1. Introduction

This document is meant to be a precursor to the HLD (high level design) for the capture service of the GALORE product. The capture service has been broken down into multiple use cases and each of them depicted in terms of classes/sets of classes. The capture process flow will not be discussed here.

1. *Design*

ConcreteEngine2
CapEngineImpl
findDevice()
openDevice()
grabPacket()
CaptureController
createCapEngine()
createPacketWriter()
setupCapSession()
CapEngine
setupCapture()
startCapture()
stopCapture()
ConcreteEngine1
string host
uint port
int capsize
sizebased
timebased
libpcap
customcap
Caplet

NetworkFilter
setCondition()
applyCondition()
ProtoProcessor
pack()
unpack()
ethernet
tcp

GenericFilter

bool check()

1. ***Discussion***

CaptureController **–** This is the entry point into the capture service. It is meant to customize the build and run of a capture session. For instance one could have a class derived from it to handle time based capture and one to handle size based capture.

CapEngine – This provides the interface for a capture engine. One can derive from this class to extend the basic interface of a capture engine.

CapEngineImpl – This isolates the actual implementation of a capture engine. Platform specific or custom implementations of the low level primitives used in a network level capture library would be done here.

The CapEngineImpl and CapEngine are a manifestation of the structural pattern which is used to isolate the interface from an implementation. This interface allows one to vary the implementation independent of the interface.

Libpcap and Customcap can be considered specific ways of implementing the low level primitives of a network capture library.

Caplet – This is an encapsulation of the attributes which comprise a capture session. Eg: - host, port, size of capture etc.

GenericFilter – This provides a way of filtering packets based on various criteria. Concrete filter classes would derive from this abstract class and override the check() method. The check() method would simply return true/false indicating whether a packet is filtered or not.

NetworkFilter – This class is used to provide filters based on the specific information stored in the network headers.

ProtoProcessor – This is a utility class for serializing/deserializing network protocol specific data. An EtherProcessor, for instance, would override the pack() and unpack() methods by converting to/from ethernet structure and a stream of bytes.

An network filter would use an EtherProcessor to implement an Ethernet header based filter.

1. Design justification

- The use of patterns is to give flexibility and extensibility to the framework.
- The filter interface is designed with the assumption that “filtering” implies ignoring a packet based on a certain condition. I have assumed one set of conditions as “header field <operator> value” to come up with the network filter class. More discussion on filter requirements is needed to come up with other kinds of filters or refine existing one.
- The methods mentioned in the diagrams are an approximation of the functionality that is exposed by the respective class. The final cut would have a larger or slightly different interface as we get more clarity on implementation aspects.
- A deep dive into the capture process flow starting from the GUI uptil the spawning of a capture agent might result in refining the entry level interfaces to the capture service.
- The packet writer,compressor,encryption interface has not been explored yet.