Daikon: Invariant Detection of Nemo in association with OMP Project

Eun-young Cho, Minho Jeung, Kyu Hou, Varun Dutt and Monica Page
{echo1, mjeung, kyuh, vdutt, mpage}@andrew.cmu.edu

1. Project Objective

Apply Daikon, a tool for dynamically discovering program invariants, to NEMO, an Overlay Multicast Protocol, in order to gain insight into the group management aspect of OMP and transfer this knowledge for usage in carrying out the MSE POSDATA studio project.

2. Background

2.1 About Daikon:

Daikon (refer to Figure 1 given below for architectural description) is an analysis tool for the purpose of making inferences concerning invariants of a system [Ernst+]. Daikon is useful because it is not necessary to annotate code in order to discover the invariants [Ernst+]. Instead by running test suites on the code, invariants can be inferred from the test suites [Ernst+]. The invariants of the system can provide guidance in the event that code needs to be modified by outlining those occurrences that need to be preserved within the system [Ernst+].

By running a program multiple times using a particular test suite and choosing a path so as to track the values of particular variables at particular points in the code, Daikon is able to infer invariants from the running system[Ernst+].

For the OMParchitectability project this is particularly of interest because the POSDATA, SI company in Korea, studio project requires the use of Overlay Multicast Protocol (OMP) in order to broadcast video stream to particular nodes through the use of group management. Understanding invariants that are inherent in a multicast protocol could help:

- Understand the types of invariants that may be needed for implementation of OMP with POSDATA
- Provide an overview of constraints and limitations that occur with multicast protocols and group management

Though the focus of the Daikon testing is not for OMP in general, the program of interest is Nemo, a multicast protocol that uses group management. The main types of nodes represented by Nemo are:

- **Bootstrap Nodes**: Nodes that are the leaders of cluster or a segment of a group. They serve as the rendezvous point for publishers and subscribers
- **Publishers**: Publish data and send data to subscribers. They may also be known as co-leaders

- **Subscribers**: Receive data from publishers

There are number of operations that are possible through Nemo, but in terms of group management, two are of interest, joining a group and leaving a group. In terms of Daikon, it would be of interest to see what invariants are inherent when joining a group and which are inherent when leaving group. More specifically, what are the boundaries for parameters associated with join and leave methods for a bootstrap node, a publisher and subscriber.

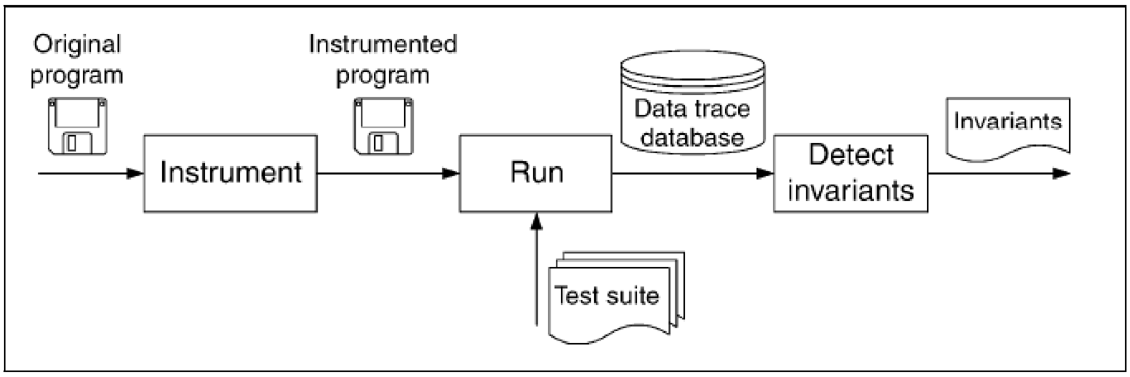


Figure 1: Architecture of the *Daikon* tool
(Taken from Daikon's website at MIT)

2.2 About NEMO and Daikon's scope on NEMO:

NEMO is an open source project that implements the concept of Overlay Multicast Protocol (OMP, a networking protocol to share a single data stream between a large number of connecting clients without degradation of the performance or increase in network cost). In the present situation or scenario there is a requirement to deal with the high degree of variability in the network. This variability arises from the dynamic situation of a large number of nodes connecting (joining) or disconnecting (leaving), the network. The aim of any multicast protocol is to achieve this variability without giving up on end to end latencies and incurring additional network costs. NEMO, a peer to peer overlay multicast protocol example provides these objectives by:

1. "*Co-leaders*" [NRPP04] i.e. have multiple leaders rather than a single leader in the network to provide for connectivity to a number of clients.
2. "*NACKS*" [NRPP04] to provide for negative acknowledgements on lost packets.

Due to the above two approaches, NEMO has been able to perform better than other existing competing multicast protocols like NICE and NICE-PRM. NEMO is proposed as an academic reference for the MSE studio project for team Trinity.

An important aspect of any OMP is group management (i.e. how to manage the existing nodes and cater to them with data streams when there may be nodes leaving and joining the group at runtime). The team decided to use Daikon's invariant detection to learn about the parameter values that may exist in such dynamic situations of group management when some nodes join and leave the network.

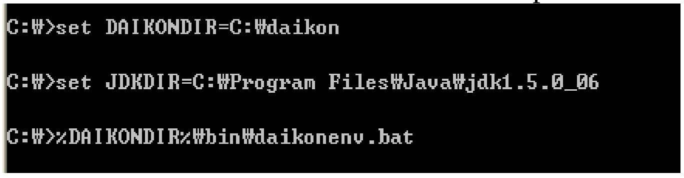
3. Experimental Setup

3.1 Daikon installation and Setup

In order to run Daikon, several steps should be done. First, download the latest version (daikon 4.2.4) of daikon which is daikon.zip in the website, http://pag.csail.mit.edu/daikon/download/ and unzip the file. Second, front-end program which enable daikon to run on a specific language such as C and C++ is needed, but daikon already include the front-end program for java.

3.1.1. Setup the environment.

For windows user below environment setup is needed.

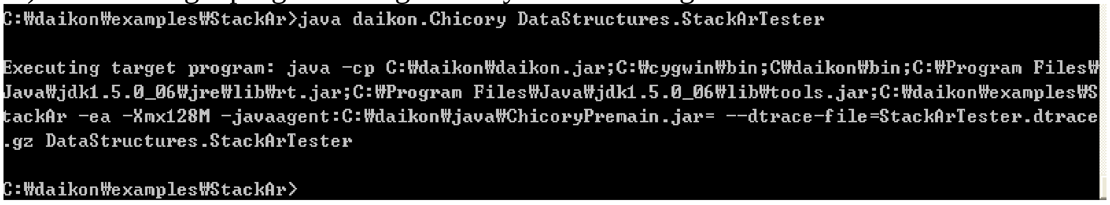


```
C:₩>set DAIKONDIR=C:₩daikon

C:₩>set JDKDIR=C:₩Program Files₩Java₩jdk1.5.0_06

C:₩>%DAIKONDIR%₩bin₩daikonenv.bat
```

3.1.2. Run daikon

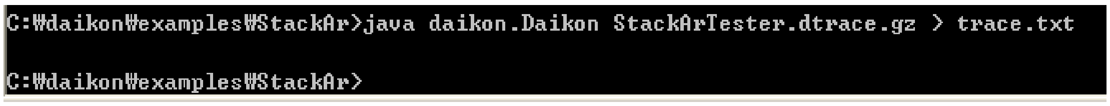
After successful compilation of the target programs, invariants of a program can be obtained by daikon in two steps.

1) Run the target program using Chicory front end to get the trace file



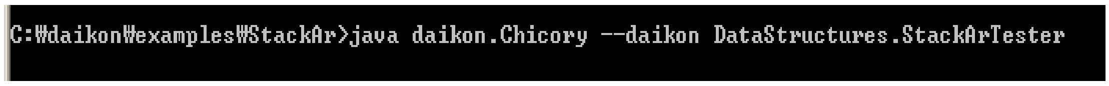
```
C:₩daikon₩examples₩StackAr>java daikon.Chicory DataStructures.StackArTester

Executing target program: java -cp C:₩daikon₩daikon.jar;C:₩cygwin₩bin;C₩daikon₩bin;C:₩Program Files₩
Java₩jdk1.5.0_06₩jre₩lib₩rt.jar;C:₩Program Files₩Java₩jdk1.5.0_06₩lib₩tools.jar;C:₩daikon₩examples₩S
tackAr -ea -Xmx128M -javaagent:C:₩daikon₩java₩ChicoryPremain.jar= --dtrace-file=StackArTester.dtrace
.gz DataStructures.StackArTester

C:₩daikon₩examples₩StackAr>
```

2) Run the java program with daikon option to get the invariants. The invariants are printed out the console so that it is better to store the invariants in the file



```
C:₩daikon₩examples₩StackAr>java daikon.Daikon StackArTester.dtrace.gz > trace.txt

C:₩daikon₩examples₩StackAr>
```

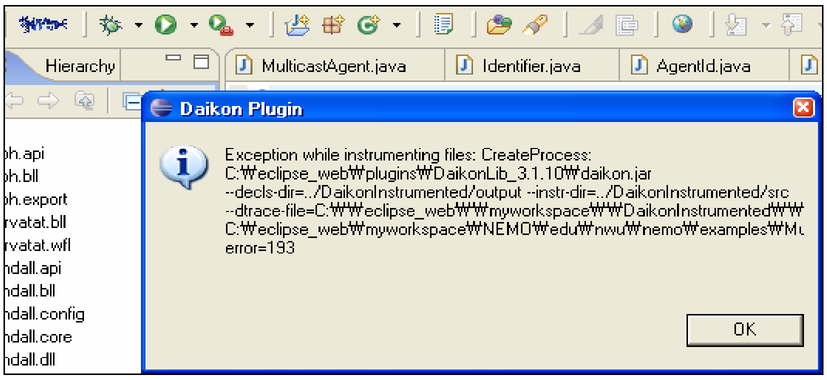
Alternately, two steps for generating invariants explained above can be done using one command as below.



```
C:₩daikon₩examples₩StackAr>java daikon.Chicory --daikon DataStructures.StackArTester
```

3.1.3. Eclipse plug-in

Even though Daikon provides eclipse plug-in, it only supports daikon 4.0.0. Because we could not find any website to download daikon 4.0.0 we tried to run the eclipse plug-in using daikon 4.2.4, but it did not work.



```
Exception while instrumenting files: CreateProcess:
C:₩eclipse_web₩plugins₩DaikonLib_3.1.10₩daikon.jar
--decls-dir=../DaikonInstrumented/output --instr-dir=../DaikonInstrumented/src
--dtrace-file=C:₩₩eclipse_web₩₩myworkspace₩₩DaikonInstrumented₩₩
C:₩eclipse_web₩myworkspace₩NEMO₩edu₩nwu₩nemo₩examples₩Mu
error=193
```

3.2. NEMO Installation

Nemo is an open-source overlay multicast protocol for streaming applications provided by Northwestern University. In order to run NEMO, NEMO source file and additional jar files are need to Download. The source website is http://www.aqualab.cs.northwestern.edu/projects/nemo/download.php and an additional jar files belongs to apache project so they can download at http://jakarta.apache.org/.

3.3. NEMO execution

NEMO provides sample program named MulticastAgent.java to test overlay multicast protocol provided by NEMO. The program can be run using three different set of parameters. When it runs using one parameter which is port number, the agent program runs as a bootstrap. A subscriber needs one more parameter, the address of bootstrap agent. A publisher needs additional packet sending interval. In our experiment, we focused on the bootstrap so that we put only one parameter in the MulticastAgent.java.

3.4. NEMO experiment challenges

The program that we focused on was the multicast agent which runs forever like server daemon program. Daikon also ran forever and the trace file kept increasing. For that reason, we slightly changed the NEMO program not to run forever and finally generated the trace file. Another challenge was daikon needed large amount of memory to make trace file and get the invariants from it. We had to increase java memory heap size up to 1024MB whiling running daikon. Last, it took more than 20 minutes to get the trace file even though daikon consumed 100% of CPU (Pentium 4 2.0MHz) and 1024MB memory

4. Analysis of Results

4.1 Group Management through NEMO:

To organize peer nodes into groups (or perform group management), NEMO organizes the peers into clusters, where the clusters are put in layers such that every peer is a member of the cluster at the lowest layer. In a cluster, a leader node is chosen from among the peers and this leader becomes the member of the next higher layer. The formation of such clusters in each layer is based on:

1. Network proximity
2. Bandwidth (in terms of latency) and
3. Peer lifetime

The size of a cluster is measured in terms of the "*degree*" [NRPP04] degree of what? The approach by the team was to use Daikon to collect the invariants at run time on these parameters (see Table in Appendix 1), and make an analysis of these parameters. The analysis would further provide guidance concerning whether NEMO as an OMP could be used on the MSE project.

4.2 Member Join:

The typical join operation occurs when a new node wants to join an existing hierarchy of nodes; this is an important part of group management. When a new node comes into the network it:

1. Polls a typical node (i.e. the first node ever to enter the hierarchy, which is in the highest layer) and based on this polling (also called the "*rendezvous point*" [NRPP04]), it gets to know the network addresses of the leaders or co-leaders in the current (top layer)
2. The new node then polls each of the nodes in the existing layer (top for the first time) and tries to find the leader which minimizes its cost function to the greatest extent in that layer. It again tries to get the leaders' addresses in the next lower layer, until it reaches the bottommost layers. At this point, whichever leader node minimizes the cost function to the greatest extent is chosen and the new node joins its associated layer.

4.3 Member Leave:

The leave of a member could be in two different situations:

1. An announced leave
2. An unannounced leave

In case of an announced leave, a peer member from a cluster in a layer announces its leave to the leader and leaves the cluster in that layer. If the peer member leaving is the leader itself, the leader must announce its leave to the peers, the peers must select a new co-leader and then the leader must leave the cluster in a layer by announcing to the new leader.

The team wanted to simulate this operation of join and leave (when there was already a layer consisting of node(s) and a new node joined the cluster) and use Daikon's invariants to provide network data on such join and leave operations. To accomplish this task, the team created a single node (called bootstrap node or publisher node) and allowed another node (called subscriber node) to join the network. But the team soon found that as Daikon's analysis is performed at runtime, the program should end for Daikon to produce a trace file for meaningful and compliable results on invariants.

In the case of the join and leave operations (for a new node joining/leaving existing nodes), the program doesn't come to an end; instead, like a server, it (passively) accepts new connections and keeps running. This caused Daikon to run nonstop on the Nemo code of interest. The team tried various ways of killing the process just after a join or a leave operation, but Daikon's trace file failed to capture such situations primarily as the tool running on the code also got killed with the process.

The end result was that the team could only simulate and generate the invariants when there was a single node in the network (i.e. when there was no node in the network and the first node, called the bootstrap, joined or left the network) for a time bound (short) period. No join requests or leave requests to the bootstrap node were considered and the situation was kept very simple. This allowed to program to end after a short period of time and thus Daikon could capture invariants on this piece of running code due to its termination.

The team realized that Daikon as a tool is not very useful for NEMO (as it could only simulate very simple situations for the join or leave of a single bootstrap node). When the invariants were checked for a single node it was found that these invariants were very informative and contained data on constants just before such a join request.

Based on these pre-join invariants, however, the team could still conclude many things. Although the invariants were produced for simple situations, they proved to be useful for an evaluation on the project in terms of whether the MSE team could conclusively carry on with NEMO. The team realized the potential (more details are provided in later sections) of using NEMO on the MSE project and Daikon's invariants enabled the team to formulate this reasoning.

4. Lessons Learned

Throughout the project, the team learned a large number of lessons by using Daikon on an OMP project like NEMO. Daikon's dynamic invariants produced enabled the team to get knowledge on NEMO's setup as an OMP project and also form an opinion on whether NEMO would provide useful support to the overall MSE project for the client POSDATA.

4.1 General characteristics of *Daikon*

1. The Daikon tool is a dynamic invariant detector. The tool's results, due to the run time characteristics, are largely dependent upon the test cases that are used to run the tool. The team faced two situations, the choice of a number of test cases (in terms of different port numbers) and the right choice of the test cases (which made the program terminate and thus enabled Daikon to produce the relevant invariants that could be compiled into a readable format).
2. The tool will only visit those portions of the code that are motivated by the test cases. The quality of test cases also determines to a large extent the information contained in the invariants.
3. The tool is very specific in its search for invariants. This is seen in the fact that the tool only checks for invariants on the entry and exit of method definitions in classes and also on the classes and objects of those classes. This can cause invariants on variables to be detected if the same are defined inside a constructor rather than being defined outside in the class definition.

4.2 Benefits to our project

1. The tool produced invariants on variable value thresholds and other important networking parameters like bandwidth, latency, heartbeats etc. This information was very useful to predict the condition of the network where a node exists and another tries to join the existing node, but the join is yet to happen.

2. Based on the expectations set by the team and the results produced by Daikon, the analysis proved to be beneficial in understanding a simple OMP situation. The same situation was used to make decisions of making NEMO a part of the MSE project.

3. The invariants detected by Daikon are automatic and the developers do not need to provide annotations in the code for producing them.

4. The invariants detected are primarily of 4 types:
    - **Enter**: For pre conditions that would be true when entering a program
    - **Exit**: For post conditions that would be true when leaving a program
    - **Class**: For any new class definition which is encountered in program execution, where the invariants are static in nature and true for the class in general
    - **Object**: True for the objects of the class, i.e. the invariants are true for all methods for that class

    These invariants provide useful information as described above.

4.3 Drawbacks

1. As already mentioned, the tool cannot scale up to situations where the program keeps running forever. The team found the dynamic situations of join and leave in the NEMO project almost impossible to check using Daikon. This is largely due to the fact that Daikon is a dynamic invariant detector and runs as a normal plug-in to the existing java compiler (Java 5). The tool would be good to use on code fragments that come to an end of execution, but in the case of NEMO the code keeps running and the Daikon tool keeps producing the trace of the invariants. As running Daikon is a 2 step process of first generating the traces and then running the Daikon tool itself, the approach cannot work for the never ending code situations.
2. Again, the tool uses a machine learning approach and based on test cases, produces a large number of invariants, to the extent that some of them are spurious. A random number generator function was used in the NEMO project, and Daikon produced one invariant for each of the random values. This provided an example of spurious invariants where the tool could have avoided this situation.
3. It took the team a large amount of time to make the NEMO project compile for the first time; thus sometimes if the code does not compile properly and reports compile time errors, testing it using dynamic tools like Daikon can almost become impossible.
4. Again the overhead involved in generating the test cases (identification of the right type of test case for a project) took the team a large amount of time. Sometimes the time spent could be comparable to time spent in doing a static code walkthrough or review on code that is the size of NEMO (~3 MB).

5. Conclusions

In our approach Daikon as a dynamic invariant detecting tool was moderately helpful. The reasons are as follows:

1. The tool could provide invariants for only very simple cases of NEMO. This was when there is a single node (bootstrap) that joins and leaves the network. This was due to the inability of Daikon to run successfully on non-terminating and forever running programs. This fact is again attributed to the dynamic nature of Daikon.

2. The test-cases did not attain coverage of the entire code fragment of interest. This only happened with some of the classes (related to the bootstrap node), whereas general publisher and subscriber join and leave methods were never explored by Daikon. The tool only traverses those paths in the program that are supported by the test cases and in our case the test cases were constrained by the inability of the tool to execute on forever running code fragments (as typical of NEMO)

However, there are several benefits that result as well. Below are some of the overall benefits of using Daikon:

1. The tool could execute a simple case of an OMP on NEMO and provide insight into the threshold values of network parameters.
2. The invariants produced with Daikon enabled the team to understand the OMP concepts as depicted by NEMO in a better and more productive way.
3. Although the invariants could have been more useful if the Daikon tool could run on a typical join or leave scenario, the team could draw an estimation of NEMO capability by the simple analysis through the use of Daikon.

6. References

[NRPP04] Stefan Birrer and Fabián E. Bustamamte, “Nemo-Resilient Peer-to-Peer Multicast without the Cost”
[Noh05] Kuyul Noh, Changki Kim, Jonggul Park, Jaeha Song, “The Evaluation of Daikon: untilization of *Daikon* in the POI Data Inspection System”
[Kim02] Miryung Kim and Andrew Peterson, “An Evaluation of Daikon: A Dynamic Invariant Detector”
[Prakash02] Amol Prakash, Mausam, “Tool Evaluation : DAIKON”
[Ernst+] Ernst, Michael D., Griswold, William G. Notkin, David. “Dynamically Discovering Likely Program Invariants to Support Program Evolution”
http://pag.csail.mit.edu/daikon/
http://www.aqualab.cs.northwestern.edu/projects/nemo/gettingstarted.html

Appendix 1. Test Case

Test Case:
There is only one bootstrap (leader node, like a server) in the network and this node is created using different port numbers like 1050 etc.

Code Segments	Output from Daikon

package edu.nwu.nemo.examples;
 	
public final class MulticastAgent
{ public static void main(String[] args)
   {
 	
org.apache.log4j.Logger.getRoot().setLevel(org.apache.log4j.Level.WARN);
// comment out for more debug output
    }
}
	edu.nwu.nemo.examples.MulticastAgent:::CLASS
	edu.nwu.nemo.examples.MulticastAgent.logger has only one value
	edu.nwu.nemo.examples.MulticastAgent.logger.getClass() ==
	org.apache.commons.logging.impl.Log4JLogger.class

Expectations: The expectations of the developers were to identify some aspects of logging, and the tool’s output confirmed that the logger class had only one value to log. **Were the Expectations met?** The expectations were met in terms of tool telling the number of values getting logged and also in terms of the class used by the logger for the logging operation. Some values that the tool missed was the value that was the actual value that was logged.	

package edu.nwu.nemo.examples.MulticastAgent;
switch (args.length)
        {
        case 3: …
        case 2:
           bootstrap = new SocketAddress[]
              {
                  AgentId.parseAgentId(args[1]).getSocketAddress()
              };
        case 1:
 int port = Integer.parseInt(args[0]);
 InetSocketAddress addr = null;
 if (port > 0)
{
addr = new InetSocketAddress(InetAddress.getLocalHost(),
                    port);
}
 	
socketFactory = new SharedPacketSocket(new PacketSocket(addr));
	edu.nwu.nemo.examples.MulticastAgent.main(java.lang.String[]):::ENTER
	args has only one value
	args.getClass() == java.lang.String[].class
	args[] == [32541460]
	args[] elements == "32541460"
	args[].toString == [1050]
	args[].toString elements == "1050"
	size(args[]) == 1
	edu.nwu.nemo.examples.MulticastAgent.SESSION_IDENTIFIER[] elements ==
	size(args[])-1
	size(args[])-1 in edu.nwu.nemo.examples.MulticastAgent.SESSION_IDENTIFIER[]

break;
}	

Expectations: The expectations were that the tool indicates the test case before starting the analysis. **Were the Expectations met?** The expectations were met in terms of tool telling the number of values that were input by the programmers and also the nature and the type of this value.	
package edu.nwu.nemo.bll.NemoBootstrapService; public void setup(IPacketSocketFactory socketFactory, ITimestampFactory tsmpFactory, IEpochServiceFactory epochServiceFactory, IReefConfiguration config, SocketAddress[] bootstrap)	edu.nwu.nemo.bll.NemoBootstrapService.setup(edu.nwu.net.api.IPacketSocketFactory, edu.nwu.util.api.ITimestampFactory, edu.nwu.reef.api.IEpochServiceFactory, edu.nwu.reef.api.IReefConfiguration, java.net.SocketAddress[]):::ENTER this.sock == null this.nodeId == null this.agent == null this.epochService == null this.config == null this.callback == null this.handler == null socketFactory has only one value socketFactory.getClass() == edu.nwu.net.wfl.SharedPacketSocket.class tsmpFactory has only one value tsmpFactory.getClass() == edu.nwu.util.bll.TimestampFactory.class epochServiceFactory has only one value epochServiceFactory.getClass() == edu.nwu.reef.bll.EpochServiceFactory.class config has only one value config.getClass() == edu.nwu.nemo.bll.NemoConfiguration.class bootstrap has only one value bootstrap.getClass() == java.net.SocketAddress[].class bootstrap[] == [] edu.nwu.nemo.bll.NemoBootstrapService.setup(edu.nwu.net.api.IPacketSocketFactory, edu.nwu.util.api.ITimestampFactory, edu.nwu.reef.api.IEpochServiceFactory, edu.nwu.reef.api.IReefConfiguration, java.net.SocketAddress[]):::EXIT edu.nwu.nemo.bll.NemoBootstrapService.logger == orig(edu.nwu.nemo.bll.NemoBootstrapService.logger) edu.nwu.nemo.bll.Configuration.loggerConf == orig(edu.nwu.nemo.bll.Configuration.loggerConf) this.config.clusterType == orig(edu.nwu.nemo.core.EClusterType.CONST) edu.nwu.nemo.core.EClusterType.LINEAR == orig(edu.nwu.nemo.core.EClusterType.LINEAR) edu.nwu.nemo.core.EClusterType.EXPONENTIAL == orig(edu.nwu.nemo.core.EClusterType.EXPONENTIAL)

	this.config.defaultJitter == orig(size(bootstrap[])) bootstrap[] == orig(bootstrap[]) this.sock has only one value this.nodeId has only one value this.agent has only one value this.epochService has only one value this.config has only one value this.callback has only one value this.handler has only one value bootstrap[] == [] edu.nwu.nemo.bll.NemoBootstrapService.logger.getClass() == edu.nwu.nemo.dll.RpPacketHandler.logger.getClass() edu.nwu.nemo.bll.NemoBootstrapService.logger.getClass() == orig(edu.nwu.nemo.bll.NemoBootstrapService.logger.getClass()) edu.nwu.nemo.bll.NemoBootstrapService.logger.getClass() == orig(edu.nwu.nemo.bll.Configuration.loggerConf.getClass())
Expectations: The configuration is set right and setup is initialized in a proper order. **Were the Expectations met?** The expectations were met in terms of tool telling the number of values that were setup by the code but there was no information on the nature and the type of values.	
package edu.nwu.nemo.bll.NemoBootstrapService; On each and every method of NemoBootstrapService class namely: setup, teardown, getName, toString	this.config.degree == this.config.numPrmAgents this.config.degree == this.config.crewSize this.config.degree == this.nodeId.id.id[this.config.numStreams] this.config.useLowPriority == this.config.useLossRate this.config.useLowPriority == this.config.usePrm this.config.useLowPriority == this.config.useScheduling this.config.useLowPriority == this.config.useDynamicPositioning this.config.useLowPriority == this.config.schedBySuccessors this.config.useLowPriority == this.config.ignoreDuplicates this.config.useLatency == this.config.useNacks this.config.useLatency == this.config.useHistory this.config.useLatency == this.config.useProactiveForwarding this.config.useLatency == this.config.useRefinement this.config.costEstimationProb == this.config.leaderRefinementProb this.config.costEstimationProb == this.config.clusterMergeProb this.config.costEstimationProb == this.config.alternateAgentsProb this.config.costEstimationProb == this.config.responseTimeAlpha this.config.minClusterRefinementThreshold == this.config.minLeaderRefinementThreshold

	this.config.minClusterRefinementThreshold == this.config.costAlphaDown this.config.degree == 3 this.config.numStreams == 1 this.config.useLowPriority == false this.config.cacheSize == 16 this.config.useLatency == true this.config.prmProb == 0.02 this.config.clusterType has only one value this.config.packetInterval == 100 this.config.dataSize == 1000 this.config.costEstimationProb == 0.05 this.config.clusterRefinementProb == 0.2 this.config.undersizedLeaderRefinementThreshold == 0.5 this.config.oversizedLeaderRefinementThreshold == 1.0 this.config.maxClusterRefinementThreshold == 0.9 this.config.minClusterRefinementThreshold == 0.1 this.config.maxLeaderRefinementThreshold == 0.8 this.config.timePerThresholdIncrease == 5000.0 this.config.jitterFactor == 4.0 this.config.minimalJitter == 200 this.config.randomSubsetSize == 10 this.config.streamPenalty == 10.0 this.config.heartbeatInterval == 10000 this.config.gracePeriod == 15000 this.config.rpHeartbeatInterval == 2000 this.config.rpGracePeriod == 7000

Expectations:
The values are initialized and the developers learn some rules in terms of the network parameters.
Were the Expectations met?
The expectations were met in terms of tool telling:

1. The number of parameter agents, crew size (number of nodes in a layer) and number of independent node streams all are same and map to the degree.
2. The low priority (i.e. priority among the network nodes in a layer is set), to, the loss rate, network parameter used, scheduling, dynamic positioning, scheduling by leader nodes or ignoring duplicates. The low priority assignment could be a function of the following parameters.
3. The latency of the network (or the delay in Overlay Multicast group management) could be based on use of negative acknowledgements, history of operation of the network, proactive forwarding or refinement. Thus, the latency could be a function of these parameters.
4. The cost estimation probability of the network is a function of the leader refinement probability, cluster merge probability, alternate agent probability or response time.
5. The thresholds for minimum cluster refinement and minimum leader refinement are equal, whereas the minimum cluster refinement threshold is a function of the cost parameter.
6. Some constant values are assigned to degree (3), number of streams (1, as it is a bootstrap node only) and parameter probability of 0.02. The cluster type has

only one node (bootstrap) and is correctly assigned a value of 1.

7. Nemo uses a machine learning genetic algorithm and this algorithm assigns hard values which are improved as the network performs the overlay multicast protocol. Some of these parameters include the grace period between packets, heart beat and jitter factor.

```
package edu.nwu.nemo.dll.Callback;
callback = new Callback(sock);


public class Callback implements IStreamAgentCallback,
IStreamMulticastCallback,
   IStreamRpCallback
{…
}
```

```
edu.nwu.nemo.dll.Callback:::CLASS
edu.nwu.nemo.dll.Callback.UNIFORM.min == cern.jet.random.Uniform.shared.min
edu.nwu.nemo.dll.Callback.UNIFORM.max == cern.jet.random.Uniform.shared.max
cern.jet.random.Uniform.shared.min ==
edu.nwu.nemo.dll.Callback.PROB_RELIABLE
edu.nwu.nemo.dll.Callback.logger has only one value
edu.nwu.nemo.dll.Callback.logger.getClass() ==
org.apache.commons.logging.impl.Log4JLogger.class
edu.nwu.nemo.dll.Callback.UNIFORM has only one value
cern.jet.random.Uniform.shared has only one value
cern.jet.random.Uniform.shared.max == 1.0
```

Expectations:

To see what are the callback values when a single bootstrap node. Callback is used in the scriber and publisher nodes to join and release a connection.

Were the Expectations met?

The expectations were met in terms of tool telling:

1. The values of minimum and maximum for callbacks were set to reliable transmission probability between two nodes (initialized to value 0) and value 1.0 (one bootstrap node) respectively.

```
package edu.nwu.nemo.bll.StreamRp;

public class StreamRp implements IStreamRp
{ …..

  public StreamRp(Configuration config, AgentId id,
      IStreamRpCallback callback, ITimestampFactory factory)
        {…. }
}
```

```
edu.nwu.nemo.bll.StreamRp.StreamRp(edu.nwu.nemo.bll.Configuration,
edu.nwu.reef.api.AgentId, edu.nwu.nemo.api.IStreamRpCallback,
edu.nwu.util.api.ITimestampFactory):::ENTER
config.degree == config.numPrmAgents
config.degree == config.crewSize
config.degree == id.id.id[config.numStreams]
config.useLowPriority == config.useLossRate
config.useLowPriority == config.usePrm
config.useLowPriority == config.useScheduling
config.useLowPriority == config.useDynamicPositioning
config.useLowPriority == config.schedBySuccessors
config.useLowPriority == config.ignoreDuplicates
config.useLatency == config.useNacks
config.useLatency == config.useHistory
config.useLatency == config.useProactiveForwarding
config.useLatency == config.useRefinement
config.prmProb == config.clusterSplitProb
config.clusterType == edu.nwu.nemo.core.EClusterType.CONST
config.clusterType.id == config.defaultJitter
config.packetInterval == config.epochInterval
```

	config.costEstimationProb == config.leaderRefinementProb config.costEstimationProb == config.clusterMergeProb config.costEstimationProb == config.alternateAgentsProb config.costEstimationProb == config.responseTimeAlpha edu.nwu.nemo.bll.StreamRp.StreamRp(edu.nwu.nemo.bll.Configuration, edu.nwu.reef.api.AgentId, edu.nwu.nemo.api.IStreamRpCallback, edu.nwu.util.api.ITimestampFactory):::EXIT config.degree == config.numPrmAgents config.degree == config.crewSize config.degree == orig(config.degree) config.degree == orig(config.numPrmAgents) config.degree == orig(config.crewSize) config.degree == id.id.id[config.numStreams] config.degree == orig(id.id.id[post(config.numStreams)]) config.degree == id.id.id[orig(config.numStreams)] config.degree == orig(id.id.id[config.numStreams]) config.numStreams == orig(config.numStreams) config.numStreams == size(this.rps[]) config.useLowPriority == config.useLossRate config.useLowPriority == config.usePrm config.useLowPriority == config.useScheduling config.useLowPriority == config.useDynamicPositioning config.useLowPriority == config.schedBySuccessors config.useLowPriority == config.ignoreDuplicates config.useLowPriority == orig(config.useLowPriority) config.useLowPriority == orig(config.useLossRate) config.useLowPriority == orig(config.usePrm) config.useLowPriority == orig(config.useScheduling) config.useLowPriority == orig(config.useDynamicPositioning) config.useLowPriority == orig(config.schedBySuccessors) config.useLowPriority == orig(config.ignoreDuplicates) config.cacheSize == orig(config.cacheSize) config.useLatency == config.useNacks config.useLatency == config.useHistory config.useLatency == config.useProactiveForwarding config.useLatency == config.useRefinement config.useLatency == orig(config.useLatency) config.useLatency == orig(config.useNacks)

	config.useLatency == orig(config.useHistory) config.useLatency == orig(config.useProactiveForwarding) config.useLatency == orig(config.useRefinement) config.costEstimationProb == config.leaderRefinementProb config.costEstimationProb == config.clusterMergeProb config.costEstimationProb == config.alternateAgentsProb config.costEstimationProb == config.responseTimeAlpha config.costEstimationProb == orig(config.costEstimationProb) config.costEstimationProb == orig(config.leaderRefinementProb) config.costEstimationProb == orig(config.clusterMergeProb) config.costEstimationProb == orig(config.alternateAgentsProb) config.costEstimationProb == orig(config.responseTimeAlpha)
Expectations: The values are set and the methods are worked for get the rendezvous-point. **Were the Expectations met?** The expectations were met in terms of tool telling: 1. The number of parameter agents, crew size (number of nodes in a layer) and number of independent node streams all are same and map to the degree. 2. The low priority (i.e. priority among the network nodes in a layer is set), to, the loss rate, network parameter used, scheduling, dynamic positioning, scheduling by leader nodes or ignoring duplicates. The low priority assignment could be a function of the following parameters. 3. The latency of the network (or the delay in Overlay Multicast group management) could be based on use of negative acknowledgements, history of operation of the network, proactive forwarding or refinement. Thus, the latency could be a function of these parameters. 4. The cost estimation probability of the network is a function of the leader refinement probability, cluster merge probability, alternate agent probability or response time.	
package edu.nwu.nemo.dll.JoinPacket; ON each and every method of JointPacket class Namely: JoinPacket, getLayer, decode, encode	edu.nwu.nemo.dll.JoinPacket:::CLASS edu.nwu.nemo.dll.JoinPacket.VERSION == size(edu.nwu.nemo.dll.JoinPacket.FIELDS[])-1 edu.nwu.nemo.dll.JoinPacket.FIELDS has only one value edu.nwu.nemo.dll.JoinPacket.FIELDS.getClass() == edu.nwu.net.api.IPacketField[].class edu.nwu.nemo.dll.JoinPacket.FIELDS[] contains no nulls and has only one value, of length 1 edu.nwu.nemo.dll.JoinPacket.FIELDS[] elements has only one value edu.nwu.nemo.dll.JoinPacket.FIELDS[].getClass() == [edu.nwu.net.dll.SimpleField] edu.nwu.nemo.dll.JoinPacket.FIELDS[].getClass() elements == edu.nwu.net.dll.SimpleField.class size(edu.nwu.nemo.dll.JoinPacket.FIELDS[]) == 1 edu.nwu.nemo.dll.JoinPacket.FIELDS[edu.nwu.nemo.dll.JoinPacket.VERSION+1..] == [] edu.nwu.nemo.dll.JoinPacket.FIELDS[] elements ==

	edu.nwu.nemo.dll.JoinPacket.FIELDS[edu.nwu.nemo.dll.JoinPacket.VERSION]
Expectations: JointPacket is used for joining the requested packet (node). **Were the Expectations met?** The expectations were met in terms of tool telling: 1. The invariants as shown above demonstrate that one node joins the network and forms a part of a FIELD (which represents a cluster in a layer). 2. The version maintains the information of who is the leader in a particular FIELD and in our case of one node, the single node is also the leader	

package edu.nwu.nemo.dll.LeavePacket;
 	
public class LeavePacket extends Packet
{ …
  public LeavePacket()
    {
      …
    }
}
	edu.nwu.nemo.dll.LeavePacket:::CLASS
	edu.nwu.nemo.dll.LeavePacket.VERSION ==
	size(edu.nwu.nemo.dll.LeavePacket.FIELDS[])
	edu.nwu.nemo.dll.LeavePacket.FIELDS has only one value
	edu.nwu.nemo.dll.LeavePacket.FIELDS.getClass() ==
	edu.nwu.net.api.IPacketField[].class
	edu.nwu.nemo.dll.LeavePacket.FIELDS[] == []
	edu.nwu.nemo.dll.LeavePacket.FIELDS[].getClass() == []

Expectations: LeavePacket is used for leaving the requested packet (node). **Were the Expectations met?** The expectations were met in terms of tool telling: 1. The invariants as shown above demonstrate that one node leaves the network and forms a part of a FIELD (which represents a cluster in a layer). 2. The version maintains the information of who is the leader in a particular FIELD and in our case of one node; the single node is also the leader. This leader gives up the leadership before leaving the network.	