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

JAVIER CÁMARA, Carnegie Mellon University
GABRIEL A. MORENO, Carnegie Mellon University
DAVID GARLAN, Carnegie Mellon University
BRADLEY SCHMERL, Carnegie Mellon University

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. Furthermore, lower-impact but quicker adaptations may be ignored altogether, even if over time the accrued impact is actually higher. In this paper we introduce a novel approach to choosing adaptations that considers these latencies. To show how this improves adaptation decisions, we use a two-pronged approach: *(i)* model checking of stochastic multiplayer games (SMGs) enables us to understand best- and worst-case scenarios of optimal 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 runtime, 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.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms: Verification, Algorithms

Additional Key Words and Phrases: proactive adaptation, stochastic multiplayer games, latency-aware, Latency

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research, CNS-0834701 from the National Science Foundation, and by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research or the U.S. government. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. (DM-0002414).
Author's addresses: J. Cámara, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, and G.A. Moreno, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, and David Garlan, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, and Bradley Schmerl, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213,

DOI:http://dx.doi.org/10.1145/0000000.0000000

ACM Reference Format:
Javier Camara, Gabriel A. Moreno, David Garlan and Bradley Schmerl. 2015.
Analyzing Latency-aware Self-adaptation using Stochastic Games and Simulations. *ACM Trans. Autonom. Adapt. Syst.* V, N, Article A (January YYYY), 28 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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; a pattern commonly used to construct self-adaptive systems.

Current self-adaptive proposals 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 system, 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.[1]

However, in many domains (such as cloud computing and wireless sensor networks, or where human operators need to be involved) 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. We refer to the interval between the time instants in which a tactic's execution is triggered and its effects are observed in the state of the system as *tactic latency*.

While considering 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 needing to develop 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-

[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.

case scenarios with little specification effort, compared to employing other alternatives (e.g., prototyping, simulation). 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) An 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 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); (ii) generalizing our adaptation algorithm to support multiple tactics, determining adaptation feasibility via formal analysis of tactics using Alloy [Jackson 2012]; and (iii) exploring how latency-awareness can improve adaptation in systems involving humans in the execution of adaptation, illustrating our approach in the context of an industrial middleware.

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 better exploit the availability of tactics with different latencies and obtain higher utility than Poladian et al.'s algorithm, which is optimal under the assumption of no tactic latency.

We anticipate that our approach will improve adaptation effectiveness in at least the following kinds of systems, where latency is an important factor:

— *Cloud computing.* One of the advantages of cloud computing is providing elastic computing capacity that can adjust dynamically to the load on the system. One limitation of current approaches is that they assume that the control actions used to make these adjustments are immediate, when in reality they are not [Gambi et al. 2013]. Our approach could help improve the effectiveness of adaptation for cloud computing. For example, by considering the latency of enlisting additional capacity, it could proactively start the adaptation, or decide that a short workload burst is better handled by another tactic, or perhaps that both tactics are needed concurrently. Furthermore, it could even dynamically decide which provider to use at different times based on their provisioning time. In fact, the decision would not necessarily always favor faster provisioning, if for example, the system can afford, thanks to the proactive adaptation, to wait longer for the provisioning of the new capacity by a cheaper provider.
— *Wireless sensor networks.* In general these systems present a tradeoff between the frequency of sensor reading reports and their battery life. Proactive adaptation

can help improving the battery life without compromising the mission supported by the sensor network by adapting the reporting frequency ahead of environment changes [Paez Anaya et al. 2014]. Furthermore, some adaptations may require updating the firmware of the nodes, an operation that can take over a minute for updating a single node [Maatta et al. 2010].

— *Cyber-physical systems.* Some adaptations that could be used in cyber-physical systems (CPS) have latency that is due to physics. For example, since different formations in multi-robot teams have different qualities, an adaptation may require switching between them. Doing this has latency because of the time required for the robots to physically move in relation to their teammates. As another example, a GPS may be turned off to save power, however, turning it back on is an adaptation that is not instantaneous because the time to first fix may be about a minute [Liu et al. 2012].

— *Systems with human actuators.* Even though the goal of self-adaptation is to minimize the dependency of humans, self-adaptive systems typically rely on humans to actuate on the physical world. For example, scaling out in industrial control systems may require the connection of a device by a human operator [Cámara et al. 2013]. Adaptation tactics that involve human actuators have considerable latency, which must be taken into account when deciding how to adapt.

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. Next, Section 5 shows how our analysis technique can be used to show the benefits of incorporating latency-aware adaptation in the context of systems that employ human actuators in adaptation mechanisms. Finally, Section 6 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-

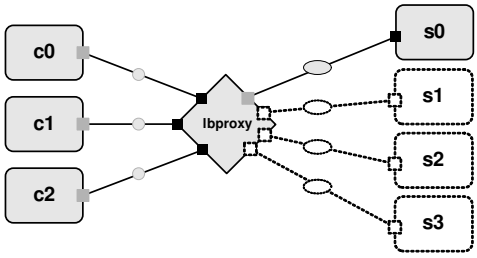


Fig. 1: Znn.com system architecture

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 does not support deciding between different tactics to address the load problem.

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 front-end 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.

The main objective for Znn.com is to provide content to customers within a reasonable response time, while considering the operating cost of the server pool. Znn.com may experience spikes in requests that it cannot serve adequately in its current configuration. To allow it to better serve its clients in such circumstances, it provides various hooks, or effectors, that can be used to adapt it. For example, the number of servers used can be increased, or the fidelity of the content can be decreased to only serve textual content instead of including images or videos. To help decide between these different adaptations, a self-adaptive system must balance three quality objectives: (i) performance, which depends on request response time; (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.

Table I: Utility functions and preferences for Znn.com

U_R			U_F	U_C	
0 : 1.00	500 : 0.90	2000 : 0.25	1 : 0.25	0 : 1.00	3 : 0.30
100 : 1.00	1000 : 0.75	4000 : 0.00	2 : 1.00	1 : 1.00	4 : 0.10
200 : 0.99	1500 : 0.50			2 : 0.90	

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:[2]

— *Tactic:* is 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.[3] 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.

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

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 effort to specify compared to employing other alternatives (e.g., simulation, prototyping), without the need to develop 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 [Norman et al. 2002] or wireless communication protocols [Kremer and Raskin 2001; Hoek and Wooldridge 2003], to biological systems [Kwiatkowska et al. 2008]. 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.

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.

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.[4] Properties written in rPATL can state that a coalition of players has a strategy which can ensure that the probability of an event's occurrence or an expected reward measure meet some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL [Alur et al. 2002] (a logic extensively used in multi-player games and multi-agent systems to reason about the ability of a set of players to collectively achieve a particular goal), combining it with the probabilistic operator $\mathsf{P}_{\bowtie q}$ and path formulae from PCTL [Bianco and de Alfaro 1995]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{r}_{\bowtie x}$ from [Forejt et al. 2011] to reason about goals related to rewards. An example of typical usage combining coalition and reward operators is $\langle\langle\{1,2\}\rangle\rangle \mathsf{R}^{r}_{\geq 5}[\mathsf{F}^{\star}\phi]$ [5], meaning that "players 1 and 2 have a strategy to ensure

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

[5] The variants of $\mathsf{F}^{\star}\phi$ used for reward measurement in which the parameter $\star \in \{0, \infty, c\}$ indicate that, when ϕ is not reached, the reward is zero, infinite or equal to the cumulated reward along the whole path, respectively.

that the reward r accumulated along paths leading to states satisfying state formula ϕ is at least 5, regardless of the strategies of other players." Moreover, an extended version of the rPATL reward operator $\langle\langle C\rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\star}\ \phi]$ enables the quantification of the maximum accumulated reward r along paths that lead to states satisfying ϕ that can be guaranteed by players in coalition C, independently of the strategies followed by the rest of players.

3.1.2. Analyzing adaptation via model checking of SMGs. The underlying idea behind the approach is modeling both the self-adaptive system and its environment as two players of a SMG, in which the system attempts to maximize an accrued reward (in this paper, accrued utility during system execution). Although in general, the environment does not have any predefined goal, it is useful to consider it either as an adversary of the system, or as a cooperative player to enable worst- and best-case scenario analysis, respectively, of different classes of adaptation algorithms (e.g., latency-aware *vs.* non-latency-aware).

By expressing properties that enable us to quantify the maximum and minimum rewards that a system player can achieve, independently of the strategy followed by the environment, we can analyze the performance of a particular type of adaptation, giving an approximation of the reward that an optimal decision maker would be able to guarantee both in worst- and best-case scenarios (by synthesizing strategies that optimize different rewards). These properties follow the general pattern $\langle\langle \mathsf{P}\rangle\rangle \mathsf{R}^{\mathsf{U}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\omega]$, 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 ω 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.[6] 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.

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.,

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

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

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

```
1  module clk
2  t : [0..MAX_TIME] init 0; rt : [0..MAX_RT] init INIT_RT; rt_upd : bool init false;
3      [] (t<MAX_TIME) & (turn=CLK_TURN) & (!rt_upd) -> (rt_upd'=true) & (rt'=totalTime);
4      [] (t<MAX_TIME) & (turn=CLK_TURN) & (rt_upd) -> (rt_upd'=false) & (t'=t+TAU) & (turn'=ENV_TURN);
5  endmodule
6  module environment
7  arrivals_total : [0..MAX_ARRIVALS] init MAX_ARRIVALS; arrivals_current : [0..MAX_INST_ARRIVALS] init 0;
8      [] (t<MAX_TIME) & (turn=ENV_TURN) & (t=0) -> (arrivals_current'=arrivals_total-X>=0?X:0) &
           (arrivals_total'=arrivals_total-X>=0?arrivals_total-X:0) & (turn'=CLK_TURN);
9          ...
10     [] (t<MAX_TIME) & (turn=ENV_TURN) & (t>0) -> (arrivals_current'=arrivals_total-X>=0?X:0) &
           (arrivals_total'=arrivals_total-X>=0?arrivals_total-X:0) & (turn'=SYS_TURN);
11 endmodule
```

Listing 2: Environment-controlled modules

service requests arriving at the system). There are two modules controlled by the environment player: (i) clk, which tracks and controls execution time (lines 1-6); and (ii) environment (lines 7-11), 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. The behavior of the environment is parameterized by the following constants:

— MAX_TIME defines the time frame for the system's execution ([0,MAX_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. 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).

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

— In the clk module: (i) t keeps track of execution time; (ii) 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); and (iii) rt_upd is an auxiliary variable used to keep track of whether rt has been updated during the current turn.
— In the environment module: (i) arrivals_total keeps track of the accumulated number of arrivals during execution; and (ii) 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 3-4): (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 3); (ii) Execution time variable t is increased one step (TAU) (line 4). After variable updates, clk yields control to environment module by updating the turn variable.

```
1  module target_system
2  s : [0..MAX_SERVERS] init INIT_SERVERS; f : [1..MAX_FIDELITY] init MAX_FIDELITY; counter:[−1..ENLIST_LATENCY]
       init −1;
3      [] (s<=MAX_SERVERS) & (turn=SYS_TURN) & (counter!=0) −> (turn'=CLK_TURN) &
           (counter'=counter>0?counter−1:counter);
4      [enlist_trigger] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=−1) & (ENLIST_LATENCY>0) −>
           (counter'=ENLIST_LATENCY!=0?ENLIST_LATENCY−1:counter) & (s'=ENLIST_LATENCY=0?s+1:s) &
           (turn'=CLK_TURN);
5      [enlist_trigger] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=−1) & (ENLIST_LATENCY=0) −> (s'=s+1) &
           (turn'=CLK_TURN);
6      [enlist] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=0) −> (s'=s+1) & (turn'=CLK_TURN) & (counter'=−1);
7      [discharge] (turn=SYS_TURN) & (s>MIN_SERVERS) & (counter=−1) −> (s'=s−1) & (turn'=CLK_TURN);
8      [df] dualFidelity & (turn=SYS_TURN) & (f>MIN_FIDELITY) & (counter=−1)−> (turn'=CLK_TURN) & (f'=f−1);
9      [if] dualFidelity &(turn=SYS_TURN) & (f<MAX_FIDELITY) & (counter=−1)−> (turn'=CLK_TURN) & (f'=f+1);
10 endmodule
```

Listing 3: System module

(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 8: 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. 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.

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 TAU * 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.

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

(i) s corresponds to the number of active servers; (ii) f is the fidelity level of the contents being served; and (iii) 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

```
1 formula uR = (rt>=0 & rt<=100? 1:0)+(rt>100&rt<=200?1+(−0.01)∗((rt−100)/(100)):0) ... +(rt>4000 ? 0:0);
2 rewards "rIU" (turn=SYS_TURN) : TAU∗(W_UR∗uR + W_UF∗uF +W_UC∗uC); endrewards
3 rewards "rEIU" (turn=SYS_TURN) : TAU∗(W_UR∗uER + W_UF∗uF +W_UC∗uC); endrewards
```

Listing 4: Utility functions and reward structures

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 4). 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 6), 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 5).
— Deactivation of a server, which is achieved through the discharge command (line 7), 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 8). 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 9), which has the opposite effect of decrease_f.

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 line 1 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, line 2 shows how a reward structure can be defined to compute a single utility value for any state by using utility preferences (defined as constant 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 4, line 3). In this case, the utility for performance during the latency period (encoded in formula uER) is computed analogously to uR, 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: $U_{rga} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$. 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: $U_{rma} \triangleq \langle\langle \mathsf{sys}, \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$.

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:

(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: $U_{ega} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rEIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$. For the specification of this property we employ the expected utility reward rEIU (Listing 4, line 3) 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

Table II: SMG model checking results for Znn.com

Latency (s)	Latency-Aware		Non-Latency-Aware				ΔU_{rga} (%)	ΔU_{rma} (%)
	U_{rga}/U_{ega}	U_{rma}	U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{rma}		
TAU	458.38	996.60	463.86	373.70	-19.43	960.40	18.47	3.63
2*TAU	452.82	996.60	463.79	278.57	-39.9	960.40	38.48	3.63
3* TAU	447.26	996.60	463.72	221.49	-52.23	960.40	50.47	3.63
4* TAU	441.70	996.60	463.64	182.64	-60.60	960.40	58.64	3.63

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 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*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 ($w_{\mathsf{U_R}} = 0.6$, $w_{\mathsf{U_F}} = 0$, $w_{\mathsf{U_C}} = 0.4$).

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

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 (ΔU_{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 (ΔU_{er}) in non-latency-aware adaptation, we can observe that 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 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 and fidelity ($w_{\mathsf{U_R}} = 0.5$, $w_{\mathsf{U_F}} = 0.3$, $w_{\mathsf{U_C}} = 0.2$).

— *Latency-aware Adaptation*. Figure 2 (left) 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 over the one for enlisting a new server. This is a clear example of how latency awareness can improve tactic selection by considering tactics that have moderate impact on utility (in this case reducing fidelity) compared to others, but due to their low latency can extract more utility over time than others with higher utility impact and latency (e.g., enlisting a new server).

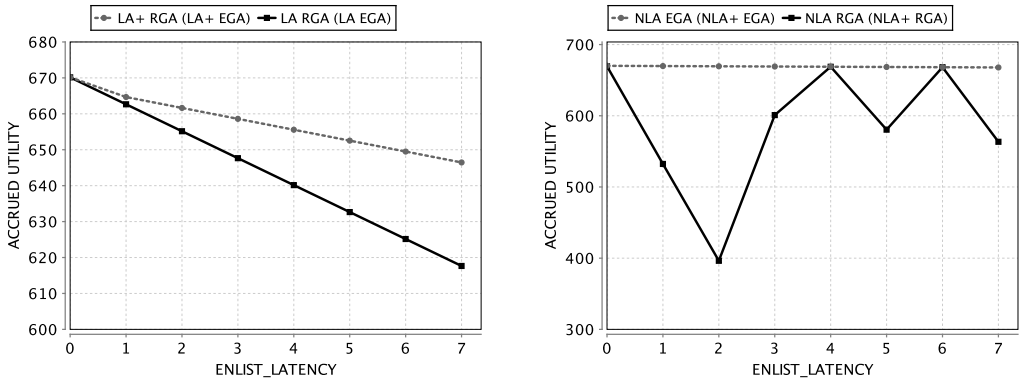


Fig. 2: Fidelity tactics impact on utility: LA (left) and NLA adaptation (right)

— *Non-latency-aware Adaptation*. Figure 2 (right) 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 explained because the new fidelity tactics never get selected by strategy synthesis. In particular, since the synthesis process is not aware of the latency of the tactic to enlist servers and only considers its positive net impact on utility (which always outweighs the impact on utility of reducing fidelity), the fidelity tactics never get selected despite being capable of extracting more system utility over time than the tactic to enlist servers. Note that the U_{rga} for both NLA and NLA+ describes a non-monotonic curve. This is due to the fact that under some particular conditions of the environment, latency-agnostic decision-making may coincidentally yield close-to-optimal or even optimal strategies with respect to guaranteeing a given level of utility in the

worst case, despite ignoring latency information. This can be observed in the figure, in which the values of U_{rga} match U_{ega} for latencies 4 * TAU and 6 * TAU).

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 both 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 estimated to happen in Δt seconds, and it could handle that with an additional server. If enlisting an additional server takes $\lambda > \Delta t$ seconds, then the desired configuration that has one additional server Δt 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. Although the input size to our algorithm is larger due to the additional states needed to keep track of adaptations in progress, it still has the same complexity.

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 λ seconds from the time a server is powered up until it can start processing requests, and ΔU_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 τ. In general, C is the set of possible configurations, and C_i is the ith configuration, for $i \in \{1 \dots |C|\}$. For our running example, $C = (S \times A \times F) \setminus \{(s, a, f) : S \times A \times F | s = 4 \wedge a \in A \setminus \{0\}\}$, where $S = \{1 \dots 4\}$ is the number of active servers in the system; $A = \left\{0 \dots \lceil \frac{\lambda}{\tau} \rceil\right\}$ 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 τ 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.[7] 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

[7]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.

$u_{i,t}$. 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 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 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} ← −∞ {assume infeasible configuration}
4:         n_{i,t} ← 0 {assume no next state}
5:         if t > 0 ∨ isReachableNow(C_i) then
6:             u_local ← τU(C_i, env(tτ))
7:             if t = H then
8:                 u_{i,t} ← u_local
9:             else
10:                {find the next best configuration after i}
11:                for j = 1 to |C| do
12:                    if u_{j,t+1} > −∞ ∧ isReachableFromConfig(C_i, C_j) then
13:                        u_projected ← u_local + u_{j,t+1}
14:                        if u_projected > u_{i,t} then
15:                            u_{i,t} ← u_projected
16:                            n_{i,t} ← j
17:                        end if
18:                    end if
19:                end for
20:            end if
21:        end if
22:    end for
23: end for
24: best ← arg max_i u_{i,0} {best starting configuration}
25: return C_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. Note, however, that this is not a limitation of the algorithm, but rather due to the example following the model of sequential tactic execution.

The algorithm uses two functions to determine the feasibility of possible adaptations. The function $isReachableNow(c')$ 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 Znn, 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 be impossible to reach any configuration other than the current one.

The function $isReachableFromConfig(c, c')$ 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 progress on a tactic with latency, if needed; and *(iii)* optionally 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 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 specification of the tactics for the Znn example, and further details about the generation of the reachability functions using Alloy are provided in the Appendix.

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 and Hornig 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 of 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 $\frac{1}{0.7}$.

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 τ 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 τ, 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.

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 λ 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.

Table III: Simulation results for Znn.com

Latency	Latency-Aware			Non-Latency-Aware				$\Delta U(\%)$			
(s)	min.	avg.	max.	min.	avg.	max.	$\hat{A}_{12}$	min.	10% quant.	avg.	max.
TAU	594.40	686.70	823.30	592.10	681.10	806.50	0.53	-1.99	-0.66	0.80	4.39
2*TAU	573.50	671.90	821.00	553.70	648.90	800.00	0.63	-2.28	0.80	3.41	9.80
3* TAU	557.40	660.60	814.20	533.00	625.70	777.50	0.69	-3.16	1.41	5.21	13.53
4* TAU	541.80	649.80	805.30	508.80	610.10	751.20	0.70	-3.50	0.89	6.02	16.05

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 ran 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 $3\frac{\lambda}{\tau} + 1$, 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,[8] 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.

[8]Note that this is a limitation of the $U(c, e)$ function used by the algorithm, and not a problem with the algorithm itself.

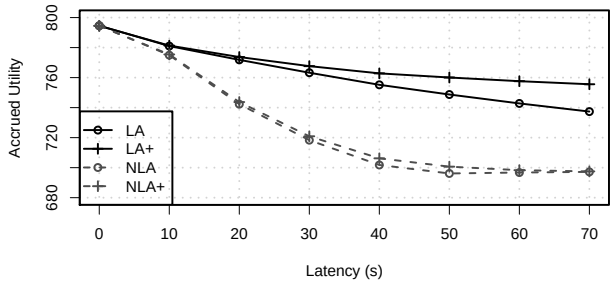


Fig. 3: Fidelity tactics impact on utility (simulation)

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 3. 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. ANALYZING LATENCY-AWARE ADAPTATION IN SYSTEMS WITH HUMAN ACTUATORS

Some classes of systems (e.g., safety-critical) and application domains can benefit by employing humans as system-level effectors to execute adaptations (e.g., in cases in which full automation is not possible, or as a fallback mechanism). However, the behavior of human participants is typically influenced by factors external to the system (e.g., training level, stress, fatigue) that determine their likelihood of successfully carrying out a particular task, or how long it will take. In this section, we describe how our SMG-based appproach can be used to compare latency-aware and latency-agnostic adaptation in the context of human-system-environment interactions, in which the latency of adaptation executed by humans has a major influence in the outcome of adaptations. We illustrate the approach in DCAS (Data Acquisition and Control Service) [Cámara et al. 2013], a middleware from Critical Software that provides a reusable infrastructure to manage the monitoring of highly populated networks of devices equipped with sensors.

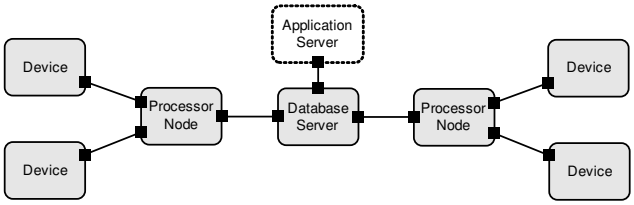


Fig. 4: Architecture of a DCAS-based system

The basic building blocks in a DCAS-based system (Figure 4) are:

— *Devices* are equipped with sensors to obtain data from the application domain (e.g., from wind towers, or solar panels). Each sensor has an associated *data stream* from which data can be read. Each type of device has its particular characteristics (e.g., data polling rate, or expected value ranges) specified in a *device profile*.
— *Processor nodes* pull data from the devices at a rate configured in the device profile, and dispatch this data to the database server.
— *Database server* stores the data collected from devices by processor nodes.

The main objective of DCAS is to collect data from the connected devices at a rate as close as possible to the one configured in their device profiles, while making an efficient use of the computational resources in the processor nodes. Specifically, the primary concern in DCAS is providing service while maintaining acceptable levels of performance, measured in terms of processed data requests per second (rps) inserted in the database, while the secondary concern is minimizing the cost of operating the system, which is directly proportional to the number of active processor nodes.

In situations in which new devices are connected to the network at run-time and all available resources in the set of active processor nodes are already being used, DCAS includes a scale out mechanism aimed at maintaining an acceptable performance level by dynamically activating new processor nodes, according to the demand determined by the new workload and operating conditions. DCAS scale out is a *manual process* carried out by a human operator, who is notified by the system whenever a new processor node must be deployed. This is a slow and demanding process in which a new processor node must be manually deployed, and devices re-attached across the different already active processor nodes to optimize the performance of the system, according to the particular situation.

Note that in this scenario, the influence of latency on the outcome of adaptation differs from the case of Znn.com, since shorter times to execute adaptations or increasing levels of stress in the operator (e.g., caused by several subsequent requests to carry out tasks within a short timespan) may negatively affect the impact of adaptation tactics on the qualities of the system (e.g., due to sub-optimal reattachment of devices across different processor nodes).

To compare latency-aware and non-latency-aware adaptation in scenarios like the one described above, we modeled a DCAS scale out scenario as a SMG in which the focus is on the interactions between the adaptation requests issued by the system, and their execution as carried out by a single human operator. Hence, in our DCAS SMG, the environment player is neutral, and only keeps track of the execution time.

The system player controls two processes: (i) adaptation_manager models the DCAS adaptation manager, including a single tactic (addPN) that requests the deployment of a processor node to the human operator and can be triggered non-deterministically at each time step of the game; and (ii) target_system models the behavior of the target system, including the human operator. The system process includes state variables to represent the performance and cost of the system, as well as a tactic latency counter to keep track of the progress of the tactic, and a queue representing requests to deploy processor nodes. Figure 5 shows the logic for the turn of the system player in the SMG:[9]

— If the DCAS adaptation manager triggers the adaptation tactic (1):
  — If the tactic latency counter is disabled (i.e., the human operator is available), the counter is enabled (initialized with value TAU) if the tactic's latency is not zero

[9]Although the target_system process is encoded following the pattern shown in Listing 3, we present here a graphical representation of the logic encoded for the sake of clarity.

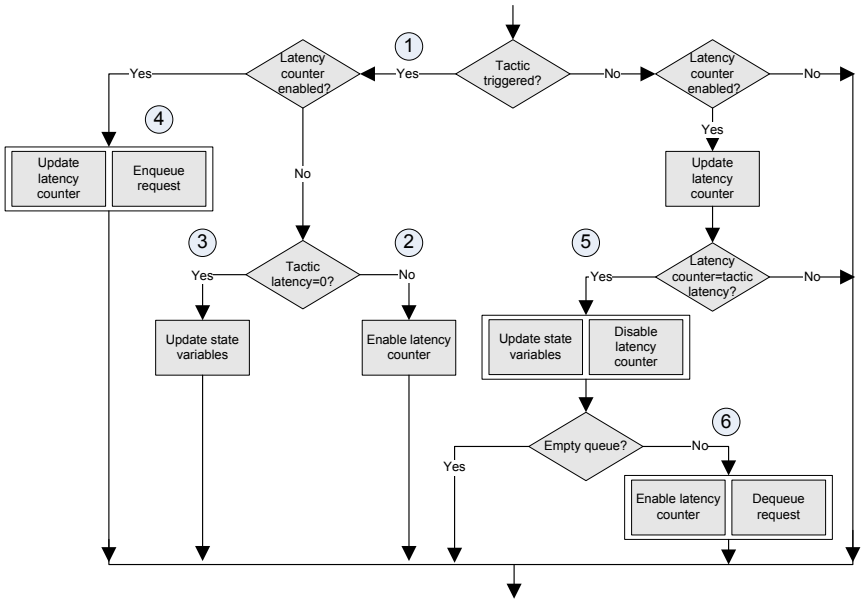


Fig. 5: System player turn logic in DCAS SMG

(2). If the tactic has no latency, the effects of the tactic's execution are updated in the state variables that encode system qualities for performance and cost (3).

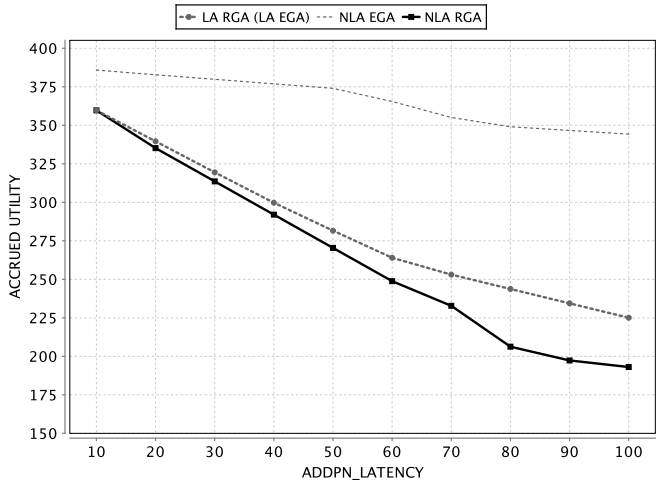
— If the tactic's latency counter is enabled (i.e., the operator is already busy deploying a processor node), the request for tactic execution is enqueued, and the latency counter is updated by increasing it in TAU (4).

— If the adaptation tactic is not triggered, the latency counter is updated. If the latency counter value has already matched the tactic's latency, the state variables encoding system qualities are updated to reflect the completion of the tactic's execution, and the latency counter is disabled (5). Next, if the queue with requests for tactic executions is not empty, a request is dequeued, and the latency counter is enabled again (6). This part of the logic, along with step (4) models the sequential execution of tactics that the human operator carries out when she receives requests from the adaptation manager while busy.

In our game, we also encode a penalty in the increment of performance upon completion of the tactic which is directly proportional to the amount of requests made by the adaptation manager during the latency period of the tactic. This models an increment in the stress level of the operator, who is more likely to perform a sub-optimal reattachment of devices when deploying the processor node under more pressing workload conditions.[10]

Our experiments compare the results for the utility extracted from the system in the worst-case scenario by a latency-aware *vs.* a non-latency-aware version of the system player, for a model of DCAS scale out that represents an execution of the system during 500 minutes. The models consider up to a maximum of 5 processor nodes, out of which one is initially active. The period duration TAU is set to 10 minutes, and for each

[10]Although there are different possibilities when modeling the influence of requests on the operator while busy (e.g., modifying the latency of the tactic according to a probabilistic distribution), we chose to employ a simple penalty in impact for the sake of simplicity.



Latency (minutes)	LA	NLA		ΔU_{rga} (%)
	U_{rga}/U_{ega}	U_{ega}	U_{rga}	
TAU	359.7	385.8	359.7	0
2 * TAU	339.6	382.8	335.1	1.32
3 * TAU	319.4	379.8	313.6	1.81
4 * TAU	299.7	376.9	292.0	2.56
5 * TAU	281.6	374.0	270.4	3.97
6 * TAU	264.0	365.4	248.8	5.75
7 * TAU	253.0	355.0	232.8	7.98
8 * TAU	243.7	349.0	206.2	15.38
9 * TAU	234.4	346.0	197.3	15.82
10 * TAU	225.0	344.0	193.0	14.22

Fig. 6: Model checking results for DCAS scale out SMG

version of the model, we compute the results for ten variants with different latencies for the activation of servers of up to 10 * TAU minutes.We employ the following utility functions and preferences: $\mathsf{U_P}(rps) = \dfrac{rps}{rps^{max}}$ maps high levels of processed requests per second inserted in the database (rps) to high utility by dividing it by the maximum level of achievable rps in the system rps^{max}, which is computed according to the number of devices in the system and the data polling rates configured in their device profiles. In contrast, $\mathsf{U_C}(pn) = 1 - \dfrac{pn}{pn^{max}}$ maps higher costs (derived from the number of active processor nodes, pn) to lower utility values. Cost utility becomes 0 when pn reaches the maximum number of available nodes pn^{max}. The utility preferences used for our experiments are $w_{\mathsf{U_P}} = 0.8$ and $w_{\mathsf{U_C}} = 0.2$.

The results shown in Figure 6 indicate that latency-aware outperforms non-latency-aware adaptation in all cases. In particular, latency-aware adaptation is able to guarantee an increment in utility extracted from the system (ΔU_{rga}) that ranges between approximately 0 and 15%, increasing progressively with higher tactic latency. This results from the fact that non-latency-aware adaptation is unable to factor in the penalty associated with requesting tactic executions before the completion of an ongoing tactic execution, and follows the strategy of requesting the activation of all processor nodes at the beginning of the system's execution.

6. 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 of 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 run-time 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., parallelizing tactic executions). We will also generalize the algorithm to consider the uncertainty of the predictions of the environment.

APPENDIX

The algorithm for proactive latency-aware adaptation presented in this paper relies on the functions $isReachableNow(c')$ and $isReachableFromConfig(c, c')$ to determine the feasibility of possible adaptations. As described in Section 4.1.1, we used formal specification of adaptation tactics in Alloy, and the Alloy analyzer to compute these functions off-line. The result is encoded as simple lookup tables for runtime use.

The basic definitions of the specification used to compute the reachability functions is shown in Listing 5. 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 6. 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

```
1  open util/ordering[S] as servers
2  open util/ordering[A] as progress
3  open util/boolean
4  sig S {} // the different number of active servers
5  sig A {} // the different levels of progress of the add server tactic
6  sig C { // each element of C represents a configuration
7      s : S, // the number of active servers
8      a : A, // the progress of the add server tactic
9      f : Bool // fidelity level
10 }
11 fact uniqueInstances { all disj c, c2 : C | !(c2.s = c.s and c2.a = c.a and c2.f = c.f) }
```

Listing 5: Alloy model of configuration reachability: basic definitions

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 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 6 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 7 is used to generate the reachability functions. Each predicates 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, $3\tau + 1 = 4$ levels of progress for the tactic, and two fidelity levels, for a total of 32 configurations.[11] 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.

REFERENCES

R. Alur, T. A. Henzinger, and O. Kupferman. 2002. Alternating-time temporal logic. *J. ACM* 49, 5 (2002).

[11] Even though some of these configuration 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.

```
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'.a = progress/last implies c'.s = servers/next[c.s] else c'.s = c.s
    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'] }
```

Listing 6: Alloy model of configuration reachability: tactics and functions

```
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
```

Listing 7: Generation of configuration reachability tables in Alloy

A. Arcuri and L. Briand. 2012. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* (2012).

A. Bianco and L. de Alfaro. 1995. Model Checking of Probabalistic and Nondeterministic Systems. In *FSTTCS (LNCS)*, Vol. 1026. Springer.

V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel. 2013. Controller Synthesis: From Modelling to Enactment. In *ICSE*. IEEE.

R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. 2011. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Software Eng.* 37, 3 (2011).

J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura. 2013. Evolving an adaptive industrial software system to use architecture-based self-adaptation. In *SEAMS*. IEEE.

J. Cámara, G. A. Moreno, and D. Garlan. 2014. Stochastic game analysis and latency awareness for proactive self-adaptation. In *SEAMS*. ACM.

T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaistis. 2013a. Automatic Verification of Competitive Stochastic Systems. *Form Method Syst Des* 43, 1 (2013).

T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaistis. 2013b. PRISM-games: A Model Checker for Stochastic Multi-Player Games. In *Proc. of TACAS'13 (LNCS)*, Vol. 7795. Springer.

S.W. Cheng, D. Garlan, and B. Schmerl. 2009a. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*. IEEE.

S-W. Cheng and D. Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012).

S-W. Cheng, V. Poladian, D. Garlan, and B. Schmerl. 2009b. Improving Architecture-Based Self-Adaptation through Resource Prediction. In *SEfSAS*. LNCS, Vol. 5525. Springer.

R.M. Chiulli. 1999. *Quantitative Analysis: An Introduction*. Taylor & Francis.

V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. 2011. Automated Verification Techniques for Probabilistic Systems. In *SFM (LNCS)*, Vol. 6659. Springer.

A. Gambi, D. Moldovan, G. Copil, H-L. Truong, and S. Dustdar. 2013. On estimating actuation delays in elastic computing systems. In *SEAMS*. IEEE.

A. Gandhi, M. Harchol-Balter, and R. Raghunathan. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4 (2012).

D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (2004).

P. Goldman, R, D. J. Musliner, and K.D. Krebsbach. 2003. Managing Online Self-adaptation in Real-Time Environments. In *Self-Adaptive Software: Applications*. LNCS, Vol. 2614. Springer.

J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. 2008. A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing. LNCS, Vol. 5377. Springer.

W. Van Der Hoek and M. Wooldridge. 2003. Model Checking Cooperation, Knowledge, and Time - A Case Study. In *Research in Economics*.

Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. The MIT Press.

J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003).

J. Kramer and J. Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *FOSE*.

S. Kremer and J-F. Raskin. 2001. A Game-Based Verification of Non-repudiation and Fair Exchange Protocols. In *CONCUR 2001*. LNCS, Vol. 2154. Springer.

M. Kwiatkowska, G. Norman, and D. Parker. 2008. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review* 35, 4 (2008), 14–21.

M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV (LNCS)*, Vol. 6806. Springer.

J. Liu, B. Priyantha, T. Hart, H.S. Ramos, A. Loureiro, and Q. Wang. 2012. Energy efficient GPS sensing with cloud offloading. In *SenSys*. ACM.

L. Maatta, J. Suhonen, T. Laukkarinen, T.D. Hamalainen, and M. Hannikainen. 2010. Program image dissemination protocol for low-energy multihop wireless sensor networks. In *SoC*. IEEE.

A. Metzger, O. Sammodi, and K. Pohl. 2013. Accurate Proactive Adaptation of Service-Oriented Systems. In *ASAS*. LNCS, Vol. 7740. Springer.

D. Musliner. 2001. Imposing Real-Time Constraints on Self-Adaptive Controller Synthesis. In *Self-Adaptive Software*. LNCS, Vol. 1936. Springer.

G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. 2002. Formal Analysis and Validation of Continuous Time Markov Chain Based System Level Power Management Strategies. In *HLDVT 2002*. IEEE.

P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A.L. Wolf. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intell. Syst.* 14 (1999), 9. Issue 3.

I.D. Paez Anaya, V. Simko, J. Bourcier, N. Plouzeau, and J-M. Jézéquel. 2014. A Prediction-Driven Adaptation Approach for Self-Adaptive Sensor Networks. In *SEAMS*. ACM.

V. Poladian, D. Garlan, M. Shaw, B. Schmerl, and J. Sousa. 2007. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In *SASO*.

B. Schmerl, J. Cámara, G. A. Moreno, D. Garlan, and A. Mellinger. 2014. *Architecture-Based Self-Adaptation for Moving Target Defense*. Technical Report CMU-ISR-14-109. Carnegie Mellon University.

A. Varga and R. Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Simutools*. ICST.

C. Wang and J-L Pazat. 2012. A Two-Phase Online Prediction Approach for Accurate and Timely Adaptation Decision. In *SCC*.

X. Zhang and C-H. Lung. 2010. Improving Software Performance and Reliability with an Architecture-Based Self-Adaptive Framework. In *COMPSAC*.