

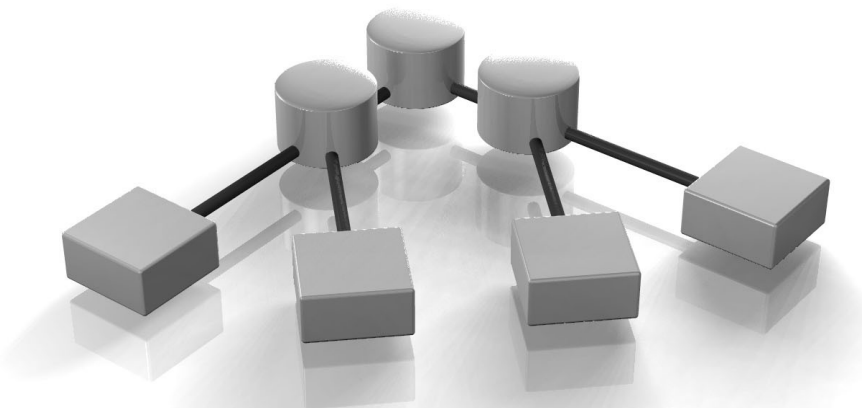
**SIXTH FRAMEWORK PROGRAMME
PRIORITY IST - 2002 - 2.3.1.8
Networked Audiovisual Systems**



Uni-Verse Project

**Addition to D3.4 Protocol Testing:
Testing on (Simulated) Bad Networks
March 1st 2006**

Distribution: Public



STREP project

Project acronym: Uni-Verse

Project full title: A Distributed Interactive Audio-Visual Virtual Reality System

Proposal/Contract no.: 002228

Table of Contents

Introduction.....	3
Test Framework.....	3
The “badnet” iptables plug-in.....	3
Results.....	4
Packet Loss Tests.....	4
Latency Tests.....	5
Packet Loss+Latency Tests.....	5
Conclusions.....	6
Omissions.....	6
Packet Modification.....	6
Multiple Delivery.....	6
Appendix A – Old vs. New Connection Code.....	7
Introduction.....	7
The Change.....	7
Results.....	7

Introduction

This document summarizes some testing done to measure the performance of the Verse network protocol on “bad” or degraded network links. Such results are useful when evaluating Verse, especially when comparing it to more known protocols. They can also be of use when trying to estimate in advance how a Verse-based system will behave.

Test Framework

When doing these kinds of measurements, it is important to have conditions that are as repeatable as possible. It should be possible to speak with certainty about the properties of the link being used during the tests.

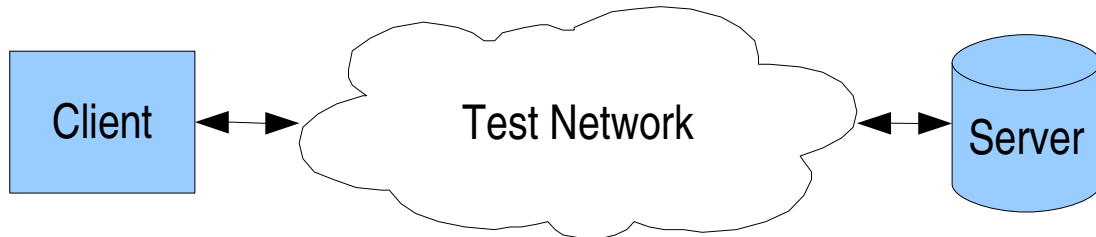


Illustration 1: Test Network

Ideally, we should test Verse over real, large and busy networks, as Illustration 1 illustrates. This would give ample opportunity to see how latency and packet loss affect the performance. However, any real network will typically not have a known, repeatable, level of such flaws. They will occur randomly, at least from the point of view of an external observer¹.

The only large IP network I have access to is the Internet, but since it definitely cannot be said to have known, repeatable, error rates, it is not suitable for these tests.

There are commercial and free network simulators available, but they are either expensive, very complex, or both. Advice from the the KTH Network Operation Centre² was to look into “dummynet”³, a FreeBSD-based tool for testing network protocols. Unfortunately, I have no experience with FreeBSD, nor access to any machine running it. I do have access to and experience with Linux, but there is no direct equivalent software available.

During the investigation of Linux variants of dummynet, I found the libipq⁴ library, and was intrigued. In a nutshell, libipq allows the creation of “plug-ins” to the standard Linux firewall subsystem. Even better, these plug-ins run in “user-space”, meaning that they are not part of the kernel proper. Such plug-ins can thus be easily written by people (such as myself) not familiar with programming directly for the Linux kernel.

After reading up on the libipq API, and some research to clarify what the documentation doesn’t tell, I implemented a simple plug-in that allows some categories of network errors to be emulated. The plug-in is called “badnet”, and its functionality is outlined below.

¹ For instance, a router some number of jumps away from me might stop working due to someone switching it off. That looks random to me, but hopefully not to the person who flipped the switch.

² See <<http://www.noc.kth.se>> for more. KTHNOC runs the SUNET and NORDUnet networks, among other things.

³ See <http://info.i.et.unipi.it/~luigi/ip_dummynet/> for more information.

⁴ Userspace iptables packet queueing library. See man libipq(3) on a Linux system, or the Web version thereof, at <<http://www.cs.princeton.edu/~nakao/libipq.htm>> for more information.

The “badnet” iptables plug-in

“Iptables”⁵ is the name given to the packet filtering framework inside the Linux kernel. Iptables is used to implement common low-level network functions such as packet filtering (firewalling) and network address translation. While iptables resides in the kernel, it is possible to ask it to send packets over to user-space, where an ordinary program using libipq can access the packets. The user-space program can then make decisions on whether or not a packet should be delivered, and when.

I wrote “badnet”, which is such a user-space program, specifically to test Verse with. In typical use, iptables and badnet cooperate so that all traffic that flows between the application under test and the external world, must first flow through badnet. Please note that in typical use, iptables is configured to send *both* incoming and outgoing packets through badnet. Badnet sees the aggregate packet stream, and currently does not take the



Illustration 2: Test Setup With "badnet"

direction of a packet into consideration when applying its problem simulation rules.

Illustration 2 above shows the layout of the network during tests. The direct link between badnet and the Verse server is in practice a switched Ethernet connection, but the switch itself doesn't participate so it was omitted from the illustration.

Badnet currently supports simulating two kinds of network problem: packet loss, and packet latency. The former is simply implemented by randomly dropping packets, i.e. totally preventing a packet from being delivered. This simulates a busy network, where routers are forced to drop UDP traffic, as well as network with actual “hard” problems, such as broken connections. Packet latency is simulated by randomly delaying a given percentage of packets. This not only delivers that data “late”, but also out of order, since a delayed packet followed by a non-delayed packet will arrive in the reverse order to the other end.

These two modes can be freely combined, and the probabilities are user-configurable. The delay when introducing latency is normal-distributed, while other probabilities are uniformly distributed.

Results

This section presents the results arrived at after the testing, in subsections according to the kind of fault that was simulated.

Packet Loss Tests

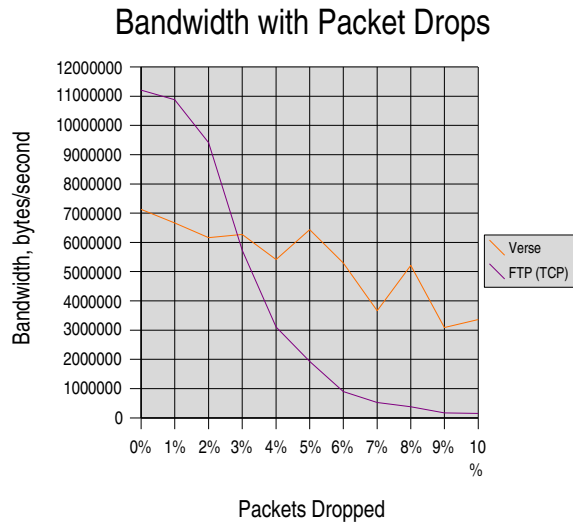
The first round of tests used packet loss only. The test case was upload of a large (5120*5120 pixels) PNG image using the “pngload” test client⁶. An option was added to pngload to compute the average bandwidth, by simply dividing the size in bytes of the raw image, and dividing with the total time needed for the upload. Note that this measures actual “payload” data size only, there is also Verse protocol overhead which is not included in these numbers.

As a reference, a large file was uploaded over FTP to an FTP server running on the same machine as the Verse server, i.e. with the network configuration shown in Illustration 2. The bandwidth figure used was computed by the “ncftp” FTP client used. The latter number is only accurate to at most five digits.

⁵ See < <http://www.netfilter.org/> > for more information.

⁶ See < <http://projects.blender.org/viewcvs/viewcvs.cgi/verse-tests/?cvsroot=verse> > for more information.

Packet Loss Probability	Verse, Average Bandwidth	FTP (TCP), Bandwidth
0%	7129628	11210000
1%	6667140	10880000
2%	6167824	9410000
3%	6270634	5730000
4%	5417532	3110000
5%	6439652	1930000
6%	5295988	898020
7%	3667993	529320
8%	5217603	382480
9%	3092913	174140
10%	3363508	156470



Verse's bandwidth starts out significantly lower than FTP over the non-disturbed link, but also does not degrade as quickly as the FTP does.

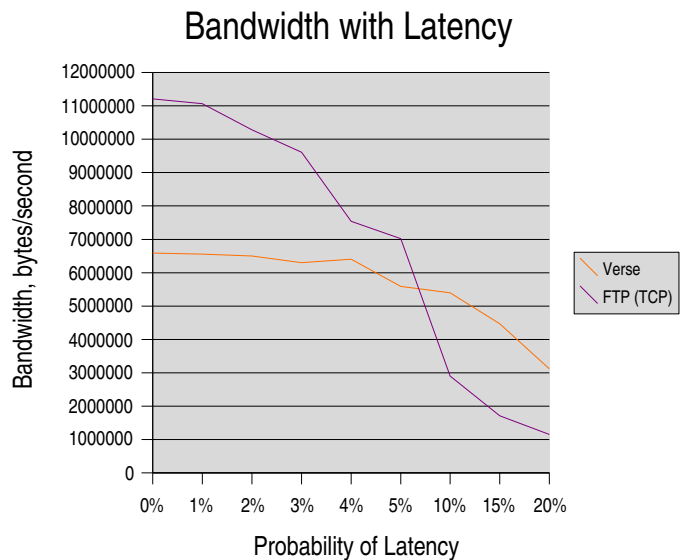
Not shown in the above table is the fact that although the Verse's transfer performance is decent as packet loss increases, it becomes increasingly harder to *connect* in the first place. Why this is I do not understand, it is something that needs looking into.

Latency Tests

Latency response testing was also performed. This uses a different mode of badnet's, that randomly picks packets to delay. The delay is also selected randomly, by picking values in a normal-distributed space. All tests were run with a mean delay of 10ms, and a variance of 5ms.

Again, the operation performed during the tests was upload of a large PNG image, using both Verse and FTP.

Latency Probability	Verse, Average Bandwidth	FTP (TCP) Bandwidth
0%	6591410	11210000
1%	6556648	11060000
2%	6502450	10280000
3%	6300092	9610000
4%	6405154	7540000
5%	5590992	7020000
10%	5398404	2900000
15%	4473854	1710000
20%	3118012	1150000



Again, we see Verse having a lower bandwidth on a network without artificial latency, but also that it degrades less sharply as the number of packets with latency increases.

Conclusions

I did not find Verse's reactions to worsening network conditions to be particularly alarming. I believe the reliance (by design) on order-independent delivery of data gives benefits in both with packet loss and latency.

I did find the very obvious problems with establishing a connection across a network with loss to be problematic, and think that part of the protocol should be investigated and, hopefully, improved. One problem here might be that Verse does resends during connection-establishment fairly seldom, so once packets begin disappearing, it takes a long time to get one through.

Omissions

The above testing, limited by the current version of the badnet tool, only includes two kinds of error: packet loss (drop) and packet latency. There are at least two more classes of error that might be interesting to test Verse under: packet modification, and multiple delivery.

Packet Modification

Packet modification is just what it sounds like: modifying one or more of the data bytes of a packet's payload. This isn't very interesting, since a random error would have a high probability of creating a packet with an illegal UDP header, and such packets are dropped by the system. So, this would just be a long-winded way of dropping packets, a test that is already supported.

Generating damaged packets with a corrected checksum, by re-computing it after introducing the error, is of course also possible. This would probably crash Verse, since Verse doesn't have any application-level data integrity testing.

Multiple Delivery

Currently, each packet processed by badnet is delivered either zero, in case of a drop, or one time otherwise. In the latter case it might be delayed and thus arrive late, but it does arrive. Real networks might exhibit the property that some packets are delivered multiple times, as the result of some kind of error. Simulating this behavior is non-trivial in badnet, since the underlying libipq API does not support it as far as I know.

Going outside libipq should be possible, but would require more engineering time spent on badnet. I do not know how important this case is to test, i.e. how often it occurs in real life networks.

Appendix A – Old vs. New Connection Code

Introduction

This appendix describes some recent changes to the core Verse network connection handling. These changes came about after initial reports about very bad effective bandwidth came in. After some investigation by me and Eskil Steenberg (who wrote most of the core code, and knows it far better than I do), we did some small changes.

The Change

The change is in a single C source file, “v_connection.c” in the core Verse module in CVS. Here we find the the very central function `verse_callback_update()`, that has the following responsibilities:

- Listen to incoming network traffic.
- “Parse” any received packets, and issue callbacks to function registered by the application programmer, as needed.
- To avoid starving, also send out any outbound traffic.

The theory was that the latter point was not being addressed well enough by the current code, leading to outbound traffic sending getting too little CPU time.

What we did was rewrite the part of the update function that sends out data, to do so for a longer period of time.

Results

The changes were committed to Verse CVS on Jan 30 2006, and thereby immediately made available to any users of Verse. To document and illustrate the improvement, here are results of a simple benchmark.

The test was to simply upload a large (5120*5120 pixels) image to a Verse server, from a PNG image. This was done on a single machine, never touching a physical network at all. The “badnet” packet filter was active, to make the numbers comparable to the ones above, but all filtering was turned off.

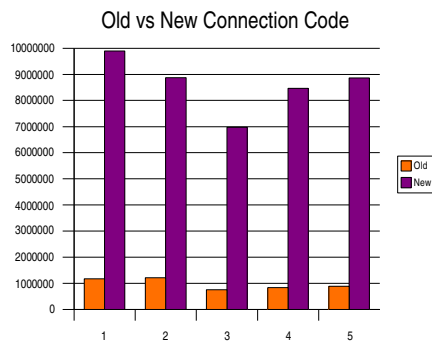


Illustration 3: Bandwidth before and after the change

Illustration 3 on the left shows the measured bandwidth before and after applying the change to both client and server Verse libraries.

The average improvement is around a 9X speedup, making this a very notable improvement.