UNIVERSIDADE DO MINHO

Anti-Ego: Tackling Byzantine fault models using accountability

by

Mohamad Baalbaki

A research report submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

in the

Faculty of Engineering

Department of Informatics

September 2017

UNIVERSIDADE DO MINHO

Abstract

Faculty of Engineering
Department of Informatics

Bachelor of Computer Science

by Mohamad Baalbaki

In this paper we present our protocol Anti-Ego for social mobile networks with rational behaviors. We assume that all the nodes are selfish. Anti-Ego shows that accountability can be an alternative solution through instant misbehavior detection. Parasite nodes are eventually detected and blacklisted. The accountability is done using a secure log with reduced resource requirements and complexity. The secure log is used to automatically detect if a node discarded a message, as in not forwarding it, thus exposing faulty nodes. Our protocol is applicable if the following requirements apply: nodes can sign messages, and that each node is periodically checked by a correct node. We demonstrate that Anti-Ego is practical by running it on a MAC machine using multi-processes and three Samsung Galaxy Tab 10.1.

Keywords: Social mobile networks, rational behaviors, selfish, accountability, secure log, blacklist.

Chapter 1

Introduction

Nowadays, cooperative systems[1] are widespread. Examples of cooperative systems are mobile phones, robots, aircraft in search-and-rescue operations, military surveillance devices and software agents. With the advancement of cooperative systems, it becomes apparent that message forwarding is a fundamental approach in this field because nodes should have incentive to forward messages between each other. Unfortunately, some nodes may have selfish behaviors because their resources are scarce, so they drop the messages to save resources. There are numerous approaches[2–5] that have been used so far to try to fix this problem, but failed to do so and most of them introduced new overheads.

For instance, the credit-based solution [2] proposes that nodes pay and get paid for providing services to others. The problem is that it must be enforced, otherwise nodes may start discarding the messages. In addition to that, it is not compatible with social mobile networks.

The reputation based approach [3] assigns nodes a metric that classifies them as good nodes or parasite nodes. In some cases, nodes might deceive the network by teaming up with each other nodes and modifying their reputation metric. This diminishes the credibility of the results of this protocol. Moreover, it is also not designed for social mobile networks like the credit based approach.

In addition to the approaches cited above, Game Theory techniques [6] have been used by Alessandro Mei and Julinda Stefa in Give2Get [4]. G2G epidemic forwarding and delegation forwarding were created because both epidemic[7] and delegation forwarding[8] do not handle selfish nodes. The problem with these protocols is that they cannot be implemented in Delay Tolerant Networks (DTN)[9]. They can only be implemented in closed area indoor networks which limits their use.

Means of accountability can be an alternative solution through instant misbehavior detection. That's what PeerReview[5] tried to do. But in a system with a many nodes, it can be difficult to ensure an absolute bound on the number of faulty nodes. In addition to that, it does not work on mobile networks and when the node has very limited resources.

This led us to develop the Anti-Ego protocol, an accountable system for Mobile systems against rational nodes. It supports anonymous messaging with accountability using secure logs with reduced resources requirements and complexity.

In this paper, we will explain our protocol and show our experimentation results, highlighting the different phases of it.

1.1 Contributions

We summarize our contribution in the following three points:

1.1.1 State-of-the-art

We made a thorough literature review for messaging protocols that address the message forwarding problem in rational and non-rational networks. It was crucial to show the existing gaps of current protocols to make use of some techniques that can help us design a better protocol.

We addressed state-of-the-art considering these points:

- Nodes may discard the messages to save resources, which will eventually ruin the message passing process.
- Some of the protocols try to solve this problem in rational environments and others in rational environments.

1.1.2 Anti-Ego made simple

Our second contribution was to make the Anti-Ego protocol easier to understand. Indeed, Anti-Ego inherits the complexity of rational behaviours from the game theory and byzantine standpoint. We tried to make the protocol easy to understand by:

- Simplifying the secure log description.
- Not mentioning timers in the algorithm, although communication between nodes is bounded by timers.

1.1.3 Documentation

Our third contribution was to provide a documentation for Anti-Ego's code; our purpose is to make the code easy to set up, configure, and understand, and thus help those who are interested in using or improving the protocol in the future. The code being written in Java, we tried to annotate the code and generate a *Javadoc* documentation following the standards and common practices. We also used this documentation to describe the workflow of the protocols by referring to some key classes and methods. We also provided a simple tutorial for installing and configuring the code.

1.2 Research report organization

The rest of the research report is organized as follows:

In Chapter 2, we start by talking about the state-of-the-art of message forwarding. Then, we mention and explain the Epidemic Routing protocol and we define fault models and explain their relation to our topic. All while showing the need for a new protocol that is accountable to Byzantine and rational behaviors. After that, we talk about the numerous approaches that try to tackle the message forwarding problem in a rational environment.

We start off Chapter 3 with the motivation of our protocol. Then, we introduce our system model by illustrating the secure log figure. In addition to that, we give a detailed description of the composition of the secure log. We define it as a concatenation of Verified Secure Logs VSL and a single Unverified Secure Log USL. Our protocol conveys 2 phases: Communication phase and Validation phase. We give a thorough and step by step description of each of these phases with figures. At the end of this chapter, we conclude.

Chapter 4 explains how to configure the host files and how to install the dependencies to run the code. After that, we enumerate the classes and we talk about each class. We give a description of the class role and we show the functions behaviors through the Javadoc (Java documentation) snippets that contain the method description, parameters, exceptions thrown and return value. The formulas are thoroughly explained with many examples are presented. The chapter explores the three system modes: Static, Dynamic, and Heuristic. Then, we evaluate these modes, and we present a method to discover when switching is worthy.

Chapter 5 is the concluding chapter. In it, we conclude everything and we open a new scope of discussion for future work.

Chapter 2

State-of-the-art

2.1 Background

Message passing in peer-to-peer networks [10] and more precisely in mobile ad hoc networks [11] is important because information exchange is very crucial in networks. A lot of protocols assume a connected path from source to destination. But unfortunately, in some scenarios there might not be a connected path between them, which will make it much harder to find a way to deliver messages between nodes. There are a lot of approaches that have been proposed such as Epidemic Routing [7], Credit based solution [4, 12–14], Reputation based solution [3, 15] and Give2Get [4]. We will first start by talking about Epidemic Routing, which doesn't handle rational behaviors [16, 17], unlike the other routing protocols. We chose to talk about Epidemic Routing because it solves the message passing problem in a non-rational network. As stated before, message passing is really helpful because information exchange is crucial in networks. But it may present some challenges when there are nodes in the network with selfish behaviors. These nodes may discard the messages to save resources, which will eventually ruin the message passing process.

2.2 Epidemic Routing Protocol

An approach that tackles this problem in a non-rational environment is Epidemic Routing. It is essentially flooding in a partially connected ad hoc network. The latter is a network that does not rely on a specific infrastructure. In Epidemic Routing, nodes continuously replicate and transmit messages to newly encountered contacts that do not already have a copy of the sent message. The current ad hoc routing protocols are not compatible with partially connected ad hoc networks, because as mentioned before, they need the presence of a connected path from source to destination. Here came the use of Epidemic Routing, where there might not be a connected path between source and destination.
The goals of Epidemic Routing are to:

- Maximize message delivery rate.
- Minimize message latency.
- Minimize the total resources consumed in message delivery.
- Deliver messages from source to destination with high probability even when there is never a connected path between them.

We assume that the sender is not in range and not near other nodes. In addition to that, we assume that he/she doesn't know where the receiver is currently located or the best route to follow.

The Epidemic Routing approach aims to distribute application messages to nodes within connected portions of ad hoc networks. In this way, messages are quickly distributed through connected portions of the network. It then relies upon nodes coming into proximity with another connected portion of the network and at this point, the message spreads to an additional island of nodes like a contagious disease. This leads to have a high probability of messages reaching their destination. Epidemic Routing supports the eventual delivery of messages to their destinations without knowledge of the network topology. It only needs pair-wise connectivity to guarantee the transitive distribution of messages through ad hoc networks.

The Epidemic Routing protocol works as follows. Each host in the ad hoc network has a buffer that contains the messages it has created and the messages it is carrying for other hosts. A hash table indexes these messages and assigns a unique identifier to each one. Each host has its own summary vector, which is a set of all messages treated by a certain host.

1. When two hosts come into communication range of one another, the host with the smaller identifier (let us call it A) initiates a session with the host with the larger identifier (let us call it B). To avoid redundant connections, each host maintains a cache of hosts that it has spoken with recently.
2. A starts by sending its summary vector SVA, to B. As stated before, the summary vector of A is the set of all the messages A has treated.
3. B then performs a logical AND operation between SVA and the complement of SVB, which we will denote by SV—B. To be more clear, it is the set of messages B has not treated. The AND operation gives us the intersection between SVA and SV—B, which in turn is the set of messages that B needs.

2.2.1 Example

Consider the set of all messages that have been sent in the network: $\{u, v, wp, r, x, y, z\}$
SVA (The set of messages A has treated): $\{u, v, w\}$
SVB (The set of messages B has treated): $\{p, u, r\}$

SV—B (The set of messages B hasn't treated): $\{v, w, p, x, y, z\}$
SVA AND SV—B (The intersection between the messages A has treated and the messages B hasn't treated): $\{v, w\}$
So v and w are the messages that A has treated and B has not yet treated, therefore B needs them.
Finally, A transmits the requested messages to B. This process is repeated transitively when B encounters a new neighbor. Given sufficient buffer space and time, these sessions guarantee eventual message delivery.

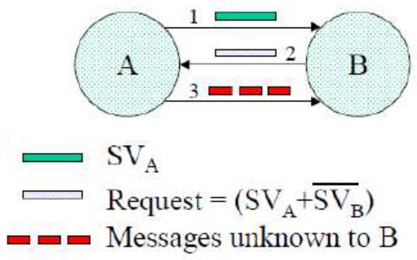


FIGURE 2.1: This figure shows the message passing from A to B in Epidemic Routing. As mentioned before, node A sends its summary vector "SVA" to node B, which is the set of all the messages A has treated. Then B performs a logical AND operation between SVA and the complement of "SVB" denoted by "SV—B". Then A sends the messages unknown to B.

2.3 Fault Models

In cooperative systems, that is, in the environment that Epidemic Routing operates in, rational/-selfish behavior may be present. This kind of behavior occurs when peers have scarce resources, and so they discard messages to save their resources. This falls under the umbrella of fault models, which are classes of something that could go wrong in the operation of a piece of equipment, or in our case, a message forwarding protocol such as Epidemic Routing. The Epidemic Routing protocol has no fault models. It is assumed to be fault-tolerant. An example of a fault model is the Byzantine fault model [18], which is an incorrect operation/algorithm that occurs in a distributed system. Another example of fault models is the fail-stop fault model [19]. In this type of failure, only crash failures are exhibited, but at the same time, we can assume that any correct entity can detect a failure. The Epidemic Routing protocol does not handle both of these fault models because it does not contain selfish nodes. Therefore, the nodes will always respond to requests and will always send correct data. It is important to focus on rational behaviors because they determine what happens in the message passing procedure. The following approaches: Credit-based, Reputation-based try to address the problem in literature.

2.4 Approaches

While the Epidemic Routing protocol [7] assumes no rational behaviors exist, several approaches have been introduced in literature to tackle this problem. Among the well known approaches are Credit based, Reputation based, and Game Theory based solutions, where we assume that all nodes have rational behaviors. Some of them partially solve this problem, and others solve it but with some overheads. We view these approaches in the following sections, focusing on the main pros and cons in each of them.

2.4.1 Credit-Based Approach

A credit-based solution [2] proposes that nodes pay and get paid for providing services to others. Since rational behaviors are present in this approach, a digital cash system is implemented in order to encourage correct behavior among them. This digital cash system stimulates a co-operative behavior between nodes and gives them an incentive to forward messages because it would benefit them. The main idea is that nodes that use a service should be charged and nodes that provide a service should get paid. Let us assume that the digital cash system currency is "coins".

Now, if Node A wants to use a service (e.g., it wants to send a message), then it has to pay for it in coins. This motivates each node to increase its number of coins, because these coins are indispensable for using the network. Thus, the probability of nodes discarding messages would be really slim because it would be against their interests to decrease their coins. It is better off providing services to other nodes because this is the only way to earn coins. The problem with this approach is that it must be enforced somehow, otherwise nodes may misbehave. To avoid forgery and to enforce protection, methods of cryptography should be applied. A secret key is needed to produce the coins. In addition to that, this way discourages parasites from collecting payments for each other. Regardless of its performance in ad hoc networks, it is not designed for social mobile networks. Social mobile networks have received much attention in recent years due to their potential applications and the proliferation of mobile devices, which increases the need of a functional message passing protocol. Unfortunately, it is not the case in the Credit-based approach.

2.4.2 Reputation-Based Approach

In this approach [3], nodes are assigned a metric that classifies them in the network as good or bad ones. If it is determined that nodes are helpful, then they should be kept in the network. On the other hand, if it shows that the nodes are parasites, they will be annihilated. The metric used in this solution is the reputation. Nodes gain a reputation based on if they discard a message or keep it to forward it. They collectively detect misbehaving members and propagate declaration of misbehavior throughout the network. Eventually this propagation leads to other nodes avoiding routes through selfish members. If Node B has the obligation to forward a message and it refuses to cooperate, then its reputation will be decreased. This leads to its exclusion if the

non-cooperative behavior persists. Other nodes will not forward messages to it as it is a parasite node and it is "blacklisted". The problem with this approach is that sometimes nodes might cheat by co-operating with other nodes and changing their reputation metric.
For example if Node A doesn't want to forward a message, it tells Node B to change it's reputation metric to 0, so that no one asks it to forward messages again.
In addition to that, it is not designed for social mobile networks like its counterpart. As mentioned before, it is important to design a message passing protocol for social mobile networks because there's a proliferation of mobile devices. This increases the need of a functional message passing protocol.

2.4.3 Game Theory Approach

Another possibility to address rational behaviours in message forwarding is to use Game Theory techniques [6] and mainly build a Nash Equilibrium [20], like the work done by Alessandro Mei and Julinda Stefa in Give2Get [4]. The message forwarding problem can be tackled by using a Game Theory based solution and therefore to achieve a Nash Equilibrium, because this motivates nodes to forward messages as it won't be beneficial for them if they don't.

2.4.3.1 Nash Equilibrium

First of all, let us define what a Nash Equilibrium is. A Nash Equilibrium is a set of strategies, one for each player, such that no player has incentive to change his or her strategy given what the other players are doing.

Example on Nash Equilibrium. Let us take the example of the Spotlight game.

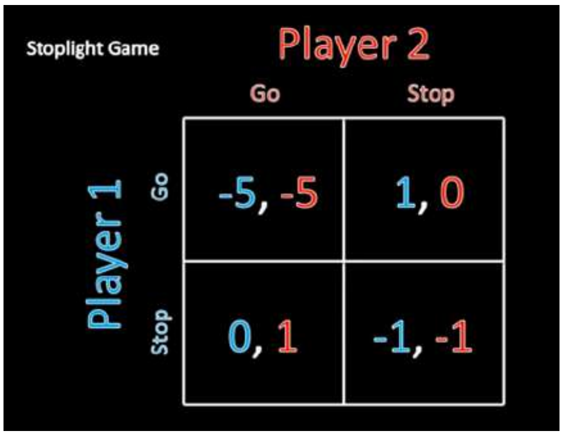
- Suppose two cars are driving at each other from perpendicular directions.
- The spotlight is red for one of them and green for the other.

If the police could not ticket the drivers, would they want to break the law?

- 1 means to go.
- 0 means to stop.
- -1 means to stop but while holding up the queue.
- -5 means to crash.

Both drivers can choose to go or to stop. Figure 2.2 shows the outcome of their decisions.

Case 1: If they both go. This will yield the outcome (-5,-5), which means that they will both crash into each other. This is the worst possible outcome. It is not Nash Equilibria because if one of them changes his/her strategy, a better outcome will be yielded.



Stoplight Game

		Player 2	
		Go	Stop
Player 1	Go	-5, -5	1, 0
	Stop	0, 1	-1, -1

FIGURE 2.2: This figure shows the different strategies that both players might take and their results.

Case 2: If they both stop. This will yield the outcome (-1,-1), which means that they will both stop and wait for each other to go. They will cause traffic, so this is a bad outcome. It is not Nash Equilibria because if one of them changes his/her strategy, a better outcome will be yielded.

Case 3: If one goes and the other stops. This will yield the outcome (1,0) or (0,1). The person who goes gets to his/her destination on time which is great. But the other person has to wait which is not so good but at least he/she will have the chance to go directly after the person who drove away, which is not so bad. If Player 2 is stopping and Player 1 is going, the latter wouldn't want to stop to not make a traffic mess (-1,-1). And Player 2 wouldn't want to go to not force an accident (-5,-5). So this clearly shows no one has an incentive to change his/her strategy, therefore it is Nash Equilibria.

The Give2Get protocol mentioned before achieves Nash Equilibrium and therefore is an effective game theory solution to the message forwarding problem. It assumes that all nodes behave in a rational manner. In the following, we will discuss its two protocols: G2G Epidemic Forwarding and G2G Delegation Forwarding.

2.4.4 Give2Get

As credit and reputation based solutions are not designed for social mobile networks, we chose to mention Give2Get [4] because it solves the problem in a rational environment. Below are the two protocols in Give2Get that solve the message forwarding problem in partially connected ad hoc networks, assuming that all nodes have rational behaviors and that the nodes are not in a delay tolerant network (DTN) [9].

2.4.4.1 G2G Epidemic Forwarding

In Epidemic Forwarding [7], the overhead of the number of messages being sent is very high, because every contact is used as an opportunity to forward messages. And as we know, Epidemic

Forwarding does not tolerate a situation where every node in the network is a selfish one.

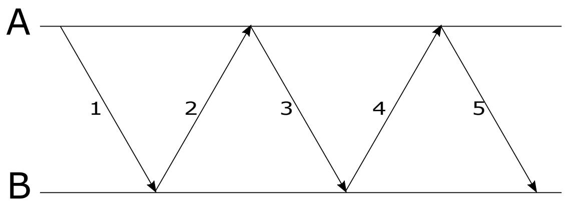


Figure 2.3: This figure shows the relay between node A and node B.

Here came the use of G2G Epidemic Forwarding, which is a protocol that works in a system where all nodes are considered selfish. It consists of three phases:

1. Message generation: If node A has a message to send to node D. The message is generated with a key called PKD which is the public key of the destination D. It is made on purpose to hide the sender of the message to all nodes/relays in the network so that no relay has an interest to drop the message.

2. Message relay: After the message is generated, A will relay it to the first nodes it encounters. Assume that node A meets node B. After starting an encrypted session together, Node A asks node B if it has already handled a message with hash H(m) (Phase a). If node B says yes, it tells A that it should not be chosen as a relay (Phase b). Otherwise, if Node B has never seen the message, Node A generates a random key k, and sends message m to B, encrypted with key k (Phase c). Then, node B sends a proof of relay to node A (Phase d) which in turn sends key k to B, who now knows whether it is the destination of the message or just a relay (Phase e).

3. Message test: Node B will follow the same protocol as done by node A. Then by the time it has collected two proofs of relay, it will be asked while re-meeting node A again if it is able to show the two proofs or to prove to have still the message in its memory. If it cannot, then node A will broadcast a proof of misbehavior (PoM) to the whole network that will blacklist node.

2.4.4.2 G2G Delegation Forwarding

On the other hand, in Delegation Forwarding [8], if node A meets node B, node A checks whether the forwarding quality of B is higher than the forwarding quality of the message. If yes, then it will create a copy of the message and assign both messages with the forwarding quality of node B, and then then it will forward one of the two copies to B. Otherwise, the message is not forwarded. As Epidemic Forwarding, Delegation Forwarding does not handle rational behaviours. Thus came the creation of G2G Delegation Forwarding, where the application of Nash Equilibrium solves this problem.

2.4.5 PeerReview

PeerReview[5] is a system that provides accountability in distributed systems. We chose to talk about PeerReview because it solves Byzantine fault models but it is not capable of solving fault models of networks with rational behaviors which is a plus for us. It is a system that ensures that Byzantine faults whose effects are observed by a correct node are eventually detected and classified as a faulty node. At the same time, it ensures that a correct node can always defend itself against false accusations (false accusations are when a node with correct behavior is accused of being faulty). This is important because in cooperative system, nodes may not trust each other and they can fail for many reasons:

- A node can suffer from a hardware or software failure.
- An attacker can compromise a node.
- Nodes may be accidentally misconfigured or may be compromised as a result of unpatched security vulnerabilities.

In cooperative systems, the lack of central administration tends to aggravate these problems. Here came the use of accountability with PeerReview to detect and expose node faults. PeerReview works by maintaining a secure record of the messages sent and received by each node. The record is used to automatically detect when a nodes behavior deviates from that of a given reference implementation, thus exposing faulty nodes. To be more elaborate, PeerReview creates a per-node secure log, which records the messages a node has sent and received, and the inputs and outputs of the application. Any node i can request the log of another node j and independently determine whether j has deviated from its expected behavior. To do this, i replays js log using a reference implementation that defines js expected behavior. By comparing the results of the replayed execution with those recorded in the log, PeerReview can detect Byzantine faults without requiring a formal specification of the system. Unfortunately, in a system with a large number of nodes, it can be difficult to ensure an absolute bound on the number of faulty nodes. To maintain PeerReviews completeness guarantees, a precise number of witnesses mentioned in [5] must be chosen; otherwise, all witnesses of a faulty node could also be faulty. In addition to that, it does not work on mobile networks which limits its use nowadays. As mentioned in the PeerReview paper, to avoid problems in finding faulty nodes and to get more accurate results, more CPUs or a CPU with multiple cores should be used. This is a bit too demanding. In addition to that, it does not work when the node has very limited resources as mentioned in the PeerReview paper [5]. This means that the PeerReview protocol is very restricted to work with.

2.4.6 CATS

CATS[21] is a network storage service with strong accountability properties. It enables nodes to read and write a shared directory of objects maintained by a CATS server. It provides them with the means to verify that the server is executing everything correctly. Nodes cannot deny their operations on a strongly accountable server.

Like PeerReview[5], CATS maintains secure logs that record the messages sent and received by each node. However, it depends on a trusted publishing medium that ensures the integrity of these logs. CATS detects faults by checking logs against a set of rules that describes the correct behavior of a specific system (a network storage service).

Yan Zhu and his colleagues said in their paper[22]: "The traditional cryptographic technologies for data integrity and availability, based on hash functions and signature schemes, cannot work on the outsourced data without a local copy of data. In addition, it is not a practical solution for data validation by downloading them due to the expensive communications, especially for largesize files. Moreover, the ability to audit the correctness of the data in a cloud environment can be formidable and expensive for the cloud users." What's stated above clearly shows that it is unpractical to use CATS.

2.4.7 Repeat and Compare

Repeat and Compare[23] is a system for ensuring content integrity in untrusted peer-to-peer content delivery networks[24]. Repeat and Compare uses accountability to ensure content integrity in a peer-to-peer CDN built on untrusted nodes.

First, it observes responses sent to clients, even though clients may lie.

Second, it needs to repeat response generation, even though origin servers may supply multiple, conflicting copies of the content.

Third, it needs to isolate misbehaving nodes, even though the nodes that "repeat and compare" are not trusted.

It detects faults by having a set of trusted verifier nodes locally reproduce a random sample of the generated content, and by comparing the results to the content returned by the untrusted nodes.

As Haeberlen said in his paper[25]: "However, existing accountability techniques fall short of the requirements for cloud computing in several ways. Since clouds are general-purpose platforms, the provider should be able to offer accountability for any service his customers may choose to run on it; this rules out application-specific techniques like CATS[23] or Repeat and Compare[23]. The application-independent technique in PeerReview[5], on the other hand, requires software modifications and assumes that the behavior of the software is deterministic, neither of which seems realistic in a cloud computing scenario. Finally, even if these limitations are overcome, the above techniques can only detect violations of a single property (correctness of execution); they were not designed to check other properties of interest in the cloud, such as conformance to SLAs, protection of confidential data, data durability, service availability, and so on." This clearly shows that Repeat and Compare and the other accountability protocols fail to serve our purpose which led us to develop a protocol called Anti-Ego. It will be explained in the following chapter.

2.5 Conclusion

In this chapter, we have discussed state-of-the-art of message forwarding under rational and Byzantine behaviours. We've addressed 3 categories of protocols:

The first protocol, Epidemic Routing protocol, only works in non-rational environments.

The next approaches are the Credit Based approach and the Reputation Based approach. They are not designed for social mobile networks which makes them unpractical.

Third of all, both Give2Get Epidemic Forwarding and Give2Get Delegation Forwarding have the problem of only working in closed area indoor networks which limits their use.

PeerReview, CATS and Repeat and Compare follow. They do not work when the node has very limited resources.

The rest of this research report describes our solution.

Chapter 3

Anti-Ego: Tackling Byzantine fault models using accountability

3.1 Motivation

The following chapter talks about our approach to solve the message forwarding problem. It is called "Anti-Ego". As it is stated previously, existing approaches such as credit[4, 12–14] and reputation based solutions[3, 15] that try to solve the message passing problem in ad hoc networks where selfish behaviors exist, do not do the job. By this we mean that both the credit based approach [2] and the reputation based approach [15] are not designed for social mobile networks. It is crucial for it to work on social mobile networks because by embracing modern technology, sharing data becomes much easier. Furthermore, the Give2Get[4] protocol solves this problem but only in closed area indoor networks. It is important to solve this problem on a higher scale such as the Delay Tolerant Networks (DTN) [9] one. This is due to the fact that this way, messages can be sent and received from really long distances. Our protocol rises to this level. Below are some of the reasons that motivated our work:

- Accountability can be an alternative solution through instant misbehavior detection.
- PeerReview [5] is not applicable for Mobile networks because of sync assumptions, limited resources and complexity.
- Our protocol handles anonymous messaging.
- The accountability is done using secure logs with reduced resource requirements and complexity.
- The implementation and experimentation are concrete. Below is the protocol with its phases which will be explained in the next paragraph.

3.2 Protocol

The protocol is composed of two phases: Validation and Communication. Validation phase is used to test if nodes are trustworthy. If a node is trusted, the Communication phase can be launched. Below is the system model along with some of the notations that are used.

3.2.1 System Model

The system is composed of multiple nodes that can interact over a wireless intermittent connection in a mobile network. Nodes are assumed to have limited resources like processing power, battery power, and storage. In addition to that, we assume that each node can be selfish (trying to save its resources) but neither malicious nor it can collude. To enforce accountability against selfish nodes, we provide a solution inspired from [26] and [5]. We require that each node has a secure log SL to store its executed operations in a way that allows other nodes to verify log correctness and consistency.

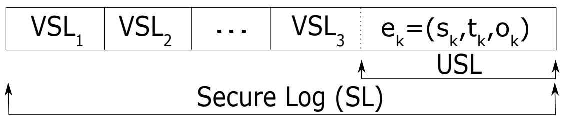


FIGURE 3.1: This figure shows the secure log.

A secure log SL is an append-only list of records appended in a chronological order; it is composed of a concatenation of Verified Secure Logs VSL and a single Unverified Secure Log USL, i.e., $SL = VSL_1||...||VSL_i||USL$. VSL_i is a sequence of hash values of the executed operations by the node itself, and appended with a verification certificate from another node. A certificate is a signed hash value of the verified log (hash of the hash values). USL is a sequence of hash values of operations executed by the node without being verified yet. In USL, each hash value is associated with an entry $e_k = (s_k, t_k, o_k)$ with a sequence number s_k , an operation type t_k , and an operation o_k. The sequence numbers must be strictly increasing. A hash value h_k is recursively computed from h_{k-1} and e_k as follows: $h_k = H(h_{k-1}||s_k||t_k||H(o_k))$. An authenticator $\alpha_k^X = \sigma^X(s_k, h_k)$ is a signed statement by node X (using its private key) that its log entry e_k^X has hash value h_k^X. In addition, each node keeps a Validation Authenticator hash Table VHT which is used to store the VSLs of the other nodes that were exchanged during validation. For any three nodes X, Y, and Z, we assume that $SL_X = VSL_Y||VSL_Z||USL$. On a fine-grained scale, $SL_X = h_1^X, h_2^X, ..., h_i^X, c_i^{X/Y}, h_{i+1}^X, ..., h_v^X, c_v^{X/Z}, h_{v+1}^X, ..., h_j^X, .., h_n^X$; where h_i^X represents the i^{th} hash value of X, and $c_v^{X/Y}$ represents the verification certificate of X's log from h_i^X to h_v^X delivered by Y. We denote by $h_k^{X/Y}$ the hash value of an entry e_k^X of X computed by node Y.

3.2.2 Communication Phase

Assume that the last entry of X's and Y's USL are h_n^X and h_m^Y, respectively.

1. **X encrypts Anonym message to hide its final destination, adds it to its log, and sends it to Y (Phase a).**

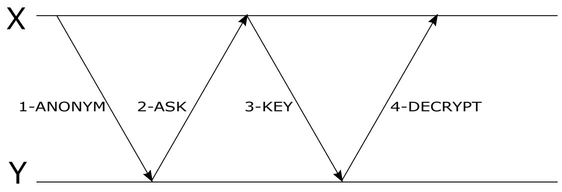


Figure 3.2: This figure shows the communication phase between node X and node Y.

Additional details. X prepares an Anonym message by encrypting the message payload m and the message destination D_m in the message content $o_{n+1} = Anonym(\sigma^X(< m, D_m >))$ using the private key of X and a randomly generated key K). Then, X creates a hash value h^X_{n+1} for a corresponding entry $e^X_{n+1} = (s_{n+1}, SEND, o_{n+1})$ and appends them to USL. Then, X creates two authenticators a^X_n and a^X_{n+1} corresponding to h^X_n and h^X_{n+1}, respectively. After that, X sends the last entry e^X_{n+1} along with the two authenticators a^X_n and a^X_{n+1} to Y.

2. **Y receives Anonym from X, verifies if X added Anonym to its log, and sends an ASK message to X asking for the key to decrypt Anonym (Phase b).**

   Additional details. Y receives Anonym from X and computes $h^{X/Y}_{n+1}$ using h^X_n and e^X_{n+1} (after verifying the signatures of the corresponding authenticators). Then Y verifies if X has added Anonym to its log by comparing $h^{X/Y}_{n+1}$ with h^X_{n+1}. If not, Y adds X to its blacklist. Otherwise, Y adds Anonym content to its log in a new entry $e^Y_{m+1} = (s_{m+1}, RECV, o_{m+1})$; where $o_{m+1} = h^X_{n+1}$ (to save storage resources) . Then, it prepares an ASK message to ask for the key of the Anonym message in order to decrypt it. It puts a^X_{n+1} in the message content o_{m+2} and creates a hash value h^Y_{m+2} with a corresponding entry $e^Y_{m+2} = (s_{m+2}, ASK, o_{m+2})$ and appends them to USL. Then, Y creates two authenticators a^Y_{m+1} and a^Y_{m+2} corresponding to h^Y_{m+1} and h^Y_{m+2}, respectively. After that, Y sends the last entry e^Y_{m+2} along with the two authenticators a^Y_{m+1} and a^Y_{m+2} to X.

3. **X receives ASK from Y, verifies if Y has added ASK to its log, and replies with a KEY message (Phase c).**

   Additional details. X receives ASK from Y and computes $h^{Y/X}_{m+2}$ using h^Y_{m+1} and e^Y_{m+2} (after verifying the signatures of the corresponding authenticators). Then, X verifies if Y has added ASK to its log by comparing $h^{Y/X}_{m+2}$ with h^Y_{m+2}. If not, X adds Y to its blacklist. Otherwise, X adds ASK content to its log in a new entry $e^X_{n+3} = (s_{n+3}, RECV, o_{n+3})$; where $o_{n+3} = h^Y_{m+2}$ (to save storage resources) . Then, it prepares a KEY message to send the key K of the Anonym message to Y. It puts the key K in the message content o_{n+4} and creates a hash value h^X_{n+4} with a corresponding entry $e^X_{n+4} = (s_{n+4}, KEY, o_{n+4})$ and appends them to USL. Then, X creates two authenticators a^X_{n+3} and a^X_{n+4} corresponding to h^X_{n+3} and h^X_{n+4}, respectively. After that, X sends the last entry e^X_{n+4} along with the two authenticators a^X_{n+3} and a^X_{n+4} to Y.

4. **Y receives Key from X and decrypts Anonym. If the final destination of Anonym is Y, this later uses the messages, otherwise, Y forwards the messages to the final destination (or another forwarder) node. If Y did not receive**

KEY, it forwards the encrypted Anonym to any N different nodes (to prevent selfishness) (Phase d).

Additional details. Y receives the KEY message from X and computes $h^{X/Y}_{n+4}$ using h^X_{n+3} and e^X_{n+4} (after verifying the signatures of the corresponding authenticators). Then Y verifies if X has added KEY to its log by comparing $h^{X/Y}_{n+4}$ with h^X_{n+4}. If not, Y adds X to its blacklist. Otherwise, Y adds KEY content to its log in a new entry $e^Y_{m+3} = (s_{m+3}, RECV, o_{m+3})$; where $o_{m+3} = h^X_{n+4}$ (to save storage resources). Now, Y used the key K to decrypt the Anonym message. If Y noticed that the message is destined to it, it executes it. Otherwise, Y forwards the message to the destination (or to another bridge node). In case Y did not receive the KEY message of Anonym (whether X has not sent it or the message was lost), Y must keep forwarding the encrypted Anonym message until receiving N ASK messages from different nodes; these messages stand as a proof in Y's log that it is not misbehaving (e.g., by dropping the KEY message from X).

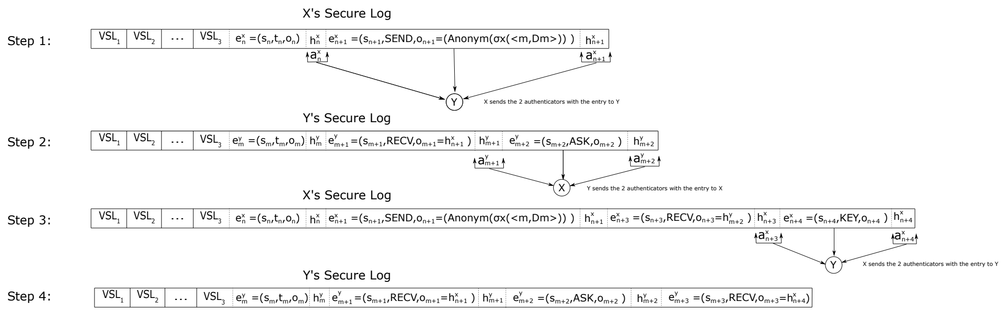

X's Secure Log
Step 1:
VSL1 | VSL2 | ... | VSL3 | e^x_n =(s_n,t_n,o_n) | h^x_n | e^x_{n+1} =(s_{n+1},SEND,o_{n+1}=(Anonym(σx(<m,Dm>))) | h^x_{n+1}
a^x_n
a^x_{n+1}
Y
X sends the 2 authenticators with the entry to Y
Y's Secure Log
Step 2:
VSL1 | VSL2 | ... | VSL3 | e^y_m =(s_m,t_m,o_m) | h^y_m | e^y_{m+1} =(s_{m+1},RECV,o_{m+1}=h^x_{n+1}) | h^y_{m+1} | e^y_{m+2} =(s_{m+2},ASK,o_{m+2}) | h^y_{m+2}
a^y_{m+1}
a^y_{m+2}
X
Y sends the 2 authenticators with the entry to X
X's Secure Log
Step 3:
VSL1 | VSL2 | ... | VSL3 | e^x_n =(s_n,t_n,o_n) | h^x_n | e^x_{n+1} =(s_{n+1},SEND,o_{n+1}=(Anonym(σx(<m,Dm>))) | h^x_{n+1} | e^x_{n+3} =(s_{n+3},RECV,o_{n+3}=h^y_{m+2}) | h^x_{n+3} | e^x_{n+4} =(s_{n+4},KEY,o_{n+4}) | h^x_{n+4}
a^x_{n+3}
a^x_{n+4}
Y
X sends the 2 authenticators with the entry to Y
Y's Secure Log
Step 4:
VSL1 | VSL2 | ... | VSL3 | e^y_m =(s_m,t_m,o_m) | h^y_m | e^y_{m+1} =(s_{m+1},RECV,o_{m+1}=h^x_{n+1}) | h^y_{m+1} | e^y_{m+2} =(s_{m+2},ASK,o_{m+2}) | h^y_{m+2} | e^y_{m+3} =(s_{m+3},RECV,o_{m+3}=h^x_{n+4})

FIGURE 3.3: This figure shows the details of the secure log in the communication phase.

3.2.3 Validation Phase

Assume that the last entry of X's and Y's USL are h^X_n and h^Y_m, respectively.

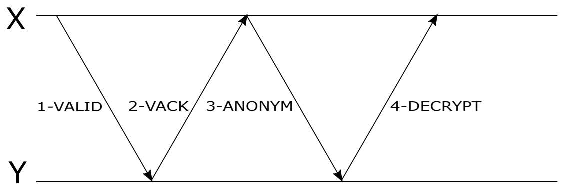


FIGURE 3.4: This figure shows the validation phase between node X and node Y.

1. **X sends a Valid message to Y with its VSL, the entries of USL, and the authenticator corresponding to h^X_{n+1} (Phase a).**

   Additional details. X prepares a validation message $Valid$; then it creates a hash value h^X_{n+1} for the corresponding entry $e^X_{n+1} = (s_{n+1}, SEND, o_{n+1})$, and appends them to USL. The Valid message must be accompanied with the verified secure log VSL, the entries $e^X_{v+1}, .., e^X_{n+1}$ (without their hash values), and the VHT table. In addition, X creates an authenticator a^X_{n+1} corresponding to h^X_{n+1} and sends it along with the Valid message to node Y.

2. **Y receives Valid message from X and verifies if X has computed its log hash sequence correctly. The it checks if its previous hash values, if any, are present in X's log or VHT; this is crucial to check for log fragmentation of nodes. Then, Y sends a VACK message to X with its VSL, the entries of USL, and the authenticator corresponding to h^Y_{m+2} (Phase b).**

   Additional details. Y receives Valid from X and logs a corresponding entry $e^Y_{m+1} = (s_{m+1}, RECV, o_{m+1})$ in its USL after verifying the signature. Then, Y validates S's log in three steps. First, it creates all the hash values of X's USL starting from h^X_v until h^X_{n+1} by using the entries $e^X_{v+1}, .., e^X_{n+1}$. if the calculated hash value $h^{X/Y}_{n+1}$ matches the received hash value h^X_{n+1} (obtained after decrypting a^X_{n+1}), then Y knows that X has not tampered with its USL. Second, Y check if X is misbehaving by dropping some Anonym messages that are supposed to be forwarded. This is done by checking whether every RECV entry of an Anonym message in X's USL is either followed by a SEND entry of the same message or followed by N different RECV entries of ASK messages corresponding to Anonym (ensuring that X used to forward Anonym messages to N nodes if it had not received their KEYs). Third, Y verifies if the hash values in X's VSL match the corresponding certificates, i.e., if $H(h^X_1, h^X_2, ..., h^X_i)$ matches $h^{X/Y}_i$ in the certificate $c^{X/Y}_i$, and $Hash(h^X_{i+1}, ..., h^X_v)$ matches $h^{X/Z}_v$ in the certificate $c^{X/Z}_v$. Fourth, Y checks whether its hash values of potential previous communication with X in present in X's SL (in our case Y has met X previously as shown in X's VSL). This step ensures that X has not deleted any previous hash values for communications with Y. Then ,Y checks whether its hash values of potential previous communication with those nodes which validated X's log (i.e., Y and Z in our case) are present in X's VHT. If Y noticed missing hash values with some node T, then Y adds T to it black list. This step ensures that other nodes are not using different log version for different nodes. As soon as this validation finishes, Y drops the entries $e^X_{v+1}, .., e^X_{n+1}$ and computes a certificate for X's $c^{X/Y}_{n+1}$ which is a signed message of the verified log hash $H(h^X_1, h^X_2, ..., h^X_{n+1})$, and adds X's SL to its own VHT (it overrides the old one if exists). Now, Y prepares a validation acknowledgment message VACK similar to the Valid message in with entry $e^Y_{m+2} = (s_{m+2}, SEND, o_{m+2})$ (where $o_{m+2} = a^X_{n+1}$), adds it to its USL, and sends the ASK message back to X (the details concerning the VACK message are quite similar to step 1 above).

3. **X receives VACK message from Y and verifies if Y has computed its log hash sequence correctly. Then it checks if its previous hash values, if any, are present in Y's log or VHT; this is crucial to check for log fragmentation of nodes. Then, X starts the communication phase by sending an Anonym message to Y with a certificate of Y's log signed by X (Phase c).**

   Additional details. X receives VACK from Y and adds a corresponding entry $e^X_{n+2} = (s_{n+2}, RECV, o_{n+2})$ to its USL log. Then, X makes sure that I has correctly validated it's log by matching the hash values received in the VHT of Y from h^X_v until h^X_{n+1}; otherwise, X adds Y to its blacklist. Then, X deletes all its entries $e^X_{v+1}, .., e^X_{n+1}$ (i.e., USL is now empty). After that, X validates Y's log in a similar way to step 2 above. Then, X trusts Y, and starts the communication phase by preparing an *Anonym* message (see Subsection 3.2.2), and adds the corresponding entry $e^X_{n+3} = (s_{n+3}, SEND, o_{m+3})$ it to its USL, and sends it along with the certificate $c^{Y/X}_{m+2}$, computed using VSL and USL of Y, which proves that X has correctly validated Y's log until the last hash value in Y's USL h^{Y}_{m+2}.

4. **Y receives Anonym message from X and verifies if X has sent a correct certificate for Y's log (Phase d).**

   Additional details. Y receives the Anonym message from X and adds a corresponding entry $e^{Y}_{m+3} = (s_{m+3}, RECV, o_{m+3})$ it to its USL. Then Y checks whether X has verified correctly its log by checking if the certificate $c^{Y/X}_{m+2}$ sent by X matches the hash value of Y itself. If this is true, Y trusts X and deletes all the entries in its own USL until e^{Y}_{m+2}, and the communication phase continues; otherwise, Y adds X to its blacklist.

3.3 Conclusion

In this chapter, we presented AntiEgo: a new message forwarding protocol for rational and Byzantine behaviours. The basic idea of the protocol is to force all nodes to contribute in the forwarding of messages and not to discard them.

Our design decision bridges the gaps of state-of-the-art protocols because unlike Give2Get, it can be implemented in Delay tolerant networks.

Our protocol may not work in the case of a node refusing to send "Anonym" since it might do so if it is selfish and only wants to validate trying to minimize its storage by deleting log entries.

In hopes of doing sufficient testing to show accurate experimentation results in the near future.

Chapter 4

Code Specifications

4.1 Setup

In this section, we will explain thoroughly how to run the code. We chose Eclipse to write our code because it is the leading Java IDE and it is easy to use and deal with.

4.1.1 Dependencies

- The code is tested on Java 6 on MAC OS X 10.6, Windows, and 3 Samsung Galaxy Tab 10.1, but it can also work on other machines.
- If 3rd party encryption is needed, then include the flexiprovider encryption library from here: `http://www.flexiprovider.de/` (or just remove flexiprovider imports in the Cryptography class in the code).

4.1.2 Installation

1. Install eclipse and include flexiprovider libraries in the class path (if needed).
2. Import in eclipse an ”Existing Projects to Workspace” indicating the AMS folder. This will import the project smoothly into eclipse.
3. Make sure that all class paths are ok. The code should not have any error.

4.1.3 Configuration

1. Use two computers and connect them through a LAN cable or WLAN.
2. Open the command prompt and write: ipconfig/all on both computers.
3. Write down the IPv4 addresses.

4. Change the IPs of your machines in the config folder to the IPv4 addresses and assign suitable keys for encryption.

4.1.4 Testing

The code is tested on a single MAC machine using multi-processes, on Windows and 3 Samsung Galaxy Tab 10.1. It is trivial to run it on multiple machines too. Running the application is straightforward, just make sure to give a reasonable time between nodes you lunch as they are configured to immediately start sending messages (by default node1 starts).

4.2 Classes

The code is comprised of one main class which is called "Launcher", and 5 other classes: Node, Cryptography, Messaging, SecureLogging and Tools. We will not talk about the Launcher class since it doesn't contain anything but the creation of a node, and we won't talk about the Tools class because it is a Utility class. In addition to that, we will mention the Cryptography class in the SecureLogging class.

4.2.1 Node class

4.2.1.1 Class role

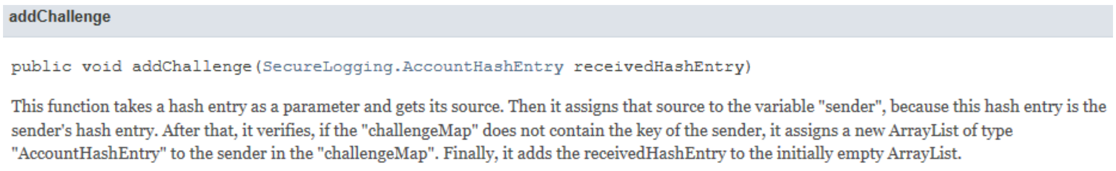
This class has the role of generating the operation and sequences ids, as well as filling the identity map.

The workflow is as follows: the Launcher class evokes the creation of a new Node with a threadpool, so the Node class's constructor does the job of creating it. At least 2 nodes should be created so that the Messaging class can create the messages between them.

4.2.1.2 Functions

This class has the following methods: addChallenge, executeOperation, fillIdentityMap, generateOpId, generateSeqId, getBlacklist, getChallengeMap, getId, getInfoBase, getIPAddress, getPrivateKey, getPublicKey, getULog, initializeOutQueue, initializeValid, newAnonymPayload. We will only cover the addChallenge and fillIdentityMap methods because they are the most important ones, the others are helper methods.

1. addChallenge(SecureLogging.AccountHashEntry receivedHashEntry):



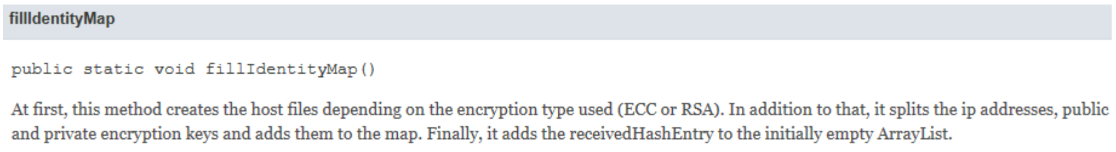
addChallenge

```
public void addChallenge(SecureLogging.AccountHashEntry receivedHashEntry)
```

This function takes a hash entry as a parameter and gets its source. Then it assigns that source to the variable "sender", because this hash entry is the sender's hash entry. After that, it verifies, if the "challengeMap" does not contain the key of the sender, it assigns a new ArrayList of type "AccountHashEntry" to the sender in the "challengeMap". Finally, it adds the receivedHashEntry to the initially empty ArrayList.

FIGURE 4.1: This figure shows the documentation of the addChallenge method.

2. fillIdentityMap():



fillIdentityMap

```
public static void fillIdentityMap()
```

At first, this method creates the host files depending on the encryption type used (ECC or RSA). In addition to that, it splits the ip addresses, public and private encryption keys and adds them to the map. Finally, it adds the receivedHashEntry to the initially empty ArrayList.

FIGURE 4.2: This figure shows the documentation of the fillIdentityMap method.

4.2.2 Messaging class

4.2.2.1 Class role

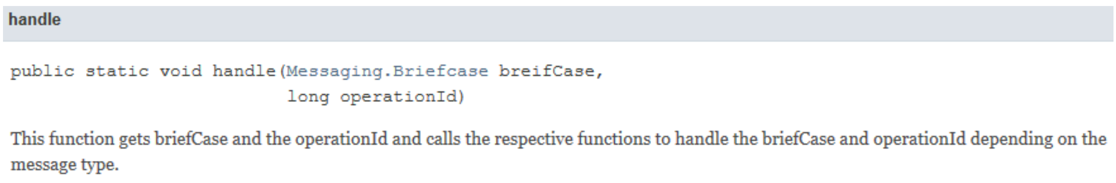
With the 2 nodes being ready, this class has the role of handling the messages being sent between them. It handles all the messages depending on their types. For example, if a message is of type ASK, it calls the method handleAsk 4.6. If it's of type Anonym, it calls the method handleAnonym 4.4 and so on... The nodes now await the accountability which is done in the secure log. We will explain that in the next section.

The following javadocs explain what is done in each method.

4.2.2.2 Functions

The Messaging class has one method called "handle" and each subclass has many methods. We will only focus on the "handle" method and the methods it calls since they are the most important ones. The main parameter used is the briefCase which is the container of messages.

1. handle(Messaging.Briefcase briefCase, long operationId):



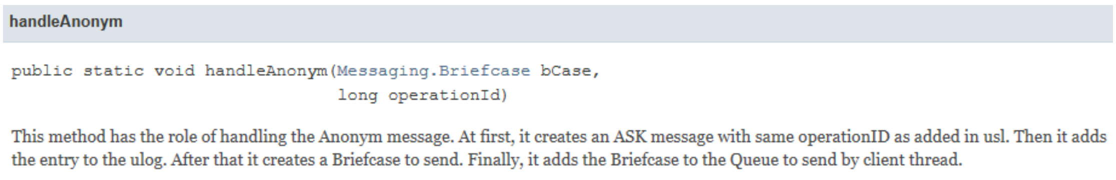
handle

```
public static void handle(Messaging.Briefcase breifCase,
                          long operationId)
```

This function gets briefCase and the operationId and calls the respective functions to handle the briefCase and operationId depending on the message type.

FIGURE 4.3: This figure shows the documentation of the handle method.

2. handleAnonym(Messaging.Briefcase bCase, long operationId):



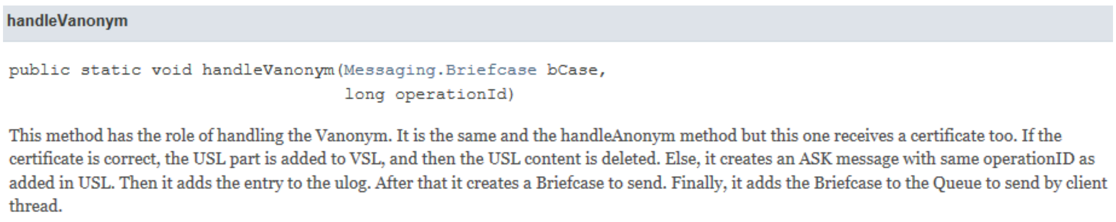
handleAnonym

```
public static void handleAnonym(Messaging.Briefcase bCase,
                                long operationId)
```

This method has the role of handling the Anonym message. At first, it creates an ASK message with same operationID as added in usl. Then it adds the entry to the ulog. After that it creates a Briefcase to send. Finally, it adds the Briefcase to the Queue to send by client thread.

FIGURE 4.4: This figure shows the documentation of the handleAnonym method.

3. handleVanonym(Messaging.Briefcase bCase, long operationId):



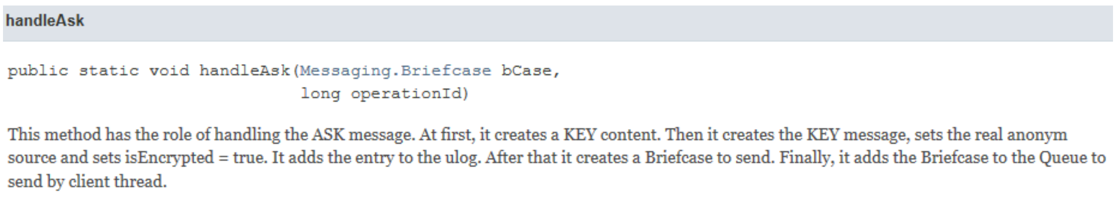
handleVanonym

```
public static void handleVanonym(Messaging.Briefcase bCase,
                                 long operationId)
```

This method has the role of handling the Vanonym. It is the same and the handleAnonym method but this one receives a certificate too. If the certificate is correct, the USL part is added to VSL, and then the USL content is deleted. Else, it creates an ASK message with same operationID as added in USL. Then it adds the entry to the ulog. After that it creates a Briefcase to send. Finally, it adds the Briefcase to the Queue to send by client thread.

FIGURE 4.5: This figure shows the documentation of the handleVanonym method.

4. handleAsk(Messaging.Briefcase bCase, long operationId):



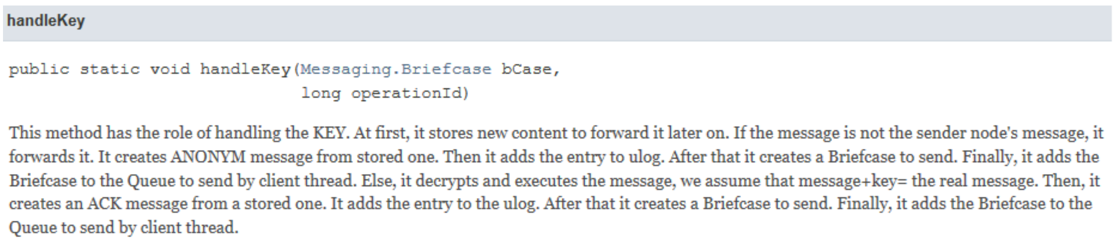
handleAsk

```
public static void handleAsk(Messaging.Briefcase bCase,
                             long operationId)
```

This method has the role of handling the ASK message. At first, it creates a KEY content. Then it creates the KEY message, sets the real anonym source and sets isEncrypted = true. It adds the entry to the ulog. After that it creates a Briefcase to send. Finally, it adds the Briefcase to the Queue to send by client thread.

FIGURE 4.6: This figure shows the documentation of the handleAsk method.

5. handleKey(Messaging.Briefcase bCase, long operationId):



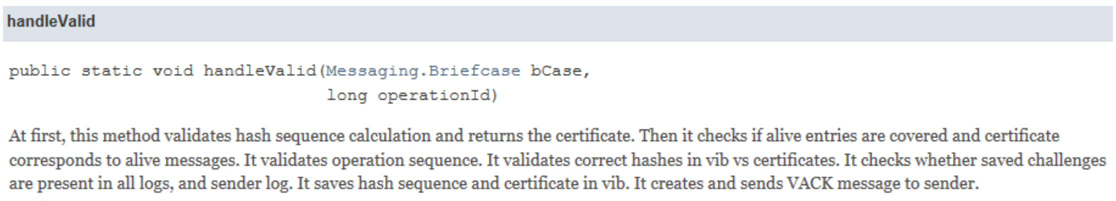
handleKey

```
public static void handleKey(Messaging.Briefcase bCase,
                             long operationId)
```

This method has the role of handling the KEY. At first, it stores new content to forward it later on. If the message is not the sender node's message, it forwards it. It creates ANONYM message from stored one. Then it adds the entry to ulog. After that it creates a Briefcase to send. Finally, it adds the Briefcase to the Queue to send by client thread. Else, it decrypts and executes the message, we assume that message+key= the real message. Then, it creates an ACK message from a stored one. It adds the entry to the ulog. After that it creates a Briefcase to send. Finally, it adds the Briefcase to the Queue to send by client thread.

FIGURE 4.7: This figure shows the documentation of the handleKey method.

6. handleValid(Messaging.Briefcase bCase, long operationId):



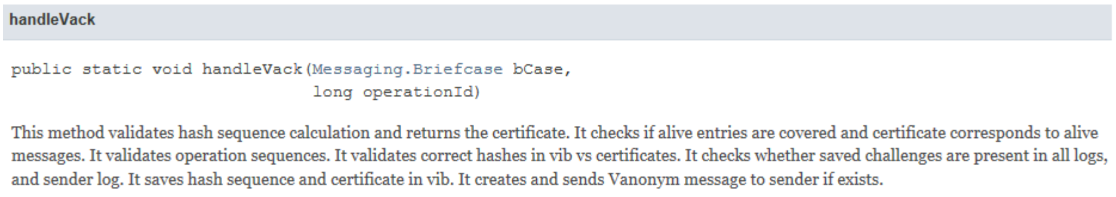
handleValid

```
public static void handleValid(Messaging.Briefcase bCase,
                               long operationId)
```

At first, this method validates hash sequence calculation and returns the certificate. Then it checks if alive entries are covered and certificate corresponds to alive messages. It validates operation sequence. It validates correct hashes in vib vs certificates. It checks whether saved challenges are present in all logs, and sender log. It saves hash sequence and certificate in vib. It creates and sends VACK message to sender.

FIGURE 4.8: This figure shows the documentation of the handleValid method.

7. handleVack(Messaging.Briefcase bCase, long operationId):



handleVack

```
public static void handleVack(Messaging.Briefcase bCase,
                              long operationId)
```

This method validates hash sequence calculation and returns the certificate. It checks if alive entries are covered and certificate corresponds to alive messages. It validates operation sequences. It validates correct hashes in vib vs certificates. It checks whether saved challenges are present in all logs, and sender log. It saves hash sequence and certificate in vib. It creates and sends Vanonym message to sender if exists.

FIGURE 4.9: This figure shows the documentation of the handleVack method.

4.2.3 SecureLogging class

4.2.3.1 Class role

Before talking about the role of the SecureLogging class, we will mention the Cryptography class because cryptography happens before the accountability.

The Cryptography class, as its name suggests has the role of handling the encryption and decryption of the messages. It does that by generating the ECC and RSA keys and using the cipher, messageDigest and two attributes from the "Node" class which are the publicKey and privateKey.

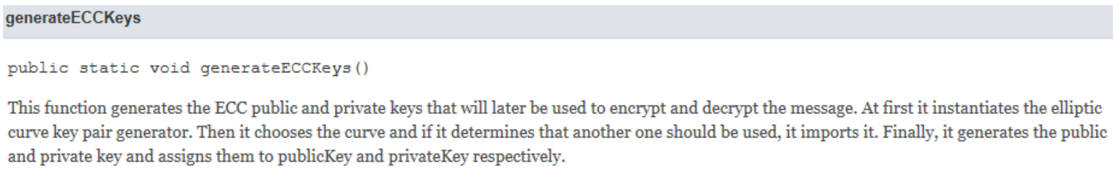
After the messages are encrypted, it is crucial to talk about the SecureLogging class which has the role of creating and handling the secure log of the nodes, as well as containing the methods that are responsible for the accountability. In other words, it contains all of the methods that verify if a node sent a message or discarded it.

The secure log figure 3.3 shows how the messages are being contained.

4.2.3.2 Cryptography class functions

This class has the following methods: digestEntry, digestHashMap, digestMessage (one that takes a String and one that takes an array of bytes), generateECCKeys, generateRSAKeys, readKeys, encryptMessage and decryptMessage (one that takes a String and one that takes an array of bytes for each of these two methods). We will only cover the generateECCKeys, generateRSAKeys, encryptMessage, decryptMessage methods because they are the most important ones, some of the others are helper methods and some are less important.

1. generateECCKeys():



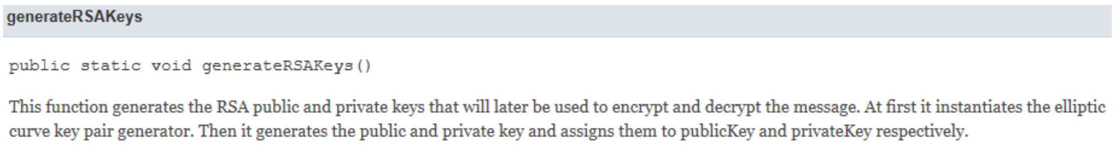
generateECCKeys

```
public static void generateECCKeys()
```

This function generates the ECC public and private keys that will later be used to encrypt and decrypt the message. At first it instantiates the elliptic curve key pair generator. Then it chooses the curve and if it determines that another one should be used, it imports it. Finally, it generates the public and private key and assigns them to publicKey and privateKey respectively.

FIGURE 4.10: This figure shows the documentation of the generateECCKeys method.

2. generateRSAKeys():

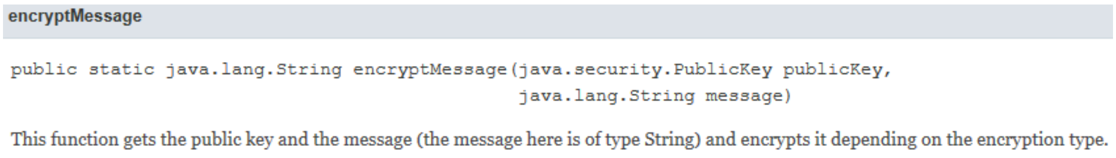

generateRSAKeys

```
public static void generateRSAKeys()
```

This function generates the RSA public and private keys that will later be used to encrypt and decrypt the message. At first it instantiates the elliptic curve key pair generator. Then it generates the public and private key and assigns them to publicKey and privateKey respectively.

FIGURE 4.11: This figure shows the documentation of the generateRSAKeys method.

3. encryptMessage(java.security.PublicKey publicKey, java.lang.String message):

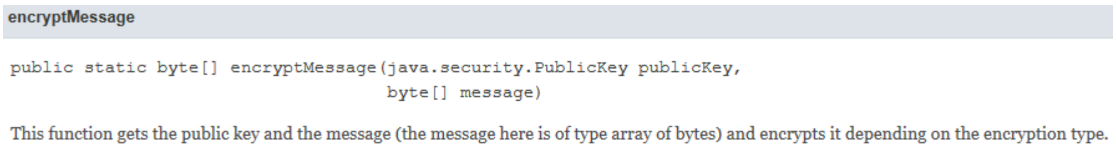

encryptMessage

```
public static java.lang.String encryptMessage(java.security.PublicKey publicKey,
                                              java.lang.String message)
```

This function gets the public key and the message (the message here is of type String) and encrypts it depending on the encryption type.

FIGURE 4.12: This figure shows the documentation of the encryptMessage method.

4. encryptMessage(java.security.PublicKey publicKey, byte[] message):

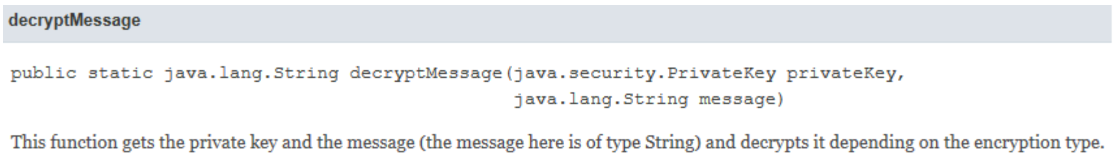

encryptMessage

```
public static byte[] encryptMessage(java.security.PublicKey publicKey,
                                    byte[] message)
```

This function gets the public key and the message (the message here is of type array of bytes) and encrypts it depending on the encryption type.

FIGURE 4.13: This figure shows the documentation of the encryptMessage method.

5. decryptMessage(java.security.PrivateKey privateKey,java.lang.String message):

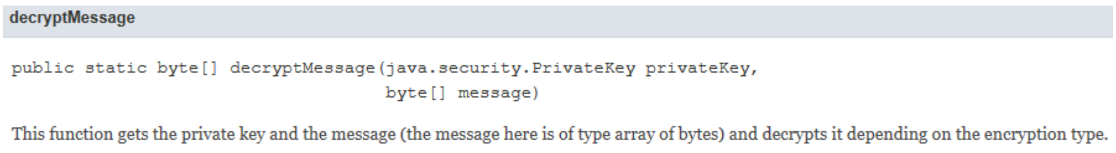

decryptMessage

```
public static java.lang.String decryptMessage(java.security.PrivateKey privateKey,
                                              java.lang.String message)
```

This function gets the private key and the message (the message here is of type String) and decrypts it depending on the encryption type.

FIGURE 4.14: This figure shows the documentation of the decryptMessage method.

6. decryptMessage(java.security.PrivateKey privateKey,byte[] message):


decryptMessage

```
public static byte[] decryptMessage(java.security.PrivateKey privateKey,
                                    byte[] message)
```

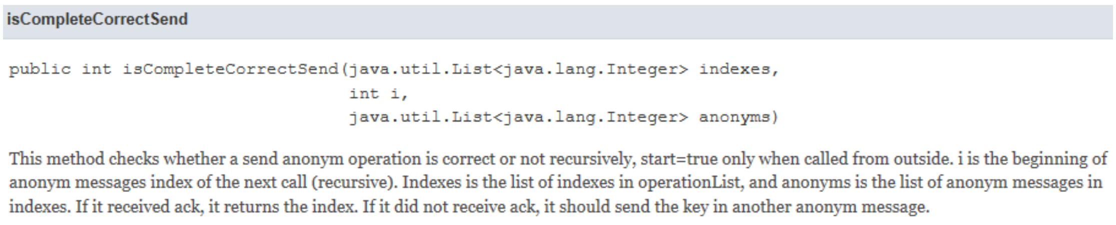
This function gets the private key and the message (the message here is of type array of bytes) and decrypts it depending on the encryption type.

FIGURE 4.15: This figure shows the documentation of the decryptMessage method.

4.2.3.3 SecureLogging class functions

The SecureLogging class's most important methods are in the SecureLogging.AccountUSecureLog subclass. They are: isCompleteCorrectSend, isCompleteCorrectForward, isCorrectSenderAndForwarder.

1. isCompleteCorrectSend(java.util.List¡java.lang.Integer¿ indexes, int i, java.util.List¡java.lang.Integer¿ anonyms):



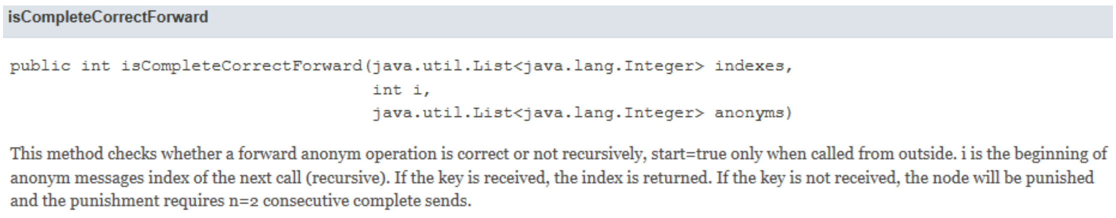
isCompleteCorrectSend

```
public int isCompleteCorrectSend(java.util.List<java.lang.Integer> indexes,
                                 int i,
                                 java.util.List<java.lang.Integer> anonyms)
```

This method checks whether a send anonym operation is correct or not recursively, start=true only when called from outside. i is the beginning of anonym messages index of the next call (recursive). Indexes is the list of indexes in operationList, and anonyms is the list of anonym messages in indexes. If it received ack, it returns the index. If it did not receive ack, it should send the key in another anonym message.

Figure 4.16: This figure shows the documentation of the isCompleteCorrectSend method.

2. isCompleteCorrectForward(java.util.List¡java.lang.Integer¿ indexes, int i, java.util.List¡java.lang.Integer¿ anonyms):



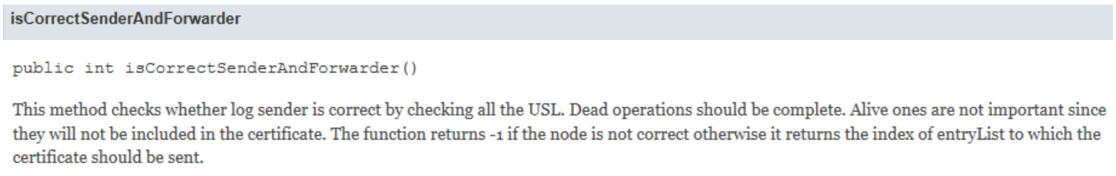
isCompleteCorrectForward

```
public int isCompleteCorrectForward(java.util.List<java.lang.Integer> indexes,
                                    int i,
                                    java.util.List<java.lang.Integer> anonyms)
```

This method checks whether a forward anonym operation is correct or not recursively, start=true only when called from outside. i is the beginning of anonym messages index of the next call (recursive). If the key is received, the index is returned. If the key is not received, the node will be punished and the punishment requires n=2 consecutive complete sends.

Figure 4.17: This figure shows the documentation of the isCompleteCorrectForward method.

3. isCorrectSenderAndForwarder():



isCorrectSenderAndForwarder

```
public int isCorrectSenderAndForwarder()
```

This method checks whether log sender is correct by checking all the USL. Dead operations should be complete. Alive ones are not important since they will not be included in the certificate. The function returns -1 if the node is not correct otherwise it returns the index of entryList to which the certificate should be sent.

Figure 4.18: This figure shows the documentation of the isCorrectSenderAndForwarder method.

4.3 Conclusion

In this chapter, we explained how to install and configure the code of our Anti-Ego protocol. In addition to that, we said that it is tested on MAC, Windows and Samsung Galaxy Tab.

After that we talked about the classes, their roles and the methods, all while showing the respective javadocs of the methods.

Chapter 5

Conclusions

In this technical report, we've presented a new message forwarding protocol for rational and Byzantine behaviors. The problem is that when selfish nodes have scarce resources, they tend to drop messages which disrupts the whole message forwarding process. This way, they save resources.

This led us to create the Anti-Ego protocol, which has the purpose of forcing any contributing node in message forwarding, in a peer-to-peer system, not to discard forwarding others messages (to save resources like energy, CPU and network bandwidth).

The main idea is to forward the message through a chain of nodes from the source to destination where each node sends the message to its successor in two round-trips: the first round-trip hides the destination of the message (using encryption) and the second sends the decryption key. In this way, the receiver won't be able to deny the message if it is not the destination, since the preceding node holds a proof that it received this message. This simplifies tracking rational (selfish) nodes and evicting them.

State-of-the-art protocols, as discussed in Chapter 2, did not completely solve the problem. For instance, Epidemic Routing[7] and Give2Get[4] were the closest protocols to Anti-Ego. Epidemic Routing solved the message forwarding problem in networks with rational behaviors only. But the problem persisted in networks with non-rational behaviors. Give2Get solves the problem in non-rational networks but with the restriction that it only works in closed area indoor networks. We expect that from the results of the Give2Get experimentation. Give2Get inspired us in hiding the destination of our message. Another inspiration was PeerReview[5], which we imitated in the use of the secure log.

As for most difficult thing we faced, a selfish node might refuse to send Anonym if it only wants to validate trying to minimize its storage by deleting log entries.

Our protocol filled lots of gaps by introducing accountability which is an alternative solution through instant misbehavior detection. The accountability is done using secure logs with reduced resource requirements and complexity. Another thing that we added was anonymous messaging.

Message forwarding proved to be a critical topic in mobile ad hoc networks. It is critical because the sender of the message is never in range, doesn't know where the receiver is currently located or the best route to follow.

In the future, we plan to improve the documentation of the code to make it more understandable. In addition to that, we plan to write test scenarios and experiment the protocol using ONE simulator (some of them are already implemented in the "simulation" folder). Finally, we plan to review the code and improve it to find mistakes overlooked in the initial development phase. However, we won't implement this program on a testbed but we strongly encourage people to do so, because it yields really accurate testing results.

Bibliography

[1] Sergiy Butenko. Cooperative Systems. http://www.springer.com/series/5788/, 2004.

[2] Sheng Zhong, Jiang Chen, and Yang Richard Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1987–1997. IEEE, 2003.

[3] Li Xiong and Ling Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE transactions on Knowledge and Data Engineering*, 16(7):843–857, 2004.

[4] Alessandro Mei and Julinda Stefa. Give2get: Forwarding in social mobile wireless networks of selfish individuals. *IEEE Transactions on Dependable and Secure Computing*, 9(4):569–582, 2012.

[5] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SIGOPS operating systems review*, volume 41, pages 175–188. ACM, 2007.

[6] John Von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton university press, 2007.

[7] Amin Vahdat, David Becker, et al. Epidemic routing for partially connected ad hoc networks, 2000.

[8] Vijay Erramilli, Mark Crovella, Augustin Chaintreau, and Christophe Diot. Delegation forwarding. In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pages 251–260. ACM, 2008.

[9] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34. ACM, 2003.

[10] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[11] Charles E Perkins et al. *Ad hoc networking*, volume 1. Addison-wesley Reading, 2001.

[12] Markus Jakobsson, Jean-Pierre Hubaux, and Levente Buttyán. A micro-payment scheme encouraging collaboration in multi-hop cellular networks. In *International Conference on Financial Cryptography*, pages 15–33. Springer, 2003.

[13] Levente Buttyán and Jean-Pierre Hubaux. Enforcing service availability in mobile ad-hoc wans. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, pages 87–96. IEEE Press, 2000.

[14] Levente Buttyán and Jean-Pierre Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *Mobile Networks and Applications*, 8(5):579–592, 2003.

[15] Ernesto Damiani, De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 207–216. ACM, 2002.

[16] Vikram Srinivasan, Pavan Nuggehalli, Carla-Fabiana Chiasserini, and Ramesh R Rao. Cooperation in wireless ad hoc networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 2, pages 808–817. IEEE, 2003.

[17] Jeffrey Shneidman and David C Parkes. Rationality and self-interest in peer to peer networks. In *International Workshop on Peer-to-Peer Systems*, pages 139–148. Springer, 2003.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[19] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3): 222–238, 1983.

[20] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

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

[22] Yan Zhu, Huaixi Wang, Zexing Hu, Gail-Joon Ahn, Hongxin Hu, and Stephen S Yau. Dynamic audit services for integrity verification of outsourced storages in clouds. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1550–1557. ACM, 2011.

[23] Nikolaos Michalakis Robert Soulé Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks.

[24] J.M. Wein, J.J. Kloninger, M.C. Nottingham, D.R. Karger, and P.A. Lisiecki. Content delivery network (cdn) content server request handling mechanism with metadata framework support, July 3 2007. URL https://www.google.com/patents/US7240100. US Patent 7,240,100.

[25] Andreas Haeberlen. A case for the accountable cloud. *ACM SIGOPS Operating Systems Review*, 44(2):52–57, 2010.

[26] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. *arXiv preprint cs/0202005*, 2002.