Self-adaptive systems must decide which adaptations to apply and when. In reactive approaches, adaptations are chosen and executed after some issue in the system has been detected (e.g., unforeseen attacks or failures). In proactive approaches, predictions are used to prepare the system for some future event (e.g., traffic spikes during holidays). In both cases, the choice of adaptation is based on the estimated impact it will have on the system. Current decision-making approaches assume that the impact will be instantaneous, whereas it is common that adaptations take time to produce their impact. Ignoring this latency is problematic because adaptations may not achieve their effect in time for a predicted event, or lower-impact but quicker adaptations may be ignored altogether when over time, the accrued impact may actually be higher. In this paper we introduce a novel approach to choosing adaptations that considers these latencies. To show that this is an improvement, we use a novel combination of two components: (i) model checking of stochastic multiplayer games (SMGs) enables us to understand best- and worst-case scenarios of latency-aware and non-latency-aware adaptation without the need to develop specific adaptation algorithms. However, since SMGs do not provide an algorithm to make choices at run time, we propose a (ii) latency-aware adaptation algorithm to make decisions at runtime. Simulations are used to explore more detailed adaptation behavior, and to check if the performance of the algorithm falls within the bounds predicted by SMGs. Our results show that latency awareness improves adaptation outcomes, and also allows a larger set of adaptations to be exploited.
1. INTRODUCTION

In order to maintain system goals during execution, self-adaptation is increasingly employed to monitor run-time conditions, analyze whether goals are being met or could be met better, choose (or plan) how to adapt the system, and then finally execute the chosen adaptation. Self-adaptive systems therefore can often be considered as adding closed-loop control, where the self-adaptive elements are the control, and the system being adapted is the plant. The activities of monitoring, analyzing, planning, and executing were first proposed in [Kephart and Chess 2003] as the MAPE loop and is a common pattern used to construct self-adaptive systems.

Current self-adaptive approaches following the MAPE pattern consider increasingly sophisticated approaches to choosing which adaptation to perform and when. The decision of which adaptation to perform focuses on trying to determine the adaptation that is expected to have the highest impact on the qualities of the resulting systems, such as performance, operating cost, and reliability [Garlan et al. 2004; Zhang and Lung 2010], or safety and liveness [Braberman et al. 2013; Goldman et al. 2003]. As to choosing when to perform an adaptation, a self-adaptive system can be reactive, meaning that it chooses adaptations to correct problems that it sees with a system, or proactive, meaning that it uses predictions to try to prevent or head off problems that might occur in the near future.

In all these cases, the time an adaptation takes to achieve its effect is ignored in the decision-making process—in fact, it is often assumed that the effect will be instantaneous. However, different adaptation tactics take different amounts of time until these effects are observed. For example, consider two tactics to deal with an increase in the load of a system: reducing the fidelity of the results (e.g., less resolution, fewer elements, etc.), and adding a computer to share the load. Adapting the system to produce results with less fidelity may be achieved quickly if it can be done by changing a simple setting in a component, whereas powering up an additional computer to share the load may take some time.

While including tactic latency in choosing which adaptation to perform should lead to an improvement, two questions need to be answered: (a) how much improvement can be achieved by optimally leveraging this additional information; and (b) what algorithms can be used to make the choice in light of latency information.

In this paper, we answer these questions with respect to using tactic latency information for proactive self-adaptation. The contributions of this paper are:

(1) A novel technique that enables us to understand the potential improvement of employing a particular type of adaptation without the need of developing a specific algorithm or implementing the infrastructure required to analyze adaptation behavior. In particular, we propose a technique that is based on model checking of stochastic multiplayer games (SMGs), which allows us to analyze best- and worst-case scenarios with little specification effort. In this paper, we exploit this technique to quantify the maximum improvement that an optimal latency-aware strategy is able to obtain with respect to a baseline optimal strategy that assumes no tactic latencies. Although we can obtain boundary cases using this technique, this component of our approach does not provide an algorithm to make latency-aware choices at run time.

(2) A novel algorithm extending a prior one that computes the optimal sequence of adaptation decisions for anticipatory dynamic configuration [Poladian et al. 2007] by considering the effects of latency in adaptation. This algorithm can be used to

---

1 Some approaches explicitly consider time when executing the chosen adaptations (e.g., [Cheng and Garlan 2012]), but these are not used in choosing the adaptations.
effectively exploit tactic latency information at run time. In order to check whether the improvement obtained by our algorithm is consistent with the boundaries predicted by SMGs, we employ simulations to explore adaptation behavior in more detail and analyze average cases of adaptation performance.

In [Cámara et al. 2014] we reported on an initial exploration of these topics that dealt with only one tactic. In this paper, we extend these results by: (i) considering multiple tactics with different latencies, enabling us to compare not only the performance of latency-aware vs non-latency-aware adaptation, but also the impact of considering latency information on tactic choices (e.g., by showing that some tactics might never get chosen, despite being able to improve the outcome of adaptation with respect to others); and (ii) generalizing our adaptation algorithm to support multiple tactics, determining adaptation feasibility via formal analysis of tactics using Alloy.

Our formal verification results show that factoring in tactic latency in decision making improves the outcome of adaptation both in worst- and best-case scenarios. Moreover, results indicate that while non-latency-aware algorithms can prevent the selection of available tactics that could help improve the outcome of adaptation, latency-aware algorithms are able to better exploit adaptation tactic repertoires. This is consistent with the results obtained for our latency-aware proactive adaptation algorithm, showing that it is able to obtain higher utility than Poladian et al.’s algorithm, which is optimal under the assumption of no tactic latency, and that it can better exploit the availability of tactics with different latencies.

The remainder of this paper is structured as follows: Section 2 introduces some background, related work, and summarizes Znn.com, the example used to illustrate our approach. Section 3 describes our technique for analyzing adaptation based on model checking of stochastic games. Next, section 4 presents our algorithm for latency-aware proactive adaptation. Finally, section 5 concludes the paper and indicates future research directions.

2. BACKGROUND AND RELATED WORK

This section first presents related work in proactive self-adaptation. Next, we overview the adaptation model that we assume in this paper. Finally, we describe Znn.com, an example that we use to illustrate our approach.

2.1. Related Work

Poladian et al. demonstrated that when there is an adaptation cost or penalty, proactive adaptation outperforms reactive adaptation [Poladian et al. 2007]. Intuitively, if there is no cost associated with adaptation, a reactive approach could adapt at the time a condition requiring adaptation is detected without any negative consequence. In their work, Poladian et al. presented two algorithms for proactive adaptation that considered the penalty of adaptation when deciding how to adapt. One of the algorithms assumed perfect predictions of the environment, while the other handled uncertainty. The latter was used to improve self-adaptation in Rainbow [Cheng et al. 2009b], where Cheng et al. considered tactic latency only to skip the adaptation if the condition that triggered it was predicted to go away by itself before the adaptation tactic completed. However, the approach did not consider all the effects that arise due to tactic latency (see Section 4).

Proactive adaptation has received considerable attention in the area of service-based systems [Calinescu et al. 2011; Hielscher et al. 2008; Metzger et al. 2013; Wang and Pazat 2012] because of their reliance on third-party services whose quality of service (QoS) can change over time. In that setting, when a service failure or a QoS degradation is detected, a penalty has already been incurred, for example, due to service-
level agreement (SLA) violations. Thus, proactive adaptation is needed to avoid such problems. Hielscher et al. proposed a framework for proactive self-adaptation that uses online testing to detect problems before they happen in real transactions, and to trigger adaptation when tests fail [Hielscher et al. 2008]. Wang and Pazat use online prediction of QoS degradations to trigger preventive adaptations before SLAs are violated [Wang and Pazat 2012]. These approaches ignore the adaptation latency.

Musliner considers adaptation time by imposing a limit on the time to synthesize a controller for real-time autonomous systems [Musliner 2001]. However, in that work there are not distinct planning and execution phases, and thus there is no consideration of the latency of the different actions the system could take to adapt. In the area of dynamic capacity management for data centers, the work of Gandhi et al. considers the setup time of servers, and is able to deal with unpredictable changes in load by being conservative about removing servers when the load goes down [Gandhi et al. 2012]. Their work is specifically tailored to adding and removing servers to a dynamic pool, a setting that resembles the running example we use in this paper. However, their work assumes the environment is unpredictable, and, consequently, does not consider the possibility of being able to predict a reduction or transient spikes in load. Our approach, on the other hand, can exploit predictions of such events and either adapt as soon as possible when removing servers, or avoid adaptations completely.

2.2. Example

Before outlining our approach, we introduce a simple example that will be used throughout the rest of the paper to illustrate and explain our approach.

Znn.com is a case study portraying a representative scenario for the application of self-adaptation in software systems which has been extensively used to assess different research advances in self-adaptive systems [Cheng et al. 2009a]. Znn.com embodies a typical infrastructure for a news website, and has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via frontend presentation logic (Figure 1). The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

![Fig. 1: Znn.com system architecture](image-url)
between these different adaptations, a self-adaptive system must balance three quality objectives: (i) performance, which depends on request response time, server load, and network bandwidth, (ii) cost, which is associated with the number of active servers, and (iii) fidelity, which maps to the fidelity level of the contents being served.

2.3. Adaptation Model

Although there are many approaches that rely on a closed-loop control approach to self-adaptation, including those that exploit architectural models for reasoning about the target system under management [Garlan et al. 2004; Kramer and Magee 2007; Oreizy et al. 1999], in this paper we use some of the high-level concepts in Rainbow [Garlan et al. 2004] as a reference framework to illustrate our approach. Rainbow is an architecture-based platform for self-adaptation, which has among its distinct features an explicit architecture model of the target system, a collection of adaptation tactics, and utility preferences to guide adaptation choice.

We assume a model of adaptation that represents adaptation knowledge using the following high-level concepts:

— **Tactic**: a primitive action that corresponds to a single step of adaptation, and has an associated: (i) cost/benefit impact on the different quality dimensions, and (ii) latency, which corresponds to the time it takes since a tactic is started until its effect is observed. For instance, in Znn.com we can specify pairs of tactics with opposing effects for enlisting/discharging servers, or increasing/reducing the fidelity of the contents being served. We assume a sequential execution model for tactics consistent with the semantics of Stitch, hence no tactic execution can be triggered during the latency period of another tactic.

— **Utility Profile**: To enable the selection of tactics at run-time, we assume that adaptation is driven by utility functions and preferences, which are sensitive to the context of use and able to consider trade-offs among multiple potentially conflicting objectives. The different qualities of concern are characterized as utility functions that map them to architectural properties. We assume that utility functions are defined by an explicit set of value pairs (with intermediate points linearly interpolated). Table I summarizes the utility functions for Znn.com. Function $U_R$ maps low response times (up to 100ms) with maximum utility, whereas values above 2000ms are highly penalized (utility below 0.25), and response times above 4000ms provide no utility. Function $U_F$ maps a low (1) level of content fidelity (e.g., textual version of contents) to a utility 0.25, whereas a high level (2) of content fidelity (e.g., including images/video) is mapped to maximum utility. Function $U_C$ maps increasing cost (derived from the number of active servers) to lower utility values. Utility preferences capture business preferences over the quality dimensions, assigning a specific weight ($w_{U_R}, w_{U_F}, w_{U_C}$) to each one of them. In the context of Znn.com, preference is typically given to performance over cost and fidelity. In fact, the weighted sum is overridden when $U_R = 0$ making the overall utility zero.

By evaluating how different tactic execution sequences might affect the different qualities of concern using a utility profile, a proactive adaptation algorithm can build a strategy with the objective of maximizing accrued utility during system execution.

---

2We use a simplified version of the concepts in Stitch [Cheng and Garlan 2012] (the language used to express adaptation tactics in Rainbow) to illustrate the main ideas in this paper.

3Stitch incorporates a different notion of timing delay to monitor the outcome of tactic executions in reactive adaptation strategies, which is not discussed in this paper.
Table I: Utility functions and preferences for Znn.com

<table>
<thead>
<tr>
<th>UR</th>
<th>UF</th>
<th>UC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
<td>500 : 0.90</td>
</tr>
<tr>
<td>100</td>
<td>1.00</td>
<td>1000 : 0.75</td>
</tr>
<tr>
<td>200</td>
<td>0.99</td>
<td>1500 : 0.50</td>
</tr>
</tbody>
</table>

3. ANALYZING ADAPTATION

This section describes our approach to analyzing self-adaptation via model checking of SMGs, which enables us to understand a priori the behavioral envelope of different types of adaptation using formal models that require little specification effort, instead of developing adaptation algorithms or costly self-adaptive infrastructure.

In the remainder of this section, we first provide an overview of model checking SMGs and describe how the technique is used in our approach. Next, we present a SMG model of Znn.com that enables the comparison of latency-aware against non-latency-aware adaptation. Finally, we describe how these models can be analyzed and show some results for different instances of the model.

3.1. Model Checking Stochastic Multiplayer Games

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains that range from power management or wireless communication protocols, to biological systems. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic behavior, effectively enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system’s use of resources, time, etc.).

Competitive behavior may also appear in systems when some component cannot be controlled, and could behave according to different or even conflicting goals with respect to other components in the system. In such situations, a natural fit is to adopt a game-theoretic perspective by modeling a system as a game between different players.

3.1.1. SMGs, strategies, and rPATL properties. Our approach to analyzing adaptation builds upon a recent technique for modeling and analyzing stochastic multi-player games (SMGs) extended with rewards [Chen et al. 2013a]. In this approach, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic.

**Definition 3.1 (SMG).** A turn-based stochastic multi-player game augmented with rewards (SMG) is a tuple \( \mathcal{G} = (\Pi, S, A, (S_i)_{i \in \Pi}, \Delta, AP, \chi, r) \), where \( \Pi \) is a finite set of players; \( S \neq \emptyset \) is a finite set of states; \( A \neq \emptyset \) is a finite set of actions; \( (S_i)_{i \in \Pi} \) is a partition of \( S \); \( \Delta : S \times A \rightarrow D(S) \) is a (partial) transition function; \( AP \) is a finite set of atomic propositions; \( \chi : S \rightarrow 2^{AP} \) is a labeling function; and \( r : S \rightarrow \mathbb{Q}_{\geq 0} \) is a reward structure mapping each state to a non-negative rational reward. \( D(X) \) denotes the set of discrete probability distributions over finite set \( X \).

In each state \( s \in S \) of the SMG, the set of available actions is denoted by \( A(s) = \{ a \in A \mid \Delta(s, a) \neq \bot \} \). We assume that \( A(s) \neq \emptyset \) for all states \( s \) in the model. Moreover, the choice of which action to take in every state \( s \) is under the control of a single player \( i \in \Pi \), for which \( s \in S_i \). Once action \( a \in A(s) \) is selected by a player, the successor state is chosen according to probability distribution \( \Delta(s, a) \).

**Definition 3.2 (Path).** A path of SMG \( \mathcal{G} \) is an (in)finite sequence \( \lambda = s_0a_0s_1a_1\ldots \) s.t. \( \forall j \in \mathbb{N} \bullet a_j \in A(s_j) \land \Delta(s_j, a_j)(s_{j+1}) > 0 \). The sets of all finite paths in \( \mathcal{G} \) is denoted as \( \Omega^+_f \).
Players in the game can follow strategies for choosing actions in the game, cooperating with each other in coalition to achieve a common goal, or competing to achieve their own (potentially conflicting) goals.

**Definition 3.3 (Strategy).** A strategy for player \( i \in I \) in \( G \) is a function \( \sigma_i : (SA)^* S_1 \to \mathcal{D}(A) \) which, for each path \( \lambda \cdot s \in \Omega_\phi^+ \) where \( s \in S_1 \), selects a probability distribution \( \sigma_i(\lambda \cdot s) \) over \( A(s) \).

In the context of this paper, we always refer to player strategies \( \sigma_i \) that are **memoryless** (i.e., \( \sigma_i(\lambda \cdot s) = \sigma_i(\lambda' \cdot s) \) for all paths \( \lambda \cdot s, \lambda' \cdot s \in \Omega_\phi^+ \)), and **deterministic** (i.e., \( \sigma_i(\lambda \cdot s) \) is a Dirac distribution for all \( \lambda \cdot s \in \Omega_\phi^+ \)). Memoryless, deterministic strategies resolve the choices in each state \( s \in S_i \) for player \( i \in I \), selecting actions based solely on information about the current game state. These strategies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures used in our models.\(^4\)

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a property in a logic called rPATL. Properties written in rPATL can state information about the current game state. These strategies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures used in our models.\(^4\)

Analyzing Latency-aware Self-adaptation using Stochastic Games and Simulations

---

\(^4\)See Appendix A.2 in [Chen et al. 2013a] for details.

\(^5\)The variants of \( F^* \phi \) used for reward measurement in which the parameter \( * \in \{0, \infty, c\} \) indicate that, when \( \phi \) is not reached, the reward is zero, infinite or equal to the cumulated reward along the whole path, respectively.
mize different rewards). These properties follow the general pattern $\langle\langle P \rangle\rangle R_{\max} U_{\max} = [F^{\omega}]$, where $P$ is a set of players that can include the system and/or the environment, $U$ is a reward that encodes the instantaneous utility of the system, and $\omega$ is a state formula that encodes a stop condition for the system’s execution. Section 3.3 details how such properties are used in our approach.

### 3.2. SMG Model

Our formal model is implemented in PRISM-games [Chen et al. 2013b], an extension of the probabilistic model-checker PRISM [Kwiatkowska et al. 2011] for modeling and analyzing SMGs. Our game is played in turns by two players that are in control of the behavior of the environment and the system, respectively. The SMG model consists of the following parts:

#### 3.2.1. Player definition

Listing 1 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the environment and clk modules), whereas player sys controls all transitions that belong to the target system module. Global variable turn in line 4 is used to make players alternate, ensuring that for every state of the model, only one player can take action. Turn-based gameplay suffices to naturally model the interplay between the environment and the system, which only senses environment information and reacts to it if necessary at discrete time points.

```
player sys target_system, [enlist], [enlist_trigger], [discharge], [if], [df] endplayer
2 const ENV_TURN=1, SYS_TURN=2, CLK_TURN=3;
3 global turn:[ENV_TURN..CLK_TURN] init ENV_TURN;
```

Listing 1: Player definition for Znn.com’s SMG

#### 3.2.2. Environment

The environment (Listing 2) is in control of the evolution of time and other variables of the execution context that are out of the system’s control (e.g., service requests arriving at the system). There are two modules controlled by the environment player: (i) clk, which tracks and controls execution time (lines 5-11), and (ii) environment (lines 13-19), in charge of controlling request arrivals to the system. Note that the choices in the environment module are specified non-deterministically to obtain a representative specification of the environment (through strategy synthesis) that is not limited to specific behaviors, since this would limit the generality of our analysis. Lines 1-3 in Listing 2 define different constants that parameterize the behavior of the environment:

- MAX_TIME defines the time frame for the system’s execution ([0,MATX_TIME]).
- TAU sets time granularity, defining the frequency with which the environment updates the value of non-controllable variables, and the system responds to these changes. The total number of turns for both players is MAX_TIME/TAU. Note that two consecutive turns of the same player are separated by a period of duration TAU.
- MAX_ARRIVALS constrains the maximum total number of requests that can arrive at the system for processing during its execution. Unconstrained arrivals result in a behavior of the environment that continuously floods the system with requests.

---

6Actions enlist_trigger, enlist, discharge, if, and df are explicitly labeled to improve readability (see Listing 3), but are still asynchronous in our model.

7Constant values not defined in the model are provided as command-line input parameters to the tool.
However, to facilitate the representation of a more realistic behavior of the environment, the total number of requests that can be placed during system execution can be limited by setting the value of this constant.

— MAX_INST_ARRIVALS is the maximum number of arrivals that the environment can place for the system to process during its turn (i.e., during one TAU time period).

Listing 2: Environment module

Moreover, two different sets of variables define the state of the environment:

— In the clk module (lines 6-8):
  — t keeps track of execution time.
  — rt is the system’s response time. Note that we choose to represent this variable as part of the environment, since the system does not have control over its value.
  — rt_upd is an auxiliary variable used to keep track of whether rt has been updated during the current turn.

— In the environment module (lines 14-15):
  — arrivals_total keeps track of the accumulated number of arrivals during execution.
  — arrivals_current is the number of request arrivals during the current time period.

Each turn of the environment consists of two steps:

(1) The clk module updates the values of the response time and time variables (lines 9-10): (i) Response time is updated according to the request arrivals during the current time period and the number of active servers (computed using of an M/M/c queuing model [Chiulli 1999], encoded by formula totalTime – line 9). (ii) Execution time variable t is increased one step (TAU) (line 10). After variable updates, clk yields control to environment module by updating the turn variable.

(2) The environment module sets the amount of request arrivals for the current time period. This is achieved through a set of commands that follow the pattern shown in Listing 2, line 18: the guard in the command checks that (i) it is the turn of the environment, and (ii) the end of the time frame for execution has not been reached. If the guard is satisfied, the command sets the value of request arrivals for the current time period (represented by X in the command), and adds the number of request arrivals for the current time period to the accumulator arrivals_total.
These updates are carried out only if maximum number of requests to be generated by the environment has not been placed yet in the system (conditional updates are guarded by $\text{arrivals}_{\text{total}} - X = 0$). In addition, the turn of the environment player finishes when this command is executed, since it modifies the value of variable `turn`, yielding the control of the game to the system player. Note that there may be as many of these commands as different possible values can be assigned to the number of request arrivals for the current time period (including zero for no arrivals). Probabilities in these commands are left unspecified, since it will be up to the strategy followed by the player (to be synthesized based on an rPATL specification) to provide the discrete probability distribution for this set of commands.

The order in the execution of commands between the clk and environment modules is the inverted for the initial turn of the environment, since the arrivals for the initial period have to be set before computing the initial response time according to the queuing model. This is achieved by a command included in the environment that yields control to clk, instead of the system for the initial time period (line 16).

### 3.2.3. System

Module target system (Listing 3) models the behavior of the target system (including the execution of tactics upon it), and is parameterized by the constants:

- `MIN_SERVERS` and `MAX_SERVERS`, which specify the minimum and maximum number of active servers that a valid system configuration can have.
- `INIT_SERVERS` is the number of active servers in the system’s initial configuration.
- `MIN_FIDELITY` and `MAX_FIDELITY`, which specify the minimum and maximum fidelity levels for served content.
- `INIT_FIDELITY` is the fidelity level of served content in the initial configuration.
- `ENLIST_LATENCY` is the latency of the tactic for enlisting a server, measured in number of time periods (i.e., the real latency for the tactic in time units is $\text{TAU} \times \text{ENLIST_LATENCY}$). In our model, tactic latencies are always limited to multiples of the time period duration.
- `MAX_RT` and `INIT_RT`, which specify the system’s maximum and initial response times, respectively.

```plaintext
const MIN_SERVERS, MAX_SERVERS, INIT_SERVERS;
const MIN_FIDELITY, MAX_FIDELITY, INIT_FIDELITY;
const ENLIST_LATENCY;
const MAX_RT, INIT_RT;
module target_system
  s : [0..MAX_SERVERS] init INIT_SERVERS;
  f : [1..MAX_FIDELITY] init MAX_FIDELITY;
  \[ (s < MAX_SERVERS) \& (turn=SYS_TURN) \& (counter=0) \rightarrow (s'=s+1) \& (turn'=CLK_TURN) \& (counter'=-1); 
  (s < MAX_SERVERS) \& (turn=SYS_TURN) \& (counter=-1) \& (ENLIST_LATENCY>0) \rightarrow (s'=s+1) \& (counter'=ENLIST_LATENCY-1) \& (turn'=CLK_TURN) \& (ENLIST_LATENCY=0)?s+1:s); 
  \[enlist\] (s < MAX_SERVERS) \& (turn=SYS_TURN) \& (counter=0) \rightarrow (s'=s+1) \& (turn'=CLK_TURN) \& (counter'=-1); 
  \[discharge\] (turn=SYS_TURN) \& (f>MIN_FIDELITY) \& (counter=0) \rightarrow (f'=f-1); 
endmodule
```

Listing 3: System module
Moreover, the module includes variables which are relevant to represent the current state of the system:

- \( s \) corresponds to the number of active servers.
- \( f \) is the fidelity level of the contents being served.
- \( \text{counter} \) is used to control the delay between the triggering of a tactic and the instant in which it becomes effective. In this case, the variable is used to control the delay between the activation of a server, and the time instant in which it ends booting up.

During its turn, the system can decide not to execute any tactics, returning the turn to the environment player by executing the command defined in line 10, Listing 3. Alternatively, the system can execute one of these tactics:

- **Activation of a server**, which is carried out in two steps:
  1. Triggering of activation through the execution of the command labeled as enlist_trigger (line 11). This command only executes if the current number of active servers is less than the maximum allowed, and the counter that controls tactic latency is inactive (meaning that there is not currently a server already booting in the system). This command activates the counter by setting it to the value of the latency for the tactic, and yields turn to the environment player.
  2. Effective activation through the enlist command (line 14), which executes when the counter that controls tactic latency reaches zero, incrementing the number of servers in the system, and deactivating the counter.

Note that the special case in which the latency of the enlist server tactic is zero, the execution of the tactic is carried out in a single step (line 12).

- **Deactivation of a server**, which is achieved through the discharge command (line 15), which decrements the number of active servers. The command fires only if the current number of active servers is greater than the minimum allowed and the counter for server activation is not active.

- **Lowering the fidelity of all active servers**, setting them to textual mode through the execution of the command df (line 15). This tactic decreases the value of the fidelity variable \( f \), and thus increases the service rate, which in turn causes a reduction in the system's response time.

- **Raising the fidelity of all active servers**, setting them to multimedia mode through the execution of command if (line 16), which has the opposite effect of decrease(df).

Note that the latency of all tactics, except for the one to enlist servers, is zero.

### 3.2.4. Utility profile

Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of instantaneous utility. Specifically, formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences.

Listing 4 illustrates in lines 1-5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function \( U_r \) in the first column of Table I. The formula in the example computes the utility for performance, based on the value of the variable for system response time \( rt \). Moreover, lines 8-10 show how a reward structure can be defined to compute a single utility value for any state by using utility preferences (in this case defined as weights \( W_{UR} \), \( W_{UC} \), and \( W_{UF} \)). Specifically, each state in which it is the turn of the system player is assigned with a reward corresponding to the entire elapsed time period of duration \( TAU \), during which we assume that instantaneous utility does not change.

In latency-aware adaptation, the instantaneous real utility extracted from the system coincides with the utility expected by the algorithm's computations during the
tactic latency period. However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance (i.e., the server is booting up, but not processing requests yet). To enable analysis of real vs. expected utility in non-latency-aware adaptation, we add to the model a new reward structure that encodes expected instantaneous utility $rEIU$ (Listing 5). In this case, the utility for performance during the latency period (encoded in formula $uER$) is computed analogously to $uR$ in Listing 4, but based on the response time that the system would have with $s+1$ servers during the latency period.

3.3. Analysis

In order to compare latency-aware vs. non-latency-aware adaptation, we make use of rPATL specifications that enable us to analyze (i) the maximum utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario), and (ii) the maximum utility that adaptation is able to obtain under ideal environmental conditions (best-case scenario).

3.3.1. Latency-aware Adaptation. In this case, the real adaptation extracted from the system coincides with the utility that adaptation uses for decision making.

— Worst-case scenario analysis. We define the real guaranteed accrued utility ($U_{rga}$) as the maximum real instantaneous utility reward accumulated throughout execution that the system player is able to guarantee, independently of the behavior of the environment player: $U_{rga} \triangleq \langle\langle \text{sys} \rangle \rangle R_{\text{max}}^{rIU}[F \, t = \text{MAX\_TIME}].$ This enables us to obtain the utility that an optimal algorithm would be able to extract from the system, given the most adverse possible conditions of the environment.

— Best-case scenario analysis. To obtain the real maximum accrued utility achievable ($U_{rma}$), we specify a coalition of the system and environment players, which behave cooperatively to maximize the utility reward: $U_{rma} \triangleq \langle\langle \text{sys, env} \rangle \rangle R_{\text{max}}^{rIU}[F \, t = \text{MAX\_TIME}].$

3.3.2. Non-latency-aware Adaptation. In non-latency-aware adaptation, the real utility does not coincide with the expected utility that an arbitrary algorithm would employ for decision-making, so we proceed with the analysis in two stages:

---

Listing 4: Utility functions and reward structure

```plaintext
const W_{UR}, W_{UF}, W_{UC};
rewards "rIU" (turn=SYS\_TURN) : TAU * (W_{UR} * uR + W_{UF} * uF + W_{UC} * uC);
endrewards
```

Listing 5: Expected utility reward structure

```plaintext
rewards "rEIU" (turn=SYS\_TURN) : TAU * (W_{UR} * uER + W_{UF} * uF + W_{UC} * uC);
endrewards
```

---

formula $uR = (rT > 0 & rT < 100? 1:0)$

formula $+(rT > 100 & rT < 200? 0.01 * ((rT - 100)/(100)):0)$

formula $+(rT > 2000 & rT < 4000? 0.25 + (-0.25) * ((rT - 2000)/(2000)):0)$

formula $+(rT > 4000? 0:0);$

...
(1) Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based on an rPATL specification that obtains the expected guaranteed accrued utility ($U_{ega}$) for the system player: $U_{ega} \triangleq \langle \langle \text{sys} \rangle \rangle R_{\text{rEIU}}^{\text{max}} F t = \text{MAX\_TIME}$. For the specification of this property we employ the expected utility reward $rEIU$ (Listing 5) instead of the real utility reward $rIU$. Note that for latency-aware adaptation $U_{ega} = U_{rga}$.

(2) Verify the specific property of interest (e.g., $U_{rga}, U_{rma}$) under the generated strategy. We do this by using PRISM-games to build a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, once we have computed a strategy for the system player to maximize expected utility, we quantify the reward for real utility in the new game in which the system strategy is fixed.

3.4. Results

In this section, we first compare worst- and best-case scenario analysis of a version of Znn.com that includes only the pair of tactics to enlist/discharge servers that are affected by latency in order to compare latency-aware and non-latency aware adaptation. Second, we provide some results to quantify the impact in utility that adding tactics to increase/reduce content fidelity introduce in the system. The improvement of introducing the new tactics is shown for latency-aware and non-latency-aware adaptation.

3.4.1. Comparing Latency-aware vs. Non-Latency-aware Adaptation. Table II compares the results for the utility extracted from the system by a latency-aware vs. a non-latency-aware version of the system player, for a model of Znn.com that represents an execution of the system during 1000s. The models consider a pool of up to 4 servers, out of which 2 are initially active, and includes a repertoire of tactics limited to enlisting/discharging servers. The period duration $\text{TAU}$ is set to 10s, and for each version of the model, we compute the results for four variants with different latencies for the activation of servers of up to $4^t \text{TAU}$ s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s. The fidelity level in this set of experiments is fixed, therefore we factor it out of the utility calculation ($w_U = 0.6, w_r = 0, w_c = 0.4$).

We define the delta between the expected and the real guaranteed utility as: $\Delta U_{er} = (1 - \frac{U_{ema}}{U_{rga}}) \times 100$. Moreover, we define the delta in real guaranteed utility between latency-aware and non-latency aware adaptation as: $\Delta U_{rga} = (1 - \frac{U_{nrga}}{U_{lrga}}) \times 100$, where $U_{nrga}$ and $U_{lrga}$ designate the real guaranteed accrued utility for non-latency-aware and latency-aware adaptation, respectively. The delta in real maximum accrued utility ($\Delta U_{rma}$) is computed analogously to $\Delta U_{rga}$.

Table II: SMG model checking results for Znn.com

<table>
<thead>
<tr>
<th>Latency (s)</th>
<th>Latency-Aware</th>
<th>Non-Latency-Aware</th>
<th>$\Delta U_{rga}$ (%)</th>
<th>$\Delta U_{rma}$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{TAU}$</td>
<td>468.38</td>
<td>458.38</td>
<td>0</td>
<td>996.60</td>
</tr>
<tr>
<td>$2^t \text{TAU}$</td>
<td>452.82</td>
<td>452.82</td>
<td>0</td>
<td>996.60</td>
</tr>
<tr>
<td>$3^t \text{TAU}$</td>
<td>447.26</td>
<td>447.26</td>
<td>0</td>
<td>996.60</td>
</tr>
<tr>
<td>$4^t \text{TAU}$</td>
<td>441.70</td>
<td>441.70</td>
<td>0</td>
<td>996.60</td>
</tr>
</tbody>
</table>

Table II shows that latency-aware adaptation outperforms in all cases its non-latency-aware counterpart. In the worst-case scenario, latency-aware adaptation is
able to guarantee an increment in utility extracted from the system, independently of the behavior of the environment ($\Delta U_{rga}$) that ranges between approximately 18 and 58%, increasing progressively with higher tactic latencies. In the best-case scenario (cooperative environment), the maximum utility improvement that latency-aware adaptation can achieve with respect to non-latency-aware adaptation is rather moderate (staying in the range 3-4%), and does not experience any variation with latency. This is an expected result, since independently of the quality of the decisions made by the system player, the environment is always going to favor utility maximization both in the latency-aware and non-latency aware cases. Regarding the delta between expected and real utility that adaptation can guarantee, we can observe that $\Delta U_{sr}$ is always zero in the case of latency-aware adaptation, since expected and real utilities always have the same value, whereas in the case of non-latency-aware adaptation there is a remarkable decrement that ranges between 19 and 60%, also progressively increasing with higher tactic latency.

3.4.2. Quantifying the Impact of Tactics on Utility. In this section, we compare the results for the utility extracted from the system for the worst-case scenario, using four different model variants of Znn.com. Two of the variants correspond to the latency-aware adaptation case when it includes: (i) only the pair of tactics to enlist/discharge servers (LA), and (ii) an extended set of tactics that include the tactics to enlist/discharge servers, plus the pair of tactics to increase/reduce content fidelity (LA+). The other two variants include the same sets of tactics for the non-latency-aware adaptation case (indicated by NLA and NLA+, respectively).

All models represent an execution of the system during 1000s, and consider a pool of up to 4 servers, out of which 2 are initially active. The period duration $\tau$ is set to 10s, and for each version of the model, we compute the results of a latency range for the activation of servers between 0 and $7\tau$ s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s for high fidelity, and 0.7s for low fidelity. The utility preferences used for the experiments give preference to performance over cost ($w_{UR} = 0.5$, $w_{UF} = 0.3$, $w_{UC} = 0.2$).

— Latency-aware Adaptation. Figure 2 compares the two variants of latency-aware adaptation. In the LA variant, it can be observed that the progressive increment in latency of the enlist server tactic results in a proportional reduction of the real guaranteed utility $U_{rga}$. However, for increasing latency values in the LA+ variant, $U_{rga}$ only decreases moderately in comparison with the LA variant, due to the fact that the optimal strategy synthesis algorithm starts favoring the selection of the tactic to reduce fidelity instead of the one to enlist a new server. Although decreasing fidelity has a negative influence in the value of the fidelity utility $U_F$ and has a moderate impact in reducing response time compared to adding a new server, increasing latencies of the enlist server tactic dwindle its capability to extract system utility over time, compared to reducing content fidelity.

— Non-latency-aware Adaptation. Figure 3 compares the two variants of non-latency-aware adaptation. In contrast with the latency-aware adaptation case, there is a clear discrepancy between real guaranteed utility $U_{rga}$, and the expected guaranteed utility $U_{ega}$ both for the NLA and NLA+ variants. Interestingly, it can be observed how the addition of the pair of fidelity tactics does not represent any difference in $U_{ega}$ nor $U_{rga}$ between the NLA and NLA+ variants. This is a direct consequence of the strategy synthesis process not being aware of the latency of the tactic to enlist servers. Concretely, the synthesis only considers the positive net impact on utility of enlisting a server, which outweighs the impact on utility of reducing fidelity, effectively preventing the selection of the fidelity reduction tactic, independently of
the fact that it is capable to extract more system utility over time than the tactic to enlist servers. Moreover, it can be observed that the $U_{rga}$ for both NLA and NLA+ describes a non-monotonic curve. This is due to the fact that there are many factors that influence the outcome of the optimal strategy with changing conditions, in particular because in the non-latency-aware case there is a difference between the behavior of the environment predicted by the strategy synthesis, and the real behavior represented in the game (although sometimes, they may coincide, leading to peaks in $U_{rga}$ that approach $U_{ega}$).

4. LATENCY-AWARE ADAPTATION

In the previous section, we showed the improvement that latency-awareness could bring if we had an optimal adaptation algorithm. The SMG analysis approach presented allows us to quantify that improvement without actually encoding any adaptation algorithm in the system model. In this section, we present a latency-aware adaptation algorithm, and the results of its evaluation in simulation. Using simulation allowed us to run many repetitions of the experiments with randomly generated behaviors of the environment, and to replicate exactly the same conditions for the new algorithm and the baseline NLA algorithm.

Latency-aware adaptation takes into account the tactics’ latency when deciding how to adapt. In our approach, the goal is to consider the latency of the tactics so that the sum of utility provided by the system over time is maximized. The effect of tactic latency on utility is that for tactics that have some latency, the system does not start to accrue the utility gain associated with the tactic until some time after the enactment of the tactic. Moreover, negative impacts of the tactic may have no latency, and start without delay. For example, when adding a server to the system, the server takes some time to boot and be online, whereas it starts consuming power—and thereby increases cost—immediately. In this example, it means that the tactic to add a server causes a drop in utility before it results in a gain.

Another consequence of tactic latency is that some near-future system configurations can be infeasible. For example, let us suppose that the system has to deal with an increase in load within 5 seconds, and it could handle that with an additional server. If enlisting an additional server takes 10 seconds, then the desired configuration that has one additional server 5 seconds into the future is infeasible. Current approaches that do not take latency into account would consider that solution regardless of whether it is feasible or not. When proactively looking ahead, taking adaptation latency into account allows the adaptation mechanism to rule out infeasible configurations from the adaptation space.

A complication arises when tactic latency is longer than the interval between adaptation decisions. When that is the case, it is possible that during an adaptation decision, a tactic that has been previously started has not yet reached the point where its effect will have been realized. If the decisions are made based only on the currently observed state of the system, ignoring the expected effect of adaptations in progress, the system will overcompensate, starting unnecessary adaptations. What is needed is a model of the system that not only represents the current state of the system, but also keeps track of the expected state of the system in the near future based on the tactics that have been started but have not yet completed.

4.1. Algorithm

The algorithm we present is an extension of an algorithm developed by Poladian et al. to compute the optimal sequence of adaptation decisions for anticipatory dynamic configuration [Poladian et al. 2007]. Using dynamic programming and relying on a perfect prediction of the environment for the duration of a system run, their algorithm can find the adaptation decision that at each time step maximizes the future aggregate utility, while accounting for the penalty of switching configurations. They showed that the algorithm had pseudo-polynomial time complexity, and was therefore suitable for online adaptation.

The key improvement our algorithm brings is how the latency of tactics is taken into account. On the one hand, there is an adaptation cost that latency induces. For example, if adding a server takes \( \lambda \) seconds from the time a server is powered up until it can start processing requests, and \( \Delta U_c \) is the additional cost the new server incurs, then the adaptation cost is \( \lambda \Delta U_c \). This cost could be partially handled by the original algorithm, as a reconfiguration penalty. However, that is not sufficient to handle the other issues previously mentioned that latency brings, namely, the infeasibility of configurations and the need to track adaptation progress. Our algorithm for latency-aware proactive adaptation explicitly handles the issues that arise due to tactic latency.

The algorithm requires iterating over all the possible configurations of the system, where a configuration describes variable aspects of the system relevant to the adaptation decision. In the Znn example, a configuration indicates how many servers are in the pool of servers, and what is the fidelity level of the content being served. To keep track of adaptation progress, a configuration also encodes information about the
progress of adaptations that have non-zero latency. In our example, that means that a configuration indicates whether a new server is being added, and how much progress that tactic has made. It is important to note that the information about progress is only needed at the granularity of the evaluation period \( \tau \). In general, \( C \) is the set of possible configurations, and \( C_i \) is the \( i \)th configuration, for \( i \in \{1 \ldots |C|\} \). For our running example, \( C = (S \times A \times F) \setminus \{(s, a, f) : S \times A \times F|s = 4 \land a \in A \setminus \{0\}\} \), where \( S = \{1 \ldots 4\} \) is the number of active servers in the system; \( A = \{0 \ldots \lceil \frac{1}{2}\rceil\} \) is the number of evaluation periods until the addition of a server completes, with 0 indicating that the tactic is not being executed; and \( F = \{1, 2\} \) is the fidelity level. Since the tactic to add a server cannot be used when the system already has the maximum number of servers, all the configurations with 4 servers and the tactic running are not included in \( C \).

The algorithm also needs to determine whether a particular configuration can be reached at a particular time, and tactic latency plays a key role in that determination. More specifically, the algorithm needs to determine if configuration \( c' \) can be reached from configuration \( c \) in one evaluation period—the boolean function \( isReachableFromConfig(c, c') \) encapsulates that. In addition, it needs to know if configuration \( c' \) can be reached at the current time—the function \( isReachableNow(c') \) determines that. In addition to latency, blocking effects between tactics are also considered by these functions. For example, in our running example, only one tactic can be used in an evaluation period. Details about these two functions are provided in section 4.1.1.

In reactive adaptation, the decision algorithm is typically invoked upon events that require an adaptation to be performed. However, for proactive adaptation, the decision must be done periodically, looking ahead for future states that may require the system to adapt. Our algorithm is therefore run periodically, with a constant interval \( \tau \) between runs. We limit the look-ahead of the algorithm to a near-term horizon of \( H \) evaluation periods, which in turn limits how far into the future the environment state needs to be estimated.\(^8\) The estimation of the future environment state is accessed by the algorithm via the function \( env(x) \), which returns the expected environment state \( x \) time units into the future.

Employing a dynamic programming approach, the algorithm (Algorithm 1) uses two matrices, \( u \) and \( n \), to store partial solutions. The element \( u_{i,t} \) holds the utility projected to be achieved from the evaluation period \( t \) (with \( t = 0 \) being the current period, \( t = 1 \) the next one, and so on) until the horizon if the system has configuration \( C_i \) at evaluation period \( t \). An infeasible partial solution is marked by a value of \(-\infty\) assigned to \( u_{i,t} \). The element \( n_{i,t} \) holds the configuration that the system must adopt in period \( t+1 \) to attain the projected utility \( u_{i,t} \) if the configuration in period \( t \) is \( C_i \).

The main loop (lines 1-23) works backwards from the horizon, computing the partial solutions using the partial solutions previously found. For each configuration (lines 2-22), it computes its projected utility or deems the configuration infeasible. For evaluation periods \( t > 0 \), all configurations are assumed feasible, and the only concern is whether one potential configuration is reachable from another potential configuration. However, for the current evaluation period \( t = 0 \), only those configurations that can be reached are deemed feasible. The projected utility a configuration can achieve can achieve is the sum of the utility the configuration obtains in that particular evaluation period (line 6), and the maximum utility it can achieve in the periods after that. Computing the former relies on the function \( U(c, e) \), which is the instantaneous utility provided by configuration \( c \) in environment \( e \). To compute the latter, the algorithm iterates (lines 11-19) over all the feasible configurations that can follow (as determined by \( isReachableFromConfig(C_i, C_j) \)) to find the configuration that the system should

\(^8\) Environment state estimation is beyond the scope of our work, but techniques such as Poladian et al.’s calculus for combining multiple source of predictions [Poladian et al. 2007] can be used.
have in evaluation period $t+1$ to maximize the projected utility of having configuration $C_i$ in evaluation period $t$ (lines 14-17). Once all the possible solutions have been computed, the algorithm selects the configuration the system should have at the current time to maximize the projected utility (line 24). By comparing the current system configuration with the selected configuration along the different dimensions ($S,A,$ and $F$ in our example), it is easy to determine what adaptation tactics have to be started at the current time, if any.

**ALGORITHM 1:** Latency-aware proactive adaptation

1: for $t = H - 1$ downto 0 do
2:   for $i = 1$ to $|C|$ do
3:     $u_{i,t} \leftarrow -\infty$ (assume infeasible configuration)
4:     $n_{i,t} \leftarrow 0$ (assume no next state)
5:     if $t > 0 \lor \text{isReachableNow}(C_i)$ then
6:       $u_{\text{local}} \leftarrow \tau U(C_i, \text{env}(t))$
7:       if $t = H$ then
8:         $u_{i,t} \leftarrow u_{\text{local}}$
9:       else
10:       {find the next best configuration after $i$}
11:       for $j = 1$ to $|C|$ do
12:         if $u_{j,t+1} > -\infty \land \text{isReachableFromConfig}(C_i, C_j)$ then
13:           $u_{\text{projected}} \leftarrow u_{\text{local}} + u_{j,t+1}$
14:           if $u_{\text{projected}} > u_{i,t}$ then
15:             $u_{i,t} \leftarrow u_{\text{projected}}$
16:             $n_{i,t} \leftarrow j$
17:         end if
18:       end if
19:     end for
20:   end if
21: end for
22: end for
23: best $\leftarrow \arg \max_i u_{i,0}$ (best starting configuration)
24: return $C_{\text{best}}$

4.1.1. Adaptation Feasibility. An important part of the proactive latency-aware adaptation algorithm is determining whether it is possible reach a particular system configuration through adaptation at a particular time in the near future. Obviously, tactic latency plays a fundamental role in this determination, not only because a tactic needed to reach a configuration may take some time to execute, but also because it can block other tactics while executing. In our running example, only one tactic can be executed at a time, and, additionally, only one tactic can be started in each evaluation period. The algorithm uses two functions to determine the feasibility of possible adaptations.

The function $\text{isReachableNow}(c')$ returns true if it is possible to reach configuration $c'$ immediately from the current configuration of the system. For example, if no tactic is executing in $Z_{nn}$, it would be possible to reach immediately a configuration in which the fidelity level has been changed, or one in which the tactic to add a new server has been started, but not both. On the other hand, if the tactic to add a server was executing (i.e., it was started in a previous evaluation period and has not completed yet), it would not be possible to reach any configuration other than the current one.

The function $\text{isReachableFromConfig}(c, c')$ returns true if configuration $c'$ can be reached from configuration $c$ in one evaluation period. More specifically, it assumes that (i) $c$ will be the configuration at the beginning of the period, including the possible effect of tactics that could have been started at that time; (ii) one evaluation period will elapse allowing for example progress on a tactic with latency; and (iii) optionally
Analyzing Latency-aware Self-adaptation using Stochastic Games and Simulations

Listing 6: Alloy model of configuration reachability: basic definitions

```alloy
open util/ordering[S] as servers
open util/ordering[A] as progress
open util/boolean

sig S {} // the different number of active servers
sig A {} // the different levels of progress of the add server tactic
sig C {} // each element of C represents a configuration
s : S, // the number of active servers
a : A, // the progress of the add server tactic
f : Bool // fidelity level

fact uniqueInstances { all disj c, c2 : C | !(c2.s = c.s and c2.a = c.a and c2.f = c.f) }
```

A tactic can be started at the end of the period. For example, assuming that \( c \) is a configuration in which the tactic to add a server has one period left to complete, and \( c' \) is a configuration with one more active server and a different fidelity would be feasible because the tactic adding a server would complete in the elapsed period, and the fidelity can be changed immediately.

These two functions can be implemented in different ways as long as they satisfy their specification. Furthermore, since they are independent of the state of the environment, they can be computed offline, generating a lookup table to be used at runtime.

Taking advantage of this, we used a Alloy [Jackson 2012] to formally specify system configurations, and adaptation tactics, and to compute the reachability functions offline. Alloy is a language based on first-order logic that allows modeling structures—known as signatures—and relationships between them in the form of constraints. One advantage of using Alloy is that it is a declarative language, and, in contrast to imperative languages, only the effect of operations—tactics in our case—on the model must be specified, but not how the operations work. The Alloy analyzer can then be used to find structures that satisfy the model.

The basic definitions of the specification used to compute the reachability functions is shown in Listing 6. These definitions introduce the sets \( S, A, \) and \( C \), representing the number of servers, the progress of the tactic to add a server, and the possible configurations the system can have. The elements of \( S \) and \( A \) are not numbers, but just abstract elements of an ordered set. It is possible to refer to the first and last element of the set that represents the possible levels of active servers as servers/first and servers/last, respectively. The relationships prev and next allow referring to the previous and next element in the ordered set. The signature \( C \) defines the set of all possible configurations. Without additional constraints, Alloy could generate elements of \( C \) with the same number of servers, progress of the tactic, and fidelity level. The constraint in line 11 is used to force all elements of the set to be unique configurations.

The specification of the tactics is shown in Listing 7. The tactic to add a server is decomposed into two predicates due to its latency. The first predicate (lines 1-5) specifies the start of the tactic. This predicate has two arguments \( c \) and \( c' \), representing the pre- and post-state, respectively. For the tactic to be able to start, it is required that no tactic is running, and that the configuration in the pre-state is not the last level of servers (i.e., the configuration has less than the maximum number of servers). In the post-state, the only change to the configuration is that the level of progress of the tactic is the first one. The tactic in the post-state has been started and will in subsequent steps go through all the levels of progress until it reaches the last one when it completes. The other predicate (lines 6-11) specifies how the configuration changes when the tactic makes progress in one evaluation period. The tactic can only make progress if it has not completed in the pre-state. In the post-state, the configuration
Listing 7: Alloy model of configuration reachability: tactics and functions

```alloy
pred addServerTacticStart[c, c' : C] {  
  !tacticRunning[c] and c.s != servers/last  
  c'.a = progress/first  
  c'.s = c.s and c'.f = c.f  
}

pred addServerTacticProgress[c, c' : C] {  
  c.a != progress/last  
  c'.a = progress/next[c.a]  
  c'.s = c.s and c'.f = c.f  
}

pred tacticRunning[c : C] {  
  c.a != progress/last  
}

pred removeServerTactic[c, c' : C] {  
  !tacticRunning[c] and c.s != servers/first  
  c'.s = servers/prev[c.s]  
  c'.a = c.a and c'.f = c.f  
}

pred changeFidelityTactic[c, c' : C] {  
  !tacticRunning[c]  
  c'.f = Not[c.f]  
  c'.s = c.s and c'.a = c.a  
}

pred noOp[c, c' : C] {  
  !tacticRunning[c]  
  c'.s = c.s and c'.a = c.a and c'.f = c.f  
}

pred isReachableNow[c, c' : C] { // is c' reachable now if current config is c?  
  c = c' or removeServerTactic[c, c'] or addServerTacticStart[c, c] or changeFidelityTactic[c, c']  
}

pred isReachableFromConfig[c, c' : C] { // is c' reachable from config c in one evaluation period?  
  one temp : C | timeStep[c, temp] and isReachableNow[temp, c]  
}

pred timeStep[c, c' : C] { noOp[c, c'] or addServerTacticProgress[c, c'] }  
```

will have the same fidelity level, and the next level of progress. If the latter is the last level of progress, then the tactic has completed and the post-state configuration has one more active server. Otherwise, the number of servers stays the same. The tactics for removing a server (lines 13-17) and for changing the fidelity (lines 18-22) do not have latency, and, therefore, do not need to be split into start and progress as the other tactic. If no tactic is running, the system can just stay in the same configuration. In the model, the predicate in lines 23-26 is used to allow that behavior.

Listing 7 also defines the configuration reachability predicates. For isReachableNow (lines 27-29), configuration c' can be reached from c trivially if they are the same configuration, or if c' is the configuration resulting from starting a tactic when the system is in configuration c, and no passage of time is allowed. In the case of isReachableFromConfig (lines 30-32), configuration c' can be reached from c after one evaluation period if the configuration that results from letting one period to elapse, configuration temp, is such that configuration c' can be reached from temp without any more passage of time. The predicate timeStep (line 33) is used for the first part of this condition, and isReachableNow is reused for the second part.

The Alloy code in Listing 8 is used to generate the reachability functions. Each predicate is used to generate the elements of the relationship Result.reachable for each of the reachability functions. The commands in lines 8-9 run the Alloy analyzer to generate the relationships that satisfy the corresponding predicates. The command specifies how many elements the solution should have in each set. For our example, when the latency of the tactic to add a server is 3τ, the solution must have 4 servers, 3τ + 1 = 4
Analyzing Latency-aware Self-adaptation using Stochastic Games and Simulations

__Listing 8: Generation of configuration reachability tables in Alloy__

```plaintext
sig Result { reachable : C -> C }
pred isReachableFromConfigGeneration {
  one r : Result | all c1, c2 : C | c1 -> c2 in r.reachable <--> isReachableFromConfig[c1,c2]
}
pred isReachableNowGeneration {
  one r : Result | all c1, c2 : C | c1 -> c2 in r.reachable <--> isReachableNow[c1,c2]
}
run isReachableFromConfigGeneration for exactly 4 S, exactly 4 A, exactly 32 C, exactly 1 Result
run isReachableNowGeneration for exactly 4 S, exactly 4 A, exactly 32 C, exactly 1 Result
```

levels of progress for the tactic, and two fidelity levels, for a total of 32 configurations. Additionally, the run should produce one result. The output of the Alloy analyzer can be exported, and transformed to a format suitable for its use at runtime.

### 4.2. Simulation

We implemented a simulation of a self-adaptive Znn with two goals. One was to evaluate the improvement that our algorithm for latency-aware (LA) proactive adaptation achieves compared to a non-latency-aware (NLA) approach. The second one, was to compare the theoretical results obtained with the SMG for generic NLA and LA algorithms with the results obtained with a concrete algorithm.

The simulation was implemented using OMNeT++, an extensible discrete event simulation environment [Varga et al. 2008]. It simulates the arrival of requests from clients, randomly generating requests. The requests arrive at the load balancer of Znn, and are forwarded to one of the idle servers. If no server is idle, then the requests are queued in FIFO order until one server becomes available. Each server processes one request at a time, with a service time distributed with an exponential distribution whose rate depends on the fidelity of the content being served. In the case of high fidelity, the rate is 1, and for low fidelity the rate is $\frac{1}{10}$.

The inter-arrival times between client requests are generated randomly with a rate in the interval $[0, 2]$ that changes periodically. To create trends and the possibility of sustained load, the request generator maintains a trend for the request arrival rate that can be upward, downward, or sustained. Every $\tau$ units of time, the trend is changed with a probability of 0.5. Also with the same interval, and when the trend is either upward or downward, a new arrival rate is selected randomly from a uniform distribution between the current rate and the lower or higher end of the $[0, 2]$, for downward or upward trend respectively. That rate is then used to generate exponentially distributed inter-arrivals. To be able to simulate the execution of the system with the same random pattern of client requests using each of the two algorithms, the request inter-arrival times and the service times are drawn from two separate random number generators.

The self-adaptive layer of the simulated system works as follows. The system is monitored by keeping track of request inter-arrival times when a client request arrives, and of the response times every time a request processing completes. Once every evaluation interval $\tau$, these observations are used to compute their average and standard deviation for the period since the last evaluation. Using the average response time, the fidelity level, and the number of servers in the system, the utility accrued since the last evaluation is computed using the utility function shown in Table I.

---

9Even though some of these configurations are not valid, namely those where there are 4 servers and the add server tactic is executing, we chose to keep the model simpler, even if it produces some elements in the reachability relationships that will never be used.
Next, the adaptation algorithm is used to determine if the system should self-adapt and how. We implemented both the latency-aware algorithm (Algorithm 1) and a non-latency-aware algorithm. The latter is basically the same as the former, except that it does not account for latency other than by considering the adaptation penalty induced by the cost of having a server powered until it becomes active.

When the algorithm is run in each evaluation period, it needs to know what is the current configuration of the system, including whether the tactic to add a server is running, and how much progress it has made. This is achieved by maintaining a model of the system configuration that keeps track of the number of servers in the system, and how many of them are active. In addition, the model keeps a list of expected changes in the future. For example, when a new server is added to the system, an expected change reflecting that the server becomes active is recorded with an expected time of $\lambda$ into the future. In that way, it is possible to determine how much time is needed until the tactic completes. When a server actually becomes active in the simulation, the model of the current system configuration is updated to reflect that change and the corresponding entry is removed from the list of expected system changes.

The predictive model of the environment, $env(x)$, was implemented as an oracle that can predict perfectly the average and variance of the request inter-arrival times for the same horizon used by the algorithm. Although the request arrivals are randomly generated in the simulation, a perfect prediction can still be achieved by generating the inter-arrival times before they are consumed by the simulation.

Implementing the $U(c, e)$ function requires first estimating the average response time for requests when the system has configuration $c$, and the environment is $e$. In this case, the relevant properties of the environment are the average and variance of the inter-arrival times. To estimate the average response time needed for the utility calculation, we used queueing theory with an M/M/c queueing model. Once the average response time is estimated in this way, the utility is estimated using the utility functions and preferences shown in Table I. After the adaptation algorithm has determined how the system has to be changed, the execution of the adaptation tactics is carried out by adding or removing servers, and changing the fidelity as needed.

4.3. Results

We used the simulation to perform comparisons analogous to those done with the SMG analysis in 3.4, namely, comparing latency-aware with non-latency-aware adaptation, and assessing the impact of an additional pair of tactics for adapting the fidelity level of the system. The simulation was run with the same parameters used for the SMG analysis. The horizon used for the algorithms was computed so that if the system was running with one server, it had a horizon large enough to be able to compute the effect of adding the three remaining servers. For that reason, the horizon was calculated as $3\frac{\lambda\tau}{\lambda} + 1$, the number of periods needed to enlist three servers plus one more period to consider the impact on utility of the change.

4.3.1. Comparing Latency-aware vs. Non-Latency-aware Adaptation. For each combination of parameters, the simulation was run 100 times to obtain the statistics shown in Table III. On average, the latency-aware algorithm outperformed the non-latency-aware one. The improvement of the LA algorithm increased with the latency of the tactic. The standardized effect size measure statistic $\hat{A}_{12}$ [Arcuri and Briand 2012] shows that LA outperforms NLA 53% to 70% of the times, depending on the parameters. For several combinations of parameters, the minimum percentual utility difference $\Delta U(\%)$ was negative, meaning that NLA did better. This is due to a limitation of the queueing model used by the algorithms to estimate the response time of different configurations, because it computes the steady-state response time, and, therefore, ignores the effect
of arrival spikes that may leave a backlog of arrivals to be processed in later periods. The LA algorithm avoids adaptation when there are transient increases in load if the cost of enlisting a server will be higher than the negative impact of not adding it. Because of the limitation of the queueing model, it sometimes underestimates that negative effect. Since the NLA algorithm does not account for the latency of the tactic, it is more prone to add servers, and that gives it an advantage in these cases. These situations were not very common in our experiment runs, as indicated by the 10% quantile, which, except for the cases with the lowest tactic latency, was positive.

It is worth noting that the results shown in Tables II and III are not directly comparable, because the SMG analysis and the simulation quantify different statistics. The SMG analysis determines the utility the system can accrue in the worst and best case for each approach. For example, the worst-case environment can be different for LA and for NLA, and the difference reported is between those two environments under each of the approaches. The simulation, on the other hand, is unlikely to reach those extremes, because the behavior of the environment is randomly generated, and constrained by trends. Furthermore, each of the differences $\Delta U(\%)$, whose statistics are reported in Table III, is the difference in percentage between the utility the two algorithms obtained for the same environment behavior. For these reasons, it is expected that the differences reported in Table III will be more modest than those in Table II.

### 4.3.2. Assessing the Impact of Tactics on Utility

We also used the simulation to assess the impact that adding tactics to control the fidelity level has on system utility. The results are shown in Figure 4. Each data point is the average of 100 runs of the simulation, each of which had a simulated duration of 1000 seconds. The LA and NLA variants show latency-aware and non-latency-aware adaptation respectively, and in both cases, only the tactics for enlisting and discharging servers are available. The LA+ and NLA+ variants additionally have the pair of tactics to increase and decrease content fidelity.

In general, the results match qualitatively those obtained with the SMG analysis. The LA(+) variants are superior than the NLA(+) variants. Adding the fidelity tactics barely makes a difference when non-latency-aware adaptation is used. However, latency-aware adaptation is able to better exploit the added tactics by taking into account how they differ in latency. For example, if the proactive adaptation is dealing with a short-lived increase in load, latency-awareness uses the faster fidelity tactic, instead of incurring the cost of adding another server.

### 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a novel approach to choosing adaptations that considers latency information, using Znn.com for illustration purposes. The approach makes a complementary use of model checking of stochastic multiplayer games (SMGs) and a novel latency-aware proactive adaptation algorithm. While model checking of SMGs enables us to understand potential improvements by analyzing boundary cases of

---

Note that this is a limitation of the $U(c,e)$ function used by the algorithm, and not a problem with the algorithm itself.
adaptation even before developing specific algorithms or infrastructure, it does not provide a suitable mechanism to make adaptation choices at runtime. To fill this gap, we proposed a proactive adaptation latency-aware algorithm, which we simulated to obtain a more detailed understanding of adaptation behavior and check whether the results fall within boundary cases predicted by SMGs.

Regarding model checking of SMGs, we have shown how this technique is used to compare latency-aware and non-latency aware adaptation. Our results show that latency-aware adaptation performs better than non-latency-aware adaptation both in the worst- and best-case scenarios, with progressively increasing improvements with higher tactic latencies in the worst case. Moreover, results indicate that not considering latency information in decision-making can potentially inhibit the selection of adaptations that could help improve the performance of adaptation.

Concerning the latency-aware proactive adaptation algorithm, we compared it against the proactive algorithm presented in [Poladian et al. 2007], which does not consider latency, showing that latency-aware adaptation achieves higher utility. In addition, we have presented an approach to compute adaptation feasibility, a key component of latency-aware adaptation, using formal specification and analysis of adaptation tactics using Alloy.

A current limitation of the model checking part of the approach is that its scalability is limited by PRISM-games, which currently uses explicit-state data structures and is to the best of our knowledge the only tool supporting model-checking of SMGs. However, the foreseeable development of symbolic (BDD-based) versions of SMG analysis tools will improve scalability. Another limitation concerns the current lack of an implementation (e.g., based on Rainbow) to measure the actual improvement obtained by our latency-aware algorithm.

With respect to future work, our two-pronged approach combines the strengths of algorithm-agnostic formal verification with those of simulating specific adaptation algorithms, and can be generalized to different contexts. In particular, we are working on applying this approach to self-protecting systems, studying how different adaptation alternatives can minimize the damage that an attacker can inflict upon a defending system [Schmerl et al. 2014]. We also aim at refining the approach to do runtime synthesis of proactive adaptation strategies based on SMGs. Concerning latency-aware adaptation, we aim at exploring how tactic latency information can be further exploited to attain better results both in proactive and reactive adaptation (e.g., by parallelizing tactic executions).
REFERENCES


Javier Cámara, Gabriel A. Moreno, and David Garlan. 2014. Stochastic game analysis and latency awareness for proactive self-adaptation. In SEAMS. ACM.


András Varga and others. 2008. An overview of the OMNeT++ simulation environment. In Simutools. ICST.
