Automatic Simulation of Network Problems in UDP-Based Java Programs

Eitan Farchi
IBM Haifa Research Lab,
farchi@il.ibm.com

Yoel Krasny
IBM Haifa Research Lab,
yoelk@il.ibm.com

Yarden Nir
IBM Haifa Research Lab,
yarden@il.ibm.com

Abstract

This paper describes a tool for black-box testing of UDP-based distributed Java programs. UDP provides little guarantee for correct delivery of data, and therefore requires the application to verify the integrity of communication according to its needs. Debugging such application is hard, since it is hard to create at will bad network conditions. The tool describes here creates an intermediary layer above the Java API which simulates network noises. It therefore enables stress-testing the application even on a flawless network environment. We describe a field experience of testing an application, using the tool vs. using specially-written testing code. We show the two approaches to be complementary.

1. Introduction

TCP and UDP are the two main networking protocols that can be employed in distributed applications. TCP guarantees that once a connection has been established between two parties, data sent from one side will be received at the other side in correct order. UDP does not guarantee this; it is based on dividing the data to *packets*, also called *datagrams*. While the data in each packet is guaranteed to be delivered, there is no guarantee about the packets themselves; packets may be lost, duplicated, or arrive in a different order from the order of sending. Java contains built-in support for both protocols.

Most applications need to verify at least some degree of integrity of data received. TCP spares this burden from the programmer, but at the price of efficiency and flexibility. Therefore, many performance oriented applications will prefer to use UDP. This means that the programmer needs to implement herself some protocols to ensure the required integrity of data, including handling of lost or misplaced data. Designing and implementing such a protocol would normally be very bug-prone.

Testing such protocols is hard, because usually the network environment which the developer has when testing the application is too friendly: it will in fact deliver all packets, or almost all, in perfect order. Thus there will be no effective test of the protocol's paths for handling misplaced packets. The bugs are likely to be found only late in the development cycle, or in the field.

The programmer may write testing code which simulates network "noises" – packets lost or misplaced. This approach is powerful in finding and analyzing bugs, but requires programming overhead, and careful design.

This paper presents a tool – a component of "Contest: Concurrent Testing Tool" [4] – which gives the developer automatic simulation of network noises. By instrumenting the bytecode, calls to Java UDP API are replaced by calls to code of our tool. This code creates an intermediary level above the Java API, which simulates network disturbance (introduces "noise") in a controlled fashion. The developer can then test, with no additional effort, the behavior of the application in arbitrarily bad network conditions.

Chapter 2 of this paper gives a brief introduction to the UDP API in Java. Chapter 3 details what we want our automatic noise tool to be able to do. Chapter 4 describes the core techniques we use, special problems and their solutions. Chapter 5 describes a field experience in testing a UDP-based application – group communication [2] – with the automatic noise tool as well as with specially-written testing code. This experience sheds light on the pros and cons of both approaches.

2. UDP in Java

Our tool works with the `java.net` package API, available since version 1.1 of Java. For simplicity, we describe only the API of this package. Java 1.4 introduced new API as part of the `java.nio` package, in particular the `DatagramChannel` class. The application of the techniques described here to the new API is mostly straightforward.

A packet is represented in Java by class `DatagramPacket`. It contains a byte buffer, length and offset (indicating which part of the buffer is relevant), and socket address (IP address and port). The address is of the other side: the sender in case of received packets, the target in packets for sending.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Class `DatagramSocket` handles sending and receiving of packets. It is connected to a local port. The method `send(DatagramPacket)` gets a packet, and sends it to the address specified in it. The method `receive(DatagramPacket)` reads an incoming packet, and fills the given `DatagramPacket` with the data and address received (for brevity, we'll refer to these methods simply as `send()` and `receive()`). It blocks until a packet arrives; however, timeout may be defined for the socket, limiting the blocking time. If the timeout expires, `receive()` throws an `InterruptedIOException` (which we shall refer to as "timeout exception").

A `DatagramSocket` may or may not be *connected* to a remote address. If it is connected, it may only send and receive packets to/from that remote address.

3. Features of the automatic noise

The basic requirement from our tool is that all the problems we simulate are indeed possible without our tool, provided the network conditions are bad enough. That is, we must not cause any false alarm. The tool is implemented as part of the ConTest tool [4], which introduces noise in the multi-threading level to reveal concurrent bugs. Virtually all distributed applications are also multi-threaded, so the combination with ConTest gives additional benefit for the programmer, compared to a tool that would simulate *only* network problems. ConTest, with the UDP noise feature, is activated by instrumenting the bytecode class files – it has no effect on the source, and can be done by testers as well as by developers, in any environment.

The tool provides simulation of arbitrarily bad network conditions, up to complete network breakdown. This is useful even for applications which are not designed to perform well in such conditions: they should at least fail gracefully, and it is desirable to test this behavior. The simulation of less noise than complete breakdown is random: all packets have an equal probability of being interfered with (although the selection can be tuned in some aspects, as described below).

The tool works locally for each JVM. A distributed application with several nodes can apply the tool to only a subset of the nodes, or set different parameters of noise creation for each node.

There are four parameters defining the behavior of the tool. The parameters are given in a preference file, which is read by our code in the beginning of the program's run. In addition, they can be changed on the fly, either by calls to our code in the test program source, or by a simple application that we provide, which runs in an independent JVM and communicates with the noise mechanism via a TCP/IP socket. The four parameters are:

- **Remote Nodes**. Our noise can be applied only to packets coming from, or going to, certain other nodes (one or more, specified by their IP addresses). By default, *all* remote nodes are considered – no distinction is done. Note that this parameter, like all others, is set independently for each node. In addition, random nodes selection can be specified: once in a while our tool randomly picks some IP addresses from all those seen so far, and sets them as the selected remote nodes. After some duration, measured in number of send/receive calls, the selection is changed.
- **Direction**. The noise can be applied on incoming packets only (we intervene in `receive()` calls), on outgoing packets only (`send()` calls), or both. Note that whether or not a `send()` operation has succeeded has no direct effect on the sending node. Tampering with outgoing packets actually tests how the *receiving* nodes cope with the disturbances. Looking at two interacting nodes, tampering with the outgoing packets in one of them is equivalent to tampering with the incoming packets in the other. However, if there is one node A which communicates with several others B1, B2, ..., it may be convenient to tamper only with outgoing messages from A, and see how B1, B2, ... are affected.
- **Mode**. we define five modes:
  1. **No noise**: packets are passed through our code with no interference.
  2. **Delay** (simulate heavy network delay at this node): all packets are accumulated and are not passed to the application. If the mode is changed to 1, 4 or 5, these packets will be sent/received ASAP, in their original order (the meaning of ASAP will become clear in section 4). The accumulated packets will be dropped (and lost) if the type is changed to 3.
  3. **Block** (simulate network failure at this node): all packets are lost.
  4. **Random noise, conservative**: packets are randomly dropped, duplicated, and reordered. The probability of these actions depends on the *strength* parameter – see below.
  5. **Random noise, radical**: same as 4, but with some small probability, the node can start behave as if it were in state 2 or 3. After a time interval T (randomly chosen, measured in number of send/receive operations), the special behavior will cease, and the node will return to behave as if it were in mode 4, until the next time it "decides" to change the mode. When returning from the temporary special mode, if it was as in mode 2, the packets that were meant to be sent/received during T will be sent/received ASAP. If the temporary mode was as in mode 3, the packets will be lost.

- **Strength**: Applies when one of the two random modes is in force, and determines how much noise is made. Specifically, it determines the probability that each send/receive call will undergo a "bad thing" (omission, duplication, reordering).

In the random modes, there is another type of noise for receive calls, in addition to omission, duplication and reordering. This is a deliberate throw of timeout exception, provided that timeout is defined for this socket. The idea, again, is that this is something that may happen in the field as a result of network problems, but is not likely to happen in the environment of the developer, and is therefore at risk of being wrongly dealt with in the program – unless it is forced in the test.

The motivation behind the different parameters is to give the tester some power and flexibility to test scenarios of the application which she finds interesting. In section 5.3 below, we give an example. In principle, other parameters and modes could have been used (e.g., select a remote port, or set different probabilities for different noises). We chose those which we judge to be most useful to cause typical interesting scenarios, while keeping the interface reasonably simple.

A "lazy" tester, which is interested in stress-testing the application but not with specific scenarios, would normally set *Remote Nodes* to *all* or *random*, *direction* to *both*, and *mode* to one of the *random* values. It then remains only to play with the strength, testing different levels of stress.

It is important to point out one limitation of the tool – all the decisions are done locally for a given node; our tool has no overall view of the application. Some scenarios would involve simultaneous events in two or more nodes, and this cannot easily be forced by the automatic tool (an example is given in section 5.4). It can still be done with an external intervention, by changing the parameters on the fly simultaneously in the different nodes. However, doing so requires an additional work of incorporating tool calls into the test program. Giving the tool capability to make coordinated decisions in different nodes would require incorporating communication protocol into the tool itself, making it extremely more complicated, and probably less easy to use.

3.1 Example

To demonstrate the behavior, consider a simple program composed of two nodes. One is a "getter". It listens on a given port, using `DatagramSocket`. When it receives a packet, it simply echoes the bytes in the packet, and listens again. The received strings are printed to the screen separated by a space. If no packet has arrived in 10 seconds, a warning is given to the screen, but the program just returns to listen This is implemented by defining timeout to the socket, and by catching and printing a possible timeout exception in each iteration of `receive()`. The getter continues the loop until it gets a packet with a pre-defined "finish" code.

The other node is a "sender". It sends to the getter's address 100 packets, containing the numbers 0 through 99 sequentially, represented as strings. Then it sends the "finish" code. The finish code is sent three times, so that if it lost once (or twice) due to communication problems, the receiver is still likely to get it (if the first one did succeed, the other two will be sent and just ignored by the closed socket at the getter). If there are no network noises, the output of the getter will just be the numbers from 0 through 99 sequentially. If there are network noises, some of the numbers will be lost, some will be duplicated, and some will arrive out of order.

First we run the program without our tool, using two command-line consoles on the same machine. Here is the output of the getter:

```
>listening...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 finishing
```

The numbers appear in perfect order. Now we instrument the application, and use our noise-creation tool. We set *mode* to *random-conservative* (#4 above), *strength* to 10 (meaning that roughly 1 out of 10 packets should undergo some type of disturbance), and, at this stage, *direction* to *outgoing only*. This means that the noise-insertion will directly affect only the sender. The output of the getter follows, with highlighting added:

```
>listening...
0 1 2 3 5 6 8 9 7 10 11 12 13 11 14 15
16 17 18 17 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53
54 55 56 57 59 60 61 62 64 63 65 66 67
68 69 70 72 71 73 74 75 74 76 73 77 73
78 79 80 82 83 81 84 85 86 87 89 88 90
91 92 93 94 95 96 97 98 99 finishing
```

We see that generally the order remained, but with some disturbances: 4 and 58 were missing. 7 and 81 appeared "too late". 11 and 17 arrived twice, 73 arrived three times. 63 was swapped with 64, and 88 with 89. Since the decision is random, another run will show similar disturbances in different places. For the programmer, it means that she gets value from running the same test many times, since different decisions will be taken, and presumably different paths in the program will be executed.

Now we run it again, with direction *incoming only* – the noise insertion directly affect the getter.

```
listening...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17
exception:
java.net.SocketTimeoutException
datagram   socket   receive:   time   out
exception   artificially   generated   by
ConTest
18 19 20 21 22 23 24 26 27 25 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 44
45 46 47 43 48 49 50 51 52 53
exception:
java.net.SocketTimeoutException
datagram   socket   receive:   time   out
exception   artificially   generated   by
ConTest
54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 75 76 77 74 78 80
81 82 79 83 84 85 86 88 89 90 91 93 94
95 96 97 98 99 finishing
```

We see the same disturbances as before (at different places, of course), but in addition, timeout exception was thrown twice. It didn't actually take 10 seconds with no packet arriving; our tool just threw the exception, to simulate condition that *might* have happened.

4. Underlying techniques and solutions

4.1 Where our code operates

The noise-insertion work is done at a class of ours called `Ct_DatagramSocket` ('Ct' for ConTest). This class extends `java.net.DatagramSocket`, and overrides the `send()` and `receive()` methods.

The instrumentation replaces calls to `DatagramSocket` constructors with calls to `Ct_DatagramSocket` constructors (which in turn call the super-class – `DatagramSocket` – constructors, with the same parameters). From this moment, calls to `send()` and `receive()` on the created objects will call our methods. Most other methods of `DatagramSocket` are not touched by our code, and so allow the object to behave as usual – except the noises at `send()` and `receive()`.

Our `send()` and `receive()` filter the packets – omitting and reordering some. When we want to actually send or receive a packet, we simply call `super.send` or `super.receive` – these do the "real" sending or receiving. In what follows we'll denote these "real" operations as send* and receive*.

4.2 Doing noise in `send()`

`send()` is easier to manipulate than `receive()`, because whether it succeeds or not has no direct effect on the local JVM (provided it didn't take IO Exception). In one call of `send()` we can do one send*, zero, or more than one.

`send()` receives a `DatagramPacket` to deliver. If we decide to send the packet without disturbance, we just send* it, of course. If we want omit this packet, we return from `send()` without sending*.

We also need to be able to store packets, when in *delay* mode. For this we keep a *pool* – a queue of `DatagramPacket` objects that were given to `send()`. In delay mode, we store the given packet in the pool, and return without sending*. When the mode changes to *no-noise* or *random* (or the *direction* or *Remote Nodes* parameters change), we just remove all packets from the pool and send* them. There is one pool per each `Ct_DatagramSocket` object.

The same pool is used for reordering and duplication in random mode. Reordering is achieved by deferring a packet until the next packet, or several packets, have been sent*. The deferred packets are stored in the pool instead of being sent* immediately. In each `send()`, after sending* the new packet given to this `send()` (or omitting it, or deferring it), in a fixed probability we poll the pool: send* a packet from the pool if there is one. Sending the pooled packet after the newer one causes the order among these two packets to be reversed. For duplicating a packet, we create an identical `DatagramPacket` (sometimes more than one copy), send* one copy and put the other ones in the pool.

Usually (depending on the strength parameter) the pool would contain at most one packet, but sometimes it will contain more. It is ordered by time of insertion. When the pool is polled, the packet that is taken is either the first or the last, randomly in equal probability. This gives some more mixing of packets.

4.2 Doing noise in `receive()`

The types of intervention in `receive()` are symmetrical to those of send, plus the timeout exception (as explained in section 3 above). However, things are more delicate here. As opposed to `send()`, `receive()` has effects on this JVM: It is given a `DatagramPacket` (which may or may not be empty), and this packet is modified – filled with data – before `receive()` returns.

Until it returns, the method blocks. If timeout is defined, the method might leave by throwing timeout exception.

It is therefore illegal for us to return from `receive()` without changing the given packet, or with changing it to an arbitrary string – this is not a possible result in the real world, regardless of the network conditions. Rather, we need to fill the given packet, and we need to fill it with data (including sender's address) that was actually received*, either in this `receive()` or earlier.

Omitting one packet (in random mode) means to do two receive* calls: the first is given a dummy packet (which is then discarded). The second is given the real packet to fill (the one given to this `receive()`).

For each DatagramSocket we hold a *receive pool*, in addition to the send pool. Deferring one packet (for reordering) is the same as omitting, except that the first packet is not discarded but pooled.

Duplication is done by one receive*. The data is copied to the given packet, and also to one or more temporary packets, which are pooled.

In block and delay modes, if no timeout is defined, we do receives* repeatedly, either discarding the packets or pooling them, until the mode changes. Each `receive()` call then blocks until the mode changes – which is indeed what would happen in the real world in complete network halt (which is what we simulate in these modes). If timeout is defined, things get a lot more complicated – we solved this problem, but it is beyond the scope of this paper.

Polling the receive pool means to take a packet, and copy its data into the `DatagramPacket` given to the `receive()`. This means that in a `receive()` that decides to poll the pool, no receive* is done. After leaving delay mode, we need to empty the pool ASAP: this is done by turning on a flag, meaning the socket is now in a state of emptying the receive pool: every `receive()` copies a packet from the pool, avoiding doing receive*. When the pool has been emptied, the flag is turned off.

Copying data from a pooled packet to a given `DatagramPacket` presents another delicate issue, stemming from the double limitation imposed on the data buffer by the API: Suppose the DatagramPacket given to `receive()` (not under the noise mechanism) has a buffer of m bytes, and the packet arriving from the network contains n bytes. The number of bytes that will be written in the packet is min(m,n). When copying a pooled packet, there are three numbers to consider – the length of the original packet from the network, the buffer length of the pooled packet (which was originally used to receive* from the network), and the new packet which will be handed to the application. It is complicated to guarantee that the final result in the handed packet is a possible result that would have been obtained if this packet was really used to receive* the original packet. The exact explanation of the problem and the solution is again beyond the scope of this paper.

5. Comparing the automatic noise tool to a white box approach in testing of group communication

For a given black-box test, automatic simulation of network noise randomly creates different scenarios at each run. We call this the *black box generic approach*. In contrast, a tester might simulate some network behavior by manipulating internal software events. We call this the *white-box non-generic approach*. Each approach has its pros and cons. For example, defining and forcing a test scenario for a specific program under test might be easier with the white-box approach, taking into account the semantic of the program. On the other hand, it is easier to create many interesting tests with the black-box approach. In this section we compare the two approaches, based on our experience in using the two approaches in testing a specific group communication application.

5.1 Stack architecture group communication

Group Communication is a mean for providing multi-point to multi-point communication for a group of processes. Groups are usually dynamic, in the sense that the set of group members continuously changes. A group communication service provides means to deliver a message from a specific member to all of the group members. In addition, the group communication service usually supplies a group membership service, which tracks the set of group members and reports these changes to the members. The output of the membership service is called a *view*, consisting of the set of the currently active members in the group. The membership service strives to deliver the same view to all active members.

A group communication service can support various guarantees for its messages delivery: best effort unreliable delivery, reliable delivery (ensures that messages sent among non-faulty processes are not lost), and Virtual Synchrony (VS) delivery, which means that all messages from a group member are delivered atomically to the group. That is, either all group members or none of them receive the message [1], [2].

The Java group communication system VRI (Versatile Replication Infrastructure), whose design is based on the Clue (The AS /400 Cluster engine [6]), supports the above mentioned guarantees for its message delivery, and supplies a membership service to track the group members. The architecture of the VRI is based on a stack of layers. All layers support a generic interface which includes calls related to either message delivery or membership change. In addition, every layer is responsible for a certain task: the Membership layer is responsible for the membership change service, the Virtual Synchrony (VS) layer is responsible for keeping the VS delivery guarantees, and the Transport layer is responsible for sending the messages, using Java UDP, guaranteeing the delivery of those messages to the group members.

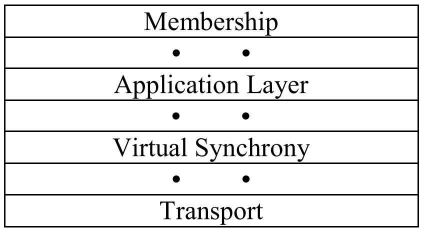

Fig. 1: The VRI layers architecture. There is one interface for up calls and one for down calls. The two interfaces are used by all the layers

5.2 The manipulator layer (decorator) test technique

The white box testing approach of the VRI is implemented by the Manipulation layer. The Manipulation layer is responsible to inject noise into the VRI stack in order to test specific uncommon scenarios. Following the Decorator design pattern [5], the Manipulation layer implements the generic interface of a layer in the stack, and it can be inserted at any point in the stack without affecting other layers. The manipulation is done by delaying stack calls, changing the parameters of stack calls and rejecting stack calls. By doing so, the manipulation layer simulates delays of messages delivery and bugs in other layers in the stack. The decision whether to manipulate a stack call can be either random or deterministic, and depends on the member's name, the call parameters and the state of the stack.

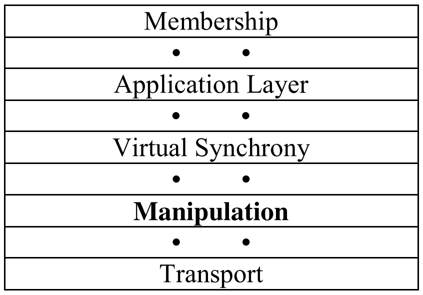

Fig. 2: The VRI layers architecture, with manipulation layer added. The Manipulation layer can be inserted anywhere in the stack, because it follows the up-calls and down-calls interface.

5.3 Example 1: testing of message completion in virtual synchrony

Given a communication group, the virtual synchrony layer guarantees that all messages from a group member are delivered atomically to the group. That is, either all group members or none of them receive a message. Missing messages typically occur as a result of a failing group member. When the membership layer identifies that a group member is failing, a new group is created. As part of the group creation process, the Virtual Synchrony layer performs a messages completion procedure so that all living group members will get the same set of messages.

In order to test the Virtual Synchrony's messages completion, we would like to test the following scenario:

1. (A, B, C) are part of a certain view.
2. C sends messages to the group members.
3. C fails. Before it failed, it had succeeded to deliver some VS messages to A but not to B.
4. During the view change, A should guarantee the atomicity, i.e. deliver to B all the messages from C that B missed.

It is easy to create a black-box test that potentially leads to the above scenario. Simply create a group (A, B, C). Let C send messages to the group and then make C fail. However, the probability of the scenario is very low, as the probability of losing a message in a normal environment is low, and the probability of losing one right before a member fails is even lower. Using the VRI implementation, we have succeeded to create this scenario only after 18 hours of testing, which included a group of six members dynamically changing every two minutes (i.e. members were leaving and joining the group), and a messages load of 100 messages per second for each member.

Thus, control over the timing of a member failure and message sending is required. This type of control can be obtained either in a white-box manner, using the manipulation layer, or in a black-box manner, using the automatic simulation of network noise and failures.

Here is how to obtain the scenario with the automatic simulation of network noise:

1. Set the automatic noise mechanism on C only, on mode *random-radical*, direction *outgoing*, and set *Remote Nodes* to be the IP of C.
2. (A, B, C) are part of a certain view.
3. At first, C sends messages to A and B, with just little disturbance with the latter.
4. Sooner or later the noise-creation mode changes to delay or block – no packets will be sent to B. Packets to A continue to arrive undisturbed.

5. The Membership layer of B identifies that C has failed, and notifies A that a group change is in order.
6. During the group change protocol, The Virtual Synchrony Layer of A delivers the missing messages to B.

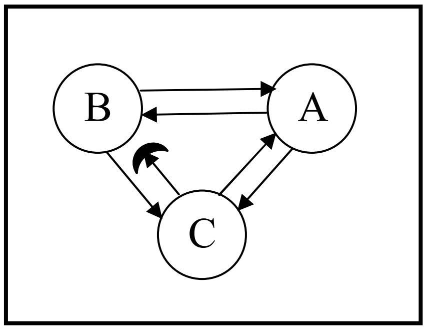

Fig. 3: Strongly disturbing outgoing messages from member C to B will result in the removal of C from the view, and the activation of the VS message completion procedure.

Instead of telling C to make noise explicitly with B, we can set *Remote Nodes* to *random*. In high probability, B will be chosen; alternatively, if A is chosen, a symmetric scenario happens, with A and B changing roles. This demonstrates how interesting scenarios happen also with "stupid", general-purpose setup of the noise mechanism parameters.

To obtain the above scenario with the white-box Manipulation layer, we set the Manipulation layer between the Virtual Synchrony layer and the transport layer, and make the manipulation layer of B lose a range of messages (From M to N) from C. Doing so would yield the following sequence:

1. (A, B, C) are part of a certain view.
2. C sends messages at the range 0 to (M-1) to both A and B.
3. B loses the incoming messages at the range M to N, while A receives them.
4. B receives message (N+1), identifies that it is not a consecutive message (i.e. B performed an illegal action) and notifies A that a group change, in which C should be thrown out of the group, is in order.
5. During the group change protocol, The Virtual Synchrony Layer of A delivers the missing messages to B.

The advantage of using the manipulation layer is the complete control over the scenario. Using the manipulation layer, the tester knows exactly which messages were lost and should be completed. The automatic noise cannot support this. The disadvantage is that the requirement of the manipulation layer has to be programmed specifically for this test, whereas with the automatic noise it only requires setup of preferences. In addition, the automatic noise test forces a correct and possible environment state. Thus, the test is perfectly black box, i.e. does not require a change in the program under test. In addition, losing the range of messages, as the manipulation layer does here, is only possible if the transport layer has a bug. Finally, with automatic noise the same test can be easily enhanced by randomly blocking incoming or outgoing messages on any of the group members. This will create additional interesting scenarios with very little additional work. For example, a partition scenario between (A, B) and C is easily obtained when A and B are blocked.

5.4 Example 2: testing of mew view in the transport layer

The Membership layer of the VRI stack is responsible to track the set of group members and report these changes to the group members. The output of the membership service is called a view, consisting of the list of the currently active members in the group and a unique identifier (*view id*). The membership service strives to deliver the same view to all active members. One of the group members is selected to be the view leader – the manager of the view change protocol. Having decided that the group members changed and a view change is in order, the membership layer of the view leader distributes a New View message to the membership layers of the group members. The membership layer of each group member invokes a New View call down the stack, to inform the stack layers that the group has changed. The New View call influences the stack layers, including the transport layer.

When the transport layer sends a message to all view members, it adds the view id to the message. The transport layer at the receiver side should deliver this message only if the message's view id is equal to the current view id of the receiver. That way, the transport layer guarantees that messages sent to all of the view members will arrive, indeed, only to the view members.

Because the New View calls in different members can not be totally synchronized, it is possible that the transport layer of a certain member C will be in a certain view V1, and the transport layer of another member B will be in the new, consecutive view V2. In this case, when B sends a message to the group with a View id V2, C should not deliver it up the stack, but rather save the message until its New View call is invoked.

In order to test the above behavior, we need to perform a test that causes one member of the group to be in a new view V2 and to send a message to another member which is still in the old view V1. Following is the scenario that will implement the required test:

1. (A,B,C,D) are the group members. The view id of the group is V1. A is the view leader.
2. D fails.
3. The Membership layer of A, the view leader, sends a New View message to (A,B,C)
4. Both A and B receive the New View message and invoke a New View call down the stack. The new view is (A,B,C) and its View id is V2.
5. B sends a message M to the group, marked with view id V2.
6. C receives the message from B and saves it (C has not received the New View message yet, and therefore can't deliver message M upwards).
7. C receives the New View message from A, and invokes a New View call down the stack.
8. C delivers the saved message M up the stack.

The probability of creating the specific scenario when running the black box test is very low, as usually the New View call in both stacks happens almost simultaneously (since the New View message arrives almost simultaneously to B and to C).

Using the automatic noise tool will not help much: it can cause the New View message to be delayed when coming to C, and then message M will be handled before it and the scenario will occur. However, since the tool chooses randomly which messages are delayed, the probability that this specific message is chosen is very low. Using higher strength will increase the probability, but will cause so much noise that the test will fail long before the desired scenario can happen.

By contrast, it is easy to create this scenario using the manipulation layer. The manipulation layer in member C can be set above the transport layer, and delay the new view event for a few seconds. This period of time gives member B enough time to send messages that will follow the scenario's requirements.

6. Conclusion

The automatic network noise simulation tool provides a powerful utility to test, in a normal environment, how UDP-based programs would behave in a less friendly environment. It can therefore help in revealing bugs early in the process. It requires very little effort, and can be done by testers with no effect on the code. On the other hand, it is "blind" to the semantics of the tested application, and is therefore not always suitable to test well-defined scenarios. For such tests, it may be more beneficial to write specially-tailored testing code. The two approaches can complement each other.

7. References

1. K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. Proceedings of the Eleventh Symposium on Operating Systems Principles, pages 123-138, Austin, Texas, November 1987.
2. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys 33(4), 2001.
3. E. Dekel and G. Goft. ITRA - Inter-Tier Relationship Architecture for End-to-end QoS. Proceedings of the IASTED PDCS 2001.
4. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. IBM System Journal. Vol. 41, No. 1, 2002.
5. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Addison-Wesley, 1995.
6. G. Goft and E. Yeger Lotem. The AS /400 Cluster engine: A case study. Proceedings of the 1999 ICPP Workshop.