A WAN Emulator for CORBA Applications

Mohammad Alsaeed and Neil A. Speirs
School of Computing Science, University of Newcastle Upon Tyne,
Newcastle Upon Tyne, NE1 7RU, UK.
{mohammad.alsaeed, neil.speirs}@ncl.ac.uk

Abstract

It is usually infeasible to test distributed systems and protocols for dependability, fault-tolerance and performance in the real operation environment. This is attributed to the cost of setting up the hosting distributed environment and the difficulty of controlling such a dynamic environment. The factors are too many and are very dynamic. This paper presents a general purpose tool that emulates a distributed networked environment and gives the user full control over the emulation environment.

1. Introduction and Related Work

There are several architectures that enable building distributed systems that communicate over networks. Unfortunately, it is hard to test these systems over different environments without modifying the sources or the networking libraries of the hosting operating system. Such approaches are not always possible or desirable. Moreover, it has to be done for all the platforms that the system is expected to run on. We have developed a general purpose fault injection architecture and implemented it for CORBA applications. Testing and fault injection are allowed without any modification to the application. First we discuss the limitations of related work that led to the development of this tool. We then provide an overview of the design of the tool and discuss some of the challenges faced during the design and implementation.

Network simulation and emulation is a well studied research field. The need for cost effective environments to test the distributed systems and networking protocols keeps the field alive. It is worthwhile to emphasize the difference between simulation and emulation. While both create virtual networks, emulators differ in the fact that they do not create virtual applications. Instead, they work with the real applications.

Some well-known network emulators are the National Institute of Standards and Technology's NIST Net [1], Dummynet [2], The Ohio Network Emulator [3] and University of Newcastle's Delayline [4]. NIST Net is a Linux kernel module that emulates common effects of networks such as delays, loss, congestion and bandwidth limits. NIST Net's architecture is extensible and allows users to supply their own modules to have more control over the emulation. Luigi Rizzo's Dummynet is a tool that runs on FreeBSD and intercepts IP packets as they go through the protocol stack. Originally designed as a protocol testing tool, Dummynet was improved to become a good tool for enforcing bandwidth and queue limitations, packet delays and loss. The Ohio Network Emulator, ONE, is an emulation tool similar to NIST Net but runs on Solaris systems instead and it has fewer features. Finally, Delayline is a link emulation tool that can emulate delays, queuing and drops. It intercepts packets going through the UNIX protocol stack and pads them with special emulation signals. The interesting feature of Delayline is that it emulates the network at the application level. However, a major drawback is that it requires the applications to be recompiled and linked using its own libraries.

Regardless of how stable and accurate these tools are, they lack the end-to-end application level realization of a wide area network. They are all (except for Delayline) low level tools, requiring either special hardware or modifications to the networking stack (kernel patches.) Some even need modifying or recompiling the application. One other problem is that they were developed for specific environments (e.g. Linux, FreeBSD, etc) and are not easy to port to other environments. Another key drawback is that they lack synthesized traffic generation, i.e. they only emulate physical networks but do not simulate the
existence of other sources of traffic in the network. With the diversity of today's network applications, ignoring other data sources in a wide area network can render test results useless.

In our work, we try to overcome the shortcomings of these earlier emulators. The architecture was designed in such a way that minimizes dependence on the underlying OS and distributed computing platform. Although the implemented version we describe in this paper is for CORBA applications, the architecture is generally applicable to any distributed computing platform (e.g. OGSA) that allows the installation of hooks into the communication subsystem. Our emulator is transparent to the application and requires no modifications, recompiling or patching. It is also independent of the hosting environment for portability. Finally, it gives the applications the sense that there are other - synthetic- applications running at the same time and sharing the networking resources without a perceivable emulation overhead. We have implemented a WAN emulator capable of emulating network faults (loss, delay, corruption, reordering, etc.). Future work will support message interpretation and application-protocol-level fault injection.

2. Architecture

The WAN emulator uses CORBA interceptors [7] to intercept the requests and replies going through the ORB. Figure 1 shows the architecture of the system.

![Figure 1: The WAN Emulator for CORBA Applications](image)

The interceptors are modules that attach themselves to the ORB while the WAN emulator is a CORBA object itself. All communication going through the ORB is intercepted by the interceptors and an emulation request is sent to the WAN emulator. The WAN emulator replies with emulation decisions. Attaching the interceptors to the ORB insures that all applications using the ORB will be subjected to the WAN emulation. This design enables handling multiple applications without changing the architecture or restarting the emulation.

The emulator itself is a composed of three major components; the emulation engine, the trace file reader and the traffic generator. Each of these components will be discussed in more details in later sections. The emulator starts by initializing a trace file reader, a traffic generator and the core emulation engine. Whenever it is started (i.e. put in running state) the core emulation engine, during each unit time, asks the trace file reader for an arrivals count that it uses to start the traffic generator. The traffic generator then generates packet arrival events with arbitrary sizes and passes them to the emulation engine.

When a CORBA application issues a request or replies to a request with results, redirections or exceptions the ORB notifies the installed interceptors before transferring the data to the other end. The interceptors in turn contact the emulator asking it what to do with the data. This is shown as a CORBA arrival event in figure 2 below.

![Figure 2: The architecture of the WAN Emulator for CORBA Applications](image)

Here is a step by step explanation of what goes on during a simple request/reply scenario:

1. The CORBA client issues an invocation request for an operation on the CORBA server.
2. The ORB checks if there are interceptors installed for this client and notifies them if they exist.
3. The interceptors generate a CORBA arrival event to the core emulation engine asking what to do with the request.
4. The emulation engine somehow decides the fate of this request. Then it notifies the interceptors of the emulation decision which could be a delay, a drop, an error or simply
no operation (when the emulator is in stopped state). Let us assume for illustrative purposes that the decision in this scenario is to delay the message by 7 milliseconds.

5. The interceptors follow the emulator decision and delay the request for 7 milliseconds.

6. The ORB transmits the request to the server side.

7. The CORBA server performs the operation and returns the results. We assume in this example that there were no exceptions generated.

8. The ORB checks and finds that the server also has interceptors installed and it notifies the interceptors.

9. The server-side interceptors also generate a CORBA arrival event to the core emulation engine asking what to do with the server reply.

10. The emulation engine, again, makes a decision on how to process the reply. Suppose this time it determines that the message should be delayed for 11 milliseconds. It notifies the server-side interceptors.

11. The server-side interceptors follow the emulator decision and delay the reply for 11 milliseconds.

12. The ORB continues its job and transmits the reply back to the client side.

13. The client receives the reply and continues with its work.

This describes the most straightforward scenario. Complications can ensue when there are redirections, exceptions, communication errors and so on.

3. The WAN Emulator

3.1. Arrivals Trace Files

Simulating Network traffic accurately has been the topic of research for a long time. In the early days of public switched telephone networks, phone companies used (and still are using) arrival models to engineer their networks to achieve the best performance with the minimum cost. Independent and simple Poisson models were used initially. More sophisticated models like Markov-modulated Poisson and Bernoulli processes were used later to show the short range dependence of network traffic. However, based on captured traffic traces [8], researchers in the 1990s became more convinced that Poisson and short range dependence processes cannot model network traffic accurately [9]. Those traces showed that long range dependence is present in both LAN [8] and WAN [9] network traces.

Many researchers however, believe that a WAN has different characteristics. It shows long range dependence (LRD), but it can not be modeled using only LRD models alone. They believe that it is not LRD, not Poisson, but something in between. This is still an open debate that has yet to be resolved.

Our network emulator allows the use of different models to generate traffic traces. It is also possible to use traces captured from real networks. The traces may even be created by hand to express independent arrivals. A simple trace file generation wizard is included with the emulator. It supports Poisson and Self-Similar models. One can create self-similar traces using two different algorithms; the Fast Fourier Transform approximation [5] and the Random Mid-point Displacement [15] algorithms. These two algorithms both model self-similar processes. A screenshot of the trace file generation wizard is shown in Figure 3.

![Figure 3: Modeling Algorithm Selection Screen for the Trace file generation wizard.](image-url)
assumption that makes it easier to plot the trace graphically. The output of the wizard is an arrivals trace file (.att) that can be fed directly into the emulator. The wizard asks the user to specify parameter depending upon the choice of algorithm used for modeling arrivals.

For Poisson arrivals the parameter $\lambda$ denotes the mean number of packet arrivals per unit time. This parameter is called Mean in the FFT and Random Displacement algorithms. The self similar algorithms require two further parameters – the Hurst and $\alpha$ values. The Hurst value is the self-similarity parameter. It may be thought of as the degree of long range dependence in the system. Finally, $\alpha$ is the variance (or degree of peakedness) of the traffic to the mean in a unit time. From this data the wizard generates the arrivals trace file and can display a plot of the generated trace.

3.2. Synthetic Traffic Generator

The synthetic traffic generator is an internal component of the system. It generates network arrival events based on the number of arrivals during a unit time. The included generator generates arrival events that are evenly distributed within the unit time (the bin). Although this is very optimistic, it should not affect the emulation under normal conditions. However, the entire system is designed in such a way that it is easy to build customized generators that can use specific arrival distribution policies. Random distribution and short range dependent distribution are two reasonable policies. Random distribution policy is a planned feature that might be implemented in future releases.

Other than arrival events, the generator is also responsible for the sizes of the generated packets. The included generator generates packets of sizes that follow the distribution of packet lengths on the Internet. It is based on traces from NASA's Amex Internet Exchange [10] and Sprint IP Backbone [6]. Once again, it is possible to build a generator that follows one’s own packet size distribution.

3.3. Emulation Engine

The emulation engine is the core of the entire system. Based on the configuration parameters it calculates the load on the network resources and gives an estimation of how long it will take to transfer a request between two ends. It handles synthetic arrival events from the traffic generator and actual CORBA arrivals from the interceptors. The emulation decisions are affected by some major parameters such as the buffer size, drop and error rates, bandwidth and delays.

The emulation engine is designed in such a way that gives the user full control over the emulation at all times. All emulation parameters including the trace file and unit time could be changed at any time using the graphical user interface. Furthermore, the engine supports forcing certain actions through the GUI like forcing bursty errors, drops, delays and packet arrivals. If a GUI is registered with the emulation engine then the engine generates emulation statistics during each unit time and reports them to the GUI. When the emulator is stopped, or the trace file is exhausted, it becomes transparent and always tells the interceptors to forward the CORBA requests with no delays.

One design goal was limiting the use of resource to minimize the overhead of the emulator. For example, no queues are kept in memory. Only counters are maintained. Moreover, updating these counters is done only when needed. For example, if the unit time was 5 seconds (just for the sake of this example) and there were no synthetic or CORBA arrivals then nothing will be updated for 5 seconds. This minimizes the load on the CPU.

The job of the emulation engine is to co-ordinate the operations of the trace file reader and the traffic generator. It also handles real arrival events sent by the interceptors. The emulation decisions are based on the current load on the networking resources and any run-time emulation events injected by the user during the operation of the emulator.

The emulation engine is controlled via a simple GUI as shown in Figure 4.
The GUI provides everything needed to control the operation of the emulator. It is possible to force the emulator to do certain things to achieve the dynamic nature of networks. Before starting the emulator the emulation parameters that best describe the target network must be set. Preset parameters can be loaded and saved for each target network configuration using the file menu.

Upon starting, the GUI registers itself as a statistics client with the emulation engine. At the end of each unit time, the emulation engine sends some emulation statistics to the registered GUI. Figure 5 shows the GUI’s visualization of these statistics. The emulator statistics panel shows four charts: how much has been sent (a sort of bandwidth utilization), the current buffer load, how many packets have been dropped and finally, how many packets were damaged. The GUI may be modified in the future to log these statistics for future analysis.

Figure 6 below shows the emulation parameters panel where all the emulation parameters can be specified.

Figure 6: The Emulation Parameters Panel

The trace filename is the fully qualified file name of the arrivals trace file to be used. Trace files for particular arrivals models can be generated automatically as described in section 3.1. One can use the browse button to look for an arrivals trace file (.att) in a particular system.

Using the “loopback when trace file ends” checkbox one can configure the emulator to loop back and start at the beginning of the trace file in case the file is exhausted. This is particularly useful if the trace file contains traces for one hour or one day and the user wishes to reuse the data again because their network traffic is repetitive in nature.

The unit time is the duration in milliseconds that each entry in the trace file corresponds to. This value must be consistent with the value used when creating the trace file. Setting the unit time to a very small value (e.g. 10 milliseconds) will decrease the emulator performance and may result in inaccurate emulation.

The propagation delay is the time in milliseconds that a bit takes to travel from one end in the network to the other end. The user can use values straight from Ping or any other such tools. The bandwidth is the width of the link being used. If there is a long path between the two ends, then the bandwidth of the bottleneck link is a reasonable setting. The buffer size of the bottleneck hop is also a reasonable size to use for the buffer size field. The buffer size is used for emulating queueing and packet drops due to overloading the routers. In future versions of our system, these three parameters will be changed so that they can be specified for each component of the network. This will allow accurate emulation of multi-hop networks.

The packet size parameter is the size in bits of the standard packet. It is used to estimate the size of
CORBA requests together with the CORBA packet size modifier parameter. This size is also used when packets are manually forced into the network. It is not used if the traffic generator generates its own packet sizes when it generates synthetic packet arrival events.

The drop and error rates parameters specify the probability of network loss and errors. The example data in figure 6 mean that 1 packet in every 10 million packets would be lost in the network. They also show that 1 in every 10 million packets would suffer bit transmission errors in the network.

The CORBA packet size modifier is used together with the standard packet size to give an estimation of the size of CORBA packets. If a way is found to accurately calculate the actual size of a CORBA request then this parameter will be dropped in future versions of our system.

The GUI also includes a Control Panel to issue commands to the emulator engine. The Emulator Control Panel is shown in Figure 7.

To test CORBA applications over a really dynamic environment, the GUI can be used to force the emulation engine to inject delays, errors, drops and packets. Simply click on the action desired in the Emulator Control Panel. For example, clicking on the “Inject Delays” button in Figure 7 will force the next 10 arrival events to be delayed for 20 milliseconds. The emulation engine will not distinguish between synthetic arrivals and CORBA arrivals. The next 10 arrivals will suffer the 20 millisecond delay. After the 10 arrivals, the engine will go back to its normal operation. The same logic is used when injecting errors and drops. For example, clicking on “Inject Errors” in Figure 7 will cause the emulation engine to inject transmission errors into the next 10 arrivals. The emulator will then return to its normal operation.

The user can inject synthetic packets into the network in the same way. For example, clicking on the “Inject Packets” button in Figure 7 will cause 10 synthetic packets to arrive at the same time into the network. The size of each is the standard packet size specified in the parameters panels.

4. Conclusions and Future Work

We have proposed a general purpose architecture for testing distributed systems running over wide area networks. We have chosen how this architecture can be realized using CORBA as out implementation vehicle. The goals of the system are platform independence and non-modification of the sources and hosting operating systems. This has been achieved using CORBA interceptors. The system is developed in Java and interceptors are plugged to the CORBA applications at runtime eliminating the need to modify or recompile the sources.

The dynamic nature of networked environments and extreme user-controlled conditions are supported using a CORBA graphical user interface that we have designed. The GUI gives the user full control over the emulation at any instance without the need to restart the experiment. It also gives the user an updated overview of the load on the networking resources.

The system currently runs only for single hop networks. We are in the process of extending it to work with multi-hop networks. To emulate a large multi-hop network we have to address scalability and
overhead issues. The emulator must scale well for networks with hundreds or even thousands of nodes while maintaining a limit on the overhead of the emulation. We intend to host the emulation over a local area network where every physical node is responsible for a clique of virtual nodes. One of the nodes will take the position of managing the overall view of the virtual network and starting the emulation. It may be any of the physical nodes participating in the emulation. However, it must carry out the extra functionality of organizing the emulation and maintaining routing information. The current version of the emulator can be downloaded from http://homepages.cs.ncl.ac.uk/mohammad.alsaeed/gpi/

References


