# Hybrid Planning For Self-Adaptation

Ashutosh Pandey and David Garlan
School of Computer Science
Carnegie Mellon University, Pittsburgh, PA
ashutosp@cs.cmu.edu · garlan@cs.cmu.edu

***Abstract*—Self-adaptive software systems make decisions at run time that seek to change their behavior in response to faults, changing environments and attacks. Therefore, having an appropriate decision making approach to find an adaptation strategy is critical to successful self-adaptation. Ideally, when an adaptation is triggered, one would like to have a decision making approach that is both quick and finds an optimal adaptation strategy. However, often designers have to compromise between an approach that is quick to find an adaptation strategy and an approach that is slow but finds an optimal adaptation strategy. To deal with this trade-off between quick decision making and finding an optimal adaptation strategy, this position paper proposes a *hybrid planning* approach that combines more than one decision making approach to bring their benefits together. As an initial proof of concept, the paper presents the results of a small experiment to demonstrate the potential of a hybrid planning approach in dealing with this trade-off.**

## I. Introduction

A typical control loop in many self-adaptive systems has four fundamental computational elements: Monitoring-Analysis-Planning-Execution (MAPE), where planning is responsible for determining a repair strategy for self-adaptation. For the planning element various approaches, including automated planning, have been suggested by the research community to determine a repair strategy [11] [25] [4].

Often there is a trade-off between determining a repair strategy quickly vs. determining an optimal repair strategy. Looking at the spectrum of proposed approaches, at one end of the spectrum are approaches that determine a repair strategy quickly but the strategy could be sub-optimal. At the other extreme are approaches that are slow but the identified strategy is optimal.[1]

For instance, in the Rainbow framework [5], the approach to determine a repair strategy is quick because the repair strategy is not generated at run time, but rather selected from a fixed repertoire of predefined strategies based on applicability conditions of a strategy and the goals of the system.[2] However, the selected strategy might not be optimal because it is impossible to have an optimal strategy beforehand for all unforeseen scenarios. At the other extreme, approaches such as AI planning, which generate the repair strategies at run time [11], could be slow but can come up with a nearly optimal strategy, since the planning element dynamically generates a repair strategy based on the current state of the executing system and possibly its external environment.

Under some circumstances, such as planning for the shortest path to the destination for a robot navigating in a stable environment, finding an optimal path could be the primary concern rather than planning quickly. In contrast, there are situations, such as a self-driving car navigating on a busy street, where a quick but sub-optimal decision to stop the car, in order to avoid a collision, might be preferred over finding an optimal path that takes longer to plan.

However, there are systems such as Amazon Web Services (AWS), that not only need to find the adaptation plans quickly, but also require the plans to be optimal. As per its SLA, Amazon Web Services(AWS) is required to maintain an up-time percentage of at least 99.95% in any monthly billing cycle, even though there might be other quality concerns such as cost minimization.[3] In other words, the utility for such systems is dominated by certain quality constraints: the utility would drop drastically if any of these quality constraint(s) is violated.

For systems such as AWS, in case of a failure a rapid response is required to keep the system in a desirable state (in this case, maintaining availability). However, to maximize overall long-term utility of the system, the adaptation plan should ideally also be optimal in terms of metrics such as operating cost. In such cases, the requirements to plan quickly and to find optimal plans, are often conflicting in nature, since finding optimal plans typically requires dealing with uncertainty, which significantly increases the planning time.[4] Therefore, selecting a single decision making approach from the spectrum of various approaches, would either be slow to react or find sub-optimal adaptation plans.

Ideally we would like an approach to self-adaptation that combines the best of both worlds: providing plans quickly when timing is critical, while allowing optimal plans to be generated when the system has sufficient time to do so. In this position paper, we propose a novel *hybrid planning* approach that does just that. The key idea behind hybrid planning is to combine more than one decision making approaches to obtain the benefits of each. For instance, a fast decision making approach could be used in combination with an optimal but slow decision making approach: the fast planning approach provides plans quickly and the slow planning approach provides optimal plans. This idea of hybrid planning is akin to human decision making: depending upon factors such as available planning time, humans apply different levels of deliberation while making real life decisions [27].

Even though hybrid planning seems to be an attractive idea, however, a successful application of the approach needs to address multiple research challenges. The first challenge is to

[1]Optimal in terms of an appropriately defined notion of utility.

[2]This approach is similar to many commercial systems, which have a fixed set of responses to problems, such as rebooting servers, redistributing stored information, and adding new servers.

[3]https://aws.amazon.com/ec2/sla/

[4]AWS-like systems could have different kinds of uncertainties such as uncertainty in the client request arrival rate, and hardware failure.

identify appropriate (i.e., applicable in the operating domain) decision making approaches that could be used in combination to instantiate a hybrid planner. Another key challenge is to figure out the criteria for knowing when to invoke each of the constituent planners and how to coordinate their plans.

In this paper, we provide one possible solution to these challenges in the context of an experimental AWS-like simulated cloud-based self-adaptive system. To instantiate the fast and the slow planning approach, two different automated planning technologies are used: deterministic planning as a fast approach, and Markov Decision Processes (MDP) planning [22] as an optimal, but a slow approach. Specifically, in case of a constraint violation (such as non-availability or high response time) deterministic planning ignores uncertainty in the client request arrival rate by assuming a constant request arrival rate and finds a tactic that can be executed immediately in the process of satisfying the constraint. However, while the tactic is being executed, an MDP planner takes uncertainty in the request arrival rate into account and finds a more-nuanced, and hence nearly optimal, MDP policy.[5]

In our experiments, we used the combination of deterministic planning and MDP planning since seamless transfer of execution is possible from a plan generated by the deterministic planning to a policy generated by the MDP planning. As explained later, if deterministic and MDP planning have the same initial state, once the MDP policy is ready, it takes over the plan execution from the deterministic plan in any of the future states and ensures that an optimal MDP policy is executed thereafter.

Our initial experimental results show that in the context of certain cloud-based self-adaptive systems, a hybrid planning approach performs better than either of the two planning technologies used in isolation. The hybrid planning performed better because it can not only plan quickly, and hence provide rapid response to constraint violations, but also finds repair plans that are optimal in the long run by taking into consideration uncertainty and other quality attributes such as cost – hence combining the benefits of both, deterministic and MDP planning.

The rest of the paper is organized as follows: in Section II, we introduce a motivating example that will be used throughout the paper as a reference problem. Section III provides background on automated planning. In Section IV, we explain hybrid planning approach in detail. Section V presents the initial experimental results. Section VI presents related work in the area of self-adaptive systems and automated planning. Finally, Section VII concludes with a discussion on the possible future directions.

## II. Motivating Example

Suppose we have a cloud-based web application, which has a typical three layered architecture: a presentation layer, an application layer, and a database layer. When the application layer receives a client request from the presentation layer, it processes the request and exchanges data with the database layer if required. We will assume that the system has servers of different capacity:[6] however, the cost of a server increases with increase in capacity. The workload on the system depends on the request arrival rate, which is uncertain as it depends on the external demand.

To increase profitability, the system needs to minimize operating cost and maximize revenue. To bring down the operating cost, the system needs to minimize active servers. To achieve this, the system has an adaptation tactic, *removeServer*, to deactivate a surplus server.

Since higher perceived user response time results in revenue loss [24], it is desirable to maintain the response time for user requests below some threshold, say $T$. The system can manage the response time either by adding more servers (using *addServer* tactic) or by controlling optional content, such as advertisements and recommendations. For each user request, there is some mandatory content along with some optional content presented to the clients. The optional content helps to generate revenue; however, it also uses additional bandwidth, which increases the response time. The system uses a *brownout* [21] mechanism to control the optional content through a dimmer variable. Value of the variable varies in the range [0..1], which represents the probability of having the optional content in a response. The *increaseDimmer* and *decreaseDimmer* tactics are used to increase and decrease the value of the dimmer variable respectively. In case of a high workload, the number of responses having optional content can be reduced by decreasing the dimmer value; however, it can also bring down the revenue generated through advertisements [20].

Since the system has servers of different capacity, a round-robin strategy for assigning client requests to active servers would not be efficient. The number of client requests delegated to a server depends on its capacity. The load-balancer uses queueing theory [29] to decide on the optimal load-distribution among the active servers. To distribute the load efficiently, there is a tactic, *divert_traffic*, which helps the load-balancer in managing the percentage of client requests, assigned to each server.

We assume there is a penalty, say $P$, for each request having a response time above the threshold. Therefore, in case of a high average response time, the system needs to react quickly either by adding servers or decreasing the dimmer value. However, once response time is under control, system should execute adaptation tactics to bring down the operating cost i.e., maximize overall long term utility.

The goal of the system is to maximize the utility, which depends on the revenue generated, the penalty for response time above the threshold, and the cost of active servers. If system runs for duration L, the utility function would be defined as:

$$U = R_O x_O + R_M x_M - P x_T - \sum_{i=1}^{n} C_i \int_0^L s_i(t)dt \qquad (1)$$

where $R_O$ and $R_M$ is revenue generated by a response with optional and mandatory content respectively; $x_O$, $x_M$, and $x_T$ are the number of requests with optional content, only mandatory content, and having response time above the threshold; $C_i$ is the cost of server type $i$ and $s_i$ is the number of active servers of type $i$; $n$ is number of different types of servers.

## III. Background

AI researchers have developed various automated planning approaches [23]. These approaches can be categorized into two

[5]The plan generated by MDP planning is known as a MDP policy.

[6]As measured by average number of requests handled per second.

broad categories: deterministic planning, and planning under uncertainty. Deterministic planning is applicable to deterministic domains when there is no uncertainty in action outcomes or observations of the underlying state.

Planning under uncertainty includes approaches such as MDP planning [22], and Partially Observable Markov Decision Processes (POMDP) [28]. These approaches handle uncertainty in the planning domain, required to generate optimal plans in terms of maximizing the expected reward. However, since uncertainty in the planning domain drastically increases the state space, these approaches are generally much slower in comparison to deterministic planning approaches.[7]

In our experiments, MDP planning is used as an approach that is slow but generates nearly optimal plans. Specifically, MDP planning helps in dealing with uncertainty in the request arrival rate that helps in generating an optimal policy, which will maximize expected utility (reward) for a predefined notion of utility.

Due to uncertainty in action outcomes, if an adaptation tactic is applied to a state, the system can end up in one of the many possible states. In such situations, a linear plan, from the initial state to a goal state, can fail because the system might end up in an unanticipated state, not covered by the linear plan. However, this is not a problem with MDP planning because it generates an MDP policy, which is a state-action pair for all reachable states from the initial state. A single state-action pair in an MDP policy indicates the next action that needs to be taken for the corresponding state to maximize expected utility.

## IV. Approach

The key idea behind hybrid planning to combine more than one decision making approaches to obtain their benefits. For instance, in our cloud-based system, to deal with the conflicting requirements of planning quickly (in case of a response time above the threshold) and finding an optimal plan, we combine a fast but sub-optimal planning approach with an optimal but slow planning approach. The intuition is: (1) using a slow planner alone introduces a gap in decision-making time that can lower utility; (2) using a fast planner alone provides a quick response but may not be nuanced enough to provide an optimal solution. Therefore, the hybrid planning approach uses a fast planner to handle an immediate problem, but simultaneously uses a slow planner to provide a nearly optimal solution.

To formulate a hybrid planner in our experiments, we instantiated a fast and a slow planning approach, using two different automated planning technologies: deterministic planning and Markov Decision Processes (MDP) planning respectively. To coordinate the two planners we used the following logic: In case of response time above the threshold, deterministic planning provides an adaptation plan quickly since it ignores uncertainty in the request arrival rate to reduce the planning state-space, which eventually reduces the planning time. However, while the deterministic plan is under execution, MDP planning is executed in the background. Our MDP specification models uncertainty in the request arrival rate as uncertainty in the outcome of actions made by the external environment. As explained later, we use a time-series predictor to anticipate the future average interarrival time (inverse of average request arrival rate) between two consecutive requests.

[7]In the deterministic domain, an action when applied to state $s_0$ would result in a certain state $s_1$. However, if there is uncertainty in the action outcomes, an action when applied to state $s_0$ could result in one of the many possible states, thereby increasing the state space, and hence increasing the planning time.

However, this hybrid planning approach will only be effective when a seamless transfer of execution is possible from a plan generated by the deterministic planner to a policy generated by the MDP planner. There are two properties of MDP policies that improve the chances of a smooth transition from the deterministic plan to the MDP policy. First, the MDP policy defines a state-action pair for all reachable states from the initial state. The second property is the Markovian nature of its state transition model, which means the action specified for a particular state in the MDP policy solely depends on that state rather than any of the previous states. Therefore, if deterministic and MDP planning have the same initial state, once the MDP policy is ready, more likely it takes over the plan execution from the deterministic plan in any of the future states and ensure that optimal policy is executed thereafter.

However, if predictions for the interarrival time, used for MDP planning, are not correct then the MDP policy can fail to take over the plan execution. To explain this further, suppose an MDP policy is available at time $t_r$ but the predicted value (used for MDP planning) for the interarrival time at $t_r$ is not the same as the actual value, then the current state would not be found in the MDP policy. Therefore, the MDP policy would not be applicable at time $t_r$. In such cases, MDP planning is triggered using the new current state as its initial.

### A. MAPE-K Loop

To bring self-adaptive capability to our cloud-based system, we implemented a MAPE-K loop [19], consisting of a knowledge base, and four components: monitoring, analysis, planning, and execution. The knowledge base maintains information including the architectural structure of the system [26], systemic properties such as average response time and operating cost, tactics in progress, the environment model, number of active servers, dimmer settings, and the portion of the workload assigned to different servers.

**Algorithm 1** hybrid_planning function returns the list of tactic(s)

```
Require: $S_{curr}$, $\pi$, $R_{curr}$, $R_{threshold}$
 1: List $tactics$ ←get_tactics($S_{curr}$, $\pi$)
 2:
 3: if $tactics$ = null then
 4:     Thread $thread \leftarrow new$ Thread()
 5:     $thread$.run(MDP_planning())
 6:
 7:     if $R_{curr} > R_{threshold}$ then
 8:         $tactics$ ←deterministic_planning()
 9:
10: return $tactics$
```

The monitoring component observes the request arrival rate and the response time perceived by the clients. Our MAPE-K loop merges the analysis component and the planning component into a single component, called the analysis-planning component.

This component evaluates the monitored system periodically, with fixed time period $\tau$, to figure out if there is a chance for adaptation defined as a opportunity to improve utility.

### B. Hybrid Planning Algorithm

At the beginning of each evaluation cycle, to figure out if adaptation is required, the analysis-planning invokes the function *hybrid_planning* shown as Algorithm 1. The inputs to *hybrid_planning* algorithm are: state $S_{curr}$, which includes the state of system and environment; MDP policy $\pi$ if it exists; response time $R_{curr}$; and upper threshold, $R_{threshold}$, for the response time. As an output, the algorithm returns a list of tactics that need to be executed at the current point to improve utility.

**Algorithm 2** get_tactics function returns the list of tactic(s)

```
Require: $S_{curr}, \pi$
1: if $\pi$ = null or $\pi$.find($S_{curr}$) = false then
2:     return null
3: else
4:     return $\pi$.get_tactics_from_policy($S_{curr}$)
```

The *hybrid_planning* algorithm calls the function *get_tactics* listed as Algorithm 2, which takes state $S_{curr}$ and MDP policy $\pi$ as input. If the policy exists in the knowledge base then $S_{curr}$ is searched in the policy. If $S_{curr}$ is found in the policy, *get_tactics* (Algorithm 2: line-4) returns the tactics for all states, starting from $S_{curr}$, within the evaluation cycle. Since our MDP specification supports concurrent execution of tactics [17], these tactics are triggered simultaneously at the beginning of that evaluation cycle. The returned list of tactics could be empty, which implies further increase in utility is not possible: adaptation is not required. However, if an MDP policy does not exist or $S_{curr}$ is not found in the MDP policy, *get_tactics* function returns *null* indicating that re-planning is required (Algorithm 2: line-2).

Since MDP planning is generally slow, the *hybrid_planning* algorithm does not wait for the MDP planning to complete; instead, it assigns MDP planning to a newly created thread (Algorithm 1: line 4-5). However, if a constraint is violated, deterministic planning is triggered for a quick response (Algorithm 1: line 7-8). Here the assumption is that the deterministic planning is quick compared to MDP planning.

### C. MDP Planning

We use the PRISM model-checker [25] as an MDP solver. Our MDP specification models each tactic as a separate module to support concurrency [17]. To handle uncertainty in request arrival rate, we model the environment as a probability tree, where the root node represents the current average interarrival time between two consecutive requests and its children represent realizations conditioned on the parent, with the edges representing the probability of the child realization given that the parent was realized. We use a time-series predictor to anticipate the future average interarrival time.

As we predict farther into the future, the uncertainty in predictions rises, which could lead to a sub-optimal policy. Therefore, MDP planning is done for finite horizon, say $N_{horizon}$, which means that when MDP planning is triggered, we predict the request arrival rate up to $N_{horizon}$ evaluation cycles and generate a policy for that horizon. This period of $N_{horizon}$ evaluation cycles (i.e., $N_{horizon} \times \tau$) is known as a planning cycle.

In our MDP planning specification, the execution of actions is done at the granularity of evaluation period $\tau$. However, when MDP planning is triggered by the *hybrid_planning* algorithm, completion time for the planning process might not necessarily overlap with the start of an evaluation cycle. In cases when the MDP policy is ready in the middle of an evaluation cycle, the execution of the policy begins only at the start of the next evaluation cycle.

*1) Storing Policy:* Since MDP planning is slow, once the MDP policy is generated, it is stored for successive evaluation cycles within the look-ahead horizon. If the system and the environment behave as expected, the policy will be applicable at the beginning of each evaluation cycle within the look-ahead horizon. The policy needs to be re-generated in either of two cases: the planning horizon is over, or there is an unexpected change in environment leading to a state not covered by the MDP policy.

*2) State matching heuristic:* At the beginning of each evaluation cycle, the current state, consisting of state variables representing the system and environment, is searched in the policy to determine which tactic(s) is to be executed for adaptation. However, in our approach, the prediction discretizes the values of interarrival time, so it is very likely that the actual value in the current state is not one of the discrete values. Therefore, current state would not exist in the MDP policy.

To deal with this problem, we use a heuristic to match the current state to a state in the MDP policy. The current state is matched to a state that meets three criteria: (1) all state variables (except average interarrival time) have same value; (2) among the states meeting criterion 1, the value of average interarrival time for the matched state should be closest to the current average interarrival time; and (3) once the first two criteria are met, the difference between the values of the average interarrival time in the current state and the matched state should be within 10% of the current average interarrival time.

### D. Deterministic Planning

For the deterministic planning, we use a modified version of MDP specification by ignoring uncertainty. To transform the MDP specification into a deterministic planning specification, probabilistic environment model is ignored. Instead, it is assumed that request arrival rate will remain fixed until the next planning cycle. Once environmental uncertainty is removed, the state space reduces significantly, resulting in an exponential reduction in planning time. In our experiments, the MDP planning state space of about 2 million states is reduced to about 23, 000 simply by ignoring environmental uncertainty; the planning time reduced from (approximately) 40 seconds to less than a second.

## V. Proof of Concept

To validate the hybrid planning approach, we conducted experiments in a simulated setup using a discrete event simulator, *OMNeT++*.[8] We implemented various architectural components

[8]https://omnetpp.org/

such as a load-balancer, and different types of servers in the simulator. Besides the fundamental functionality, these components also have logic to support the tactics described earlier. For instance, servers allow increments/decrements of dimmer values and the load-balancer allows addition/removal of servers.

## A. Experiment Setup

The system has three types of servers: A, B and C. Among these three types, server type-C is the costliest but has the highest capacity in terms of its ability to handle requests per second: server type-A is the cheapest but has the least capacity among the three types of servers. Therefore, we assign a cost per minute to be \$0.5, \$0.7, and \$1 for server type-A, type-B, and type-C respectively. The capacity of server type-A is 50 when serving with the optional content and 150 without serving the optional content; the capacity of server type-B is 130 when serving with the optional content and 200 without serving the optional content; and the capacity of server type-C is 150 when serving with the optional content and 300 without serving the optional content.

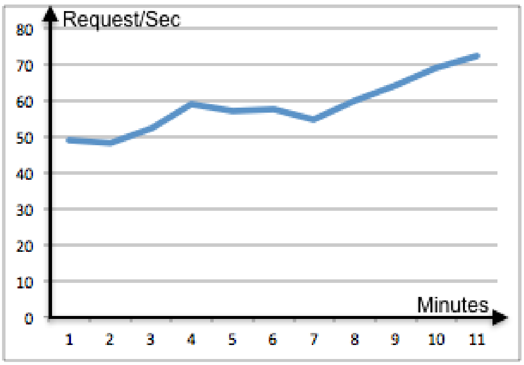

Fig. 1. Request arrival rate

Suppose the cost of a server can be covered by the revenue of handling 1/10 of its maximum capacity with optional content and the revenue of handling 2/3 of its maximum capacity without optional content. If the server cost per minute is C, capacity with optional content is $c_O$ and without optional content is $c_M$, then the revenue for a server with optional content would be $R_O = \frac{10}{c_O}C$ and without optional content would be $R_M = \frac{3/2}{c_M}C$. For each request having response time above the threshold value of 1 second, there is a penalty of -3 units.

The evaluation period, $\tau$, for the experiment is configured as 1 minute. For the experiments, we have 3 dimmer levels and 1 server of each type i.e. the total number of servers is 3. We assume a fixed boot-up time of 2 minutes for each server type. Since at a given point of time, we would have maximum 2 inactive servers and each server requires 2 minutes of boot up time, our look-ahead horizon for MDP planning is 5 minutes. This heuristic gives a long enough horizon to go from 1 active server to 3 active servers plus 1 additional evaluation cycle to observe the resulting utility.

For a realistic workload pattern, we used a request arrival trace of from the World Cup '98 website [18]. As shown in Fig. 1, the trace includes a considerable load increase from 48 requests per minute to 73 requests per minute. This trace is scaled down so the request arrival rate does not exceed the capacity of the simulation setup. Of the 11 minutes, the last minute is simply an observation point, therefore, the remaining time is divided into 2 planning cycles of 5 minutes each. This trace contains time-stamps representing interarrival rate between two client requests. To simulate the clients, we send requests to the load-balancer with the interarrival rate indicated by the trace.

We conducted the experiments on a Ubuntu 14.04 virtual machine having 8GB RAM and 3 processors at 2.5 GHz. The state space for the MDP planning varies approximately between 1.6 million and 2.2 million and the MDP planning time varies between 35-45 seconds. The state space for the deterministic planning is about 23,000 states with planning time less than a second, which is considered negligible in the experiments.

## B. Results

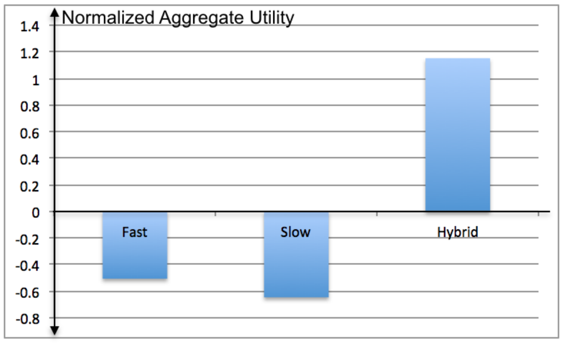

Fig. 2. Normalized aggregate utility for different approaches

We look at the aggregate utility, which is the sum of utility accumulated at the end of each evaluation cycle, to compare three approaches: deterministic planning alone, MDP planning alone, and the two combined in the hybrid approach as described above. Fig. 2 shows the normalized aggregate utility for three approaches. As it can be seen, there is a significant performance gap between the hybrid planning and the other two approaches.

Fig. 3 explains the outperformance of the hybrid planning approach by analyzing the first MDP planning cycle of the two cycles. At the beginning of the first evaluation cycle, one server of type-A is active and dimmer value is 3. Initially, the request arrival rate is about 48 requests/seconds, which results into an average client response time greater than 1 second, thereby a highly negative utility. As shown in Fig. 3, at time step $t0$, fast planning comes up with a tactic to add a server of type-C, which brings down the response time below the threshold at time step $t3$ because server takes 2 minutes to boot up. Since fast planning is instantaneous in the experiments, it is invoked at each evaluation cycle to determine if there is a need for adaptation.

However, the slow planning approach takes one evaluation cycle to generate a MDP policy, therefore once generated, the policy is stored for successive evaluation cycles within planning horizon. Once MDP planning is triggered at time step $t0$, due to planning delay, the policy is available at time step $t1$. Therefore, a server of type-C is added at time step $t1$, which results in a higher utility at time step $t4$. Since there is a high negative utility for the response time above the threshold, fast planning has an edge over slow planning during the time period between $t3$ and $t4$.

At time step $t0$, since the average response time is above 1 seconds, the hybrid planning approach triggers both the fast and the slow planning. The fast planning suggests the tactic to add a server of type-C (at $t0$), which results in higher utility between $t3$ and $t4$. However, from $t1$ onward, the hybrid planning refers to the MDP policy for optimal adaptation decisions. Since MDP planning deals with uncertainty in the requests arrival rate, it provides optimal decisions compared to the fast planning approach. For instance, the hybrid approach achieves higher utility between $t1$ and $t3$ due to an adaptation decision (as suggested by the MDP policy) of decreasing the dimmer value at $t1$.

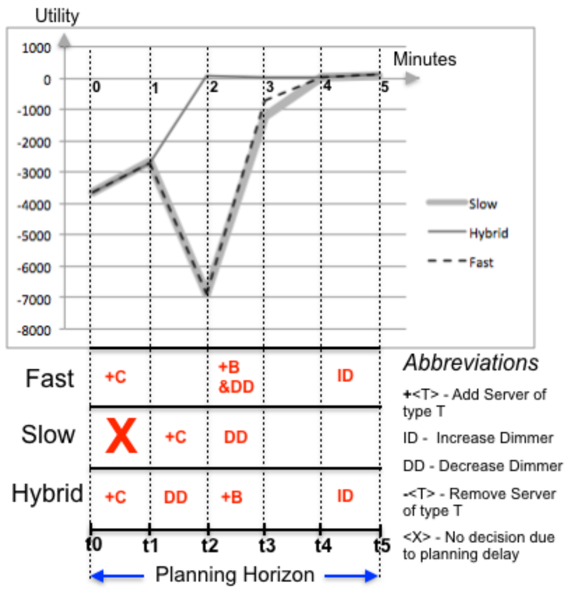

Fig. 3. Adaptation decisions made by different approaches during a planning horizon and impact on utility

## VI. Related Work

Decision making has been a challenge in self-adaptive community [10]. Even though researchers have demonstrated successful use of automated planning [12] [11] in the context of self-adaptive systems, such work tends to focus on domains, where timing is not a critical factor. Generally, the focus of the existing work has been to demonstrate that automated planning could be useful in self-adaptive software systems.

To support automated planning, several execution architectures [8] have been suggested to deal with the problem of planning delay. For architectural based self-adaptation, Kramer and Magee proposed a layered architecture [6] inspired by Gat [7], which aims to deal with the problem of planning delay through hierarchical decomposition of the planning domain. This is a good first step, however, planning delay also depends on the planning approach deployed at each layer. Tajali et. al. [4] extended the hierarchical design by suggesting two types of planning: application planning and adaptation planning. However, two types of planning would further add to the performance overhead. Quite different from these layered architectures, Musliner et al. [9] proposed a framework, CIRCA, which generates control plans at run time that meet real-time constraints. However, the framework requires hard deadlines to be mentioned in the planning specification.

Planning delay is also a problem that has been considered by artificial intelligence researchers. Over the years, numerous heuristics [3] [1] [13] [2] have been suggested to reduce planning time. Moreover, researchers have proposed case-based reasoning, which deals with planning delay by storing previously generated plans for future reuse, thereby reducing the amount of replanning [14]. Furthermore, researchers have suggested the idea of incremental planning known as "anytime planning": the planning process can be interrupted at any time to get a sub-optimal plan, however, more planning time leads to better plan [15]. The hybrid planning approach is close to anytime planning since both use execution time to find optimal plans. However, hybrid planning is different because it manipulates the state space to reduce planning time.

The hybrid planning approach falls into the algorithmic/heuristic category. However, it differs from the existing approaches since none of them use more than one planning approach simultaneously. The closest heuristic is FF-Replan [16], which solves a relaxed version of a planning problem by transforming its planning domain with uncertain action outcomes into a deterministic domain by considering only the most probable action outcome. From an experimental perspective, our work is close to Moreno et al [17]. However, their work focuses on the problem of tactic execution time rather than planning time: planning time is considered negligible in their experiments.

## VII. Conclusion

This paper proposes a hybrid planning approach to deal with the trade-off between quick decision and optimal decision making in self-adaptive systems. Using a realistic client request arrival pattern in the context of a cloud-based self-adaptive system, our experiments provide evidence of the effectiveness of hybrid planning approach.

However, there are various future directions that need to be explored to develop a principled theory of hybrid planning. For instance, in our experiments, fast planning is triggered only when the response time constraint is violated. However, for a hybrid planner, we need to develop a generalized theory for deciding: when to invoke each of the constituent planners.

For the experiments, we used the combination of two planning approaches to instantiate a fast and a slow planning approach. However, we need to figure out if more than two decision making approaches could be combined. Moreover, we would like to explore whether AI techniques other than automated planning could be used in combination.

To answer these research questions, we would like to apply hybrid planning approach on self-adaptive systems from different domains. Experimenting with variety of domains would improve our understanding of the approach.

## Acknowledgments

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research (ONR), and FA87501620042 from the Air Force Research Laboratory (AFRL). Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ONR, AFRL, DARPA or the U.S. Government.

## References

[1] Alfonso Gerevini, Alessandro Saetti, Ivan Serina, "LPG-TD: a Fully Automated Planner for PDDL2.2 Domains" (short paper), in 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04), booklet of the system demo section, Whistler, Canada, 2004.

[2] B. Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, in: Journal of Artificial Intelligence Research, Volume 14, 2001, Pages 253 - 302.

[3] Blai Bonet, Hector Geffner, Planning as heuristic search, Artificial Intelligence 129 (2001) 5-33

[4] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic, PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation, ASE-10, September 20-24, 2010, Antwerp, Belgium

[5] Shang-Wen Cheng, David Garlan, Stitch: A language for architecture-based self-adaptation, The Journal of Systems and Software 85 (2012) 2860-2875

[6] Jeff Kramer and Jeff Magee, Self-Managed Systems: an Architectural Challenge, Future of Software Engineering(FOSE'07)

[7] E. Gat, Three-layer Architectures, Artificial Intelligence and Mobile Robots, MIT/AAAI Press, 1997.

[8] David Kortenkamp, Reid Simmons, Handbook of Robotics, Chapter 8: Robotic Systems Architectures and Programming

[9] David Musliner, Ed Durfee and Kang Shin, "World Modeling for Dynamic Construction of Real-Time Control Plans", Artificial Intelligence, 74:1, 1995.

[10] Frank D. Macìas-Escriva, Rodolfo Haber, Raul del Toro b, Vicente Hernandez c, Self-adaptive systems: A survey of current approaches, research challenges and applications, Expert Systems with Applications 40 (2013) 7267-7279

[11] Daniel Sykes, William Heaven, Jeff Magee, Jeff Kramer, Plan-Directed Architectural Change For Autonomous Systems, Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

[12] Carlos Eduardo da Silva, Rogèrio de Lemos, Dynamic Plans for Integration Testing of Self-adaptive Software Systems, SEAMS, 2011, Waikiki, Honolulu, HI, USA

[13] Silvia Richter, Matthias Westphal, The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks, Journal of Artificial Intelligence Research 39 (2010) 127-177

[14] Veloso, M. (1994). Planning and Learning by Analogical Reasoning. Number 886 in Lecture Notes in Computer Science. Springer, Berlin.

[15] Shlomo Zilberstein, Using Anytime Algorithms in Intelligent Systems, American Association for Artificial Intelligence. 0738-4602-1996

[16] Sungwook Yoon, Alan Fern, Robert Givan, FF-Replan: A Baseline for Probabilistic Planning, American Association for Artificial Intelligence, 2007

[17] Gabriel A. Moreno, Javier Càmara, David Garlan, Bradley Schmerl, Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach, ESEC/FSE-15, August 30 - September 4, 2015, Bergamo, Italy

[18] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. IEEE Network, 14(3):30-37, 2000.

[19] J. O. Kephart and D. M. Chess. The vision of autonomic computing. Computer, 36(1):41-50, 2003.

[20] M. B. Dias, D. Locher, M. Li, W. El-Deredy, and P. J. Lisboa. The value of personalised recommender systems to e-business. In Proceedings of the 2008 ACM Conference on Recommender Systems - RecSys '08, page 291, New York, New York, USA, Oct. 2008. ACM.

[21] C. Klein, M. Maggio, K.-E . Irz̀en, and F. Hernàndez-Rodriguez. Brownout: building more robust cloud applications. In Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, pages 700-711, New York, New York, USA, May 2014. ACM.

[22] Mausam, Andrey Kolobov, Planning with Markov Decision Processes: An AI Perspective, Synthesis Lectures On Artificial Intelligence and Machine Learning.

[23] Dana Nau, Malik Ghallab, Paolo Traverso, Automated Planning: Theory and Practice

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In 10th USENIX Symposium on Networked Systems Design and Implementation, pages 313-328. USENIX Association, Apr. 2013.

[25] Javier Caǹara, David Garlan, Bradley Schmerl, Ashutosh Pandey, Optimal planning for architecture-based self-adaptation via model checking of stochastic games, SAC 2015: 428-435

[26] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, Peter Steenkiste: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. ICAC 2004: 276-277

[27] Daniel Kahneman, Thinking, Fast and Slow

[28] Leslie Pack Kaelbling, Michael L. Littman, Anthony R. Cassandra, Planning and Acting in Partially Observable Stochastic Domains, Journal of Artificial Intelligence Research, Volume 101, Issues 1-2, May 1998, Pages 99-134

[29] Mor Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action