The Progression of DyMonD: A Novel Framework for Application Monitoring

Aaron Lohner, Supervisors: Dr. Bettina Kemme and Mona Elsaadawy

McGill UNIVERSITY

Introduction

The DyMonD framework is a tool designed to monitor the health of a large-scale application. Initially, it was lacking in functionality, efficiency, and interconnectivity. The framework operated with three independent components (see Figure 1). The agent component would sniff for network packets on an application (or read a capture file) and write detected communication flows to a text file. Then, the controller component would read the text file and use it to prepare an incomplete call graph that would be rendered by the visualization component.

Objectives

- Improve transfer of communication flows
- Enable the framework to be run in a distributed setup
- Restructure controller code
- Develop useful framework configurations

Methods

- Implemented a more effective way for flows to be transferred from the agent to the controller using TCP and Google Protocol Buffers (Protobuf)
- Created the "interfaces loader" and the "next hop extractor" controller modules
- Structured the controller code to allow for production of a complete application call graph
- Created various setup configurations as the framework evolved

Results

Originally, the DyMonD controller was required to be on the same machine as the agent in order to access the text file written by the agent, which contained the detected flows. The new process instead uses a TCP connection and Google Protocol Buffers to communicate the flows from the agent to the controller. This way of transferring flows allows for a distributed setup of the framework (note that for all of the figures below, components of the same colour must all run on the same machine).

In addition to the restructured controller code (Figure 5), there are now three principal configurations of DyMonD. The primary setup is to have the framework capture packets from a live application (Figure 2). In this setting, the controller is started by the user with the IP address of the first component in the monitored application as input. The controller determines which network interface corresponds to this IP using its interfaces loader module, which loads a dictionary that maps interfaces to IP addresses. The interface is then sent to the agent, which produces the communication flows. The controller uses the flows to determine the next interface to sniff using its next hop extractor module. This back-and-forth process between the controller and agent continues until no new interfaces are detected by the hop extractor. At this point, the controller calls its call graph producer module. The module creates a JSON file that is used as input to the visualization tool, which generates the call graph.

The "capture from file" setup is similar to the previous setup, except that packets are read from a file rather than sniffed from a live application (Figure 3). The standalone setup does not make use of the controller and instead simply produces a log of communication flows (Figure 4).

An additional configuration option for the framework includes the ability to transfer flows to the controller using logs or over TCP.

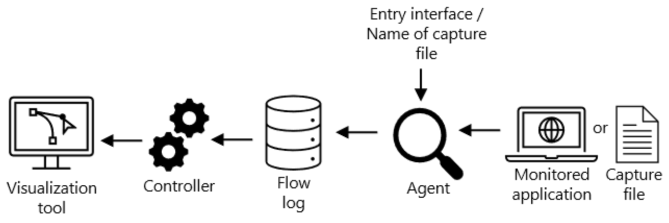

Figure 1: Original architecture

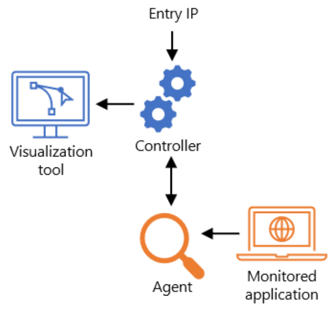

Fig. 2: Capture from live application

```
input   : $IP$:start service IP.
output  : $G$: Call graph of $IP$.
Data    : let $Q$ be a queue and $F$ and $L$ empty lists.
$Q.enqueue$(IP)
while $Q$ is not empty do
    $IP^*$=$Q.dequeue()$.
    $F \leftarrow$ FLOWS DETECTED FOR $IP^*$.
    $L \leftarrow$ (L∪F)
    $IPs \leftarrow$ NEXT HOP EXTRACTOR ($F$, $IP^*$)
    for $IP$ in $IPs$ do
        $Q.enqueue(IP)$.
    end
end
$G \leftarrow$ CALL GRAPH PRODUCER ($L$)
return $G$
```

Figure 5: Controller pseudocode
From Mona ElSaadawy (internal communication)

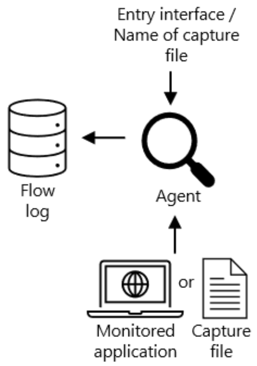

Figure 4: Standalone

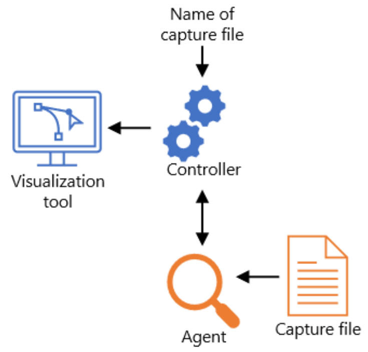

Figure 3: Capture from file

Conclusion

DyMonD evolved from a collection of independent scripts into a structured, distributable framework. It can send flows in a more effective manner, produce a complete application call graph and is highly configurable. Although there is still work to be done for DyMonD to be completed, it is now much more functional and closer to demonstrating that a truly holistic and independent application monitoring tool can exist.

Future Work

- Enhance usability with a UI
- Introduce multiple agents using multithreading
- Replace interface dictionary with flow detector module
- Ignore flows on certain ports
- Automatic updates to call graph

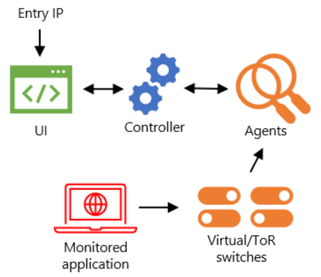

Fig. 6: Eventual architecture

Acknowledgements

I would like to thank Dr. Bettina Kemme and Mona Elsaadawy for supervising this project and providing their support and assistance on it whenever it was needed.