Formal modelling and analysis of behaviour grading within a peer-to-peer storage system

Samira Chaou, Franck Pommereau

Formal Modelling and Analysis of Behaviour Grading within a Peer-to-peer Storage System

Samira Chaou
UbiStorage, 20 av. Claudel, 80000 Amiens, France
samira.chaou@ubistorage.com

Franck Pommereau
IBISC, University of Évry, 91000 Évry, France
franck.pommereau@ibisc.univ-evry.fr

Keywords: peer-to-peer storage, security protocols, malicious peers detection, Petri nets, model-checking, simulation

Abstract

In this paper, we extend a peer-to-peer based storage system in order to cope with malicious nodes. To do so, we introduce a grading system allowing peers to evaluate the outcome of their transactions with others, and consequently allowing to detect misbehaving peers.

We evaluate this extension by two means. On the one hand, we have built a formal model of the system and used model-checking to verify whether malicious peers can be always detected or not. On the other hand, we have implemented the system and used simulation to assess malicious peers detection in realistic situations.

Thanks to this analysis, we can guarantee that some attacks are necessarily detected while others remain undetected and, worse, may yield false positive (*i.e.*, some peers are graded as being malicious while they are not). We propose solutions to improve this situation at the end of the paper.

1. INTRODUCTION

Ubiquitous Storage (UbiStorage) implements and commercialises a storage solution based on a peer-to-peer network allowing each user to securely store and retrieve data [2, 1]. The system is provided to customers as a Linux server running the peer-to-peer software as well as end-users services.

In [14], the security of the underlying protocols has been assessed from a qualitative point of view, resorting to a formal modelling of the protocols combined with automated model-checking of typical scenarios as well as a manual proof. This allowed to discover and fix flaws in the protocols in presence of a Dolev-Yao attacker [5], that is by definition external to the peer-to-peer network. Then, in [3], we considered attackers within the system, *i.e.*, malicious peers, and showed by simulation the intrinsic resistance of the system to such misbehaving nodes.

To tackle the problem of malicious peers, one solution proposed in [3] was to introduce a grading system allowing peers to evaluate the outcome of their exchanges with others. With such a system, when a peer is misbehaving, others will be able to lower its grade and hopefully identify it as malicious.

In the current paper, we introduce and evaluate this grading system. The work is twofold: on the one hand we have implemented the grading system and evaluated it with simulations similar to those presented in [3]; on the other hand, we have formally modelled the protocol, including grading, and used model-checking to verify if malicious nodes are systematically detected. Both works have been related in the sense that interesting traces of the model have been manually checked to be effectively realisable in the implementation. The conclusions of our analysis is that systematics attacks on the data itself necessarily lead to lowering grades in such a way that malicious nodes are detected. But attacks on meta-information remain undetected and, worse, lead to false positive, *i.e.*, honest nodes are graded as malicious ones. Several solutions to this problem are proposed at the end of the paper.

In the sequel we focus on the modelling and analysis of the system, its implementation and simulation is a straightforward extension of that described in [3] and thus will be treated more briefly. The rest of the paper is organised as follows. In the next section, we present the UbiStorage system. Then, section 3 presents the formal model, section 4 compare honest and malicious agents modelling, section 5 introduces the grading system, and section 6 discusses the analysis we conducted. We finally conclude with some perspectives.

2. THE UBISTORAGE SYSTEM

The system developed by UbiStorage is fully distributed using peer-to-peer (P2P) communication. Each peer in the system is a network node that runs the various services and clients composing the system. The system is structured in three layers:

- *the application layer* is directly queried from the end-user interface and is based on three basic primitives: primitive *Put* is used to store a file in the system, primitive *Get* to get back a stored file, and primitive *Delete* to remove a file from the system. Each primitive is implemented as a corresponding protocol;

- *the communication layer* consists of a distributed hash table (DHT) that stores all the information needed by the application layer;

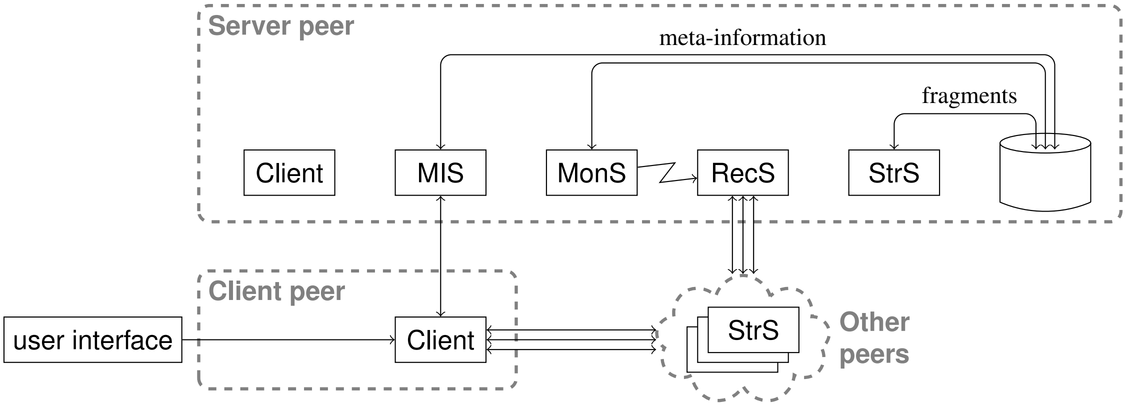

Figure 1. Internal organisation of peers and their communications. A disk stores data as well as meta-information about data.

- *the routing layer* uses a key-based routing protocol to dispatch the messages exchanged within the DHT.

Each peer is identified by a unique 128 bits identifier called a *PeerID*, it is responsible for storing files in a given range of file identifiers (128 bits *FileID*s), and it can allocate FileIDs for its own files in another range (under the responsibility of other peers).

The application layer is composed of five services also depicted in Fig. 1:

- *the client service*, depicted as Client, is responsible for executing the Put, Get and Delete primitives in response to requests from the end-user;

- *the storage service* (StrS) receives data fragments to be stored from other peers or sends them back when requested;

- *the meta-information service* (MIS) stores all the meta-information related to the data fragments locally stored. Each MIS is also responsible to keep all the meta-information for a range of FileIDs, the corresponding files being stored within the *leaf-set* composed of the peer's neighbours in the network. Notice that each peer belongs to several leaf-sets as a storer but to only one as a MIS;

- *the monitoring service* (MonS) monitors the MIS and issues warnings when critical files are found, *i.e.*, files for which there exists just enough data fragment to allow reconstruction;

- *the reconstruction service* (RecS) is responsible for actually reconstructing files and redistributing fragments within the leaf-set when an alert is raised by the monitoring service.

Each file stored in the system is actually split into $s+r$ fragments using the Reed-Solomon code [13], which allows to reconstruct the whole file from any s fragments (the remaining r fragments are for redundancy). Each storage service normally holds only one fragment of any file, identified by its *FragID* and the corresponding *FileID*. A meta-information service is thus responsible for remembering the storers that hold the fragments of a given FileID. This information can also be reconstructed from the storers themselves (which occurs when a peer, and thus its MI service, goes offline or crashes).

Communication in the system is based on the distributed hash table (DHT) Pastry [8]. More details about communication can be found in [3], for our current purpose, it is enough to note that a message sent by s to a node identified by i is routed through the network and eventually received by the peer whose identifier j is the closest to i among the peers that are actually online when the communication occurs. This is sketched in Fig 2.

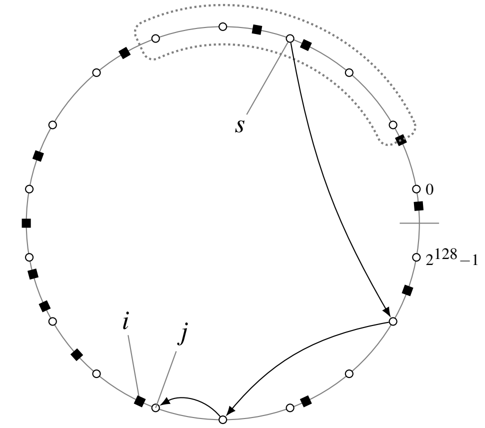

Figure 2. Organisation of the Distributed Hash Table. White dots denote online peers and black squares denote offline peers. The dotted region corresponds to the leaf-set of s.

3. FORMAL MODELLING

The UbiStorage system has been modelled using the ABCD formalism [11, sec. 5.2] that is an algebra of coloured Petri nets, *i.e.*, a process algebra with a coloured Petri net semantics. In this case, the colour domain is Python language [12], *i.e.*, data and computation are expressed in Python. For concision, we introduce ABCD directly while presenting the model, the required notions will be presented as needed. More details about ABCD may be found in [11]. Moreover, we will present only simplified excerpts of the model in order to focus on its most important aspects.

An ABCD process comprises a number of buffers that store data and can be accessed and shared by processes. The most primitive process is an atomic action that performs accesses to buffers, possibly guarded by a Boolean condition. Then, processes can be combined using control flow operations: sequence (operator "`;`"), choice (operator "`+`"), iteration (operator "`⋆`") and parallel composition (operator "`|`"). In order to bring modularity, one can use parametrised sub-processes that can be later instantiated. Our modelling makes an extensive use of sub-processes in order to structure the model in a way similar to the structure of the real system.

Fig. 3 shows a fragment of our model. Let us first concentrate on the **net** declarations that define processes (with nested declarations allowed): a process declaration `UbiSystem` (lines 3–49) includes two sub-processes declarations `Peer` (lines 7–44) and `User` (lines 46–47), to model a node in the P2P network and its user. At the end of `UbiSystem`, one instance of `Peer` and one instance of `User` are composed in parallel (line 49) to define the behaviour of process `UbiSystem`. A `Peer` declaration includes the declaration of the main services shown in Fig. 1, except that we omit the reconstruction part (and so monitoring and reconstruction services) as well as file deletion, but we include a `Router` sub-process responsible for passing messages throughout the P2P network. Monitoring and reconstruction has been omitted because we have abstracted away peers churning (*i.e.*, peers that transiently go offline) to concentrate our analysis on finite executions (*e.g.*, the storage of one file followed by its retrieval). Without churn, reconstruction and thus monitoring are simply pointless because no file ever needs reconstruction. Deletion has been omitted because it is very similar to the Get primitive: the only difference is that fragments are discarded by storers instead of being sent back. Routing has been modelled as an abstraction of the reality: as shown below, a message destinated to peer k is sent through $k-1$ that will forward it to k, so we have only one hop while in the real system there may be a lot more; moreover, a message always reaches its exact destination instead of the closest online peer as in the real system.

```
 1 buffer nw : object = ()
 2
 3 net UbiSystem (n, this, tasks, ●●●):
 4     buffer params : str⋆int = ()
 5     ●●●
 6
 7     net Peer () :
 8         ●●●
 9
10         net Router () :
11             [nw-(s,this,d,m), nw+(s,d,m)]
12
13         net Client () :
14
15             net GetClient () :
16                 buffer fileid : int = ()
17                 buffer storers : tuple(int) = ()
18                 buffer frags : int⋆tuple(str) = ()
19
20                 [params-("Get", Fid),
21                  frags+(Fid, ()), fileid+(Fid),
22                  nw+(this, (Fid-1) % n, Fid % n,
23                       (Fid,●●●,"MSG_GETFMI"))]
24
25                 ; [fileid?(Fid), storers+(strs),
26                    nw-(src, this, (Fid,●●●,strs))]
27
28                 ; ●●●
29
30             net PutClient () :
31                 ●●●
32
33             (GetClient() + PutClient()) ⋆ [False]
34
35         net Storer () :
36             ●●●
37
38         net MIService () :
39             ●●●
40
41         (Client() ⋆ [False])
42         | (Storer() ⋆ [False])
43         | (MIService() ⋆ [False])
44         | (Router() ⋆ [False])
45
46     net User () :
47         [params<<(tasks)]
48
49     Client() | User()
50
51 UbiSystem(6, 1, (("Get", 2), ●●●), ●●●)
52 | ●●●
```

Figure 3. Excerpt of the ABCD model of the UbiStorage system. Symbol "●●●" denotes parts that have been skipped for the sake of clarity. Like in Python, indentation is used as the blocks delimiter. This fragment also includes simplifications for the sake of clarity (*e.g.*, management of sessions numbers is not shown).

At the end of `Peer` declaration (lines 41–44), its sub-processes are instantiated, each is put into an infinite loop and the resulting processes are composed in parallel. For instance, `Client()⋆[False]` is a loop that allows to execute `Client()` repeatedly, the loop being able to exit whenever the corresponding exit action, here `[False]`, is executed. But, in ABCD, `[False]` is an atomic action that can never be executed. So we have an infinite repetition of `Client()`, *i.e.*, we have a service that repeatedly handles requests without terminating.

Process `Client` is itself split into the two basic subprocesses `PutClient` and `GetClient` modelling the corresponding primitives. Delete primitive has been omitted as explained above. The main process for `Client` is a choice (+) between a Put and a Get, infinitely repeated.

Let us look now at the process parameters and buffers declarations. Process `UbiSystem` is parametrised by `n` the number of peers in the system, `this` the identity of the peer that will be instantiated from this declaration, and `tasks` a list of operations that the user of this node will request to the client service. Line 4, a buffer `params` is declared, its type is `str*int` which denotes the Cartesian product between `str` and `int`, *i.e.*, the buffer can hold pairs whose first components are strings and second components are integers, and it is initially empty (as denoted by "`=()`"). This buffer is filled by process `User` line 47: atomic action `[params<<(tasks)]` denotes that buffer `params` is populated by the content of collection `tasks`, *i.e.*, `tasks` is iterated over and every of its values in it is added to the buffer. We can see line 51 that when `UbiSystem` is instantiated, `tasks` is actually a tuple of pairs (whose type is indeed `str*int`).

Let us now examine sub-process `Router` and buffer `nw`. The latter is used to model network communication, messages to be routed are exchanged as tuples (s,r,d,m) where s is the source, r is the router, d is the destination and m is the message content. Direct messages are simply exchanged as tuples (s,d,m), such a message is consumed from `nw` and used directly at its destination. Usually, a node is first contacted through message routing, and the answer is sent directly since the node receives its peer's address within the message. A `Router` process whose identity `this` is r repeatedly consumes 4-tuples (s,r,d,m) from `nw` and produces 3-tuples (s,d,m) instead, which models routing and corresponds to the atomic action line 11: `nw-(`$\cdots$`)` is a consumption on buffer `nw` while `nw+(`$\cdots$`)` is the production of a value onto `nw`.

Finally, let us discuss sub-process `GetClient` that models the execution by a client service of the Get primitive. Fig. 3 shows only the two first actions of this execution, which is enough to demonstrate how protocols are modelled. After three buffer declarations, sub-process `GetClient` is defined as a sequence ("`;`") of atomic actions. In the first one, the following accesses to buffers are atomically performed (if this possible, otherwise the action is blocked and none of its access is performed):

- line 20, a pair whose first element is string `"Get"` and second element is bound to variable `Fid` is consumed in buffer `params`. This models the receiving of a Get request from the user;
- at the beginning of line 21, a pair whose first value is the freshly bound `Fid` and second value is an empty tuple is produced in buffer `frags`. This will be used to store the file fragments received during the Get protocol in order to reconstruct the requested file;
- at the end of line 21, `Fid` is also saved to buffer `fileid` so it can be reused in the sequel of the process;
- lines 22–23, a message is sent onto the network by producing a tuple in buffer `nw` following the pattern (s,r,d,m) explained above where here s is `this`, the current peer, r is `(Fid-1) % n`, the peer that immediately precedes the target (peers are organised as a ring, so we need the modulo "`%n`"), d is the peer that has the same number as the file (modulo `n`) and m is a triple (f,i,t) where f is a FileID, i is a session identifier (not shown here), and t a string indicating the type of the message (`"MSG_GETFMI"` reads as "message get file meta-information").

This message will be routed and eventually reach the MI service responsible for the requested file that will answer providing a list of storers holding fragments for this file. In the second action of the sequence, this answer is retrieved from `nw`:

- line 25, the FileID is retrieved from buffer `fileid` and bound to variable `Fid`. Using `fileid?` instead of `fileid-` allows to get the value without consuming it;
- line 26, the message is received (and consumed) from the network, the message part being a triple composed of the FileID, the session number and the expected list of storers;
- this list is saved into buffer `storers` at the end of line 25.

The protocol continues in a similar way, requesting fragments from the storers, gathering them in buffer `frag` and finally reconstructing the file from the retrieved fragments. The whole model is about 300 lines long, following the same principles. Further details about `Storer` and `MIService` are given in the next section.

4. MALICIOUS PEERS

Various malicious behaviours have been considered in [3] and their impact have been evaluated through simulation. We recall now the malicious behaviours we have considered and describe how malicious services have been modelled using ABCD, comparing them with the model of the corresponding honest services.

A meta-information service is responsible for maintaining a list of nodes that store fragments of the files added to the P2P system, this list is sent to a client during the execution of the Get protocol so that the client can query the appropriate storers to get back the required fragments. A *malicious MI service* can send an invalid list of storers and thus block a client from retrieving its files. The information is not lost and can be retrieved from the storers themselves, but this is not feasible for the client that only knows a FileID, and its only contact in the network for it is the corresponding peer running the MI service. This peer is the only one in the network that exactly knows which other peers form its leaf-set and can possibly store fragments for this FileID.

A storage service is responsible for storing a fragment of a file. One possible malicious behaviour, that we call *malicious success on store*, is when the storage service pretends it stores a fragment but it actually does not. When requested for this fragment, it sends dummy data instead. This may lead to file loss since the number of fragments actually stored in the system is lower than expected.

Another possible malicious behaviour for a storage service is the *malicious fail on store* where the service pretends that it fails to store a fragment while it actually succeeds. Because of the declared failure, the storage service is not recorded as holding a fragment for this file and may be proposed later on another fragment for the same file. By repeatedly failing, the malicious service may try to collect enough fragments so that it can reconstruct the file. We have shown in [3] that this attack is statistically hard to achieve, moreover, since files are encrypted before to be fragmented, this is not a dangerous attack. However, our purpose in this paper is to assess whether using a grading system can help to detect the attack or not, so we have modelled this attack also.

```
 1 net MIService():
 2     buffer meta : int*tuple(int) = •••
 3
 4     ([nw-(src, this, (Fid,ids,"MSG_GETFMI")),
 5       meta?(Fid,str),
 6       nw+(this, src, (Fid,ids,str))]
 7      + [nw-(src,this, (Fid,ids,"MSG_GETSTRLST")),
 8         nw+(this, src, (Fid,ids,•••))]
 9      + [nw-(src, this, (Fid,src,"MSG_PUBFMI")),
10         meta-(Fid,s), meta+(Fid,s+(src,))])
11     * [False]
```

Figure 4. Simplified model of a honest MI service, which should be inserted line 38 in the model of Fig. 3. Two levels of indentation have been removed for the sake of legibility.

```
 1 net MaliciousMIService():
 2     buffer fakedmeta : int*tuple(int) = •••
 3
 4     ([nw-(src, this, (Fid,ids,"MSG_GETFMI")),
 5       fakedmeta?(Fid,str),
 6       nw+(this, src, (Fid,ids,str))]
 7      + [nw-(src,this, (Fid,ids,"MSG_GETSTRLST")),
 8         nw+(this, src, (Fid,ids,•••))]
 9      + [nw-(src, this, (Fid,src,"MSG_PUBFMI"))])
10     * [False]
```

Figure 5. Simplified model of a malicious MI service, intended to replace that shown in Fig. 4.

4.1. Meta-information service

The ABCD model of a honest MI service is shown in Fig. 4. Line 2, a buffer `meta` is declared to store the meta-information as pairs (i,s) where i is a FileID and s a tuple of storers. This buffer is initialised with pairs $(i,())$ for all i in the leaf-set of the `MIService` instance (which is not shown here). Type `int*tuple(int)` denotes the set of pairs whose first components are integers and whose second components are tuples of integers. Then, the process is basically an infinitely repeated choice between three actions:

- handle a request from a `GetClient` that needs meta-information (lines 4–6): a message is received lines 4, requesting for the meta-information associated to FileID `Fid`, with a session number `ids`; the requested information is retrieved line 5 from buffer `meta`; the answer is sent back line 6, all within the same atomic action;
- handle a request from a `PutClient` that needs a list of storers available to store new fragments (lines 7–8): the request is received line 7 and immediately answered line 8, the list of storers, not shown here, is computed from the set of this peer's neighbours (*i.e.*, its leaf-set);
- handle a request from a `Storer` that declares it stores a fragment for a file (lines 9–10): the request is received line 9 and processed line 10 by getting the current meta-information for file `Fid` in buffer `meta` and updating the second part of the pair by adding the PeerID `src` of the storer that has sent the message.

The actual model is a bit more complex because it handles the coupling of the second and third actions: when a `"MSG_GETSTRLST"` message is handled, the MI service sends a number of storers identifiers to the client that will request them to store a fragment; so, the MI service shall expect as many `"MSG_PUBFMI"` messages (or timeouts not shown here) from exactly the same storers and for exactly the same FileID.

With respect to the honest MI service, the malicious one differs on essentially three points as show in Fig. 5:

- line 2, buffer `meta` is replaced with `fakedmeta` that plays the same role but is initialised with dummy data;
- line 5, the answer to a `"MSG_GETFMI"` message is taken from buffer `fakedmeta`, but the request handling is otherwise the same;
- line 9, `"MSG_PUBFMI"` messages are received but then discarded so that buffer `fakedmeta` is never updated.

4.2. Storage service

The model of a honest storage service is shown in Fig. 6 and consists of an infinite repetition of two actions:

```
1  net Storer():
2      buffer data : int*str = ()
3
4      ([nw-(src, this, (Fid,ids,"MSG_GET")),
5            data?(Fid,frag),
6            nw+(this,src,(Fid,ids,frag))]
7       + [nw-(src, this, (Fid,ids,"MSG_PUT",frag)),
8           data+(Fid,frag),
9           nw+(this, Fid%n, (Fid,this,"MSG_PUBFMI")),
10          nw+(this, src, (Fid,ids,"MSG_OK"))])
11     * [False]
```

Figure 6. Simplified model of a honest storage service, which should be inserted line 35 in the model of Fig. 3. Like in Fig. 4, indentation has been shortened.

```
1  net WriteKOStorer () :
2      ([nw-(src, this, (Fid,ids,"MSG_GET")),
3        nw+(this, src, (Fid,this,"MSG_TIMEOUT"))]
4       + [nw-(src, this, (Fid,ids,"MSG_PUT",frag)),
5          nw+(this, Fid%n, (Fid,this,"MSG_TIMEOUT")),
6          nw+(this, src, (Fid,ids,"MSG_KO"))])
7      * [False]
```

Figure 7. Simplified model of a malicious storage service that fails on store, which should replace code shown in Fig. 6.

```
1  net WriteOKStorer () :
2       ([nw-(src, this, (Fid,ids,"MSG_GET")),
3         nw+(this, src, (Fid,ids,"DUMMY"))]
4        + [nw-(src, this, (Fid,ids,"MSG_PUT",frag)),
5           nw+(this, Fid%n, (Fid,this,"MSG_PUBFMI")),
6           nw+(this, src, (Fid,ids,"MSG_OK"))])
7       * [False]
```

Figure 8. Simplified model of a malicious storage service that fakes success on store, which should replace code shown in Fig. 6.

- lines 4–6, a request to send a fragment is handled: the request message is first received line 4, then, the corresponding fragment is retrieved from buffer `data` line 5, and the answer is sent back line 6;

- lines 7–10, a request to store a fragment is handled: the corresponding message is received line 7, storage takes place line 8, and two messages are sent back. Line 9, a `"MSG_PUBFMI"` message is sent to the MI service so it can record that the storer now holds a fragment for this file; line 10, an acknowledgement is sent to the client that requested the storage.

Like previously, we have not shown the possibility of a timeout. Since ABCD does not include an explicit notion of time, this is simply modelled with a non-deterministic choice between success and timeout. Notice also that, from the declaration of buffer `data` line 2, we can deduce that fragments are modelled as strings.

The malicious storage service comes in two flavours, respectively shown in Fig. 7 and Fig. 8. The former always fails on store as show in the answers lines 3, 5, and 6. The latter always pretends it succeeds on store but returns dummy data on a Get request as show line 3. In both cases, we do not need a buffer `data` at all since no fragment is ever stored. This would be useful only to model attempts to reconstruct data from accumulated fragments. But we did not model this aspect because this is a very combinatorial operation and, as already explained, this is a worthless attack because data is encrypted before to be fragmented.

5. GRADING SYSTEM

Peers can be extended with an evaluation of the outcome of their exchange with others, allowing to build a grading system. Each peers stores for each other a grade in interval $[0,1]$ and update this value on each communication. If a grade lowers below 0.5 then the corresponding peer is considered as potentially malicious. Conversely, those peers with a grade greater than 0.5 are considered as probably honest.

Two exchanges can be basically evaluated, corresponding to the Get and Put protocols. Indeed, in both cases, the outcome can be evaluated as satisfactory or not. The question of evaluating exchanges with the MI service is more complex (except if the MI service is unresponsive, which is a trivial case but also not a good strategy for a malicious node). In every case, the MI service is supposed to send a list of storers (to put or to get fragments), but, if some of these storers do not provide the service as expected (for instance, they fail to send back a fragment), this may be because they are malicious or because the MI service had wrongly provided them in its response. And actually, we will see later on that this latter case occurs in our model. Because of this ambiguity, we do not consider here the grading of MI services and this is left to future works.

Modelling grades is made by adding a buffer `grade` to every peer, initialised with all grades set to 0.5 and stored as pairs (i,g) where i is a peer identity and g its grade. Then, each communication outcome is evaluated, either by raising or by lowering the grade by steps of ± 0.1. For instance, Fig. 9 shows the reception of the last message of a Put by the client (*i.e.*, the reception of, *e.g.*, the message sent on line 10 in Fig. 6 or that sent on line 6 in Fig. 7). As one can see, this is a very simple grading system, yet it is already quite useful as we shall see below.

```
    ([nw-(src, this, (Fid,ids,"MSG_OK")),
      grade-(this,src,x), grade+(this,src,x+0.1),
      •••]
     + [nw-(src, this, (Fid,ids,"MSG_KO")),
        grade-(this,src,x), grade+(this,src,x-0.1),
        •••])
```

Figure 9. Adjusting the grade of a storer during the execution of the Put protocol.

The evaluation of a Get is a bit more complex since the client first tries to reconstruct the file before to evaluate whether the fragments retrieved from storers allowed to do so or not.

It is worth noting that a real failure of a peer (*i.e.*, a failure that is not faked) results in lowering its grade. Fortunately, this is not a problem because in the case of a temporary failure, the lowering will be compensated by a raising at the next successful exchange; and in the case of a permanent failure, the peer will be banned because it will be considered as malicious while it is actually crashed or damaged. But both situations require UbiStorage to manually check the peer and possibly replace it. So, banning peers is just an automated way to eliminate nodes that are broken in one or another way.

6. ANALYSIS

Several variants of our ABCD model have been considered, including one or another combination of malicious peers (including none), and considering various execution scenarios consisting of series of Get and Put. The Petri net semantics has been computed using the ABCD compiler distributed with SNAKES toolkit [10]. Then, this net has been passed to Neco compiler [6] that could build an optimised implementation of the Petri net allowing to accelerate its state space exploration. A typical state space for our models has about 400,000 states that can be computed in about 10 minutes on an Intel® Core$^{\mathrm{TM}}$ i5-520M CPU at 2.40GHz equipped with 4Gb of RAM; then, 2 to 4 additional minutes are typically needed to analyse the state space.

In every case, we could verify that a malicious storer cannot avoid obtaining a bad evaluation from the peers it interacts with, *i.e.*, in every execution, whenever a storer sends a dummy fragments or fails storing a fragment, the requester will necessarily lower this peer's grade. From this it results that a systematic malicious behaviour of a storage service is necessarily detected. However, a storer that would alternate between the correct and malicious behaviour would be able to keep an average grade (globally over the nodes).

This results from the formal verification of the abstract model have been compared with a concrete implementation of the grading system within the actual UbiStorage system. The software has been extended similarly to the model and the simulation infrastructure used in [3] has been reused, allowing to observe that malicious peers were indeed detected by other peers thanks to the grading system. By excluding from the network a peer detected as malicious by enough other peers, we could actually reduce the number of files lost during simulations.

However, the overall situation is not as good as it could appear. Indeed, being a systematic approach that checks every possible execution, model-checking allowed to discover situations where honest storers would get bad grades even if they do not fail at any time during the execution. This occurs when a malicious MI service sends a wrong list of storers to answer a `"MSG_GETFMI"` message. When doing so, the client is amened to ask honest storers for fragments they do not have, which results in lowering their grades.

One obvious reason for this situation is that we do not model a reputation system (like, *e.g.*, [4, 7]), in which peers exchange the grades each gives and aggregate them to compute a reputation that reflects the overall perception of each peer's honesty from the others' point of view. This is somehow what we implemented in our simulation since peers' grades have been observed from the simulation engine, allowing to combine local results into a more global view. However, this was possible within a simulation but cannot be realised this way in a real implementation that is distributed.

We intend to extend our grading system so it becomes a reputation system, in particular, we would like to reuse the two-level reputation system defined and analysed in [9]: one level is a reputation that is built over a grading system like ours, a second level is a confidence about reputations that each peer builds by comparing the reputation it computes with that computed by others. It is shown in [9] that using these two levels allows a faster detection of malicious peers, including when some peers are "protecting" others (*i.e.*, they are honest in their data exchange, but dishonest when grading malicious peers in order to hide that these peers are malicious). We believe that our problem is similar to this behaviour (instead of malicious peers being reported as honest in [9], honest peers are reported as malicious in our case) and so that the good results of [9] should also hold in our case.

Two additional solutions are envisaged to cope with this false positive problem. First, we could distribute the grades updating over the peer running the storage service and the peer running the MI service when the failure of the former may come from the latter. If a MI service is honest, is would get bad marks because of dishonest or failing storers, but this will be compensated by the good marks it will get also by providing the correct storers. Similarly, a honest storer will get bad marks when it is wrongly proposed by a MI service, but, in a reputation system that does not rely on a single peers grades, this will be compensated by the other marks it gets from other transactions.

Another solution would be to change the Get protocol so that a list of storers is sent by a MI service together with proofs that these storers actually hold the fragment they are supposed to hold. This proof could be for instance a digital signature of the fragment by the storer, that the MI service would receive with the `"MSG_PUBFMI"` message (see Fig. 4, line 9). Using such a system, a MI service sending a wrong list of storers would be detected as doing so because it would not be possible for it to produce faked signatures.

7. CONCLUSION

We have shown how a peer-to-peer based storage system can be formally modelled in order to design and assess its extension with a grading system. This system allows each peer to evaluate the outcome of its transactions with others in order to detect misbehaving peers, in particular malicious ones. By using formal verification, we could show that malicious peers are necessarily detected if their incorrect behaviour is systematic and directly concerns data storage. This conclusion has been confirmed using simulations of the real system, extended with grading. Formal verification also allowed to discover that peers lying about meta-information may introduce false positive in the detection of malicious peers. We have proposed three complementary ways to solve this problem: extending grading to a fully fledged reputation system, distributing bad marks on data storers and meta-information services when a fault could come from both, and resorting to cryptographic signature to validate meta-information.

Future work will consider these extensions, following the same methodology as in this paper, *i.e.*, formal modelling and analysis followed by implementation and simulation. The former approach allows to quickly obtain results since the model is used as a prototype, and formal verification provides a good confidence with respect to the qualitative properties of the system. Then, simulation allows to check the feasibility of the abstract model and to assess quantitative properties of the system.

REFERENCES

[1] SPREADS project. `http://www.spreads.fr`.

[2] Ubiquitous Storage. `http://www.ubistorage.com`.

[3] S. Chaou, F. Pommereau, and G. Utard. Evaluating a peer-to-peer storage system in presence of malicious peers. In *Proc. of SPCLOUD/HPCS'11*, IEEE Digital Library. IEEE, 2011.

[4] T. Cholez, I. Chrisment, and O. Festor. A distributed and adaptive revocation mechanism for P2P networks. In *Proc. of the 7th International Conference on Networking*. IEEE Computer Society, 2008.

[5] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

[6] Ł. Fronc and F. Pommereau. Optimising the compilation of Petri net models. In *Proc. of SUMO'11*, volume 726 of *Workshop proceedings*. CEUR, 2011.

[7] F. Lesueur, L. Mé, and V. Viet Triem Tong. Detecting and excluding misbehaving nodes in a P2P network. *Studia Informatica Universalis*, 7(1), 2009.

[8] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. of IPTPS'03*, 2003.

[9] M. Morvan and S. Sené. A distributed trust diffusion protocol for ad hoc networks. In *Proc. of ICWMC'06*. IEEE Press, 2006.

[10] F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, October 2008.

[11] F. Pommereau. *Algebras of coloured Petri nets, and their applications to modelling and verification*. LAMBERT Academic Publishing, 2010.

[12] Python Software Foundation. Python programming language. `http://www.python.org`.

[13] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM journal on applied mathematics*, 8(2), 1960.

[14] S. Sanjabi and F. Pommereau. Modelling, verification, and formal analysis of security properties in a P2P system. In *Proc. of COLSEC'10*, IEEE Digital Library. IEEE, 2010.