2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Analysing Selfishness Flooding with SEINE

Guido Lena Cota*, Sonia Ben Mokhtar†, Gabriele Gianini*, Ernesto Damiani*§,
Julia Lawall‡, Gilles Muller‡, and Lionel Brunie†

*Università degli Studi di Milano, †Université de Lyon, CNRS LIRIS INSA Lyon, ‡Sorbonne Universités/UPMC/Inria/LIP6,
§EBTIC/Khalifa University, Abu Dhabi, UAE

***Abstract*—Selfishness is one of the key problems that confronts developers of cooperative distributed systems (e.g., file-sharing networks, voluntary computing). It has the potential to severely degrade system performance and to lead to instability and failures. Current techniques for understanding the impact of selfish behaviours and designing effective countermeasures remain manual and time-consuming, requiring multi-domain expertise. To overcome these difficulties, we propose SEINE, a simulation framework for rapid modelling and evaluation of selfish behaviours in a cooperative system. SEINE relies on a domain-specific language (SEINE-L) for specifying selfishness scenarios, and provides semi-automatic support for their implementation and study in a state-of-the-art simulator. We show in this paper that (1) SEINE-L is expressive enough to specify fifteen selfishness scenarios taken from the literature, (2) SEINE is accurate in predicting the impact of selfishness compared to real experiments, and (3) SEINE substantially reduces the development effort compared to traditional manual approaches.**

I. Introduction

Selfish behaviours are an inherent and critical problem of cooperative distributed systems such as peer-to-peer (P2P), grid and volunteer computing, and self-organising networks. Such behaviours are performed by participants that benefit from the system without contributing their fair share to it [13], [14], [30]. An example of selfish behaviour in a P2P live streaming system is to download a given video file without sharing it with other nodes in order to save local bandwidth. It has been measured that such behaviours may severely impact system performance, causing a significant degradation of the system's reliability and efficiency [11], [18], [20]. For instance, in the above-mentioned live streaming system, if 25% of nodes free ride by not sharing the received video chunks, then half of the remaining nodes receive a degraded stream [11].

In this context, understanding the impact that selfish behaviours have on the system performance is crucial to the design of effective selfishness countermeasures. However, this can be done only by modelling and injecting selfish behaviours into the system under consideration, which is a non-trivial task. Indeed, to carry out this task, the system designer has to first analyse the functional specification of the considered system and identify those steps (e.g., functions) for which selfish nodes may behave in a non-cooperative way. Then, for each of the identified steps, the designer has to decide what are the possible selfish behaviours that are meaningful in the context of her application and implement the corresponding behaviours. Finally, the designer has to invest considerable effort in experiments to assess the impact of the introduced behaviours on the performance of the system.

For the purpose of these experiments, a designer can rely on frameworks for developing and evaluating real distributed systems [16], [19] or simulations of such systems [2], [27]. However, existing frameworks do not provide any specific support for modelling and injecting selfishness, which has to be done manually. In practice, the designer hard codes both the control and logic of selfish behaviours into the parts of the system implementation that are affected by those behaviours. This activity typically results in generating variant implementations of the same system (i.e., one for each node behaviour), or in creating a single implementation that incorporates all the possible behaviours as well as the algorithmic functionalities for their control (e.g., variables, if-clauses). The increased complexity and redundancy of the source code reduce its readability and maintainability. Finally, once a system implementation is available, the designer usually proceeds with an extensive experimental campaign to quantify the harm caused by various manifestations of different proportions of selfish behaviours. To the best of our knowledge, such a domain-specific evaluation has to be conducted manually by the system designer, which is tedious and time-consuming.

In order to help system designers in this task, we propose *SEINE*, a framework for modelling various types of selfish behaviours in a given system and automatically understanding their impact on the system performance through simulations. *SEINE* relies on a *Domain-Specific Language* (DSL), called *SEINE-L*, for describing the behaviour of selfish nodes, along with an annotation library to associate such specifications with a system implementation for the state-of-the-art simulator PeerSim [27]. To design *SEINE-L*, we have conducted a domain analysis on state-of-the-art papers related to building selfish-resilient systems. *SEINE-L* provides a unified semantics for defining *selfishness scenarios*, which allow describing capabilities, interests and behaviours of different types of nodes participating in the system. The *SEINE* framework provides a compiler and the run-time system supporting the automatic and systematic evaluation of different selfishness scenarios in the PeerSim simulation framework. Simulations return a set of statistics on the behaviour of the system when in the presence of the specified types of selfish nodes.

The use of the *SEINE* framework supports a clear separation of selfishness concerns from the main logic of a cooperative distributed system. This separation improves overall

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.51

maintainability, reuse, and reproducibility of both the system implementation and experiments. Particularly, the *SEINE-L* specification allows system designers to describe and easily compare the same experimental conditions in different versions of the same cooperative system.

Overall, the present work makes the following contributions:

- We present the design of *SEINE-L* by conducting an analysis of 15 state-of-the-art papers related to the subject. We evaluate the expressiveness of the language by showing that *SEINE-L* can capture the semantics of the 38 selfish behaviours described in the papers analysed.
- We assess the impact of a selfishness scenario in a PeerSim simulation. Through the evaluation of three complete use cases, namely, a gossip-based dissemination protocol, a live streaming protocol (i.e., BAR Gossip [20]) and a file sharing system (i.e., BitTorrent [5], [22]), we show that the simulations enabled by *SEINE* are accurate with respect to real measurements performed on these systems.
- We show that *SEINE* facilitates a substantial reduction in the effort required to model, code, and evaluate selfish behaviours in a given system. First, we evaluate the effort quantitatively, showing that the number of lines of code required for assessing different selfishness scenarios in the three use cases using *SEINE* is almost an order of magnitude lower than the one required by their manual implementation. Then, we present a qualitative evaluation discussing the ease of use of *SEINE* in the rapid development and test of different selfishness scenarios.

The remainder of the paper is organised as follows. Section II introduces background information on selfishness in cooperative systems and Section III presents an exhaustive analysis of the literature on the subject. Section IV provides an overview of *SEINE*, followed by a detailed description of its components: the DSL for modelling a selfishness scenario (Section V) and the support tools for injecting it into a PeerSim simulation (Section VI). Section VII presents a performance evaluation of *SEINE*. Section VIII reviews related work. Finally, the paper concludes in Section IX.

II. Background

A cooperative system is a complex distributed system that relies on voluntary resource contributions from its participants to perform the system function. File sharing systems (e.g., Gnutella [14], eDonkey [13], BitTorrent [5]) are the most widespread and well-known examples of cooperative systems. Other examples include cooperative distributed computing [1], [17], self-organizing wireless networks [24], [25], [35], anonymous communication protocols [29], and many others [3], [6], [10]. Most cooperative systems are characterised by untrusted autonomous individuals with their own objectives — that are not necessarily aligned with the system's objectives — and full control over the device that they use to interact with the system [26]. In this context, a selfish node is an autonomous, strategic and self-interested individual that cooperates with other nodes only if such behaviour increases its local benefits. In practice, a selfish node may choose to stop behaving

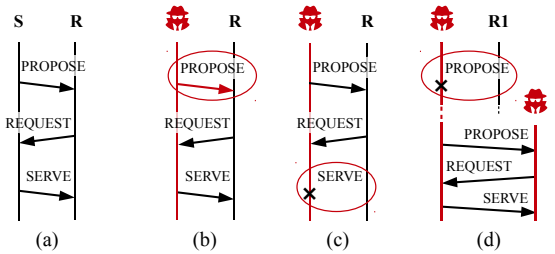


Fig. 1: Selfishness manifestations in gossip-based live streaming dissemination [11].

cooperatively if one or more of the following conditions occur: (i) the cost of contributing resources to other nodes outweighs the benefits received from the system; (ii) the system does not impose punishments for selfish behaviours, or the punishment is not fast, certain and painful enough to be credible; (iii) there are economic or social reasons for cooperating only with a restricted group of nodes; (iv) the node suffers from persistent resource shortages, due for example to hardware or software limitations of the device that hosts the node (e.g., battery-powered devices).

Selfish behaviours have been observed in many cooperative systems and may have multiple manifestations [14], [29], [30], [35]. For instance, let us consider the gossip-based live streaming system described by Guerraoui et al. [11], consisting of a source node that disseminates video chunks to a set of nodes over a P2P network. Figure 1(a) represents the simplified interaction protocol between nodes participating in this system: a node S periodically sends a `PROPOSE` message containing the video chunks it has received to a set of randomly chosen partners (R), and asks them to reply with a `REQUEST` message that indicates the chunks they are missing. Finally, S delivers the requested chunks with a `SERVE` message.

If S is selfish, it may decide to save its bandwidth consumption by reducing the number of chunks R would request. To this end, S can provide false information to R, proposing fewer chunks than it currently has available (see Figure 1(b)). Another strategy for the selfish node S to reduce its bandwidth consumption is to not serve all the requested chunks, but only a subset of them. In the extreme, S serves no chunks (see Figure 1(c)). Guerraoui et al. demonstrated experimentally that if 25% of nodes make deviations like those in (b-c), then the fraction of cooperative nodes that are not able to view a clear stream reaches up to 50%. Lastly, Guerraoui et al. considered the possibility of collusion among nodes. For example, in Figure 1(d) S does not start the dissemination protocol with nodes outside its colluding group (R1 in the figure), so as to dedicate more bandwidth to exchanging data with its colluders. Experiments have shown that a colluder can decrease its contribution up to 15% without suffering any performance degradation [11].

The examples above describe various manifestations of selfishness in a particular system and the impact they have

Reference	Domain	*Selfish deviation type* [a]				
		D	F	M	C	O
Ben Mokhtar et al. [3]	Data Distribution	√	×	×	√	×
Ben Mokhtar et al. [4]	Data Distribution	√	√	×	×	×
Guerraoui et al. [11]	Data Distribution	×	√	√	√	×
Hughes et al. [14]	Data Distribution	√	×	×	×	×
Li et al. [20]	Data Distribution	×	√	√	√	×
Lian et al. [21]	Data Distribution	×	×	×	√	×
Locher et al. [22]	Data Distribution	√	×	√	×	√
Piatek et al. [32]	Data Distribution	×	√	×	√	×
Sirivianos et al. [34]	Data Distribution	√	×	√	×	×
Anderson et al. [1]	Computing	×	√	×	√	×
Kwok et al. [17]	Computing	×	√	√	×	×
Cox and Noble [6]	Backup & Storage	×	√	×	×	×
Gramaglia et al. [10]	Backup & Storage	√	×	×	×	×
Mei and Stefa [24]	Networking	√	√	×	×	×
Ngan et al. [29]	Anonym. Comm.	√	√	×	×	×

[a] D: defection , F: free ride, M: misreport, C: collusion, O: other types.

TABLE I: The papers reviewed for the domain analysis, with information about their application domain and the selfishness investigated.

on its performance. To help a designer assess the impact of selfishness on any cooperative system, we start by presenting, in the following section, a unified model for selfish behaviours resulting from an extensive analysis of state-of-the-art works on this topic.

III. Domain Analysis

To gather domain knowledge on the problem of selfishness in cooperative systems, we performed a systematic analysis of the problem domain. Our goal is to identify possible commonalities in the motivations and executions of such behaviours, so as to build a domain-specific terminology and semantics for their representation and understanding.

Given the vast body of literature on the subject, we selected as inputs of the domain analysis 15 state-of-the-art papers that are of particular interest to the research community and that provide detailed descriptions of concrete selfish behaviours. Table I lists the selected papers and reports some of their relevant aspects, namely, the application domain of the target system (e.g., data distribution, distributed computing, networking) and other information to characterise the selfishness manifestation therein investigated. The output of our analysis is a model for the specification of *selfishness scenario*s in cooperative systems, whose formal representation is given by the feature diagram in Figure 2.[1] A selfishness scenario consists of a non-empty set of *node models*, which describe interests and capabilities of types of nodes, and a set of *selfishness models*, which describe selfish behaviours. We present each of these components below.

1) Node model: The participants of a cooperative system constitute a heterogeneous population both in their personal interests and capabilities. A node model describes a type of participant in the system and specifies their interests and capabilities in terms of *resources*. A resource is a physical or

[1]The feature diagram in the figure is a cardinality-based extension of the *FODA* notation [7].

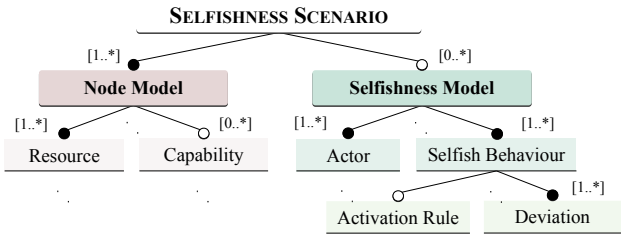


Fig. 2: Feature diagram of a selfishness scenario.

logical commodity that increases the personal utility of the nodes that possess it. A physical resource represent a node's capacity, such as *bandwidth*, *CPU* power, *storage* space, or *energy*. A logical resource can be a high-level and application-specific *service* offered by the cooperative system (e.g., file-sharing, message routing), or the *incentive* created by a cooperation enforcement mechanism (e.g., money, level of trust). The *capability* of a node defines a constraint over a resource. For example, mobile nodes usually have lower communication and computation capabilities than desktop nodes, which can be expressed as a stricter constraint on the bandwidth and CPU resource, respectively.

2) Selfishness model: A node can be the *actor* of one or more selfish *behaviours*. In a selfishness model, a behaviour is described as the implementation of a non-empty set of *deviations* from the intended execution of protocols in the cooperative system. We define a *deviation point* as the step of a system protocol in which a deviation may take place. The wide range of motivations behind selfish behaviours, as well as the application-specific nature of their implementation, generates a tremendous number of possible deviations for any given cooperative behaviour. Nevertheless, based on our review of the available literature, we could identify four recurring types of deviations, named defection, free-riding, misreporting, and collusion. As shown in the last columns of Table I, these types match almost all the selfish behaviours analysed in our review. The only exception is the rarest-first policy for requesting file pieces, which is very specific to the implementation of BitTorrent [22].

A *defection* is an intentional omission in the execution of a system protocol. A selfish node performs a defection to stop the protocol execution, so as to prevent requesters from consuming or even asking for its resources. *Free-riding* is a reduction in the amount of resources contributed by a node without stopping the protocol execution. The literature on cooperative systems offers other definitions of free riding, such as the complete lack of contribution [22], [29], [34], or downloading more data than what is uploaded [14]. Our definition is more general because it applies to any resource, and it is more precise because it can be clearly distinguished from deviations that achieve a similar result by stopping the system protocols. A *misreport* consists in the communication of false or inaccurate information, to avoid contribution or gain better access to resources. Finally, a *collusion* is the coordinated execution of a selfish behaviour by a group of

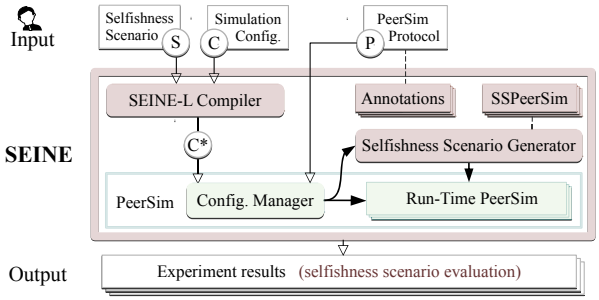


Fig. 3: Overview of the *SEINE* framework.

nodes that act together to increase their benefits. Collusions are more difficult to detect than individual deviations [3], [11], because colluders can reciprocally hide their misbehaviours. Examples of misreporting and collusion have been discussed in the previous section and shown in Figure 1(b) and (d).

In the selfishness model, the *activation* rule describes the conditions that motivate a node to start behaving selfishly. Examples of activation rules are exceeding a threshold amount of resource consumption or the delivery of a service (e.g., a file download). Another typical situation consists in providing false information to a monitoring mechanism to cover up previous deviations. Thus, a selfish behaviour may be the activator of other selfish behaviours.

IV. *SEINE* Overview

The *SEINE* framework aims to help cooperative system designers to evaluate the impact of selfish behaviours on the system performance. The framework builds on the lessons learnt from the domain analysis presented in Section III, providing designers with modelling and simulation tools to describe and experiment with selfishness scenarios in a given system. *SEINE* relies on the PeerSim open-source simulator for large-scale distributed systems [27]. The results of the simulation experiments are the output of *SEINE*.

Figure 3 provides an overview of the *SEINE* framework. To begin, the system designer (hereafter "Designer", for brevity) produces the input files required by the framework, namely, a selfishness scenario (*S* in the figure) specified using the *SEINE-L* DSL, a configuration file to set up the simulation (*C*), and a Java implementation of the protocols underlying the system (*P*). The clear separation between selfishness and implementation concerns facilitates maintenance and reuse of the *S* and *P* artefacts. To associate the DSL declarations to the affected protocol implementation components (e.g., classes, variables, methods), the Designer decorates such components using a library of *Annotations* provided by the *SEINE* framework.

Upon creating all the input files, the Designer uses *SEINE* to study the behaviour of the system defined by *C* and *P* when faced with the selfishness scenario described in *S*. First, the *SEINE-L Compiler* generates the configuration file *C**, which extends *C* with instructions for injecting selfish behaviours into *P* as well as for monitoring the system performance. Second, *SEINE* calls the *Configuration Manager* included in the PeerSim library to build the experiment at run-time via reading *C** and instantiating the specified simulation components. These components are Java classes that implement (i) the nodes that compose the network, (ii) the set of protocols hosted by each node (including *P*), (iii) the observers that monitor or modify the behaviour of the simulated system, and (iv) the *Selfishness Scenario Generator* that injects the selfishness scenario into the simulation run-time. In particular, the Selfishness Scenario Generator uses Aspect-Oriented Programming techniques [8] and relies on a library of Java classes (*SSPeerSim*, in Figure 3) to interact with the PeerSim simulator. Finally, the *SEINE* framework presents the results of the simulation as a collection of statistics describing the behaviour of the simulated cooperative system for the given selfishness scenario.

V. Modelling selfishness in *SEINE-L*

SEINE-L provides a clear and concise description of the capabilities, interests and behaviours of different classes of nodes participating in the protocols of a cooperative system. The semantics of the DSL builds on the domain analysis presented in Section III, while its syntax is based on Java property files, i.e., collections of pairs associating a property name to a property value.

Figure 4 illustrates the outline of a *SEINE-L* program. The entry point is the keyword `seine` followed by a dot and the name of the selfishness scenario. Then, the DSL provides five top-level language constructs: *resource*s of interest, *indicator*s of the system state, *node* models, *selfishness* models, and *observers* to monitor the system behaviour. The declarations of the first four constructs have the format `keyword.[construct_name]`, whereas the observers are defined inside a block of statements in curly braces.

```
seine.[selfishness_scenario_name] {
    resource.[r1_name]
    ...
    indicator.[i1_name]
    ...
    node.[n1_name] { ... }
    ...
    selfishness.[s1_name] { ... }
    ...
    observers { ... }
}
```

Fig. 4: The outline of a *SEINE-L* specification.

We illustrate the usage of the *SEINE-L* constructs by describing in detail the specification of a selfishness scenario for the live streaming system presented in Section II and originally described by Guerraoui et al. [11]. Listing 1 shows the *SEINE-L* specification of this scenario, called *LSS*, which we refer to in the remainder of this section. In *LSS*, a node can be either mobile or desktop, depending on its device type (lines 6-11). The scenario shows that mobile nodes have severely constrained resources, and, particularly, their upload bandwidth capacity is half of the desktops' capacity (line 9). The selfish behaviours declared in the *LSS* example are based on those illustrated in Figure 1, which aim to reduce the

```
1  # Three-phase gossip-based live streaming
2  seine.LSS {
3    resource.bwCapacity uniform(1000) # kbps
4    indicator.batteryLeft

6    node.mobile {
7      fraction 0.3
8      selfish 0.5
9      capability bwCapacity(500)
10   }
11   node.desktop { selfish 0.1 }
12   node.exclude 0 # the source of the streaming

14   selfishness.smMobile {
15     actor mobile(0.8)
16     behaviour.bhvAggressive {
17       activation batteryLeft < 30
18       freeriding { degree 0.8 on send_SERVE }
19     }
20     behaviour.bhvWeak {
21       freeriding { degree 0.3 on send_SERVE }
22       misreport { degree 0.3 on send_PROPOSE }
23     }
24   }
25   selfishness.smColluders {
26     actor desktop mobile(0.2)
27     behaviour.bhvCollusive {
28       collusion.probability 0.15
29     }
30   }
31   observers {
32     period 100
33     name package.path.LSSObserver
34   }
35 }
```

Listing 1: A *SEINE-L* specification of a selfishness scenario for the live streaming system described in [11]. Keywords are shown in bold.

bandwidth dedicated to other nodes (lines 14-24 in Listing 1) or to non-colluders (lines 25-30).

1) Comments: Comments begin with # and continue to the end of the line, as illustrated in lines 1, 3, and 12.

2) Resources: The keyword `resource` introduces the declaration of a physical or logical resource. The declared resources must be associated with a value, which can function as an indicator of its current state. Line 3 of the *LSS* scenario declares the *bwCapacity* resource, which refers to the upload bandwidth capacity of nodes. A resource declaration can also provide instructions to initialize its values. In our example, all nodes are initialised with the same upload capacity (mode `uniform`) of 1000 kbps. *SEINE-L* allows other two initialisation modes: `random` and `linear` (i.e., a linearly increasing distribution of values within a specified range).

3) Indicators: An `indicator` declaration specifies a quantifiable attribute of either the system or a node type that depends upon its current state. For instance, the *batteryLeft* indicator in line 4 of Listing 1 can be used to guard the battery level of mobile nodes. Indicators cannot be initialised within a *SEINE-L* program.

4) Node model: Each node model is declared using the `node` keyword followed by a name and a block of properties delimited by curly braces. The *LSS* scenario declares mobile and desktop nodes, respectively, in lines 6-10 and line 11. The properties that can characterise a node model are listed below:

- `fraction` is the proportion of nodes in the system that hold this node model. If omitted, the fraction is set evenly by the preprocessor so that all node fractions sum up to 1. For instance, *desktop* nodes in the considered scenario will have the fraction set to 0.7.
- `selfish` is the fraction of selfish nodes within this node model. The default value is 1, i.e., all nodes holding this model are selfish.
- `capability` is a list of constraints over the values of the declared resources. Line 9 of the *LSS* scenario, for example, halves the upload bandwidth capacity of mobile peers, to 500 kbps. The DSL syntax prevents the definition of capabilities on resources that are not specified in the program.

There might be reasons to exclude a given set of nodes from the scope of the selfishness scenario, for instance because they represent special devices or trusted parties. In *SEINE-L*, this can be achieved using the `node.exclude` keywords followed by the identifiers (i.e., integers) of the nodes to exclude. As an example, line 12 of Listing 1 excludes the first node from the *LSS* scenario, because it represents the streaming source, which is assumed to be always cooperative.

5) Selfishness model: The `selfishness` declaration specifies the selfish behaviours adopted by a certain configuration of nodes. Such a configuration is expressed by the `actor` keyword followed by a list of terms, each defining the fraction of nodes of a given model to associate with the selfishness under specification. In Listing 1, the selfish behaviours of the *LSS* scenario described above are grouped into two selfishness declarations, namely, *smMobile* and *smColluders*. The actors of the *smMobile* model are defined in line 15 as 80% of the selfish population of the *mobile* nodes. In practice, given that 15% of nodes in the live streaming system were described in lines 7-8 as selfish mobile nodes (i.e., 50% of 30% of the overall population), the percentage of nodes that adopt the *smMobile* selfishness model is 12%. Notice that in the `actor` declaration in line 26 in Listing 1, the fraction of *desktop* nodes is not specified. In this case, the default value is 1, which means that all selfish *desktop* nodes are actors of the *smColluders* model.

Each selfish behaviour that constitutes a selfishness model is described by a `behaviour` declaration. A behaviour is a list of selfish deviations from the correct execution of a system protocol; such deviations are strategically interrelated and triggered by the same activation rule, which is introduced in *SEINE-L* by the `activation` keyword. An activation rule defines a condition (e.g., greater than or equal to) over the current value of a resource or indicator declared in the selfishness scenario. The *LSS* specification, for example, indicates in line 17 that every mobile node with a *smMobile* selfishness model switches to a more aggressive behaviour to reduce bandwidth consumption when it is running out of battery (i.e., the battery level drops below 30%). In contrast, if no activation rule is specified, then the selfish behaviour is always triggered. This is the case of the *bhvWeak* (lines 20-23) and *bhvCollusive* (lines 27-29) behaviours. Support for the specification of

logical expressions to combine multiple activation rules is left to future work.

A selfish behaviour specifies a non-empty set of deviations from the correct execution of certain steps (deviation points) of the system protocols. The *SEINE-L* syntax allows to declare five types of deviations, based on the classification developed from the domain analysis. Each deviation type is introduced by its own keyword, namely, `defection`, `freeriding` (`free-riding` is also accepted), `misreport`, `collusion`, and `other`, if none of the previous types applies. The execution of a deviation can be further characterised by the following additional properties of the deviation declaration:

- `probability` indicates the probability to deviate if the activation rule of the corresponding behaviour is met. The default value is 1.
- `on` constrains the possible deviation points of a deviation. For instance, in the free-riding declaration of the *bhvAggressive* behaviour (line 18), the `on` property ties the execution of this deviation to the deviation point named *send_SERVE* (see Figure 1(c)). Multiple deviation points can be listed as illustrated below, separated by whitespace.

  ```
  || on send_PROPOSE send_REQUEST send_SERVE
  ```

  SEINE-L also allows specifying the steps of a system protocol execution in which the deviation *cannot* take place, by preceding the name of a deviation point with an exclamation mark. For instance, the code fragment below specifies a free-riding deviation that affects all deviation points except the one named *send_SERVE*.

  ```
  || freeriding { on !send_SERVE }
  ```

- `degree` is a real value between 0 and 1 that specifies the intensity of free riding and misreport deviations (default value 1). In particular, the degree quantifies the reduction in the amount of resources contributed by a node in the case of a free-riding deviation, and the reduction in the reliability of the information provided in the case of a misreport deviation.

Different deviations of the same type may affect the same deviation points. For instance, in the *LSS* scenario, the *smMobile* model includes two behaviours that specify a free-riding deviation on the *send_SERVE* deviation point. Note that if the value of the *batteryLeft* indicator is below 30%, then both behaviours are active. These conflicts are resolved by triggering the first deviation in order of appearance in the program. The development of more sophisticated conflict resolution strategies is another area of future study.

To conclude, *SEINE-L* also allows a compact declaration for deviations with only one property, that is, the declaration used in line 28 of Listing 1.

6) Observers: The language constructs presented so far focus on the description of a selfishness scenario. In addition, *SEINE-L* provides a means to set-up monitoring components for assessing the performance of a cooperative system under that scenario. This can be done using the `observers` declaration. In practice, an observer is a Java object that collects statistics on the system performance during its simulation with PeerSim (see Section VI for more details). The `observers` declaration specifies the full class `name`s of each observer object to enable, as well as the `period` between two monitoring events in terms of simulated seconds (the default value is 100). For instance, the *LSS* scenario sets up the periodic execution of the *LSSObserver* object every 100 simulated seconds. This can also be specified using the compact form below.

```
|| observers.name package.path.LSSObserver
```

VI. Injecting selfishness in PeerSim using *SEINE*

The *SEINE* framework comprises the PeerSim simulator [27], a library of annotations for linking the *SEINE-L* specification of a selfishness scenario to the source code of PeerSim protocols, a compiler for *SEINE-L*, and a generator of simulation components for modelling, executing and monitoring a selfishness scenario in PeerSim. In the remainder of this section, we present each of the novel tools developed for *SEINE*.

A. Library of Annotations

Annotations are the means to link a selfishness scenario for a given system to the concrete implementation of that system, i.e., a set of PeerSim protocols. More precisely, the Designer can associate declarations of a *SEINE-L* specification to the affected program elements (e.g., classes, fields, methods) by decorating an element definition with annotations. This operation requires small and simple modifications of the original code. The *SEINE* framework provides eight types of annotation. `@Seine` decorates the declaration of each class implementing a PeerSim protocol to associate with the *SEINE-L* specification. In other words, it specifies which protocols are affected by the selfishness scenario. `@Resource` and `@Indicator` declare a field modelling a resource or indicator in *SEINE-L*, respectively.

The remaining annotation types allow indicating deviation points in PeerSim protocols. Concretely, a deviation point is the Java method that implements the part of the protocol behaviour in which one or more deviations may take place. These annotations are named after their deviation type (i.e., `@Defection`, `@Freeriding`, `@Misreport`, `@Collusion`, and `@OtherDeviation`) and target method declarations. As an example, let Listing 2 be a fragment of the PeerSim implementation of the live streaming system to experiment with the *LSS* selfishness scenario presented in Section V. The annotations in lines 4 and 8 indicate the default deviation points for any declarations of the corresponding deviation type provided in *LSS*. For instance, the `collusion` deviation in line 28 in Listing 1 is implicitly associated with any method that has been annotated with `@Collusion`, such as `send_PROPOSE` in Listing 2.

The annotations to indicate deviation points can have different attributes, depending on the deviation type that it represents. For instance, the `@Collusion` annotation in Listing 2 (line 4) specifies the attribute `ref_arg`, which indicates what argument of the *send_PROPOSE* method is the reference to

```
1  @Seine
2  public class LSS {
3    ...
4    @Misreport @Collusion(ref_arg = 1)
5    public void send_PROPOSE
6          (List cnksId, LiveStreaming req) { ... }
7    ...
8    @Freeriding
9    public void send_SERVE
10         (List cnks, LiveStreaming req) { ... }
11 }
```

Listing 2: A fragment of the PeerSim protocol implementing the system to associate with the *LSS* scenario in Listing 1.

the protocol instance that might be run by a potential colluder. This is the first argument by default. The same attribute is also supported by the `@Freeriding` and `@Misreport` annotation types, identifying the argument of the deviation point that may be affected by the deviation. For instance, the free riding annotation in line 8 of Listing 2 can modify the value of the list of chunks `cnks` to deliver to the requester `req`. For reasons of space, an exhaustive overview of all annotation attributes is beyond the scope of this paper.

B. SEINE-L Compiler

As shown in Figure 3, the *SEINE-L* compiler performs a source-to-source transformation of the *SEINE-L* specification (*S*, in the figure) into the PeerSim configuration file format.

The *SEINE-L* compiler performs statically various consistency checks on the selfishness scenario specification. Due to the declarative nature of the DSL, it is possible to verify the consistency of a specification with respect to the following properties: no omission (i.e., each referenced construct must be declared), no double declaration, correctness of the node model distribution (i.e., the proportions of the declared node models must sum to 1), and of the selfishness model distribution (i.e., for each node model, the proportions of selfish nodes adopting a selfishness model must sum to 1). If any of these properties is not fulfilled, then the compiler reports the error and stops.

In addition to the detection of errors in the *SEINE-L* specification, the *SEINE-L* compiler can also verify whether there are inconsistencies in the association between the DSL program and the annotated PeerSim protocol. More precisely, it verifies that (i) the protocol class is decorated with the `@Seine` annotation, (ii) for each resource and indicator in the specification there exists a class field with the same name that has been properly annotated, and (iii) for each deviation point explicitly defined in a deviation declaration (using the `on` property) there exists a method declaration with the same name that has been consistently annotated.

C. Selfishness scenario generation

The configuration file generated by the *SEINE-L* compiler enables the *Configuration Manager* of the PeerSim simulator to initialise the native simulation objects (e.g., nodes, protocols, monitors) as well as the selfishness scenario objects (e.g., resources, node models, deviations). Specifically, each language construct of the *SEINE-L* syntax is implemented in a Java class in the *SSPeerSim* Java library, which is included in the *SEINE* framework.

At run time, the Configuration Manager gives instructions to the *Selfishness Scenario Generator* to properly instantiate the classes in *SSPeerSim* so as to generate the objects that support the simulation of the selfishness scenario. Also, the Selfishness Scenario Generator uses Aspect-Oriented Programming (AOP) [8] techniques to modify the execution of the PeerSim protocol components that have been annotated by the Designer, in such a way as to inject selfish behaviours and node model capabilities. For coherence with the *SEINE* and PeerSim frameworks, both written in Java, we chose AspectJ [15] as the aspect-oriented language. In AspectJ, cross-cutting behaviours are described in class-like modules, called *aspects*. An aspect includes *advice* constructs for describing code to be inserted at given locations (*joinpoints*) of a standard Java program. Such locations are specified by *pointcut* constructs. Advice can insert the code *before* or *after* such locations, or it can replace existing code (*around* advice). The Selfishness Scenario Generator includes the aspects listed below.

- `SeineProtocolAspect` can extend PeerSim protocol classes by adding fields for storing selfishness-related information (e.g., the name of the node model implemented, the selfish behaviours that can be performed) as well as methods for behaving according to the selfishness scenario provided. The pointcut of this aspect intercepts all the classes decorated with the `@Seine` annotation.
- `ResourceIndicatorAspect` replaces getters/setters of the fields decorated with `@Resource` and `@Indicator` annotation types with a new implementation that (i) checks the fulfilment of each activation condition that may trigger a selfish behaviour, and, only for resources, (ii) constrains the values to the range specified by a capability condition.
- `SelfishInjectionAspect` specifies the advice that replaces the correct implementation of an annotated deviation point with that of a selfish deviation. More precisely, first it checks whether the deviation can take place, by verifying that the node executing the method can perform a deviation of that type and that the deviation is currently activated. If these conditions are verified, then the deviation implemented in the advice can be executed; otherwise, the execution proceeds according to the reference implementation.

The code snippet in Listing 3 illustrates the integration of deviation code by the `SelfishInjectionAspect` into the *send_SERVE* method. According to the *LSS* selfishness scenario presented in Listing 1, this method is a deviation point only for selfish mobile nodes that adopt the *smMobile* selfishness model, i.e., 12% of the overall system population (see Section V).

Another type of simulation component instantiated by the Selfishness Scenario Generator is the set of observers that monitor and gather statistics on the system performance. The

```
1  @Freeriding
2  public void send_SERVE( ... ) {
3    /** SelfishInjectionAspect advice */
4    boolean can_deviate = /* Verification */ ;
5    if(can_deviate) {
6      /* Execution of the deviation code */
7    }
8    /** End of SelfishInjectionAspect advice */
9
10   /* reference method implementation */ ...
11 }
```

Listing 3: A code fragment representing code injection into the *send_SERVE* method.

SEINE framework aids the Designer in creating application domain-specific observers, by providing in the *SSPeerSim* library an abstract class that defines the methods that need to be implemented. The execution of the Designer's observers is coordinated by a configurable controller that is automatically operated by the Selfishness Scenario Generator. The set-up of the controller is specified by the `observers` declaration of the *SEINE-L* program (see Section V).

D. SEINE Implementation

All tools and components in *SEINE* are written in Java. The entire implementation consists of almost 4000 lines of code, not including third-party components (i.e., the PeerSim simulator) and automatically generated code (i.e., the *SEINE-L* parser, built using ANTLR [31]). We developed the *SEINE* framework in a modular and loosely coupled manner, which promotes extensibility and reuse of its core components. For example, the interaction with the PeerSim simulator is implemented in a separate and independent module (the *SSPeerSim* library), which is less than 25% of the entire source code.

VII. Evaluation

In this section, we demonstrate the benefits of using *SEINE* to describe selfish behaviours and evaluate their impact on cooperative systems. We start by assessing the generality and expressiveness of the *SEINE-L* language by outlining some of our experiences in describing selfishness scenarios with our DSL. Then, we evaluate the accuracy of the *SEINE* output, developing and testing three use cases selected from our literature review, namely, a gossip-based live streaming protocol, a selfish-resilient media streaming protocol, and a selfish client for the BitTorrent protocol. Also, we assess the effort required by a Designer to implement and test the use cases. Finally, we show that *SEINE* imposes a small time overhead on the normal execution of the PeerSim simulator.

The *SEINE* framework is available, publicly and freely, at http://glenacota.github.io/seine/. To facilitate the reproducibility of our results, the configuration files related to the experiments reported in this section will also be available for download on the project website.

A. Generality and expressiveness of SEINE-L

We have used *SEINE-L* to express all of the selfishness scenarios described in the studies reviewed for the domain analysis (see Section III). Many of these works present various strategies to save bandwidth in data distribution applications, such as Gnutella [14], BitTorrent [22], and PPLive [32]. Other works investigate selfishness in different domains, like the paper of Kwok et al. [17] that studies Grid computing systems, and specifically the impact of task dispatching policies within a Grid site that allocates resources only to local tasks. Overall, the number and variety of the cooperative systems considered, as well as the different degrees of complexity of the selfishness scenarios therein described, demonstrate the general applicability and the expressive power of *SEINE-L*.

Table II shows that *SEINE-L* files are concise: the selfishness scenarios specified are between 14 and 38 Lines of Code (LoC), with an average of 25 LoC.

Reference	LoC	Reference	LoC
Ben Mokhtar et al. [3]	29	Sirivianos et al. [34]	24
Ben Mokhtar et al. [4]	34	Anderson et al. [1]	17
Kwok et al. [17]	24	Cox and Noble [6]	14
Guerraoui et al. [11]	25	Gramaglia et al. [10]	35
Hughes et al. [14]	19	Mei and Stefa [24]	28
Li et al. [20]	38	Ngan et al. [29]	30
Lian et al. [21]	17	Piatek et al. [32]	18
Locher et al. [22]	23		

TABLE II: Lines of Code for expressing the selfishness scenarios of the papers considered in the domain analysis review.

B. Accuracy of SEINE-R

To validate the accuracy of *SEINE*, we compared the results produced by our framework with those published in three use cases selected from the literature review. We discuss each use case separately below.

1) Live Streaming: We consider the gossip-based streaming system presented by Guerraoui et al. [11] and already described in Section V. Despite its simplicity, this system is realistic enough to serve as a representative example of a practical live streaming application. Guerraoui et al. deployed the system over PlanetLab,[2] in which a source node streams video chunks with a bit rate of 674kbps to 300 nodes having upload bandwidth limited to 1000kbps. They tested one scenario with only cooperative nodes and another scenario with a quarter of the nodes being selfish, performing the selfish behaviours described in Section II. To assess the system performance in both scenarios, Guerraoui et al. considered the fraction of cooperative nodes perceiving a clear stream (i.e., viewing at least 99% of the streamed chunks) when varying the playout deadline from 0 to 60 seconds.

We developed the gossip-based live streaming system as a PeerSim protocol and we used *SEINE* to describe and simulate the same selfishness scenario as well as the same experiment setting as investigated by Guerraoui et al. [11]. We ran ten simulations for each set of parameters, obtaining a fairly low standard deviation (0.02 on average), and we used the mean value to compare with the reference results. Figure 5 shows

[2]PlanetLab: https://www.planet-lab.org/

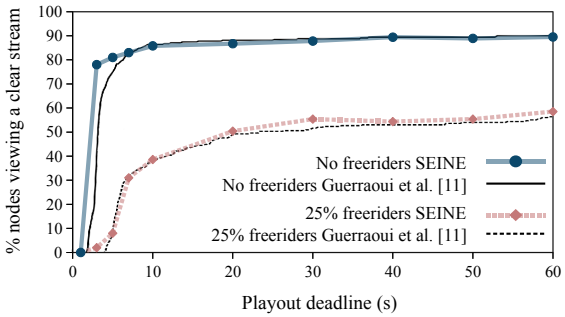


Fig. 5: Comparison between the results published by Guerraoui et al. [11] and the results obtained with *SEINE*.

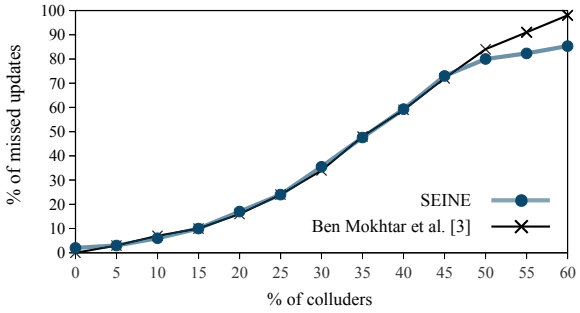


Fig. 6: Comparison between the results published by Ben Mokhtar et al. [3] and the results obtained with *SEINE*.

the high level of accuracy of *SEINE-R*, with an almost perfect match of the curves representing the scenario with selfish nodes and a satisfactory correspondence also in the selfish-free scenario.[3] Note that in the latter scenario our results show high accuracy when the playout deadline is above 7 seconds, whereas, below this time limit, *SEINE* simulations indicated better results. This is mainly due to the lower, real-world reliability of the PlanetLab network compared with the perfect but simulated reliability of the PeerSim network.

2) BAR Gossip: Proposed by Li et al. [20], BAR Gossip is a P2P live streaming system designed to tolerate both Byzantine and selfish peers. To this end, BAR Gossip includes mechanisms to enforce cooperation, namely, verifiable partner selection and data exchange mechanisms that make non-cooperative behaviours detectable and punishable. We select this use case to show that *SEINE* can also be used as a tool for testing performance and robustness of selfish-resilient protocols. For instance, BAR Gossip has been proven vulnerable to colluding nodes [3], which exchange video chunks only among each other, thereby decreasing the system efficiency and particularly the streaming experience of non-colluding nodes.

We assessed the accuracy of *SEINE* in reproducing the BAR Gossip selfishness scenario that was presented and experimentally studied by Ben Mokhtar et al. [3]. That study deployed 400 nodes in the Grid'5000 testbed,[4] each node running either a compliant version of BAR Gossip or a collusion-enabled implementation. We developed BAR Gossip [20] in PeerSim and set up its protocols using the configuration reported by Ben Mokhtar et al. [3]. Then, we simulated the system when varying the proportion of colluding nodes, from 0 to 50, and we measured the fraction of missed updates by cooperative nodes. Again, we ran ten simulations for each setting, obtaining an average standard deviation below 0.01. As clearly depicted in Figure 6, the accuracy of the results output by *SEINE-R* with respect to the ones provided by the authors of the reported study is very high (0.996 Pearson correlation score). The gap between the results when the proportion of colluders is above 50% is due to some missing parameters in the description of the experiment setting, especially the maximum of chunks that can be exchanged during the optimistic push protocol.

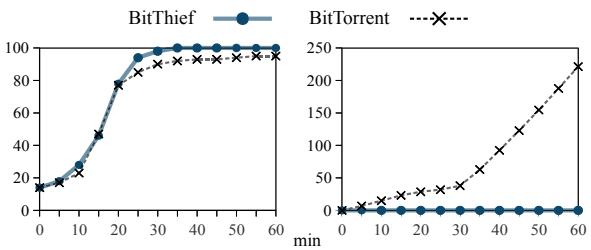
3) BitThief: Locher et al. [22] developed and released software to download files in the BitTorrent protocol without uploading any data. Specifically, they implemented and openly distributed a selfish client, *BitThief*, capable of attaining fast downloads without contributing. The purpose of this use case is twofold. First, it shows the accuracy of *SEINE* on empirical experiments performed on real-world applications. Second, it proves the simplicity of using *SEINE* when a PeerSim implementation of the system is already available. Specifically, we used the BitTorrent code published on the PeerSim project website.[5]

We used *SEINE* to reproduce the real world experiment described by Locher et al. [22], which consists in monitoring the download times for one torrent in a BitTorrent network with 5% BitThief clients. BitThief exploits several features of the BitTorrent protocol by means of a set of selfish deviations from the default client implementation. For example, a BitThief client can open up to 500 connections with other peers (the default value is 80), to increase its probability of receiving useful file pieces. In their experiment, Locher et al. showed that such deviations allow BitThief clients to download a file with performance comparable to (if not better than) the default clients, while not contributing any file pieces to other peers. The same results have been obtained in our experiment using *SEINE*, as can been observed in Figures 7(a)-(b).

C. Development effort

To show the benefits of using *SEINE* in terms of design and development complexity, we discuss the effort required to describe, implement and maintain selfish behaviours in our use cases. Using *SEINE*, the specification of the selfishness scenario to test is clearly separated from its actual integration into the cooperative system code. Such separation of concerns facilitates description and maintenance of node models and selfish behaviours, allowing the Designer to focus only on the *SEINE-L* program and on a few annotations of the system code. On the contrary, without using *SEINE*, the selfishness

[3]Our results are plotted over a copy of the figure published in [11].
[4]Grid'5000: https://www.grid5000.fr/
[5]http://peersim.sourceforge.net/code/bittorrent.tar.gz.



(a) % of clients that have downloaded the file

(b) File pieces uploaded by the clients

Fig. 7: Performance and contribution of BitTorrent and BitThief when downloading the same file, measured using *SEINE*.

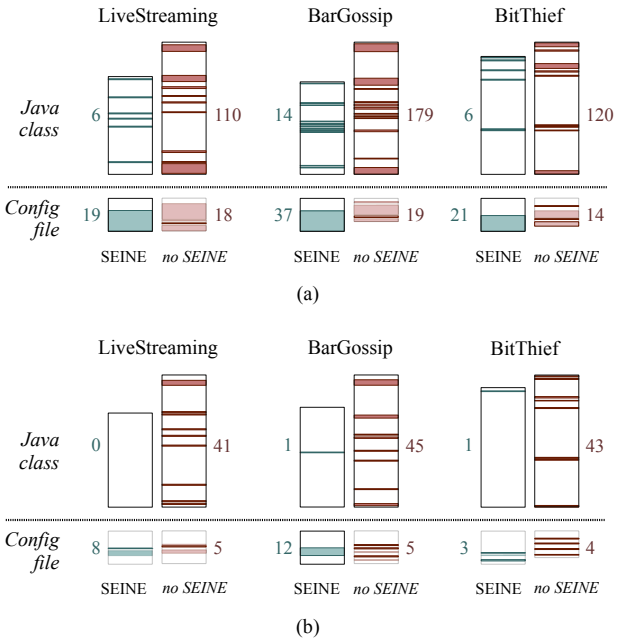


Fig. 8: Number and distribution of Lines of Code (a) to integrate the selfishness scenario into the faithful implementation of the use cases and (b) to modify such scenarios, with and without using *SEINE*.

scenario must be hard coded into the PeerSim Java programs and configuration files.

As can be noted from Figure 8(a), implementing a selfishness scenario using *SEINE* is not only easier but also extremely concise regarding Lines of Code. The bars in the figure provide a graphical representation of the volume and distribution of code to modify with respect to the faithful implementation of the cooperative system (the white part of the bar). The figure also reports the exact number of LoC to add for each use case. We can derive two observations from these results. First, implementing a selfishness scenario using *SEINE* requires writing four to five times less code than without using our framework. Second, when not using *SEINE*, the implementation (i.e., parameters, variables and methods) is scattered across the code of the PeerSim Java program and configuration file; on the other hand, the annotation library included in *SEINE-R* requires the Designer only to annotate existing fields and methods of the PeerSim program, and to write a *SEINE-L* program directly into a configuration file.

To illustrate the gain in flexibility and maintainability of testing selfishness scenarios using *SEINE*, we propose simple modifications to the scenarios of our use cases and we discuss the effort required to adapt the input files.

- *Live Streaming*: we duplicate a selfishness model specifying a different activation policy and deviation parameters (i.e., a higher degree of free riding and misreporting).
- *BAR Gossip*: we remove a selfishness model.
- *BitThief*: we remove a resource and we add a probability of execution to all deviations.

Figure 8(b) illustrates the number of Lines of Code to modify (i.e., add, remove, or edit) in each use case to implement the modifications listed above. Using *SEINE*, modifying the Java class requires modification of at most 1 line, which corresponds to inserting or dropping an annotation. Furthermore, when updating the configuration file, the Designer operates on coherent and consecutive blocks, such as the selfishness model block. In contrast, as can be observed in Figure 8(b), implementing the modifications to the selfishness scenarios when not using *SEINE* leads to more LoC to modify, which are scattered across the sources.

To demonstrate how *SEINE* facilitates fast development and testing of different selfishness scenarios, we present two test cases for the BAR Gossip cooperative system. In the first test case, we start from the selfishness scenario described in [3] and we evaluate the impact of the size and number of colluding groups. More precisely, we fix the fraction of selfish nodes in the system to 20%, and we evaluate the quality of stream perceived by selfish and cooperative nodes when varying from one big colluding group to 10 smaller groups of equal size. Results depicted in Figure 9(a) show that the percentage of updates missed by selfish nodes increases as they form colluding groups of smaller size. This is due to the lower probability for colluders to meet, given the random nature of the underlying gossip protocol for chunk dissemination, which cannot be cheated in BAR Gossip by design. On the contrary, the absence of significant changes in the performance for cooperative nodes indicates that they are not affected by how colluders organise themselves into groups. The system Designer can implement this test case using *SEINE* without modifying a single line of code in the PeerSim implementation of BAR Gossip, but only operating on the *SEINE-L* code. Specifically, the Designer first duplicates the selfishness model describing the collusive behaviour as many times as the number of colluding groups she wants to create. Then, the Designer modifies the fractions of the `actor` declarations.

As a second test case, we investigate the impact of mobile nodes with lower bandwidth capabilities on the performance of BAR Gossip. Similarly to the previous test case, this scenario modification does not change the system implementation but only the *SEINE-L* description of the selfishness scenario. In

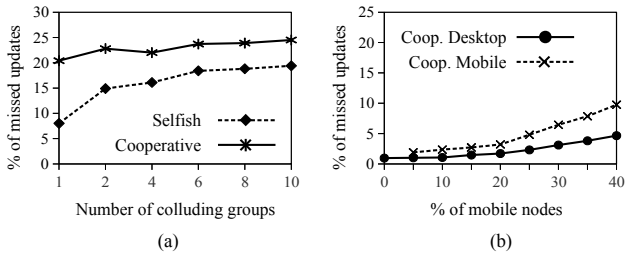


Fig. 9: Performance of BAR Gossip when varying (a) the number of colluding groups and (b) the fraction of resourceless mobile nodes.

particular, the Designer has to create a new node model block (i.e., *mobile*) which limits the bandwidth capacity with respect to the original node model (i.e., *desktop*). For example, the bandwidth capacity of desktop nodes is 1000 kbps, whereas it is capped to 300 kbps for mobile nodes. This modification to the scenario corresponds to adding 4 lines to the *SEINE-L* program and changing a few numeric values (e.g., refactoring the fraction of desktop nodes). Figure 9(b) reports the results of this test case, showing that the percentage of updates that are missed by cooperative nodes decreases as the fraction of mobile nodes increases. This result can be explained by the lower contribution that mobile nodes make to the chunk dissemination protocols, due to their limited resource capabilities.

To conclude, the test cases demonstrate how the development and testing of cooperative systems greatly benefited from the *SEINE* functionalities. Evaluating a new selfishness scenario in a complex system like BAR Gossip only took less than an hour, including the simulation time.

D. Simulation time

In this section, we evaluate the extra execution time imposed by *SEINE* on the regular PeerSim performance. To this end, we defined 10 different selfishness scenarios for each use case described in Section VII-B, and we ran 40 simulations for each scenario. The results, summarised in Table III, show that *SEINE* imposes an average extra execution time of 5% (standard deviation 0.03), ranging from the 1.6% extra time achieved by the BAR Gossip use case to the 7.8% extra time of BitThief. Such a short duration increase — 11 seconds out of 154 seconds, at most — appears to be reasonable in light of the benefits provided by the *SEINE* framework.

Use Case	**Execution time** (ms)	**Extra time** (ms)
Live Streaming	86,900	4,791
BAR Gossip	15,088	238
BitThief	154,604	11,176

TABLE III: Average execution time to evaluate a selfishness scenario using *SEINE* and the additional time it imposes.

VIII. Related Work

The ways selfishness has been evaluated in the literature of cooperative systems can be broadly divided into analytical and experimental approaches. Analytic analysis, especially game theory [28], provides mathematical tools to reason about selfishness and cooperation in competitive situations like those underlying a cooperative system [3], [4], [10], [20]. However, applying formal approaches to study real systems tends to be complex [23], [33]. Particularly, game theory approaches require manually creating a mathematical model of the system (the game), including the alternative strategies available to the system participants (the players) and their preferences over the possible outcomes of the system. Then, game-theoretic arguments have to be formulated to assess what strategy is the most likely to be played by the players. In addition to being complex and time-consuming, carrying out this process is also prone to modelling errors, due to assumptions and simplifications to make the model tractable [33]. Finally, and most relevant to our work, game theory can be helpful in understanding the decision-making process of system participants, but not to estimate the impact of their decisions on the system.

Adopting an experimental approach, like in *SEINE*, can be an appropriate solution to overcome the shortcomings of analytical modelling. Concretely, an experiment consists in implementing a selfish behaviour in a real or simulated instance of the system [12]. Testbeds such as Grid'5000 and PlanetLab provide the physical infrastructure to perform experiments with real distributed applications on real networks, in a configurable and monitorable manner. On the other hand, like many other authors [4], [10], [17], [20], [24], [29], we consider simulations a more practical tool for conducting comprehensive evaluation campaigns on large-scale systems such as cooperative systems. The use of simulations allows for a perfect control over the experimental conditions, high reproducibility, faster execution time, and the possibility of simulating millions of nodes on a single host [2]. These features come at the price of a high level of abstraction, which can introduce some bias in the experimental results [12]. Relying on a well-established and extensively tested simulator — e.g., PeerSim [27], used by *SEINE* — provides more certainty about the accuracy of the results.

Although a number of the existing frameworks suitable for the experimental evaluation of cooperative systems have the ability to script and inject events into the system, we find that almost none of them explicitly support the generation and assessment of selfish behaviours. Indeed, in most cases, the support for scripted events allows to specify system dynamics (e.g., churn management [19], [27]) or to inject simple fault events into specific system components [2], [12].

SEINE is also related to a significant body of work in the area of languages and tools for building and testing dependable distributed systems. In particular, practical frameworks such as Mace [16], Splay [19], and MOLStream [9] provide language support that enables developers to work on the different concerns that comprise a distributed system in isolation, thereby simplifying the overall process. Although performance and dependability concerns are also taken into account by these frameworks (e.g., fault handling support, performance and correctness analysis), there is no explicit guidance for addressing selfishness-related issues. To the best of our knowledge, the

only exception is RACOON [18], a framework for designing and configuring selfishness countermeasures for P2P systems. RACOON includes a custom built simulation environment that allows assessing the selfish-resilience of the designed system against a fixed set of three simple deviations. On the contrary, *SEINE* supports a considerably more expressive power and customisation of selfish behaviours, allowing it to cover most of the state-of-the-art selfishness manifestations, including the small subset supported by RACOON. Furthermore, *SEINE* supports the automatic injection of the specified behaviours into a PeerSim implementation of the system, while in RACOON such behaviours need to be manually implemented for a custom built simulator [18].

IX. Conclusion

In this paper, we presented *SEINE*, a semi-automatic framework for fast modelling and evaluation of selfish behaviours in cooperative distributed systems. At the heart of *SEINE* is the *selfishness scenario* model that we built through a systematic domain analysis on the subject. Based on this model, we developed an expressive domain-specific language (*SEINE-L*) and run-time support for the specification, implementation, and study of selfish behaviours in the state-of-the-art simulator PeerSim. We illustrated the generality of *SEINE-L* by showing that it can be used to describe the 15 selfishness scenarios extracted from the domain analysis. Then, we showed the accuracy and ease of use of *SEINE* in evaluating the impact of selfish behaviours in three use cases selected from the literature. The SEINE framework is freely available at http://glenacota.github.io/seine/.

Our future work includes the extension of the framework to offer new selfish deviations (e.g., computation or communication delays) and activation policies (e.g., game-theoretic strategies), as well as to support experiments on different simulators or real testbeds. We will also study how to design a convenient language workbench for *SEINE-L* to further ease the specification, reuse and evolution of selfishness scenarios.

References

[1] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of IEEE/ACM Int. Workshop on Grid Computing*, 2004.

[2] A. Basu, S. Fleming, J. Stanier, S. Naicken, I. Wakeman, and V. K. Gurbani. The state of Peer-to-Peer network simulators. *ACM Computing Surveys (CSUR)*, 45(4), 2013.

[3] S. Ben Mokhtar, J. Decouchant, and V. Quéma. AcTinG: Accurate freerider tracking in gossip. In *Proc. of SRDS*. IEEE, 2014.

[4] S. Ben Mokhtar, A. Pace, and V. Quéma. FireSpam: Spam resilient gossiping in the BAR model. In *Proc. of SRDS*. IEEE, 2010.

[5] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, 2003.

[6] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in Peer-to-Peer storage. In *Proc. of the ACM Symp. on Operating Systems Principles*, SOSP, 2003.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Int. Conf. on Software Product Lines*. Springer, 2004.

[8] R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented software development*. Addison-Wesley Professional, 2004.

[9] R. Friedman, A. Libov, and Y. Vigfusson. MOLStream: A modular rapid development and evaluation framework for live P2P streaming. In *Proc. of ICDCS*. IEEE, 2014.

[10] M. Gramaglia, M. Urueña, and I. Martinez-Yelmo. Off-line incentive mechanism for long-term P2P backup storage. *Computer Communications*, 2012.

[11] R. Guerraoui, K. Huguenin, A.-M. Kermarrec, M. Monod, and S. Prusty. Lifting: lightweight freerider-tracking in gossip. In *Proc. of the ACM/IFIP/USENIX Int. Conf. on Middleware*. Springer-Verlag, 2010.

[12] J. Gustedt, E. Jeannot, and M. Quinson. Experimental validation in large-scale systems: a survey of methodologies. *Parallel Processing Letters*, 2009.

[13] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems. In *Proc. of the ACM SIGOPS/EuroSys*, 2006.

[14] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: the bell tolls? *IEEE Distributed Systems Online*, 2005.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conf. on Object-Oriented Programming*. Springer, 2001.

[16] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[17] Y.-K. Kwok, K. Hwang, and S. Song. Selfish grids: Game-theoretic modeling and NAS/PSA benchmark evaluation. *IEEE TPDS*, 2007.

[18] G. Lena Cota, S. Ben Mokhtar, J. Lawall, G. Muller, G. Gabriele, E. Damiani, and L. Brunie. A framework for the design configuration of accountable selfish-resilient Peer-to-Peer systems. In *Proc. of SRDS*. IEEE, 2015.

[19] L. Leonini, É. Rivière, and P. Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proc. of NSDI*. USENIX Association, 2009.

[20] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *Proc. of OSDI*. USENIX Association, 2006.

[21] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li. An empirical study of collusion behavior in the Maze P2P file-sharing system. In *Proc. of ICDCS*. IEEE, 2007.

[22] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proc. of Workshop on Hot Topics in Networks*. Citeseer, 2006.

[23] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Experiences applying game theory to system design. In *Proc. of the ACM SIGCOMM workshop on Practice and Theory of Incentives in Networked Systems*, 2004.

[24] A. Mei and J. Stefa. Give2Get: Forwarding in social mobile wireless networks of selfish individuals. *IEEE TDSC*, 2012.

[25] H. Miranda and L. Rodrigues. Friends and foes: Preventing selfishness in open mobile ad hoc networks. In *Proc. of ICDCS Workshop on Mobile Distributed Computing*. IEEE, 2003.

[26] J. C. Mitchell and V. Teague. Autonomous nodes and distributed mechanisms. In *Software SecurityTheories and Systems*. Springer, 2003.

[27] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *IEEE Int. Conf. on Peer-to-Peer Computing*. IEEE, 2009.

[28] R. B. Myerson. *Game Theory*. Harvard university press, 2013.

[29] T.-W. Ngan, R. Dingledine, and D. S. Wallach. Building incentives into Tor. In *Int. Conf. on Financial Cryptography and Data Security*. Springer, 2010.

[30] J. F. Oliveira, Í. Cunha, E. C. Miguel, M. V. Rocha, A. B. Vieira, and S. V. Campos. Can Peer-to-Peer live streaming systems coexist with free riders? In *Proc. of IEEE P2P*, 2013.

[31] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[32] M. Piatek, A. Krishnamurthy, A. Venkataramani, Y. R. Yang, D. Zhang, and A. Jaffe. Contracts: Practical contribution incentives for P2P live streaming. In *Proc. of NSDI*. USENIX Association, 2010.

[33] R. Rahman, T. Vinkó, D. Hales, J. Pouwelse, and H. Sips. Design space analysis for modeling incentives in distributed systems. In *ACM SIGCOMM Conference*, 2011.

[34] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *USENIX Annual Technical Conference*, 2007.

[35] Y. Yoo and D. P. Agrawal. Why does it pay to be selfish in a MANET? *IEEE Wireless Communications*, 2006.