네트워크 프로토콜 학습 속도 개선을 위한 **L*** 알고리즘 쿼리 파이프라이닝

Pipelining L* Algorithm Queries to Speed Up Learning Network Protocol Behaviors

Abstract

The L* algorithm is the most commonly used algorithm to learn network protocol behavior models. It sequentially throws numerous queries to a target protocol agent and determines its behavior by observing the corresponding query outputs. However, due to the large number of queries, the execution time of the L* algorithm often becomes too high. This paper presents a modification of the L* algorithm that shortens this execution time during use in network communication environments. The synchronous querying mechanism of the L* algorithm is modified so that queries are instead done in an asynchronous manner. This new design of the L* algorithm is implemented and tested with an experiment of learning the TCP protocol over sockets.

1. Introduction

With the increase in not only the number of devices but also the required connections amongst them, protocol compatibility is becoming a more and more serious issue. The ability to connect to any system when needed is desired, while manual configuration of protocols or long update times for their synchronization are unwanted. Because there are many variations of protocols, even for systems with the same purpose or domain, it becomes harder for two systems to agree on the same protocol syntax and semantics at runtime. The introduction of the active automata learning L* algorithm by Angluin [1] into the field of communication protocols has helped improve the flexibility of communications of systems. The L* algorithm helps to model and learn a protocol's behavior as a state machine and a number of recent works have applied it to runtime protocol learning.

CONNECT project [2] presents emergent middleware that translates between the two protocols by learning the semantics of each protocol on-the-spot through the active automata learning algorithm. There have been researches [3, 4, 5, 6, 7] that aim to learn the specification of communication protocols such as SIP, TCP and MQTT with the L* algorithm and their own mapper designs. The L* algorithm sends multiple queries to the system under test and examines the resulting output to ultimately form a model of that system's protocol. However, these works miss out on a very fundamental and practical problem. Generally, the number of queries required to learn a model is much larger than the number of states in the model. Because the L* algorithm processes each query in a synchronous nature, when communication with the SUT is done over the network, the whole querying procedure takes a huge amount of time. This means that existing works cannot help user devices to interact with nearby devices of different protocols within the time period when their interaction opportunities are valid.

In this paper, we propose an asynchronous learner which consists of modifications in the membership and equivalence oracles of the original L* algorithm, to suit uses in the network communication environment. By pipelining multiple requests of a query and ignoring outputs that are unnecessary, we create an asynchronous querying behavior that results in a much faster overall execution time of the L* algorithm. To evaluate the proposed scheme, we implemented an SUT with TCP protocol on a Raspberry pi 3 and learned its three-way handshake behavior. The evaluation result shows a 46.64% decrease in execution time compared to the common structure of aforementioned existing works.

2. Proposed Scheme

2.1. Architectural Overview

To solve the limitations presented above, we advance the main building blocks from the common structure of existing works [3, 6] to enable faster learning of the target protocol behavior. Figure 1 shows interactions between the modified modules. The AsyncLearner sends pipelined

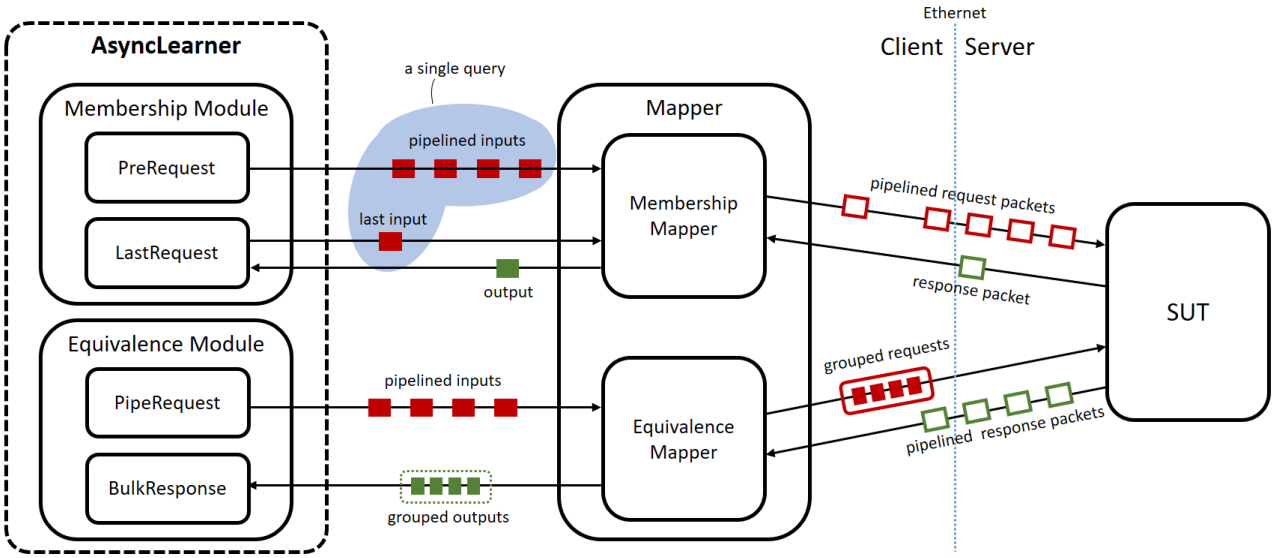

Figure 1. Architectural overview of the learning setup using the AsyncLearner

inputs of a membership query. The Mapper turns the abstract inputs into concrete inputs and sends them as packets to the SUT. When the AsyncLearner receives responses from SUT, it stores the result into the observation table. Once the observation table is completed, the Equivalence Module sends pipelined inputs of the equivalence query to the SUT through the Mapper and compares the responses to the current hypothesis on the behavior of the SUT.

2.2 AsyncLearner

2.2.1 Membership module

The Membership module is responsible for testing the individual requests of each membership query asynchronously by pipelining them. A single query, also referred to as a word, is a combination of symbols. For example, for the TCP protocol, a “SYN” would be a symbol while the sequence of “SYN-ACK-FIN” would be a word (i.e. a query). The L* algorithm, to test a single membership query, sends a word to the SUT. This means that it sends multiple symbols (of which the word is made up of) to the SUT one by one and then observes the output of the final symbol. For example, to test the membership query “SYN-ACK-FIN”, the L* algorithm would send a “SYN”, wait for the output, then send an “ACK”, wait for the output, and then finally send the “FIN” and wait for the final output. However, because the outputs of the symbols before the final symbol do not matter, these symbols could be sent continuously without waiting for their responses to enable faster querying. PreRequest sends all the symbols upto the last one continuously to the Membership Mapper, without waiting for their replies. Once PreRequest has completed, LastRequest sends the last symbol of the query and waits for the response.

2.2.2 Equivalence module

The Equivalence module is responsible for pipelining the inputs of the equivalence query. An equivalence query is made up of a long sequence of random symbols, where the output of each tested symbol is compared with the hypothesis model. Because the output of each of the symbols matters, we cannot just ignore them and observe only the output of the last symbol, like how membership queries are processed. Instead, multiple symbols may be sent together only if all of their outputs are received as well. This way, the Equivalence module processes multiple symbols at a time.

PipeRequest sends multiple symbol requests to the Equivalence Mapper at once by pipelining them. Instead of waiting for the output response after sending a symbol, it stores the symbol as pending and continues on with the next symbol in the equivalence query. This means that comparisons with the hypothesis model isn’t done after each symbol request, but instead is done later on altogether. BulkResponse is responsible for receiving the output responses. It receives a group of outputs from the Equivalence Mapper and then assigns each of them to their corresponding input symbols. Then it iterates over the sequence of outputs and compares each of them with the output of the hypothesis model.

2.3 Mapper

The Mapper consists of two modules that support asynchronous querying: Membership Mapper and Equivalence Mapper. The common functionality of both

Mappers is to translate abstract inputs to concrete inputs and forward them to the SUT as network packets. The Membership Mapper pipelines the requests from PreRequest to the SUT. However, it does not receive any responses and also returns nothing to PreRequest. When the Membership Mapper receives the final request from LastRequest, it sends the request to the SUT and receives its output, which it returns to LastRequest. The Equivalence Mapper receives the pipelined inputs from PipeRequest. When the final request arrives, it groups all the requests into a single packet and sends it to the SUT. It then receives pipelined response packets from the SUT. It parses these packets and forms it into a group of outputs which is returned to BulkResponse.

3. Evaluation

3.1. Implementation

To verify the proposed scheme with a real world scenario, we implement a TCP agent on a Raspberry Pi 3 and learn its behavior model. AsyncLearner and Mapper are implemented with Java and run on a Windows desktop with an i5-4 CPU and 8GB RAM. LearnLib [8] is used for the L* algorithm of AsyncLearner. Packet transmissions between the Mapper and the SUT are done through sockets, where both components reside under the same network.

3.2. Results

To investigate whether the devised mechanisms are effectove to speed up the learning, we measure the execution time of each mechanism and compare the result to that of the common structure used in existing works [3, 6].

Figure 2 shows the execution time of both the AsyncLearner and the common structure of existing works. The execution time of membership queries dropped from 431 seconds on LearnLib to 264 seconds on AsyncLearner and the execution time of equivalence queries dropped from 90 seconds on LearnLib to 14 seconds on AsyncLearner. This means 38.7% faster execution of membership queries and 84.4% faster execution of equivalence queries. Pipelining inputs is more effective for queries that have a long sequence of inputs. While equivalence queries are generally very long, membership queries are usually much shorter, with most of them being shorter than 8 inputs. Because of this, the decrease in execution time of equivalence queries is must larger than that of membership queries. Due to these changes, the overall execution time of AsyncLearner was 243 seconds faster, which is a difference of 46.64%.

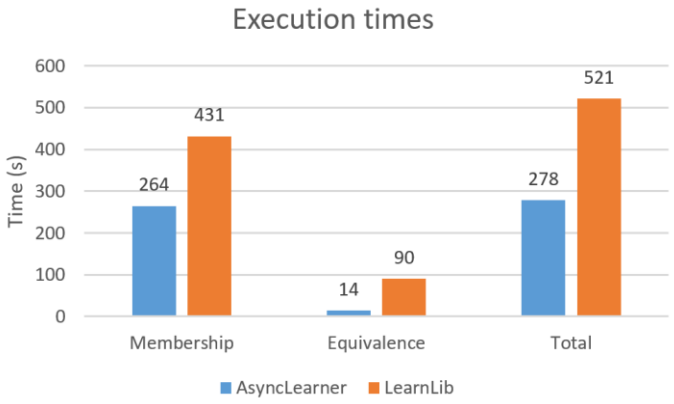

Figure 2

4. Conclusions

In this paper, we propose an asynchronous learner which consists of modifications in the membership and equivalence oracles of the original L* algorithm. To turn the synchronous nature of L* algorithm into asynchronous, we pipeline the query inputs transmitted from the learner module to a system under test through the mapper. To verify the proposed scheme, we implement a TCP protocol learning scenario with different machines and the evaluation results show that the proposed scheme decreases the execution time down to 53.36% compared to the common structure of existing works. For future works, we plan to integrate the proposed behavior learning scheme to a protocol syntax learning scheme and present a runtime protocol adapter generator.

5. References

[1] Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput.75(2): 87-106, (1987)

[2] Bennaceur, Amel, and Valérie Issarny. "Automated synthesis of mediators to support component interoperability." IEEE Transactions on Software Engineering 41.3: 221-240, (2015)

[3] Aarts, F.: Inference and Abstraction of Communication Protocols (Dissertation). (2009)

[4] Bohlin, T., Jonsson, B., Soleimanifard, S..: Inferring compact models of communication protocol entities. (2010)

[5] Aarts F., Jonsson B., Uijen J.: Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. (2010)

[6] Fiterau-Brostean, P., Janssen, R., Vaandrager, F.W.: Learning fragments of the TCP network protocol. (2014)

[7] Tappler, M., Aichernig, B.K., Bloem, R.: Model-Based Testing IoT Communication via Active Automata Learning. (2017)

[8] Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models: 393-407, (2009)