

AMIE Transmission Benchmark

Christopher A. Baumbauer
cab@sdsc.edu

July 30, 2007

1 Background

After the TGCDB upgrade, there were still problems with NPU packet processing taking much longer than they were supposed to on the order of tens of seconds instead of the expected timeframe of milliseconds. As of December 25th, Michael Shapiro discovered a flaw in the TGCDB where there was a missing index on one of the three tables used to process incoming job information. With this fixed, we began the process of determining other avenues for improvement, one of which is the transfer mechanism itself.

The transfer mechanism used by the TeraGrid is called AMIE or the Account Management Information Exchange which consists of a collection of data structures used to represent the various forms of account data (in an XML format), and a collection of Perl scripts that are used to perform the actual transfer by pulling the packet data from one database, converting it into a file, then copying the contents of the file to the destination site where that file is then uploaded into their local database. The current focus of this experiment is to look at how data gets from Site A to Site B, and determine if there is a way this could be improved in both timeliness and security.

The current approach to the data transfer utilizes SSH to provide the communication mechanism. The usage of ssh is as follows:

```
Site A generates an XML file;  
Place file in directory;  
foreach File in directory do  
    read in file;  
    open ssh connection to Site B;  
    stream contents of file;  
    close ssh connection to Site B;  
end
```

The approach just described will be considered the reference or base setup, and will provide the base time that the other methods will be compared against. Some approaches that are being sought include:

- Instead of invoking SSH to stream the data over, use SCP to copy the data directly to the destination directory. Examine this approach with transferring a single file at a time, or performing a batch send consisting of multiple files in the same call. Note that for future releases, this will no longer require AMIE to support a receiver mode.

- Adjust the AMIE receive mode to turn AMIE into a ‘server’ whereby a connection is opened once, and then multiple files can be transmitted over a single connection. Note that in this case, the data will need to be encrypted via SSL by coding in calls to the OpenSSL libraries. We will test streaming multiple files over a single connection, and then opening/closing the connection for each file.
- Perform the previous two tests without and then with compression by first compressing each file.
- Perform the first two tests and then tar and compress/zip the contents together and then send as a single connection. Note that we may need to ASCII armour the text as it goes out.

What will follow is a description of the framework that will be constructed to perform the test, what will need to happen at each site for the test to happen, the results of the test, and then finish with concluding remarks. An appendix will be included which will contain all of the code used to conduct the test.

2 Framework

As this test focuses on just the transfer mechanism, we will write some scaffolding that can be used to run the desired tests on both sides of the communication channel. While we may get more accurate numbers by modifying the AMIE program, pushing out test changes may prove to be somewhat error-prone. This way, we can collect data from the phase that matters most and that focuses specifically on the transfer mechanism instead of the entire piece.

This scaffolding will be called from within a master shell script that will invoke `date(1)` to pull the time before and after execution. The master shell script will provide cleanup operations on the remote site as well.

For the component that will live on Site A (or the TGCDB site for now), a script will be written that matches the algorithm currently used by AMIE to obtain the list of files for transmission, and then depending on the transmission type, invoke the new approach. The script will be invoked with the desired primary test. Additional options can be passed to this script to determine if we should compress the files before transmission, tar the files before sending, or perform both. Bear in mind that the `scp` test won’t have any interaction with a client side program, so to mimic the operation that will be required to happen on the destination, this script will call `tar/gzip` on the remote site to decompress the file.

In the other case, a light-weight server will be written with the sole purpose of reading in the contents of the packets, and writing them into files. The script will mimic the `amie -r` functionality for the base test while also having a separate daemon component that will listen for a connection from the Site A test component to receive the necessary files while also making sure the appropriate actions are handled (such as untarring and decompressing)

Please note that for the client/server approaches, both will stream text over a SSL encryption layer which can be controlled with finer grained SSL certificates issued from a single recognized certificate authority.

The tests will consist of sending a random number of 100, 250, 500, and 1000 packets from the source to the destination. I chose these packets to allow for an honest comparison of the possible worst case that we could look forward to during a new allocation period or high npu load. For the

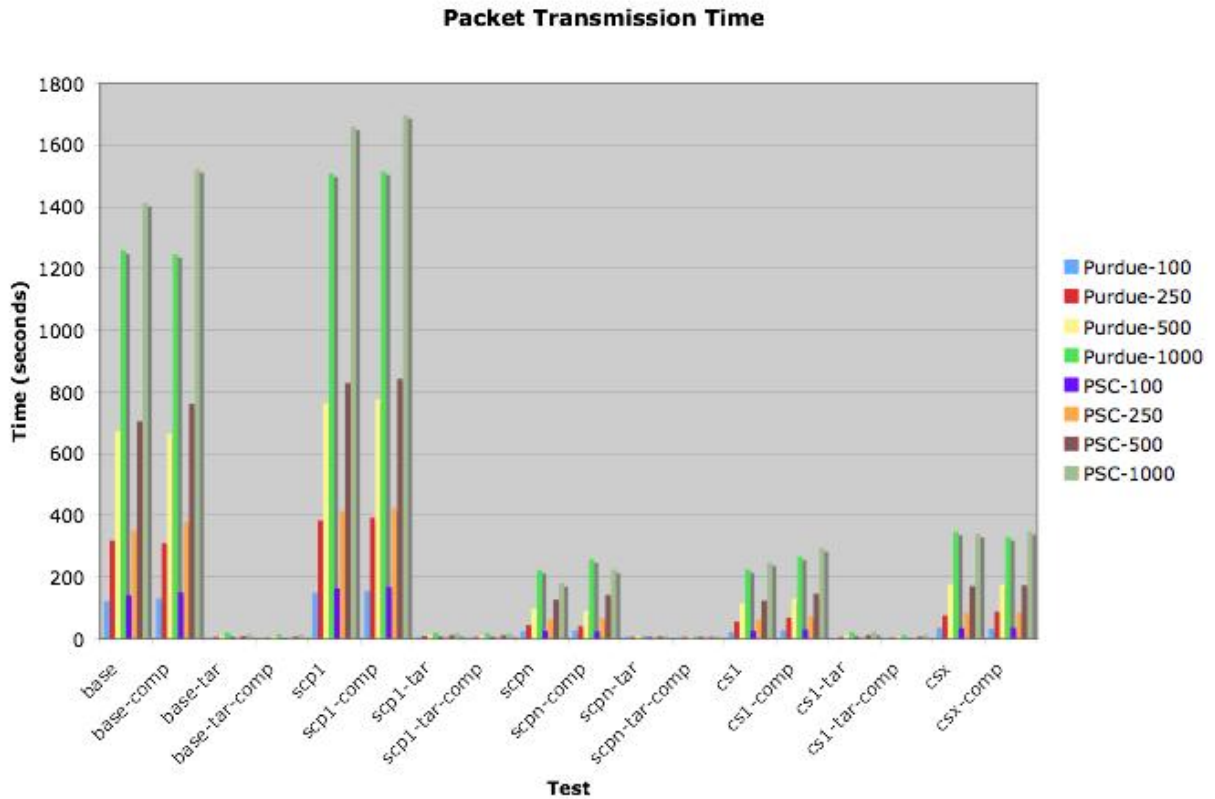
test sites, I chose both PSC and Purdue for their distance from the TGCDB, and Purdue's case where they would stand to gain the most with respect to a new solution. I went for two sites in order to ensure collaborating results.

3 Results

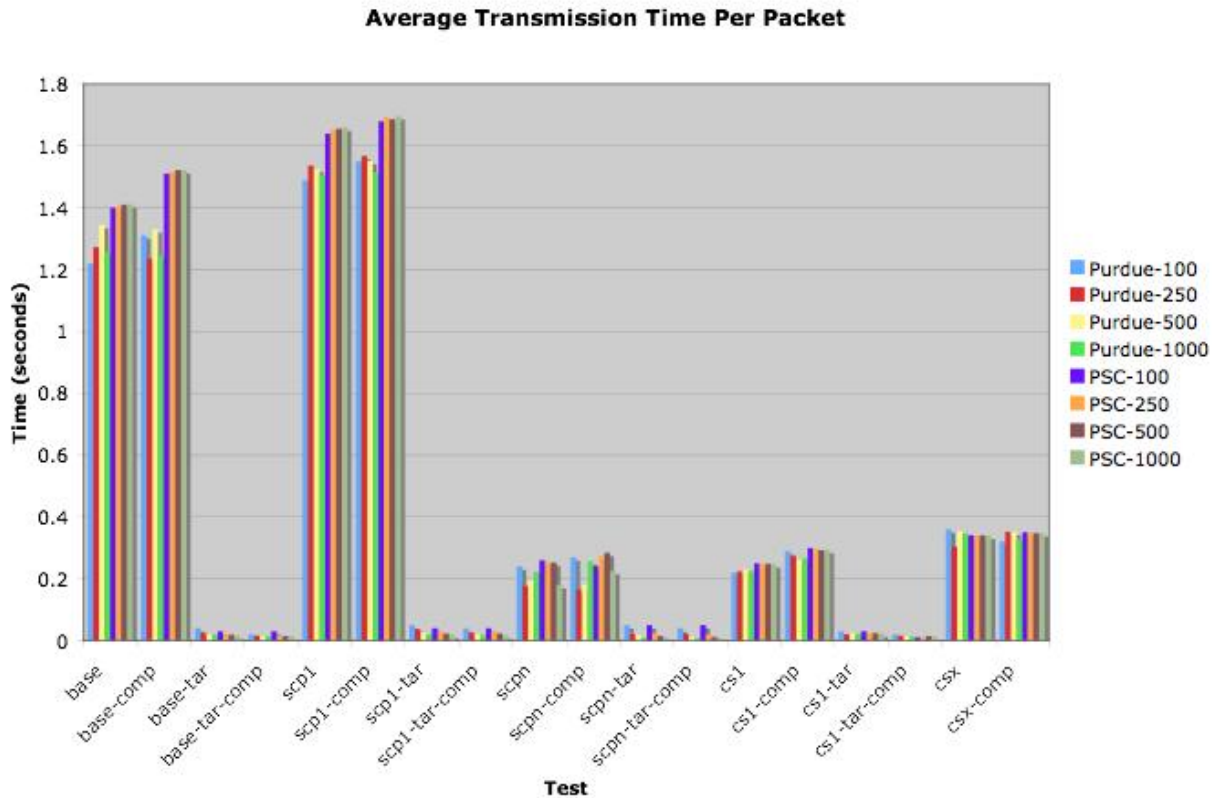
Before I can go into details of the graphical analysis, here is some information about the tests:

- base - The reference implementation which reflects how AMIE transmits packets to this day.
- scp1 - Invoke scp as the transfer mechanism for each file in the directory we want to send.
- scp_n - Invoke scp as the transfer mechanism, but pass in all files in the directory.
- cs1 - Invoke the client/server transfer mechanism by using a single connection to stream the data to the receiving site.
- cs_x - Invoke the client/server transfer mechanism by opening a connection for each file.

Each graph will have something along the lines of `-comp`, `-tar`, and `-comp-tar` which correlates to the use of compression, tarring, and both for the specified primary test. You will notice that the cs_x test is missing calls to tar which were omitted as the tar operation is equivalent to sending a single file which was performed in the cs1 test.



The above graph called the Packet Transmission Time consists of the total amount of time it took to run each test for the given number of packets per site. Each bar mentioned in the legend corresponds to the run of n packets for each site.



The second graph called Average Transmission Time Per Packet is just that. The average times are in packets per seconds, and as above, each bar posted for the test consists of the site/packet run. I opted to include this graph to assist with providing a mean transmission time for each packet regardless of its size since the inform-transaction-complete packet is considerably smaller than the request-project-create packet. Also, this graph does help better illustrate some of the tests where they had a much better transfer time than the base tests, but I'll go into more detail in the analysis section.

4 Analysis

Looking at both graphs, it is very easy to tell three things:

1. The current approach is very expensive
2. Compression will not improve performance, and may have a detrimental impact.
3. The number of files opened, read, and transmitted has more of an impact than the number of connections.

The use of scp over a list of files (scpn test) did as well as using the client/server approach. However, the client/server approach also performed evenly over the number of packets. I was very surprised to notice how combining all of the files into a single tar archive, and then transmitted that way resulted in the best performance.

5 Conclusion

There are several things to note from the results. The first is that the use of ssh/scp provides the worst performance when it comes to transferring files. The second is that the current process of reading the entire contents of the file, transmitting it, and then closing the file is also very expensive and can be streamlined considerably by using tar. There is the drawback with tar, where AMIE will lose its control of how it names files in the transmission process by defaulting to the originator's name. It also appears that the best transmission method, without compression and/or tarring, may be the use of scp with multiple files given as arguments, however it was brought to my attention that this approach may be very difficult to control safely and won't pass the security-wg. As an alternative, the client/server test over the single socket may provide the second best performance, and can be locked down.

6 Acknowledgments

I would like to express my deep thanks to Preston Smith (Purdue) and Rob Light (PSC) for their help in this collaboration and the logistics needed to pull off the tests. I would also like to thank Laura McGinnis (PSC) for administrative support during the day of the test. Dave Hart (SDSC), Michael Shapiro (NCSA), and Steve Quinn (NCSA) for providing test ideas and also a healthy debate that resulted in the need for this test.

A Source Code

Three files have been attached that drove the testing framework. The first file consists of the backend script that was run from SDSC that drove the transfer process, and would resemble what AMIE would run when it sends packets to the remote site. The second file is the receiver script that would correspond to the counterpart program used to receive the file, and possibly translate the filename into something that the main AMIE script would recognize and handle. The last script is the driver script that was used to start the entire process and also perform the necessary cleanup.

Note that I have removed any hard-coded passwords to ensure that the certificates used in the test won't be compromised.

A.1 Connectivity.pl - The Benchmark Backend Script

```
#!/usr/bin/env perl

# connectivity.pl - A collection of tests used to perform various methods of
# AMIE packet transmission given a source directory containing the packets we
# want to send. Bear in mind, this will be very dirty, but every attempt will
# be made to keep things similar to the current workings of the AMIE
```

```

# transmission mechanism.
#
# Created by Chris Baumbauer <cab@sdsu.edu>

# NOTE: This will consist of the transmission side of things. The receiving
# end is expected to run either their own client, or rely on other tools.

use strict;
use warnings;
use IO::Socket::SSL;
use File::Basename;

if ((scalar(@ARGV) < 2) || (scalar(@ARGV) > 6)) {
    print "Usage: _connectivity.pl _packet-dir _test-name _[comp] _[tar] _[nnnn]\n";
    print "Where _test-name can be one of the following:\n";
    print "\tbase*_ _Reference _test _of _current _AMIE/TGCDB _operations\n";
    print "\tscp1*_ _Copy _individual _files _from _source _to _destination\n";
    print "\tscp*_ _Copy _a _bundle _from _source _to _dest _in _one _burst\n";
    print "\tcs1*_ _Connect _to _a _destination _server, _and _stream _through _1 _socket\n";
    print "\tcs*_ _Connect _to _a _destination _server, _and _stream _through _X _sockets\n";
    print "And these are the following supported options...\n";
    print "\tcomp _Enable _gzip2 _compression (optional)\n";
    print "\ttar _Tar _the _files _before _transmission, _and _untar _at _the _end\n";
    print "\tnnnn _A _port _number _for _the _remote _host (defaults 6666)\n";
    print "\n\n";
    print "* _These _operations _support _tar.\n";
    exit(-1);
}

my $packet_dir = shift(@ARGV);
my $test = shift(@ARGV);
my $enableCompression = 0;
my $enableTar = 0;

# Some useful constants
my $username = "amie";
my $hostname = 'some-host'; # The peer host
my $testhomedir = %ENV[PWD]; # location of peer program
my $data = '/tmp'; # where the results should be stashed
my $port = 6666; # The peer port number to connect to

my @args = @ARGV;
while (my $a = shift(@args)) {
    if ($a eq "comp") {
        $enableCompression = 1;
    } elsif (($a eq "tar") && ($test ne "csx")) {
        $enableTar = 1;
    } elsif ($a =~ /\d+/) {
        $port = $a;
    }
}

```

```

# Perform the 'base' test which should resemble how things are currently done.
if ($test eq "base") {
    my @files = glob($packet_dir . "/*");
    my $starFile = '';

    # Define command to invoke
    my $cmd = "ssh_-$username@$hostname_-$testhomedir/receiver.pl";

    if ($enableCompression == 1) {
        $cmd .= "_-z";
    }

    if ($enableTar == 1) {
        $starFile = "connectivity-$$tar";
        foreach my $f (@files) {
            if ( -f $starFile ) {
                'tar -rf $starFile $f';
            } else {
                'tar -cf $starFile $f';
            }
        }

        @files = ( );
        push (@files, $starFile);

        $cmd .= "_-t";
    }

    foreach my $file (@files) {
        my $cmdfile = "/tmp/connectivity.$$";
        open FP, ">$cmdfile";

        if ($enableCompression == 0) {
            print FP "$cmd<<_ '$file '\n";
        } else {
            print FP "gzip_-c_ '$file '|_-$cmd\n";
        }
        close FP;

        my $pid;

        if ($pid = fork) {
            waitpid $pid, 0
        } elsif (defined $pid) {
            die "Unable_to_exec." unless exec '/bin/sh', $cmdfile;
            exit 127;
        } else {
            die "Unable_to_fork!\n";
        }
    }
}

```

```

        unlink $cmdfile;
        if ($starFile ne "") {
            unlink $starFile;
        }
    }
}

if ($test eq "scp1") {
    my @files = glob($packet_dir . "/*");
    my $starFile = "";
    my $i = 0;

    if ($enableTar == 1) {
        $starFile = "amie-$$tar";
        foreach my $f (@files) {
            if ( -f $starFile ) {
                'tar -rf $starFile $f';
            } else {
                'tar -cf $starFile $f';
            }
        }

        @files = ( );
        push(@files, $starFile);
    }

    foreach my $file (@files) {
        if ($enableCompression == 0) {
            'scp $file $username@$hostname:$data/amie-$i';
            if ($? != 0) {
                die "Error: _Unable_to_execute!\n";
            }
        } else {
            'scp -C $file $username@$hostname:$data/amie-$i';
            if ($? != 0) {
                die "Error: _Unable_to_execute!\n";
            }
        }
    }

    if ($enableTar == 1) {
        'ssh $username@$hostname tar -xf $data/amie-$i -C $data';
        unlink $starFile;
    }

    $i += 1;
}

if ($test eq "scpn") {
    my $starName = "";
    if ($enableTar == 1) {

```

```

    $starName = basename($packet_dir);
    $starName .= ".tar";
    'tar -cf $starName $packet_dir';
    $packet_dir = $starName;
}

if ($enableCompression == 0) {
    'scp -r $packet_dir $username@$hostname:$data';
    if ($? != 0) {
        die "Error: Unable to execute!\n";
    }
} else {
    'scp -C -r $packet_dir $username@$hostname:$data';
    if ($? != 0) {
        die "Error: Unable to execute!\n";
    }
}

# If we tar the file up, be sure to untar it as well
if ($enableTar == 1) {
    'ssh $username@$hostname tar -C $data -xf $data/$starName';
    'ssh $username@$hostname rm $data/$starName';
    unlink $starName;
}
}

if ($test eq "cs1") {
    my @files = glob($packet_dir . "/*");
    my $sock = IO::Socket::SSL->new(PeerAddr => "$hostname",
        PeerPort => "$port",
        Proto => 'tcp',
        SSL_use_cert => 1,
        SSL_verify_mode => 0x02,
        SSL_ca_file => "certs/test-ca.pem",
        SSL_key_file => "certs/client-key.enc",
        SSL_cert_file => "certs/client-cert.pem",
        SSL_passwd_cb => sub { return "*****" }) or
        die "Unable to open connection: " . IO::Socket::SSL::errstr . "\n";

    # Verify the certificate
    #print $sock->dump_peer_certificate;

    my $starFile = "";
    if ($enableTar == 1) {
        $starFile = "amie-$.tar";
        foreach my $f (@files) {
            if ( -f $starFile ) {
                'tar -rf $starFile $f';
            } else {
                'tar -cf $starFile $f';
            }
        }
    }
}

```

```

    }

    @files = ( );
    push(@files , $starFile);
}

# Iterate through the list of files and send them down the pipe
my $EOF = "ENDOFFILE\n";
foreach my $file (@files) {
    if ($enableCompression == 0) {
        open FP, "<$file";
    } else {
        'gzip -c $file > $file.gz';
        open FP, "<$file.gz";
    }

    my $base = basename($file);

    if ($enableTar == 0) {
        print $sock "$base\n";
    } else {
        print $sock "$base.tar\n";
    }

    my $result = <$sock>;

    if ($result =~ /OK/) {
        print "Successfully created: _$base\n";
    }

    while (<FP>) {
        my $buf = $_;
        syswrite $sock, $buf;
    }

    if (($enableCompression) || ($enableTar)) {
        print $sock "\n\n"; # force a newline
    }
    print $sock $EOF;

    $result = <$sock>;
    if ($result =~ /OK/) {
        print "Finished file: _$base\n";
    }

    close(FP);
    if ($enableCompression == 1) {
        unlink "$file.gz";
    }

    if ($enableTar == 1) {

```

```

        unlink "$file";
    }
}

# Close the connection
close($sock);
}

if ($test eq "csx") {
my @files = glob($packet_dir . "/*");
foreach my $file (@files) {
    my $sock = IO::Socket::SSL->new(PeerAddr => "$hostname",
        PeerPort => "$port",
        Proto => 'tcp',
        SSL_use_cert => 1,
        SSL_verify_mode => 0x02,
        SSL_ca_file => "certs/test-ca.pem",
        SSL_key_file => "certs/client-key.enc",
        SSL_cert_file => "certs/client-cert.pem",
        SSL_passwd_cb => sub { return "*****" }) or
        die "Unable to open connection:_" . IO::Socket::SSL::errstr . "\n";

    # Verify the certificate
    #print $sock->dump_peer_certificate;

    # Iterate through the list of files and send them down the pipe
    if ($enableCompression == 0) {
        open FP, "<$file";
    } else {
        'gzip -c $file > $file.gz';
        open FP, "<$file.gz";
    }

    my $base = basename($file);

    print $sock "$base\n";
    my $result = <$sock>;

    if ($result =~ /OK/) {
        print "Successfull created:_"$base"\n";
    }

    while (<FP>) {
        my $buf = $_;
        print $sock $buf;
    }

    close(FP);
    close($sock);
    if ($enableCompression == 1) {
        'rm $file.gz';
    }
}
}

```

```

    }
  }
}

```

A.2 Receiver.pl - The Benchmark Frontend Receiver Script

```

#!/usr/bin/env perl

# receiver.pl - An extremely simple perl script to mimic the amie -r mode of
# operation by reading in STDIN, and writing the contents to a known output
# directory
#
# Created by Chris Baumbauer <cab@sdsc.edu> as a part of the amie testing
# framework.
#
# 2007-06-06 - Added a command option to toggle a "server" mode that will
# accept connections for streaming files given the -d option.

use IO::Socket::SSL;
use strict;
use warnings;

my $server = 0;
my $enableCompression = 0;
my $enableTar = 0;
my @args = @ARGV;

# Some useful constants
my $hostname = 'some-host'; # Our hostname
my $data = '/tmp/cab'; # location of the results
my $port = 6666;

while (my $a = shift(@args)) {
  if ($a eq "-d") {
    print "Enable_daemon_mode\n";
    $server = 1;
  } elsif ($a eq "-dx") {
    print "Enable_single-file_daemon_mode\n";
    $server = 2;
  } elsif ($a eq "-z") {
    print "Enable_compression\n";
    $enableCompression = 1;
  } elsif ($a eq "-t") {
    print "Enabling_tar... \n";
    $enableTar = 1;
  } elsif (($a =~ /\d+/) && ($server > 0)) {
    print "Port_number:_$a\n";
    $port = $a;
  }
}

```

```

if ($server == 0) {
  my $filename = "$data/amié-test.$$";
  if ($enableCompression == 0) {
    open FP, ">$filename" or die "Unable to open tmp file: $filename\n";
  } else {
    open FP, "|zcat >$filename" or die "Unable to open tmp file: $filename\n";
  }

  while (<STDIN>) {
    print FP;
  }

  close(FP);

  if ($enableTar == 1) {
    'tar -C $data -xf $filename';
    'rm $filename';
  }
} elsif ($server == 1) {
  my $server = IO::Socket::SSL->new(
    LocalAddr => "$hostname",
    LocalPort => "$port",
    Reuse => 1,
    Listen => 10,
    Proto=>'tcp',
    SSL_verify_mode => 0x01,
    SSL_use_cert => 1,
    SSL_ca_file => "certs/test-ca.pem",
    SSL_key_file => "certs/server-key.enc",
    SSL_cert_file => "certs/server-cert.pem",
    SSL_passwd_cb => sub { return "*****" } or
    die "Unable to create socket: " . IO::Socket::SSL::errstr . "\n";

  while (my $client = $server->accept()) {
    print "Received connection from: " .
      $client->sockhost() . "\n";
    my $filename = "$data/amié-test.";
    my $buf = "";

    if (ref($client) eq "IO::Socket::SSL") {
      print "SSL has been enabled\n";
    }

    # NOTE: This is where we will perform our check that the client
    # coming in is on our white-list to allow. We can safely close
    # connections that don't match.
    #print "Cert info:\n" . $client->dump_peer_certificate;

    # Everything else should be skipped.
    while (my $cmd = <$client>) {
      chomp($cmd);

```

```

    print "Received_response:_$cmd\n";

    my $name = $filename . $cmd . "." . time();
    print $client "OK\n";
    if ($enableCompression == 1) {
        open (FP, ">$name.gz");
    } else {
        open (FP, ">$name");
    }

    while (<$client>) {
        my $buf = $_;
        last if ($buf =~ /^ENDOFFILE$/);
        print FP $buf;
    }

    close(FP);
    if ($enableCompression == 1) {
        'gzip -df $name';
    }

    if ($enableTar == 1) {
        'tar -C $data -xf $name';
        unlink $name;
    }
    print "Done_with:_$name\n";
    print $client "OK\n";
}

close($client);
}
} elsif ($server == 2) {
    my $server = IO::Socket::SSL->new(
        LocalAddr => "$hostname",
        LocalPort => "$port",
        Reuse => 1,
        Listen => 10,
        Proto=>'tcp',
        SSL_verify_mode => 0x01,
        SSL_use_cert => 1,
        SSL_ca_file => "certs/test-ca.pem",
        SSL_key_file => "certs/server-key.enc",
        SSL_cert_file => "certs/server-cert.pem",
        SSL_passwd_cb => sub { return "*****" } or
    die "Unable_to_create_socket:_". IO::Socket::SSL::errstr . "\n";

    while (my $client = $server->accept()) {
        print "Received_connection_from:_".
            $client->sockhost() . "\n";
        my $filename = "$data/amie-test.";
        my $buf = "";

```

```

if (ref($client) eq "IO::Socket::SSL") {
    print "SSL_has_been_enabled\n";
}

# NOTE: This is where we will perform our check that the client
# coming in is on our white-list to allow. We can safely close
# connections that don't match.
#print "Cert info:\n" . $client->dump_peer_certificate;

# Everything else should be skipped.
my $cmd = <$client>;
chomp($cmd);
my $name = $filename . $cmd . "." . time();
print $client "OK\n";

if ($enableCompression == 1) {
    open (FP, ">$name.gz");
} else {
    open (FP, ">$name");
}

while (<$client>) {
    my $buf = $_;
    print FP $buf;
}
close(FP);

if ($enableCompression == 1) {
    'gzip -df $name';
}
print "Done_with:_$name\n";
close($client);
}
}

```

A.3 testRun.sh - The Benchmark Backend Execution Script

```

#!/bin/sh

# testRun.sh - Run the test program through the races by collecting time samples
# before, and after each invocation of the connectivity.pl script and write it
# to a csv file suitable for parsing. The expected call is:
#
# testRun.sh logfile testdir
# Where logfile is the file to dump the results into (default to append)
# and testdir contains the files that we want to copy over
#
# Created by Chris Baumbauer <cab@sdsc.edu>

logfile=$1
testdir=$2

```

```

basedir=$PWD # location of the base scripts
timestamp="date +%s"
remotehost="username@hostname"
remotedata="/tmp/amie-test"

# Perform sanity checking with the arguments
if [ "x$logfile" = "x" ]; then
    echo "The logfile must be defined"
    exit 1
elif [ "x$testdir" = "x" ]; then
    echo "The test directory must be specified"
    exit 2
fi

# Begin with the base test. This implementation is how AMIE sends files as of
# today.
starttime='${timestamp}'
$basedir/connectivity.pl $testdir base
endtime='${timestamp}'
echo "base:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add gzip compression
starttime='${timestamp}'
$basedir/connectivity.pl $testdir base comp
endtime='${timestamp}'
echo "base-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add tarring the files together before transmission
starttime='${timestamp}'
$basedir/connectivity.pl $testdir base tar
endtime='${timestamp}'
echo "base-tar:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# A hybrid of the previous two tests using a tarball.
starttime='${timestamp}'
$basedir/connectivity.pl $testdir base tar comp
endtime='${timestamp}'
echo "base-tar-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

```

```

# Next we do the scp1 test which will rely on using scp to transfer each file
# individually instead of sending the data over stdout via ssh
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scp1
endtime='$(timestamp'
echo "scp1:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add gzip compression
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scp1 comp
endtime='$(timestamp'
echo "scp1-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add tarring the files together before transmission
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scp1 tar
endtime='$(timestamp'
echo "scp1-tar:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# A hybrid of the previous two tests using a tarball.
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scp1 tar comp
endtime='$(timestamp'
echo "scp1-tar-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Now we do the scpn test. This will copy over a directory instead of
# individual files in an attempt to mimic a single connection for multiple files
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scpn
endtime='$(timestamp'
echo "scpn:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add gzip compression
starttime='$(timestamp'
$basedir/connectivity.pl $testdir scpn comp
endtime='$(timestamp'

```

```

echo "scpn-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add tarring the files together before transmission
starttime='$(timestamp`
$basedir/connectivity.pl $testdir scpn tar
endtime='$(timestamp`
echo "scpn-tar:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# A hybrid of the previous two tests using a tarball.
starttime='$(timestamp`
$basedir/connectivity.pl $testdir scpn tar comp
endtime='$(timestamp`
echo "scpn-tar-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Now for our 'client-server' or sockets test. This will open a new connection
# for each file by going in over an SSL encrypted socket connection. Note that
# this will require that the receiver.pl script be running in a daemon mode (-d).
starttime='$(timestamp`
$basedir/connectivity.pl $testdir cs1
endtime='$(timestamp`
echo "cs1:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add gzip compression (Call receiver.pl -d -z)
starttime='$(timestamp`
$basedir/connectivity.pl $testdir cs1 comp 6667
endtime='$(timestamp`
echo "cs1-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add tarring the files together before transmission (Call receiver.pl -d -t)
starttime='$(timestamp`
$basedir/connectivity.pl $testdir cs1 tar 6668
endtime='$(timestamp`
echo "cs1-tar:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

```

```

# A hybrid of the previous two tests using a tarball. (Call receiver.pl -d -t -z)
starttime='$(timestamp'
$basedir/connectivity.pl $testdir cs1 tar comp 6669
endtime='$(timestamp'
echo "cs1-tar-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Lastly, we invoke client-server or sockets test to send multiple files over
# a single socket connection. We will skip the tar test since it would be the
# exact same thing as the tar tests done with the previous series since the tar
# file would require only one connection.
starttime='$(timestamp'
$basedir/connectivity.pl $testdir csx
endtime='$(timestamp'
echo "csx:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

# Add gzip compression (Call receiver.pl -d -z)
starttime='$(timestamp'
$basedir/connectivity.pl $testdir csx comp 6667
endtime='$(timestamp'
echo "csx-comp:$starttime:$endtime" >> $logfile

# Cleanup test data
ssh $remotehost rm "$remotedata/*"

```