might not forward authenticators from k to $w(k)$ and thus, help k provide fake evidence with no messages sent to/received from j. Note that, $O(|w(j)|.|w(k)|)$ messages are exchanged between witnesses for each message exchanged between j and k and thus, it has a message complexity of $O(|w(j)|.|w(k)|)$.

   (b) **Audit** - Once the evidence provided by a node is found to be consistent, it is now checked for correctness. For this, each witness ω in $w(j)$ asks j to return log entries starting from its last audit up to e_k corresponding to the most recent authenticator α_k^j. Then, ω initializes the reference implementation of j with the snapshot from last audit and replays the inputs and compares the outputs from the implementation with that in the log entries. Since the state machine of j is deterministic, j is detected as faulty if the outputs don't match.

   (c) **Challenge/Response** - When a node j refuses to acknowledge receiving a message m, a request is sent to the witnesses of j to challenge j. The witnesses challenge j to acknowledge m and return the evidence of receiving m. If j refuses to respond to a challenge from the witnesses (it could also be a consistency or audit challenge), then the witnesses indicate j as *suspected*. Depending on the distributed system, an appropriate action can be taken for suspected nodes.

4. **Probabilistic Guarantees**: The overhead of implementing this protocol can be huge as for each message exchanged in the actual protocol, about ψ^2 messages need to be exchanged between witnesses in the consistency check. Here ψ is the size of witness set and it needs to be greater than the number of faulty nodes so as to ensure that there is at least one correct witness in the witness set. However, if we allow a small probability for an all-faulty witness set, ψ can be $O(\log N)$, where N is the total number of nodes in the system, which reduces the message complexity to $O(\log^2 N)$. Similarly, if we allow a small probability of an instance of misbehavior or inconsistency being undetected, then the message complexity of consistency check reduces to constant via a randomized consistency check protocol.

5. **Fault Detection Power**: As described in [2], the protocol can only detect commission and omission faults. An *omission* fault is one where a faulty node refuses to acknowledge a message, in which case, the node is suspected by a witness (for example, Figure 1(e)). A *commission* fault is one where a faulty node sends an incorrect message to a correct node and thus can be caught by a witnesses during audit or consistency check (for example, Figure 1(c)-(d)) A *non-observable* or *ambiguous* fault is one where, the fault can not be observed by a correct node since it does not receive any incorrect messages. For example, faulty nodes might collude such that they appear correct to the correct nodes, even though they deviate from the protocol (for example, Figure 1(f)). In such cases, they can get away by presenting fake evidence to the correct nodes, and thus, such faults can not be detected.

3 CATS

CATS [5] presents a network storage service with strong accountability properties. It allows clients to read and write to objects in a shared directory maintained by a server. The server maintains evidence of its updates to the objects and allows clients to verify that their write requests are executed correctly and that their read requests return correct values of objects. The server also holds clients accountable for their write requests and clients can not deny sending the request later. Similar to PeerReview, CATS assumes that each client and the server has a private/public key pair and can sign messages and another node can not forge its signature. It also assumes that the server has access to a trusted external publishing medium where it can periodically publish digests over its state and that the published digests can be accessed independently and correctly by each client.

Now, we present how CATS implements different aspects of an accountability protocol.

1. **Notion of correctness**: CATS only specifies correctness properties of the server. The properties are as follows:

    - **Authenticity**: The server executes only writes issued by authorized clients.
    - **Freshness and consistency**: Writes are applied in order and reads return the value of the latest write.
    - **Completeness**: Writes to objects are visible to all authorized clients.

2. **Evidence Collection**: The server maintains two forms of evidence, action histories and published digests.

    - **Action Histories**: With each object, the server associates a sequence of write requests that affected or produced the current value of the object. The write requests are signed by clients, so that they do not deny sending the request later. The write requests also include the previous version stamp of the object so as to certify that client knows the version of the object at the time of write. This also prevents the server from reordering writes.
    - **State Digests**: The server periodically generates signed digests over its local state and publishes them to the external medium. The data structure used to store objects and their values provides a way to compute a fixed size hash over its contents. The obtained hash is signed by the server to produce the required digest and represents a summary of the current state of the server. Further, the data structure also provides a way to generate *inclusion* or *exclusion* proofs. An inclusion proof gives evidence of the fact that a given value of an object was used to generate the digest. An

exclusion proof gives evidence that the object did not exist in the state that was used to compute the digest. Inclusion and exclusion proofs can be used to verify correctness of the state with respect to its published digest. Further, previous digest value is included in the computation of a new digest. Thus, the digests form a chain which prevents the server from inserting or deleting digests.

The above forms of evidence can be used to give evidence of self-correctness and evidence of mutual-correctness.

(a) **Evidence for self-correctness -** The digests are periodically computed and published onto the external medium and thus are visible to all clients. This can be used as a proof of completeness and clients can check if their view of server is consistent with the digest. Further, as evidence for a value of an object, the server can give the sequence of signed write requests that led to this value relative to a published digest and the inclusion or exclusion proof of the object being correctly included in or excluded from the digest. The values of objects in a digest can be substantiated similarly relative to a previously published digest.

(b) **Evidence of mutual-correctness -** Clients sign any write or read requests while sending them to the server and these signed messages can be used as evidence by the server. Similarly, server signs any response to the clients and this can be used as evidence by the client.

3. **Evidence Inspection**:

   (a) **Consistency -** Unlike PeerReview where a witness collects all the messages that a node sent or received and uses them to verify the consistency of the evidence, here the server makes its evidence public and available to all clients as periodic state digests and the onus lies with the clients to check if their writes are correctly incorporated in the digest or the reads are consistent with the digest. Since each client is responsible for checking the consistency of the digests with respect to their requests, no separate mechanism exists to check consistency of the evidence provided by the server.

   (b) **Audit -** Next, we need to check the correctness of published digests once it is known that all the client requests are consistent with the digests. To do this, CATS allows clients to audit digests in some span or interval of time for the objects that they are allowed access to. When a client makes such an audit request for an object, the server returns the history of signed write requests to the object during the interval which asserts that the value of the object was correctly computed and also inclusion/exclusion proof for the digests in the interval which asserts that value of the object was correctly incorporated into the digests. This allows clients to check that the object was written by only authorized clients during the interval (authenticity), that the writes were applied in order and correctly incorporated into the digests (consistency) and also, no writes were reverted later on and thus any subsequent reads return fresh value of the object (freshness). Note that, completeness is also fulfilled since the writes are incorporated into the digests which are visible to all clients.

   (c) **Challenge/Response -** CATS does not provide any mechanism to handle the scenario where server refuses to respond to clients or provide evidence. However, the goal of this work is to have *semantic* accountability rather than *performance* accountability and thus, this is not a concern.

4. **Probabilistic Guarantees**: Auditing all digests in an interval of time can be expensive for the server and thus, CATS allows clients to audit randomly selected digests in the interval. This however leaves a possibility that the digest which reveals misbehavior might not be audited and thus server might get away. So, if the probability of selecting a digest that reveals misbehavior in the interval is p and d is number of digests that are selected from the interval, then the probability that the misbehavior is detected is $1 - (1 - p)^d$. Thus, the probability of detecting misbehavior comes exponentially close to 1 as we increase d.

5. **Fault Detection Power**: Unless server colludes with clients, any misbehavior by the server can be observed by the clients and the server can be caught via discrepancy in the evidence provided by the server.

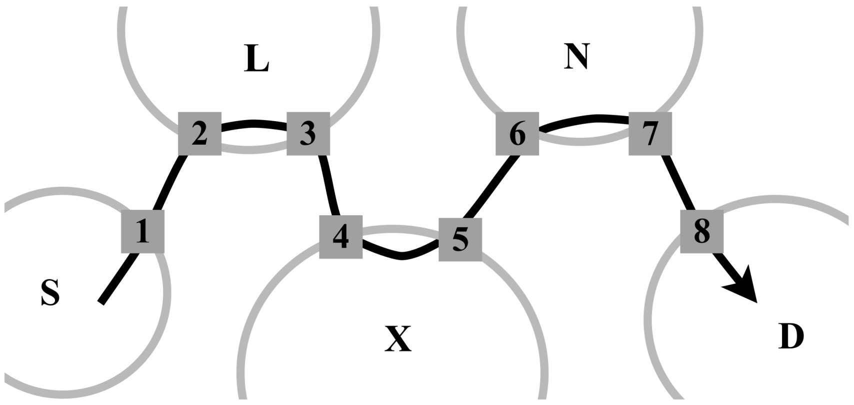


Figure 2: Example from [1]

4 Network Confessional

Internet is split into multiple domains controlled by different authorities and any packet on Internet often travels through multiple such administrative domains to reach its destination. Internet due to its best effort nature provides no way to measure performance of these domains. So, when there is loss of performance or violation of Service Level Agreement (SLA) along a route or path consisting of multiple domains, there is no way to detect which domain has the problem and this makes debugging slow. Network Confessional [1] gives a protocol via which it is possible to accountably measure domain performance on Internet routes. In the protocol, each domain self reports its performance but needs to back it up with verifiable evidence which prevents it from biasing its performance greatly. Evidence consists of traffic receipts on sampled packets at entry and exit points of the domain when the packets flow through the domain on the given route. Further, the protocol provides a way for each domain to reduce the overhead of evidence collection at the expense of reduced quality of the measurement provided.

We describe some terminology first. A *domain* is a contiguous network that falls under a single administrative domain. A *path* is a sequence of nodes in the network on which packets are streamed from the source to destination, where each node corresponds to entry or exit point of a domain lying on the route. Figure 2 describes an example from [1]. The entry point of a domain is called an *input* node and the exit point an *output* node. For example, input and output nodes of domain X are 4 and 5 respectively. Packets enter a domain at its input node and exit out of it at the output node, and thus, the performance of the domain is measured by collecting statistics about packets at these nodes. Further, adjacent nodes from different nodes are called *peering* nodes (for example, nodes 3 and 4). A required characteristic of peering nodes is that since the link between them is short, significant loss of performance should not occur in between and thus, the packet statistics at both nodes should be similar. Thus, the protocol assumes that the link between two peering nodes i and j is *faulty* if it introduces packet loss, or reordering, or a delay beyond a value Δ_{ij} that is pre-negotiated between nodes.

The protocol measures two performance characteristics for each of the domains on a path k: *Loss Rate*, λ_{ij}^{k} which denotes the amount of packet loss experienced by path-k traffic between input node i and output node j of a domain; and *Delay*, $\delta_{ij}^{k}(q)$ which denotes the minimum value such that q-th fraction of this traffic experiences a delay less than this value. To collect these measurements, nodes collect receipts on some subset of packets flowing through them which include packet id, timestamp and some node specific details. Then, they send the collected receipts securely to other domains on the path and possibly a regulator, which computes loss rate and delay using them. Further, the correctness of these measurements depends on the assumption that nodes only lose, delay or reorder packets and they do not inject new packets or modify observed packets.

Now, we describe how the protocol implements different aspects of the accountability protocol.

1. **Notion of correctness**: The goal of the protocol is to measure performance of domains in a path and identify domains with poor performance in case of performance problems with the network. Hence, for each domain, there is no discrete notion of correctness but, only a quantitative measure of how good or bad a domain performs in terms of loss rate and delay.

   Further, a link between two peering nodes i and j is considered faulty if it introduces packet loss, or reordering, or a delay beyond a value Δ_{ij} that is pre-negotiated between nodes. This can also be used to detect faults in the network.

2. **Evidence Collection**: A trivial technique to collect evidence for a domain's performance is to collect receipt for each packet that flows through its input and output nodes, i and j on the path k. Then, the loss rate λ_{ij}^k is estimated by the count of packets observed at i minus the count at j. Similarly, the delay $\delta_{ij}^k(q)$ is computed by comparing timestamps recorded for the same packet observed at both nodes i and j. However, this can have significant overhead on each node and thus, the protocol gives a way to tune the overhead at the expense of having probabilistic guarantees for approximately correct measurements i.e. the protocol ensures that the measured value $\hat{\lambda}_{ij}^k$ is such that, $|\lambda_{ij}^k - \hat{\lambda}_{ij}^k| < l_{ij}$ with probability π_{ij} for some values of l_{ij} and π_{ij} and similarly for $\delta_{ij}^k(q)$. Now, we describe how each node collects evidence.

   The key idea of the protocol is to generate receipts for only a subset of packets flowing through each node, and yet ensure that the measurements are approximately correct and are not biased by the nodes. If a node knows a-priori the packets for which the receipts are to be generated, it might treat them preferentially say by putting them in a high-priority queue and bias the measurements. Hence, in the protocol, the subset for which nodes generate receipts is selected based on *future* incoming traffic, preventing nodes from knowing a priori if a packet is selected or not and therefore, the measurements on this subset are representative of all the packets.

   The evidence collection proceeds as follows. Each node i maintains a circular buffer to store β_i most recent packets and their measurements like timestamp and packet ID etc. Next, it selects some of the incoming packets as markers. It assigns a $MarkerID$ to each incoming packet between 0 to M using a uniform hash function applied to the contents of the packet. If $MarkerID$ is less than some global marking threshold μ then, this packet is selected as *marker* and also selected for generating receipt. Whenever an incoming packet m is selected as marker, the protocol selects some of the packets in the buffer for generating receipts and discards the remaining packets from the same path. For each packet p in the buffer from the same path as m, it computes a hash value for the pair (m, p) and if the value is less than sampling threshold σ_i then, p is selected for generating receipt and otherwise discarded.

   Thus, some of the packets in the packet stream of path k are selected as markers by each node and each marker packet is used to select n_i older packets from the path for receipt generation, where n_i depends on the values of β_i and σ_i and is specific to a node. Note that, however, the above procedure ensures that a common subset of packets is selected at each node on the path even when different nodes have different values of β_i and σ_i.

   These receipts generated at input and output nodes of a domain are used both as evidence for self-correctness and mutual-correctness.

3. **Evidence Inspection**: Now, we elaborate how the protocol uses generated receipts to estimate measurements for the domains and check the correctness of the link between peering nodes.

   (a) **Consistency** - Checking the correctness of link between peering nodes corresponds to the consistency check for the corresponding domains. Thus, if the link between peering nodes is correct then, the evidence provided by the corresponding domains is consistent with each other. To check the correctness of peering nodes i and j, the protocol identifies the subset of packets that were sampled at i and must be sampled at j. A packet p must have been sampled at both nodes given a marker m, if p is within $\min(\beta_i, \beta_j)$ most recent packets when m is seen and if hash value of (m, p) is less than the sampling thresholds of both nodes, σ_i and σ_j. Note that, there is no reordering of packets at peering nodes and thus, a packet is within $\min(\beta_i, \beta_j)$ most recent packets at both nodes or none of them. Next, it checks if both nodes provide receipts for such packets and for each packet p in this set, time-stamp at i and j differs by at most Δ_{ij}. If not, then the protocol declares the link as faulty. Note that there is no way to know if the fault lies in the domain of i or j, but it is known that at least one of the two has the fault.

(b) **Audit** - Computing loss rate and delay measurements from the receipts at input and output nodes of a domain corresponds to the audit phase for a domain.

To compute the loss rate between an input node i and output node j of a domain, the protocol again identifies the subset S_i of packets that were sampled at i and must be sampled at j using the approach described in consistency check for the markers observed during a small time interval. Let the size of S_i be k_i. Next, it identifies the actual set S_j of such packets that were sampled at j and let its size be k_j. Note that S_j might include some packets from S_i and some new packets because of reordering of packets. Also, S_j might not include some packets from S_i again because of reordering. However, statistically the number of new packets because of reordering and the number of packets from S_i not sampled in S_j because of reordering must be the same. Hence, the loss rate for the domain is estimated as $\frac{k_i - k_j}{k_i}$. This value is also the expected value of the true loss rate and has a standard deviation that is inversely proportional to k_i. Thus, as the number of sampled packets used to compute the loss rate increase, the estimate $\frac{k_i - k_j}{k_i}$ becomes more accurate.

To compute the delay between an input node i and an output node j of a domain, the protocol looks at the timestamps for packets sampled at both nodes and uses that to estimate the delay using the algorithm proposed in [4]

(c) **Challenge/Response** - If a node refuses to provide receipt for a packet, then it is equivalent to assuming that the packet was lost. Hence, this check does not apply here.

4. **Probabilistic Guarantees**: We have described above how the protocol allows reduction in overhead of the nodes in a domain at the expense of providing only probabilistic guarantees on finding approximately correct measurements for the domain.

5. **Fault Detection Power**: A single node can not deviate from the protocol and generate significantly different receipts so as to improve measurements of its domain, because otherwise, the receipts would not be consistent with the ones generated at the peering node and the link would be declared faulty. However, the protocol fails to report correct measurements if two peering nodes collude with each other such that the receipts provided by both nodes are consistent but the reported values in the receipts are incorrect. For example, in Figure 2, nodes 5 and 6 might collude such that timestamps at both nodes is decreased by x so as to reduce the delay measurement of X by x. However, if node 4 and 7 are honest and report correct timestamps, then the total delay between 4 and 7 is constant and thus, delay measurement of N is increased by x. Thus, improvement in the measurement of one domain comes at expense of degradation in another domain, and so collusion is not favorable for at least one of the domains.

5 Discussion

Now, we compare and contrast the different accountability protocols described above. We again use the different aspects of the accountability protocol to describe this.

1. **Notion of Correctness**: All three protocols differ quite a lot in their notion of correctness. In PeerReview, it is implicitly stated by checking if node follows the protocol or not, in CATS, it is explicitly stated via high level correctness properties of reads and writes, i.e. writes should be applied in order and reads should return the latest writes, and in Network Professional, it is stated quantitatively via delay and loss rate measurements for domains. Thus, PeerReview has the most strict constraints on each node and requires the nodes to follow a deterministic protocol, though it is easier to check if a node follows the protocol by simply replaying the execution. CATS loosens it up by only specifying high level correctness requirements and allowing the nodes to implement the reads and writes in any manner. Network Professional is even more flexible, since it does not require absolute correctness and accepts close to correct measurement values for delay and loss rate.

In PeerReview and CATS, nodes use messages to interact with each other and hence, the messages received and sent are used to check consistency in these protocols. Whereas in Network Professional, nodes only forward packets and don't interact with each other directly unlike the other protocols. Hence, it further defines the notion of correctness of link between peering nodes which is used later to check consistency.

2. **Evidence Collection**: These protocols also differ in type of evidence that they collect. In PeerReview, the evidence is a sequence of messages sent and received by the node in a sequential order. In CATS, the evidence consists of sequences of messages sent and received by the server (action histories) and, also periodic snapshots of the state (state digests). These periodic snapshots help break the sequence of messages into smaller sequences, where each smaller sequence corresponds to messages sent and received between 2 consecutive snapshots. The integrity of these smaller sequences can be checked independently of each other. In Network Professional, the evidence consists of receipts on packets which are generated independently and thus the receipts on each packet can be checked independently. Breaking the evidence into smaller units helps these protocols to implement a probabilistic auditing scheme which is not possible in the case of PeerReview.

3. **Evidence Inspection**:

   (a) **Consistency** - PeerReview checks consistency of evidence via a set of witnesses, whereas CATS releases its digests on a trusted publishing medium. While it is easy to implement, the consistency check in PeerReview is costly and has a message complexity of the order of square of the number of nodes. Whereas in CATS, each client directly checks the consistency of published digests for the messages sent and received by them and thus, it has a constant message complexity. However, CATS requires access to a trusted publishing medium which may not be always available. Network Professional does not elaborate on how the integrity of the receipts is maintained but assumes there is some way to securely transmit receipts from a node to nodes in other domains and a regulator.

   (b) **Audit** - In PeerReview, audit involves replaying the execution of a node and checking it matches the log provided by the node. The complete execution needs to be replayed again in order to guarantee correctness of the log. Similarly in CATS, in order to guarantee correctness of the log, each action in the log needs to be checked for correctness. However, the presence of published digests leads to a probabilistic scheme where only part of the history for some of the objects is checked. Because, the correctness of a published digest can be established relative to the previous digest. Thus, the auditor can randomly select some of the digests whose correctness is established relative to the previous digest. Since, each digest is equally likely to be selected, there is a positive probability that the protocol will detect a discrepancy in the log, if one exists. Also, since the server does not known which digests will be checked a priori, it can not hide the misbehavior in some digest without the possibility of being detected. In Network Professional, a probabilistic scheme is given to compute approximately correct measurements. A small subset of packets is sampled and used to compute the measurements. Similar to CATS, since the node can not predetermine which packets will be sampled, it can not give preferential treatment to any subset of packets and bias the measurements. Thus, both in CATS and Network Professional, processing a small subset from the available units of evidence suffices to get a probabilistic guarantee on detecting the fault. Note that, in CATS, the auditor explicitly tells the server which digests to audit while, in Network Professional, each node is implicitly told which packets to sample via future incoming traffic at the node.

We have described above the key differences between the three protocols. Overall, we can say that PeerReview is the most general protocol with fewest assumptions or requirements. However, as a tradeoff, it is also an expensive protocol both in terms of the messages exchanged and the computation required. CATS uses the availability of a trusted publishing domain, to reduce the cost of consistency check and also reduce the auditing overhead at the expense of having only probabilistic guarantee of detecting the fault. Network Professional uses the following domain-specific fact: a subset of packets can be used to compute almost correct measurements given that the node does not treat them preferentially. It uses this fact to reduce the overhead of receipt generation at each node. Thus, availability of additional support from the system or domain specific information can be used to have an efficient accountability protocol.

6 Conclusion

In this report, we have presented three accountability protocols using a general framework and drawn a comparison between them. Each protocol defines a notion of correctness of nodes in the distributed system, describes how evidence is collected and how the evidence is inspected to detect faults. Further, the protocols allow reduction in the overhead at the expense of providing only probabilistic guarantees of detecting the fault. We have also seen how domain-specific information or support from the system can be used to make the accountability protocol more efficient. Particularly, the idea of breaking the evidence log using intermediate state snapshots, as done via state digests in CATS, can be used in other applications as well. Thus, once the evidence log is broken into smaller components, we can randomly choose some of the components to verify and this can reduce the auditing overhead. Similarly, we can identify parts of the evidence in a protocol that can be independently verified, for example receipts in Network Professional, and randomly select only some of them to reduce the overall auditing overhead. Thus, while complete guarantee of correctness can often be costly, probabilistic guarantees can be achieved in an accountability protocol with a very small overhead on the nodes in the distributed system.

References

[1] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 1:1–1:12, New York, NY, USA, 2010. ACM.

[2] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, Dec. 2009.

[3] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct 2007.

[4] J. Sommers, P. Barford, N. Duffield, and A. Ron. Accurate and efficient sla compliance monitoring. *SIGCOMM Comput. Commun. Rev.*, 37(4):109–120, Aug. 2007.

[5] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *Trans. Storage*, 3(3), Oct. 2007.