Goliath: A Configurable Approach for Network Testing

Wade Fagen and João W. Cangussu
Department of Computer Science
University of Texas at Dallas
{waf022000,cangussu}@utdallas.edu

Ram Dantu
Department of Computer Science
University of North Texas
rdantu@unt.edu

***Abstract*— When testing a network environment and/or application many aspects need to be taken into consideration. For example, different software needs to be deployed at different nodes and different topologies may also need to be tested. The manual execution of these tasks are very time consuming and a configurable environment to facilitate these tasks and consequently improve testing performance is desired. In this paper a virtual network environment that can be easily reconfigured is presented to address this problem. Also presented is a case study with the results of deployment and containing a worm propagation attack.**

I. Introduction

Network testing has many distinct dimensions. For example, some could be interested only in testing a communication protocol [1] while others would be interested in testing an intrusion detection system. In any case, the network must be properly set up to allow the testing. This may consist of deploying distinct software solutions to different nodes in the network. In addition, the testing must be conducted for distinct topologies to verify their adequacy to a multitude of different configurations. Based on that, two major issues must be addressed. The first refers to the configuration of the network. Clearly, a manual configuration will be very time consuming and will limit the number of distinct scenarios tested. An easily reconfigurable network environment must be available to improve testing performance. Since it is not possible to keep constantly changing the physical topology of a network, the solution must rely on a virtual network as the one proposed here, referred hereafter as *Goliath*. As described later, *Goliath* can be easily reconfigured.

The second aspect to be considered regards where to execute the testing. Depending on what is under test, a closed network must be used to address security issues. That is, the testing of a intrusion detection system may be conducted on a secure dedicated environment while a protocol testing could make the use of idle resources. *Goliath* can be used in both cases. In many modern corporations and businesses, thousands of computers either set idle or have very limited computational usage through the day. Often, the depth of work that one computer may see in its three years existence is the use of an e-mail client, a web browser, and possibly some productivity tools such as a Word Processor or a Spreadsheet. While there have many attempts to harness the idle computation time and countless research papers on the topic [2], IT departments often find there to be too many hurtles to overcome to create a distributed system across the entire enterprise [3]. One of the greatest problem that is encountered in any widely-available distributed network solution is the lack of cross-compatibility of the distributed system across multiple operating systems. Even with a homogeneous set of machines, the task of actually getting the distributed task out to each machine, managing those machines, and running the next task is extremely time-consuming and requires human interaction at each step. *Goliath* addresses these issues. The major goal of this paper is first to present the initial developments of the *Goliath* framework and then show its flexibility by testing a worm propagation deployment along with a containment solution.

Though the focus on this paper is on network testing, *Goliath* can be used to test/run general distributed systems. Many of the most widely known distributed network systems include large-scale computational tasks such as examining radio waves [4], factoring RSA keys [5], factoring particular near-prime numbers [6], simulating global warming [7], and more. As such, many of the partly or fully customizable solutions mimic these models – each of them relying completely on the underlying network structure and does not provide any customizability to network topology or network availability. For the task at hand, a simple distributed computation would be expected. The problem lies when any form of network testing, topology testing, or even software deployment testing is desired. With the approach of computational distributed networks, one would be limited to a distributed network simulator such as ns-2 [1]. However, a distributed virtual network is capable of performing real-time network tests and not simply simulations.

The remainder of this paper is organized as follows. A description of the *Goliath* approach is presented in Section II followed by a detailed description of a worm propagation case study in Section III. Section IV presents some related work as well as a comparison with the proposed approach. Section V finalizes the paper with concluding remarks and plans for future work.

II. The *Goliath* Approach

In the simplest form, a virtual network provides functionality of links between nodes without regard to the physical connection between sites. The *Goliath* approach is designed

to provide that level of functionality, without the need to specify the domain space any further. To achieve this end while allowing for a system that could be widely used, it was important that any computer could take part in the virtual network and, as such, a platform independent solution was chosen as the primary implementation. By using the Java programming language and the widely available JVMs [8], a base framework was able to be developed, deployed, and utilized. Furthermore, by standardizing the communication between the virtual nodes, specific platform-dependent implementations could be made for the most widely-used platform in the virtual network to increase performance and eliminate the need for a JVM. An overview of the *Goliath* framework is seen in Figure 1.

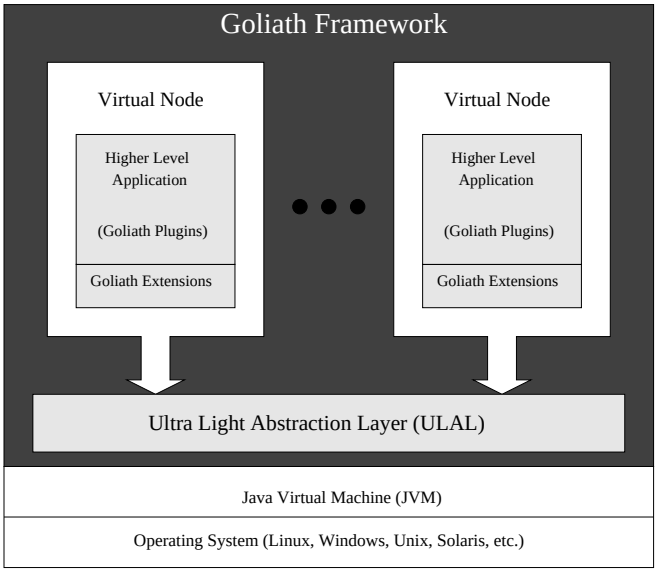

Fig. 1. Overview of the *Goliath* framework

A. Ultra Light-Weight Abstraction Level

Despite requiring a JVM, *Goliath* is an ultra light-weight abstraction layer which gives higher level applications the ability to seamlessly interact with each other over the virtual network. Since the virtual network is indistinguishable from a physical network to the application, one of *Goliath*'s main tasks was to handle routing and port-like abstractions within the virtual network without the application developer needing to be concerned about such protocols. That said, this does not limit the application developer from creating his or her own protocol on top of the *Goliath* framework for connection nodes, synchronizing data, or any other task.

The virtual network topology is set up at the deployment of the virtual network through XML files described later in Section II-D. As would be expected in a real network, when a node is three hops away (virtual hops in the case of the *Goliath* framework), a request must travel from its source, through the two intermediate computers, and finally arrive at the destination computer. To do this, the beginning of every packet sent through the virtual network is marked with an extra sixteen bytes of information: four bytes for the source address, three bytes for the source port, four bytes for the destination address, three bytes for the destination port, and a final two bytes reserved for future use. This layer could correctly be viewed as a header much like the headers in TCP, UDP, IP, etc. [9]

Similar to the Internet, each node's address is assigned uniquely by a name server. However, unlike the Internet [10], specific dedicated name servers are not necessary. Instead, the deploying computer acts as the master computer of the virtual network while the deployment process is under way (discussed in full detail later in this paper). After the deployment is complete, if additional computers are permitted to join the virtual network during the computation, a special request is made by the joining computer to seek out the master computer of that particular deployment and the master computer then assigns a virtual network address to the joining node. Similar to TCP and TCP applications, the address assignment is handled by *Goliath* and is transparent to the application running on the *Goliath* platform.

B. Plugins

As previously discussed, a virtual network is absolutely nothing without an application making use of the virtual network. Moreover, most of the requirements for virtualization revolve around the ease of deployment of applications on a test network. To satisfy each of these requirements, the *Goliath* framework allows higher level applications to plugin to the *Goliath* framework similarly to how artists can extend the functionality of Adobe Photoshop or how games can develop their own applications to interface with the in-game interface through programs such as Decal. To fully understand how the Goliath application implements this, it is important to first understand the *Goliath* framework's structure.

The *Goliath* framework consists of two core entry points which can be viewed as two separate applications: the client application and the master application. The client application is the part of the Goliath framework which is deployed on each of the computers participating in the virtual network. This client application is the application which takes higher level applications and extensions to the *Goliath* framework and provides the interfacing to the virtual network. Figure 1 shows the client application on top of the JVM and operating system of a target machine. The entire purpose of the master application is to control the *Goliath* framework from any other location. The master application may connect to any one of the client applications to communicate with the virtual network as a whole, allowing for the addition of configurations to be tested or termination of the currently running application.

C. Extensions

Throughout the past forty years, networking has become increasingly complex with new methods of performing everything from synchronization to recovery. As these new ideas gain adoption, they become the basis of new research which depends on the very existence of these ideas. Since these fundamental networking ideas would be applied in a whole

range of *Goliath* plugins, the *Goliath* framework includes a way to extend the functionality of the *Goliath* framework to allow a plugin to make use of new technologies whatever they might be in a nearly native way. As previously mentioned, a *Goliath* extension exists between the ULAL and the higher level applications, as depicted in Figure 1. Since these extensions extend the functionality of an application and are not an application in their own regard, they rely on a higher level application to make use of them and are therefore abstract classes in Java. Like all components of the *Goliath* framework, they need only to be compiled once and are automatically deployed just as higher level applications are deployed.

As the development of the *Goliath* framework progresses, more and more functionality can be added to it through the use of extensions. For example, delays due to the distance of virtual nodes have not yet been incorporated to the framework. That is, if two virtual nodes are five hops away but reside in the same machine, the delay may be much smaller then if they were physically five hops away. A delay extension is planned to be implemented using a distance vector routing algorithm. Clearly, any experiment which behavior is dependent on delays may produce compromised results without the proper extension. As a library of extensions become available, the functionality of the *Goliath* framework will increase significantly. It should be clear that extensions could be anything of interest ranging from the use of a new protocol to the collection of network metrics/measurements.

D. Network Deployment and Redeployment

One of the key features of *Goliath* is the ability to schedule multiple tasks running different programs with different options at one time. To accomplish this, a simple XML document is used. This document provides the advantages of being both human-readable and human-editable as well easily parsed by a machine. Particularly, there are two main sections of the XML file: a section to set up the virtual network and a section to topologically connect the nodes in the virtual network and deploy their task.

To set up the virtual network, a user would first have to deploy the *Goliath* application to each computer which wishes to be a part of the initial virtual network. This deployment may be running the application as a service every time the computer boots or it may be as simple as running the program from an SSH session. By designing *Goliath* to run with the most minimal required privileges, a vast set of computers may be used even if the user is not specifically in control of the entire set of computers. Since nodes are capable of joining once a task is started, it is only necessary to start *Goliath* on a single computer before beginning to deploy the network. (It is important to note that while nodes may join the *Goliath* framework while in the middle of a task, it is the choice of the developer of the higher level application as to if a node may join in the task if the node was not available at the beginning of the task.)

The *Goliath* manager is the only extra piece of the *Goliath* solution, outside of the management of the virtual network, required to begin the deployment of the network. This small application connects to any node within a virtual network and interacts with the virtual network. From this connection to within the virtual network, the management tool can send the deployment XML file and any associated files to a node in the virtual network. If no current task is being preformed, this node becomes the master node of the network, begins creating the network or deploying the new task, depending on the current state of the network.

```
...
<computer address="192.168.2.52" port="3213" >
<computer address="192.168.2.53" port="3213" >
<computer address="192.168.2.53" port="3214" >
...
<connection>
    <nodeID>31</nodeID>
    <nodeID>57</nodeID>
</connection>
...
```

Fig. 2. Snippet of XML code used in the *Goliath* framework.

Figure 2 displays a sample snippet of an XML file which manages the *Goliath* framework. The first section of the snippet shows the list of computers available initially to the *Goliath* framework to establish the virtual network for the first task. Notice that two computers have the same address – *Goliath* allows for any number of virtual nodes to exist on the same computer and leaves it to the application developer or the user to ensure that his or her application is not being slowed because of lack of available resources on a given computer. The second section of the snippet shows the connections of the virtual network itself, interconnecting nodes with one-another. Since the virtual network may grow while a task is being preformed, the use of an identification number rather than an address allows for a user to pre-plan the topology for nodes which may connect to the network which are not in the original setup.

As an example, if *Goliath* was started on ten (10) machines and then a user connects to any of the ten machines with the *Goliath* manager, he or she could then send the node he or she connected to the XML file and the application task. This XML file would specify the nine other nodes that are to be used in the virtual network within the first section and then specify the topology of the nodes and the task or tasks these nodes were to run at that topology. Then, the XML file may specify a new topology and a new task or series of tasks to run at that topology. At any time, another user can connect to any of the nodes in the virtual network and then queue more tasks, more topologies, or a combination of both. Therefore, a single virtual network is capable of running a single task with twenty different network topologies to gather results if a user was interested in topological testing.

onConnect(...) : optional, notifies the higher level application of a new node which is now connected to the virtual network

onDisconnect(...) : optional, notifies the higher level application of an existing now which has now disconnected from the virtual network

onExiting(...) : optional, notifies the higher level application of pending termination and allows for final tasks to be preformed

onMessageRecieved(...) : required, notifies the higher level application of a new network message and provides the message and the sender of the message as parameters of the method

Fig. 3. Functions *Goliath* provides for higher level application integration into the *Goliath* framework.

E. Higher Level Applications

As mentioned in previous sections, the task running on the *Goliath* framework is a simple application which runs above the virtual network itself. This application interacts with the *Goliath* framework through a simple API which provides functionality to send and receive messages and notification of termination of a task and termination of the node. Making use of the features of Java, this is implemented as an abstract class which a user must implement in order to compile the application against the *Goliath* library.

Figure 3 shows the entire set of functions a user needs to implement to interact with the *Goliath* framework. Additional distributed system concepts such as checkpointing (to ensure recover in case of failure) is left to the application to handle. However, it is trivial to implement an intermediate class which is compiled between the *Goliath* framework and the user application to implement any of these functionalities. For verification of the virtual network, such an approach was used to include a vector clock time stamp on all messages.

From the beginning of coding, a simple application such as nodes counting in a distributed fashion (where a single node initializes the counting process with the number 1, sends it randomly to another node in the network which adds one to the number and then repeats the process of randomly sending it to another node) was able to be completed in less than five minutes. While this is a simple and trivial application, this shows the ease of use and flexibility of the virtual network, *Goliath* allows for any application to run as a higher level application, performing whatever functionality it desires, thereby creating a rich testing environment. Combining this with *Goliath*'s ability to schedule a virtually unlimited number of tasks all to be executed on any desired topology which is configured in real time, the *Goliath* framework lead itself to a very flexible approach for the testing of network environments and or applications ranging from the trivial node counting to a more elaborated case studied as described in the next section.

III. Network Worm Propagation Case Study

Over the past few years, numerous studies have been done on intrusion detection systems and the prevention of a worm entering into a network. While these studies have made significant progress in slowing down potential worms and attacks on networks, every year there's another major worm which sneaks by every safeguard and gains international media attention. Current research suggests that completely avoiding worm infection and propagation is likely an impossible task, the next logical step would be to accept the worm will exist and figure out what one can do with a worm to slow its propagation.

The Network Worm Propagation Case Study (NetWormPCS) is a series of programs implemented on top of the *Goliath* framework to simulate the spreading of a network worm in the most real-life condition possible without irreversibly infecting hundreds of machines in every-day use within a test network. NetWormPCS studies the use of adaptive proportional integral derivative (PID) controllers [11] with machine learning clustering techniques [12] to identify worm packets from user packets and to prevent these packets from propagating and infecting new nodes [13], [14]. Section III-A provides a description of the worm propagation scheme used whereas Section III-B gives a more detailed description of the containment solution. The results of using *Goliath* to test the worm propagation and the containment solution are described in Section III-D followed by a discussion, in Section III-E, of how both the worm propagation and the solution were easily deployed using *Goliath*.

Each node on the virtual network represents a physical network device. Leaf nodes (nodes which only have one edge or neighbor in the virtual network) were considered to be only a user's computer, while nodes with more than one neighbor were considered to be some form of networking routing equipment which is intelligent enough to make use of the *Goliath* framework and the NetWormPCS application. This could be a server, firewall, router, or any other device. Therefore, when a X number of nodes is created, the number of leaf nodes Y is going to be less than the total number of nodes $Y < X$. Constraints can be added when generating the virtual nodes to limit the number of internal nodes. The NetWormPCS application suite was placed on each node, user and server alike, allowing for the analysis of all outgoing packets.

A. Worm Propagation

The NetWormPCS selects one node to become infected at time $t = 0$ and the worm propagation begins. For every configuration, the assumption of a 16-bit subnet was made and the addresses for each computer was randomly assigned through the 16-bit subnet. Therefore, at any given packet send by the worm, there's only a $\frac{503}{(2^{16}-1)} = 0.768\%$ chance of a packet arriving at a real computer in every case; where 503 is the total number of nodes in the experiment. This behavior closely mimics how actual worms spread in real attacks.

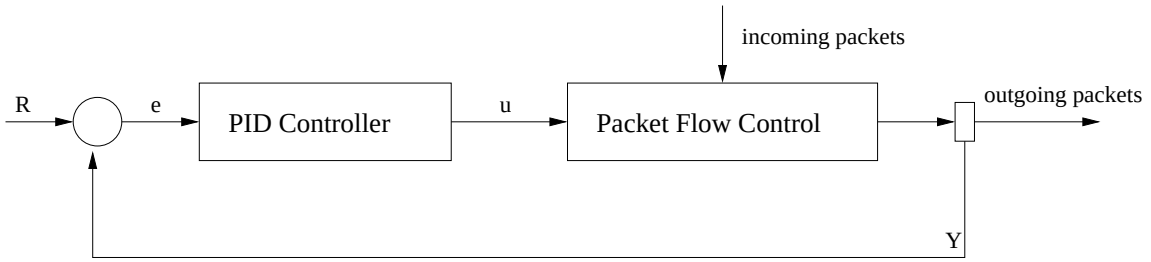

Fig. 4. Overview of PID controller

B. Attack Slow Down/Containment Solution

In order to slow down and/or contain the propagation of the worm described in Section III-A a feedback control approach is used. The approach is based on the use of a PID controller to slow down the worm propagation allowing enough time for the machine learning algorithm to identify patterns for the worm and further slow it down or even contain the attack. These are described next.

1) PID Controller: A PID controller is a simple feedback controller that can be viewed visually as a closed loop 'black box' as shown in Figure 4. The desired output value, or set point value, is denoted by R, the actual output of the system is denoted by Y. The intermediate value of e (defined in Eqn. 1) represents the error and is sent directly into the PID controller. The controller uses the error value to compensate the input and achieve the desired output value.

$$e = R - Y \tag{1}$$

Within the PID controller, there are three configurable values which are used to alter the behavior of the controller. These values are: K_p-proportional gain; K_I-integral gain; and K_d-derivative gain. These three values make up the key function of the PID controller, the transfer function which is defined by Eqn. 2.

$$f_{transfer} = \frac{K_p s^2 + K_I s + K_d}{s} \tag{2}$$

where s is a unit of time (usually seconds). From this, the final piece of the PID controller is developed: the signal function. This function is the sum of the proportional gain times the value of the error, the integral gain times the integral of the error, and the derivative gain times the derivative of the error as seen in Eqn. 3

$$u = K_p e + K_I f(e)dt + K_D \frac{de}{dt} \tag{3}$$

Previous research in PID controllers show the relationship between K_p, K_I, and K_d [11]. While these values were optimized based on the analysis of performance, it would be worthwhile future work to determine mathematical models between these values and the effect on worm propagation.

2) Set Point and Threat Level: The previous section describes the PID controller, but leaves out the definition of one critical value: the actual desired output of the system. To determine this, two values were used to make use of the current condition of the node itself and the network as a whole: a network threat level (NTL) and a local threat level (LTL). These values, each based on a scale of 0-100, are representative of the current threat of a worm within the network where 0 denotes 'no threat' and 100 denotes 'a sure threat'.

The local threat level is the first level to be determined by each node based on the outgoing packets. To determine the LTL, the technique of clustering is borrowed from machine learning [15]. In clustering, a series of data points are examined and compared against one another to form clusters which represents similarity between the data points. In the case of worm propagation, we find that a worm quickly and respectively sends out nearly the same exploit packet to nodes across the network. This creates two unique signatures of a worm: a tightly knit cluster of nodes and a wide range of destination addresses within the tightly knit cluster. If, over a two second period of time, five nodes are clustered together in the tight knit cluster with differing destination addresses, the possibility of a worm arises and the LTL is adjusted according to Eqn. 4.

$$LTL = \begin{cases} c & c \leq 10 \\ 2c & 10 < c \leq 15 \\ 4c & 15 < c \leq 20 \\ 5c & 20 < c \leq 25 \\ 100 & c > 25 \end{cases} \tag{4}$$

where c is equal to the number of packets which meet the criteria described above.

Despite the name, the network threat level is not the same through the entire network. This threat value is defined as a function of the neighbors of the computer and the local threat levels of those nodes. Given that a computer has n neighbors, LTL_i denotes the local threat level of neighbor i, and LTL denotes the local threat level of oneself:

$$NTL = \frac{LTL}{n+1} + \sum_{i=1}^{n} \frac{LTL_i}{n+1} \tag{5}$$

Therefore, when the local threat level is changed, it is only necessary to broadcast it to all your neighbors and nothing more. Since local threat levels affect only network threat levels of neighbors, there is no dependence chain spanning the entire network.

Finally, the set point is defined as a function of the LTL and NTL. The aim of this function is to maximize the

```
begin GenerateNodes()
    GN := array of Generated Nodes
    GN_0 := {address_0, null}
    for i := 1 to n do
        GN_i := {address_i, address_rand(0,i−1)}
    return GN
end
```

Fig. 5. Algorithm used by *Goliath* to generate network nodes for a basic topology.

network throughput while the network is likely uninfected but limit the throughput while the network is infected. Through experimental trial, the function in Eqn. 6 was found to produce impressive results.

$$R = \begin{cases} +\infty & (LTL + NTL < 20) \\ 500 - (2.8LTL + 2.18NTL) & \\ & (20 \leq LTL + NTL < 150) \\ 2 & (LTL + NTL \geq 150) \end{cases} \quad (6)$$

where R is expressed as the number of packets per second. With the varying values of R, during a network infection the PID controller is constantly at work sending the NetWormPCS application differing signal functions to allow or permit a certain level of packet throughput.

3) Delayed Queue: To implement a slow down in network traffic, a delayed queue is necessary to delay the process of sending packets across the network. This queue, as the name implies, queues a packet until there is bandwidth available to send the packet. In an implementation without any form of intelligence, the process of placing items into the delayed queue is simply a probability and will not place more worm packets into the delayed queue than 'good' packets. Therefore, intelligence is introduced to the system to allow for the greatest number of good packets to avoid the delayed queue and, likewise, the greatest amount of worm packets to arrive in the delayed queue.

As described in Section III-B.2, a clustering technique is used to determine the local threat level of a computer. Since this clustering is done in real time as each packet arrives, before a packet leaves it is known if it falls into a tightly knit cluster with varying destination addresses. If so, this packet is immediately sent to the delayed queue. Otherwise, the packet is served to the ready queue – a queue which must be empty before any packets from the delayed queue is passed through.

Based on the current rate of throughput and the signal value received from the PID controller, packets are allowed to flow from the ready queue out onto the network.

C. Network Topology

In the NetWormPCS case study, all network topologies were generated in the same way to ensure the greatest amount of comparability between results and differing instances. The way that was chosen to generate nodes was done such that some end nodes would connect to every server along the network – but that specific servers would have vastly more end nodes than other servers. The algorithm used is described in Figure 5. However it should be clear that virtually any topology can be created in the *Goliath* framework.

D. Results

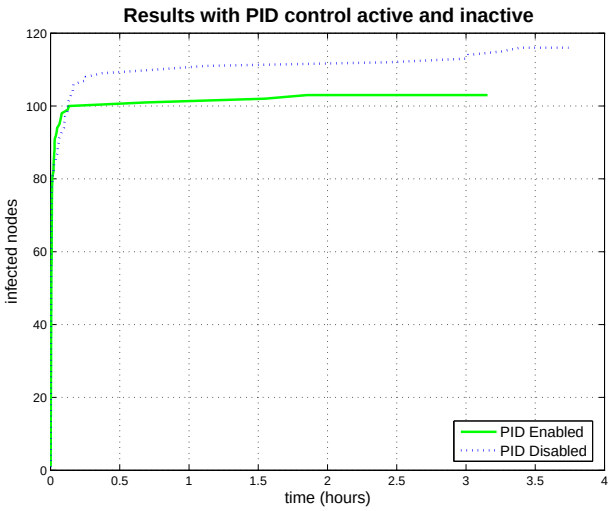

Fig. 6. Rate of infection with only PID control protection

To test the NetWormPCS application and the *Goliath* framework itself, a total of 42 physical machines were used with 12 virtual nodes ran on each machine for a total of 504 machines. Once the *Goliath* framework was set up, the command was given and the nodes self-deployed, self-organized into the given topology, and began running the NetWormPCS application. After infecting the first node at time $t = 0$ the worm propagation begins.

The first configuration ran was using no protection whatsoever and simply having the worm spread at will. The results can be seen in Figure 6. As expected, the worm saturated the network in a matter of minutes to a point where the worm couldn't even spread itself because the throughput had become so low on the physical network. After a little over three minutes, the network throughput diminished considerably and over 100 of the nodes were infected. Although not all the nodes were infected, the whole network (virtual) was unable to function due to the amount of traffic generated by the infected machines. This was determined by the throughput testing (described along with Figure 8) which resulted in a consistent and constant throughput of near 0% meaning no packets, worm or not, was able to make it to their destination.

The next configuration ran involved the use of the PID controller but did not include any intelligence whatsoever on the part of the delayed queue. While the worm spread slightly slower, the network still became congested quickly and throughput of good packets also diminished nearly as quickly as it had with no protection. However, by looking at individual nodes we were able to see the PID controller in action cutting back on the amount of network traffic contributed by the given node. Figure 6 depicts the results of applying the PID controller to a spreading worm.

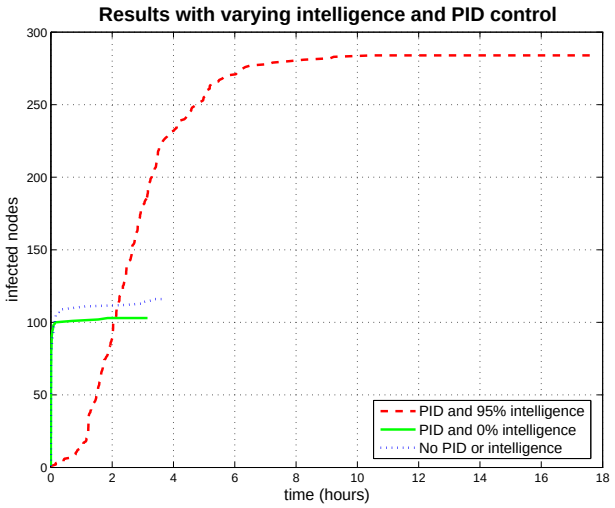

Fig. 7. Rate of infection while varying PID control and intelligence

Finally, a 95% intelligence rate was assumed for the third configuration. That is, when a worm packet arrives, there's a 95% chance it will be placed in the delayed queue. Inversely, when a good packet arrives, there is a 5% chance it will be placed in the delayed queue. After running this test for 18 hours, the results still saw a significant amount of network throughput (particularly among uninfected groups of neighbors) and the amount of time it took for the network to reach the peak connection was over four hours. Figure 7 shows the third configuration alongside the two other configurations shown in Figure 6. Notice the two lines representing the results with no intelligence terminate early – this is because the system throughput had reached nearly 0% and, even though packets were sent, the packets would never reach their destination. The third line was terminated after approximately eighteen hours despite throughput having not reached the near 0% mark. In the next three paragraphs, a better understanding of why significant throughput was still available despite the worm no longer spreading.

Finally, Figure 8 shows the network throughput of the third configuration. (Throughput testing was done by sending a packet from each node to a random destination. If the packet was received, then the throughput succeeded.) It can be seen even after the eighteen hour test cycle, throughput was much greater than the near-0% throughput experienced without any intelligence. Moreover, over two hours of time elapsed before heavy damage was done on the system throughput. These results verify the validity of the approach and leads one to believe that by fine-tuning the algorithm, increasing intelligence, and by including the termination of detected worm packet generating processes, that these results could lead to a greatly larger time window to allow human intervention to completely stop the worm before the network experiences significant downtime. Moreover, by gathering specific state information from each of the 500 virtual nodes, we found that the network had many areas which were completely isolated from the rest of the network except by a single link of two or more hops.

Using an example to explain this better, if we assume a worm on a machine three hops away from a destination computer which is in a close cluster of all uninfected nodes, the probability can be determined if a packet would be sent to that computer and, when sent, the likelihood of it reaching the computer. From section III-A, we know the probability of sending a packet to a given computer under our setup is $0.786\% = 0.00786$. If we make the most open assumption that the network is not overloaded at all, there is a 5% chance at each node that a worm packet will propagate through. Combining the probabilities, we find that $P_{infect} = 0.00786 * (0.05^3) = 9.825 * 10^{-7}$. Clearly the likelihood that one given packet successfully reach an isolated set of nodes is extremely low, even under the very best of assumptions. If you add in network traffic, the flow control provided by the PID controller, and numerous other network aspects, the probability decreases substantially. (Specifically, the probability decreases exponentially based on the number of hops away from the destination computer is from the host computer when intelligence is applied.)

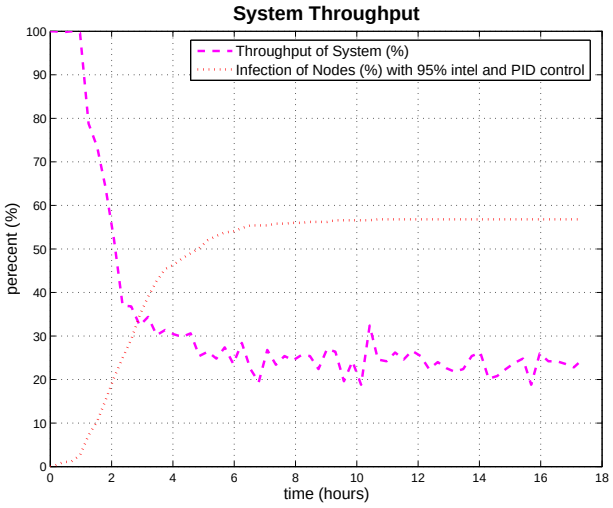

Fig. 8. Throughput of the system while running NetWormPCS with 95% intelligence and PID control enabled

Another aspect to be considered is the misidentification of either normal or worm traffic. The side effects in both cases are on the throughput. At one hand, if normal traffic is send to the delay queue it will decrease the throughput of normal traffic while the goal is to maximize it. On the other hand, if worm traffic is wrongly send to the safe queue it will increase the throughput of worm traffic while the goal is to minimize it.

The *Goliath* framework makes every approach to be as lightweight as possible and will only begin to take a significant amount of computer resources under high network traffic such as under a worm attack. In such cases, where the *Goliath* platform may have to handle multiple packets within one millisecond, the *Goliath* platform may result in packets taking

a longer amount of time to travel from one computer to another or, in the worst case, be dropped prematurely because the buffer was full when the packet arrived. As such, a simulation and the results must be understood that the simulation results would be based on a physical network which can handle the throughput as the throughput able to be handled by their current deployment of the *Goliath* framework.

E. NetWormPCS Deployment

As outlined in the previous sections, deployment of NetWormPCS with the *Goliath* framework was both easy and seamless despite having forty-four nodes across three different types of operating systems (SunOS 5.9, Microsoft Windows XP, and Fedora Core 4). The entire process is summarized in the following steps:

1. The NetWormPCS client was developed and compiled against the *Goliath* framework plug-in class file. This completion allows for NetWormPCS (the higher level application) to interface with the underlying virtual network created and managed by the *Goliath* platform.

2. The *Goliath* virtual network application was deployed to the computers that were to be used in the computation. On the Windows XP machine, *Goliath* was run under a user account by a user double clicking an executable to launch *Goliath* and then doing nothing else. Likewise on the Fedora Core machine, *Goliath* was run under a network managed account by simply invoking the JVM and the *Goliath* application by a single command line command. Finally, the machines running SunOS 5.9 were logged into via SSH and the *Goliath* platform was started by invoking a single line on the command line. The only necessary knowledge about these machines was their network name and/or their IP address.

3. The *Goliath* manager was then run from a separate machine (which could be any operating system which has a JVM). This manager is also invoked by a simple command line containing only two arguments: the location of the XML file to manage the network and the location of some node which has the *Goliath* application running. The manager communicates with a client and sends all the necessary files, topologies, and test conditions to the *Goliath* application – which will configure the virtual network to the specifications described.

At this point, the only thing left to do was to wait for the results. The entire process of setting up the 44 physical computers to host the 504 virtual nodes and launching the *Goliath* manager to distribute the settings about the network to these nodes took less than fifteen minutes and then ran unattended for nearly two days completing one iteration of test results. It is trivial to schedule multiple iterations, differing topologies, different versions of the software, or any combination of those to run unattended until completion.

IV. Related Work

In virtual networking, automated software testing, and worm detection, there is a significant body of work in each of the individual fields. However, as far as we are concerned there are no known published examples of techniques which combine any two of the three areas in any ways similar to that which is presented in this paper. That said, when most researches consider the topic of testing networks with configurable options the first thing which comes to mind is ns-2 [1]. While the *Goliath* solution is able to provide some of the same functionality of ns-2, ns-2 is completely and fully a simulation environment which cannot be deployed across 100 different unpredictable computers. Moreover, in testing of situations where the network encounters heavy traffic (such as in worm attacks, denial of service attacks, and more), variants in operating systems, routing hardware, and other physical network devices often perform in unique ways which currently cannot be simulated by ns-2. While ns-2 currently has a suite of protocols at a programmers disposal, wide adoption of the *Goliath* framework would likely produce much of the same functionality through the ability to compile intermediate classes against the *Goliath* framework between *Goliath* and the higher level application as described in Section II-E.

Possibly the closest work to the *Goliath* network framework is done by Campbell et. al [16] where they define the “The Genesis Kernel”, an entire virtual networking operating system design (not implementation) which is capable of creating entire virtual network architectures (multiple entirely isolated networks with subsets, children virtual networks, etc). In 2001, two years after its initial publication, a secondary paper on the Genesis Kernel was published [17] and a subset of the original Genesis Kernel was implemented. The focus [17] was no longer of a wide enough scope that a testing environment could easily be applied above the network architectures which the implemented Genesis Kernel could produce. Moreover, implementation of the Genesis Kernel was very platform specific and required complete control over the computers in use by the kernel – requirements which directly violate one of the core goals of the *Goliath* solution.

With regard to automated software testing, a wide range of research projects and commercially available products discuss differing flavors of automated software testing. Ranging from automated testing in Java based on formal specifications [18] to the idea of analyzing source code and creating input from the source code to be used in a model checking program [19]. In both cases presented above and in nearly all the papers published on automated software testing, the goal is to verify some code is valid (correctly implements the specifications given) across some range of input. Under that assumption, network input is simply simulated locally by an input file piping in information that would be coming from the network. While such an approach will help ensure the validity of a piece of code, it does not examine the behavior of the system as a whole in an true network – a network filled with propagation delays, failed nodes, and numerous other unpredictable events.

Finally, the motivation for the NetWormPCS case study came from the work done in [20]. In short, [20] proposed a solution to contain a network worm attack by using a feedback control mechanism. In designing NetWormPCS, care was taken to make use of the same concepts which was presented in the proposed solution. However, the proposed solution was

only tested and results were gathered from a single machine using inter-process communication mechanisms to simulate not only an entire network environment, but also the spread of the worm on that environment. However, a cross platform solution could not be tested. In addition, the topology was fixed and demand a large effort to be reconfigured. By making use of the *Goliath* framework, such features as the cross-platform environment and the true use of real (not simulated) network traffic could be used to test part of the solution proposed in [20].

V. Conclusions and Future Work

In this paper we have introduced the *Goliath* solution, an ultra light-weight virtual network layer which allows developers to create applications (or modify existing applications) that run on top of an easily configurable, reconfigurable, self-deploying virtual network. With the *Goliath* solution, software deployments, network topologies, and other conditions may be defined by the user in a simple XML file which can contain a virtually limitless number of network topologies and software deployments to deploy without any user interaction. This paper then presented a comprehensive case study on worm propagation and, by making use of the *Goliath* solution, the NetWorkPCS application was able to be easily tested, improved, and retested many times with a user typing only a single command line command to execute a new iteration of tests on the latest build of the NetWorkPCS application. In general, any type of application can be tested by *Goliath*. However, proper extensions may need to be included to accommodate specific needs of the experiment. As a library of extensions becomes available, the need for customizations will significantly decrease.

The *Goliath* solution provides a new automated network testing tool which allows for the real-time testing of distributed applications without needing anything more than a JVM on a given computer. We believe this approach could significantly reduce the time spent by software engineers in the testing phase of distributed application development.

Although the initial experiments on node counting and worm propagation serve as a basis to show the flexibility and applicability of the *Goliath* framework, we are aware that more case studies are needed. Currently we are working on the deployment of a distributed denial of service attack along with the implementation of a couple of containment solutions. Other distributed applications as the ones mentioned in Section I are also planned. Though *Goliath* demands very little user interaction, the command line needed to launch the application on each of the virtual machines will be done automatically which will further improve *Goliath*'s friendliness. Finally, a module to log user specified data and algorithms to analyze the data are also foresee.

References

[1] K. Fall and K. Fall, "The ns manual." www.isi.edu/nsnam/ns/nsDoc.ps.gz.

[2] I. Foster and C. Kesselman, *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[3] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in grid computing," *Concurrency and Computation: Practice and Experience*, pp. 4–10, January 2003.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56–61, November 2002.

[5] D. Boneh and M. Franklin, "Efficient generation of shared rsa keys," *Journal of the ACM*, vol. 48, pp. 702–722, July 2001.

[6] G. Lawton, "Distributed net applications create virtual supercomputers," *IEEE Computer*, vol. 33, pp. 16–20, June 2000.

[7] D. P. Anderson, "Boinc: A system for public-resource computing and storage," *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, June 2004.

[8] R. F. Boisvert and B. Miller, "Enhancing interactivity of software and data repositories with java," *15th IMACS World Congress on Scientific Computation*, vol. 4, pp. 767–772, August 1997.

[9] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley Professional, 1993.

[10] P. B. Danzig, K. Obraczka, and A. Kumar, "Applications, technologies, architectures,and protocols for computer communication," in *Conference proceedings on Communications architectures & protocols*, (Baltimore, Maryland, USA), ACM, August 1992.

[11] G. C. Goodwin, S. F. Graebe, and M. E. Salgado., *Control system design*. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[12] M. A. Maloof, *Machine Learning and Data Mining for Computer Security*. Springer Verlag, 2006.

[13] R. Dantu, J. W. Cangussu, and A. Yelimeli, "Dynamic control of worm propagation," in *Proceedings. ITCC 2004. International Conference on Information Technology*, vol. 1, pp. 419–423, April 5-7 2004.

[14] R. Dantu and J. W. Cangussu, "Attack containment using feedback control," in *International Conference on Intelligence and Security Informatics (ISI)*, (Atlanta, Georgia), IEEE, May 19-20 2005. (under submission).

[15] T. M. Mitchell, *Machine Learning*. Portland, Oregon, USA: McGraw-Hill, 1997.

[16] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. A. Villela, "The genesis kernel: A virtual network operating system for spawning network architectures," *IEEE Second Conference on Open Architectures and Network Programming Proceeding*, pp. 115–127, March 1999.

[17] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang, "The genesis kernel: A programming system for spawning network architectures," *IEEE Journal on Selected Areas in Communication*, vol. 19, pp. 511–526, March 2001.

[18] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *ACM Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.

[19] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, pp. 203–232, 2003.

[20] R. Dantu, J. Cangussu, and S. Patwardhan, "Attack containment using feedback control." Submitted to IEEE Transactions on Dependable and Secure Computing.