A Kernel-based Communication Fault Injector for Dependability Testing of Distributed Systems*

Roberto Jung Drebes, Gabriela Jacques-Silva,
Joana Matos Fonseca da Trindade, Taisy Silva Weber

Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – 90501-970 – Porto Alegre, RS, Brazil
{drebes,gjsilva,jmftrindade,taisy}@inf.ufrgs.br

Abstract. Software-implemented fault injection is a powerful strategy to test fault-tolerant protocols in distributed environments. In this paper, we present ComFIRM, a communication fault injection tool we developed which minimizes the probe effect on the tested protocols. ComFIRM explores the possibility to insert code directly inside the Linux kernel in the lowest level of the protocol stack through the load of modules. The tool injects faults directly into the message exchange subsystem, allowing the definition of test scenarios from a wide fault model that can affect messages being sent and/or received. Additionally, the tool is demonstrated in an experiment which applies the fault injector to evaluate the behavior of a group membership service under communication faults.

1 Introduction

Specification errors, project errors and inevitable hardware faults can lead to fatal consequences during the operation of distributed systems. Developers of mission critical applications guarantee their dependable behavior using techniques of fault tolerance. Examples of such techniques include detection and recovery from errors, and error masking through replicas. The implementation of these techniques should be strictly validated: it should be assured that recovery mechanisms are capable of masking the occurrence of faults, and that unmasked faults lead the system through a known, expected, fault process.

Experimental validation is the main goal of fault injection. A fault injection test experiment lies in the introduction of faults from a given scenario into a system under test, the target, to observe how it behaves under the presence of such faults. Many software-implemented fault injectors exist for different platforms. When dealing with communication faults, most of them, however, are specific to some proprietary technology. For example, EFA [1], runs exclusively on the A/ROSE distributed operating system, while SFI [2] and DOCTOR [3] are restricted to the HARTS distributed system. ORCHESTRA [4], was originally

* Partially supported by HP Brazil R&D and CNPq Project # 472084/2003-8

developed for the Mach operating system and later ported to Solaris. When tools are not locked to a specific architecture, they may be designed to work with a specific runtime environment, such as FIONA [5] or the network noise simulator for Contest [6], which only support Java applications.

ComFIRM (*COMmunication Fault Injection through operating system Resources Modification*) is a tool for experimental validation, built exploring the extensibility of Linux through the load of kernel modules. This allows it to minimize the probe effect in the kernel itself. The tool aims to inject communication faults to validate fault tolerance aspects of network protocols and distributed systems.

Like ORCHESTRA, the tool provides many options to inject communication faults and allows specifying the test experiments through scripting. However, different than ORCHESTRA, ComFIRM was conceived exploring specific characteristics of the open source Linux kernel, which is available to many different architectures. Even if both ORCHESTRA and ComFIRM are script based, a fundamental difference exists: the former uses a high-level interpreted language, while the latter uses a special bytecode oriented language, which gives ComFIRM more power to specify test conditions and actions, and a lower intrusiveness in the target and supporting systems.

This paper emphasizes ComFIRM's fault model, design and use. Section 2 introduces fault injection concepts. Section 3 describes the tool's components and its mechanisms. Section 4 presents design and implementation issues of ComFIRM. Section 5 demonstrates the application of the tool in a practical experiment using the JGroups middleware as a target. Final considerations and future perspectives are presented in the closing section.

2 Experimental fault tolerance validation using fault injection

A software fault injector is a code that emulates hardware faults corrupting the running state of the target system. Software fault injection modifies the state of the target, forcing it to behave as if hardware faults occur. During a test run, the injector basically interrupts or changes, in a controlled way, the execution of the target, emulating faults through the insertion of errors in its different components. For example, the contents of registers, memory positions, code area or flags can be changed.

Network protocols and distributed systems assume the role of the target when considering communication environments. The injection of faults is an important experimental tool for their dependability analysis and design validation: the developer can obtain dependability measures such as fault propagation, fault latency, fault coverage and performance penalty under faults. Fault injection can also be used to evaluate the failure process of non fault-tolerant systems when faults are present. Also, and perhaps more important in distributed systems, targets can be tested when they operate under abnormal timing conditions.

The validation of protocols and distributed systems, however, meets inherent space and timing constraints. While fault injection activities can be used to test the targets under special timing conditions by delaying messages, for instance, the intrinsic fault injector activities impose an extra load on the target which can bring unintentional timing alterations to task execution time. Thus, when developing an injector, special care should be taken to limit the injector intrusiveness on the target system to minimize the probe effect on the dependability measure.

A complete fault injection campaign includes the following phases. First, a fault scenario for the test experiment should be specified. This includes the decision of location, type and time of injected faults. Next, the effective injection of faults should take place. During this phase, experiment data must be collected, and as a final step the data must be analyzed to obtain the dependability measures of the target protocol. For this paper, our major concerns are the creation of scenarios, control of location, type and time of fault injection and the effective injection of faults.

2.1 Building kernel-based fault injectors

Software fault injectors can disturb the target system and mask its timing characteristics. This happens because the fault injector usually alters the code of the target inserting procedures that interrupt its normal processing. When dealing with the communication subsystem of an operating system, those procedures can be executed as user programs or as components of the kernel. The closer they are to the hardware, the more efficient those procedures tend to be. One alternative is placing the fault injector at operating system level, which is the approach used by ComFIRM.

The exact location inside the operating system depends on how it is structured. The fault injector can operate, for example, through system libraries, system calls or other executable resources of the system (like interruption or exception routines, device drivers or modules). Whenever the target protocol requests services from these operating system resources, the fault injector can act and change the execution flow to inject faults.

If a fault injector is located inside the operating system, the intrusiveness, or probe effect, in the target protocol code is greatly minimized, since no context switches are necessary between the target and injector, beyond those originally existing between user and kernel contexts. Even if the target protocol is not disturbed or altered directly, however, the fault injector could disturb the operating system integrity: the intrusiveness moves from the target protocol to the operating system. An efficient kernel-based fault injection tool, then, must restrict the way it alters the original kernel, both in terms of timing and code alteration.

3 ComFIRM: a communication fault injection tool

ComFIRM intends to address the main issues regarding the injection of software-implemented communication faults, that is, validating fault-tolerant mechanisms

of network protocols and distributed systems. Like ORCHESTRA, the fault types it supports are: receive and send omission, link and node crash, timing and byzantine faults. These classes represent a typical fault model for distributed systems, as detailed in Section 3.1.

Currently, ComFIRM explores the Netfilter architecture [7] available in versions 2.4 and 2.6 of the Linux kernel [8]. ComFIRM can be applied to any device running recent versions of the Linux operating system. Since it uses high-level features of Linux, it is architecture independent, and so can run both in servers and workstations as well as embedded devices. Beyond that, our choice of Linux as platform was influenced by its free software nature. This makes a large amount of Linux information available, such as user manuals, kernel documentation, newsgroup archives, books and, of course, source code.

Two basic guidelines were followed while developing the tool. First, the injector should allow the configuration of the experiment during runtime. Also, the description of fault scenarios and fault activation should be possible using simple rules, which combined could be used to describe more complex scenarios.

Runtime configuration is done through virtual files created in the `proc` file system. Four files are used: writing to `ComFIRM_Control`, general commands can be passed to the tool, like activation, deactivation, and log detail level; reading from `ComFIRM_Log`, the sequence of fault injector events can be obtained; finally, `ComFIRM_RX_Rules` and `ComFIRM_TX_Rules` receive the fault injection rules that are used for reception and transmission, respectively.

3.1 Fault model

Usual fault models for distributed systems, as the one suggested by Cristian [9], can be implemented using communication faults. A node crashes when it stops to send or receive messages. A node is omitting responses when it does not send or receive some of the messages. It is said to have timing faults when it delays or advances messages. Byzantine behavior happens when the values (contents) of messages are altered, messages are duplicated, or if a node sends contradictory messages to different nodes.

For any given experiment, a fault scenario is described from the chosen fault model. Considering a communication fault injector, the scenario depends on the way the tool handles communication messages, that is, how messages are selected and manipulated [10].

A scenario must describe what message the injector must select to manipulate, be it by deterministic or probabilistic means. It is not enough to operate randomly on messages, unless the distribution of faults follows some known model.

With ComFIRM, messages can be selected by the following criteria: message contents, message flow or counters and flags. Content based selection considers some pattern in the message, as confirmation messages or connection close requests. When selection is based on message flow, contents are ignored, and all messages which match a rule are considered as a single group. It is then possible, for example, to manipulate a percentage of the messages, or even one in each n

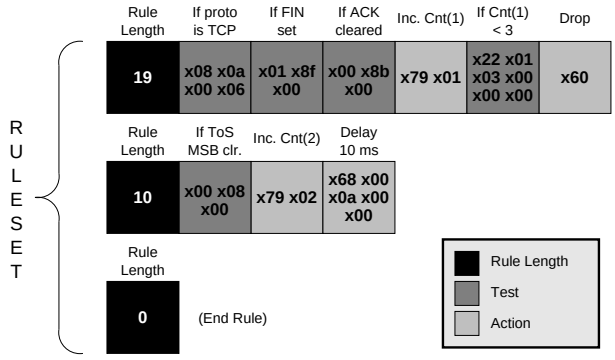

Fig. 1. ComFIRM example rules

messages. Selection by external elements considers conditions based on timers, user variables and measures of some physical parameter. It is the combination of the types of selection that gives the tool great flexibility in describing scenarios.

After determining which messages will be selected, it must be specified how to manipulate them. Three classes of manipulation can be identified: message internal actions, actions on the message itself, or unrelated to the messages. The first class defines actions that change some message field. The second, defines the most common actions, as losing messages (where the selected message is simply discarded), delaying messages (late delivery) and duplication. The last class includes actions that manipulate counters, timers, variables and user level warnings. Again, the possibility to combine several actions allows the definition of sophisticated fault scenarios for a test experiment.

The injection of timing faults can test the target under conditions that may trigger timing bugs, deadlocks or race conditions. Developers usually assumes that communication is "instantaneous", or even when that is not the case, they are unable to practically test their systems under degraded network performance in a controlled manner. With ComFIRM, not only the developer can test his applications with late message delivery – applied deterministically through message content examination, and not statistically as usually provided by network emulation tools such as NIST Net [11] and Dummynet [12] – but he can also alter message contents, forcing explicit message retransmissions which may result in timing situations.

3.2 Selection and manipulation of messages

ComFIRM's operation is performed using a bytecode language developed to describe an experiment through operation codes. The choice of a bytecode language to represent the fault scenarios is related to the lower timing intrusiveness its compact and efficient representation imposes on the experiments. Since the tool executes inside the operating system kernel, the interpretation of a higher level

x00 Test bit	x20 Test counter
x08 Test byte	x28 Test random byte
x10 Test word	x30 Test timer
x18 Test double word	x38 Test flag

Table 1. ComFIRM selection instructions

x00 Test if equal	x00 Test if cleared
x01 Test if greater	x01 Test if set
x02 Test if less	

Table 2. ComFIRM selection instruction modifiers

language would need to either be done in kernel space, which presents many challenges, or in user space, which would require frequent context switches between these modes to run the fault injection scripts, resulting on a significant performance penalty.

Using this representation, rulesets are composed of individual rules and can be independently configured for both the transmission and the reception flows. A rule is composed of a set of operations, or instructions, which determine either selection conditions or manipulation actions, arranged in a nested *if-then* structure. Instructions in a rule are evaluated in sequence until a test condition returns false, or an action specifying that a message should be dropped, duplicated or delayed is reached. If a test is evaluated false, processing is resumed in the next rule. If a message was discarded, duplicated or delayed, no further rules are processed for that packet. Figure 1 presents the relationship between instructions, rules, and rulesets. When specifying rulesets, each rule should be prepended by its rule length, and a null (zero length) rule specifies the end of the set.

The injector can select a message by its contents, through message flows, or based on external elements, like timers, counters and flags. Selection instructions are given in Table 1. It is possible to test the contents on a bit, byte, word (16 bits) or double word (32 bits) level. The modifiers presented in Table 2 should be applied to these operations, to determine relations. The bit and flag testing operations (instruction codes 0x00 and 0x38) can only determine if a bit is cleared or set (0x00 and 0x01 modifiers). The remaining operations can test if a value is equal, greater or less than the operand (0x00, 0x01 and 0x02 modifiers, respectively).

Bit testing can be used to verify specific protocol control flags. Testing of bytes, words and double words is interesting when testing other protocol fields, like addresses, commands, sequence numbers and offsets. For selection based on message flow, the injector can test a counter value or it can randomly select messages. ComFIRM provides up to 256 general use 32 bit counters, which should be manually incremented or decremented, through explicit manipulation operations.

x40	Modify bit	x70	Duplicate message
x48	Modify byte	x78	Modify counter
x50	Modify word	x80	Modify timer
x58	Modify double word	x88	Modify flag
x60	Drop message	x90	Dump message
x68	Delay message		to log

Table 3. ComFIRM manipulation instructions

x00	Assign value	x00	Clear
x01	Increment	x01	Set
x02	Decrement	x02	Complement

Table 4. ComFIRM manipulation instruction modifiers

For random selection, the injector can obtain a kernel supplied pseudo-random value. For selection of messages based on external elements, timers and flags can be used. Timer resolution when running on the x86 architecture is that of the Linux kernel, that is, 1 millisecond in current versions. Just like the counters, 32 single bit flags are available to the test operator. With all these selection operations, precise selection conditions can be specified.

Selection instructions are passive and by itself could be used for monitoring, but are not enough for active fault injection. Manipulation operations, presented in Table 3, can be combined to the previous operations for defining complex fault scenarios. ComFIRM supports manipulating bits, bytes, words (16 bits) and double words (32 bits) from the message contents, acting over the external elements: timers, flags and counters; as well as delaying, dropping and duplicating messages. An extra instruction can dump the contents of a message to the fault injector event log.

Similar to the selection instructions, there are modifiers which can be applied to manipulation instructions, as shown in Table 4. For bit and flag related operations (0x40 and 0x88), their operands can be cleared, set or complemented. These actions can be used to emulate *stuck-at* or byzantine faults. The values of bytes, words, double words, timers and counters, with the appropriate modifiers, can be assigned, incremented or decremented. Operations which drop, duplicate, delay and dump messages do not support any modifier.

3.3 Fault manifestation

It should be noted that when changing contents of a message through manipulation, all error detection and/or correction codes that the protocol may implement, become invalid. If the goal is to test the detection mechanisms of transport protocols, this approach is suitable. If it is to test higher-level protocols, we should observe which supporting protocols are used. If a supporting protocol like UDP, which provides unreliable datagram service, is used, manipulation

of message contents makes sense and the injected faults can reach higher-level target protocols. However, if lower protocols that handle errors, such as TCP, are used as the basis for multicast or membership target protocols, it will be the supporting protocol itself that will react and recover from errors. The target protocol on the upper layers will be unaware of the injected faults.

While TCP is extensively used in network applications, it is stream oriented and has FIFO semantics, what, in addition to not supporting multicast, makes it ill-suited for messaging in group communication systems [13]. UDP packet delivery, however, while unreliable, is commonly used as the basis for the development of communication protocols where the required reliability is implemented on upper layers, such as group communication systems. ComFIRM can directly alter the contents of UDP and TCP messages, but as explained above TCP alterations are masked by the protocol itself. Regarding UDP, even when its checksum is employed, the injector can make the receiving application ignore the verification code by writing a value of zero in its location. This is interpreted as if the sender did not enable the checksum mechanism. The destination, when receiving a message like this, cannot tell whether it was the fault injector or the sender who set this value. In this way, the alteration in the packet contents generated by ComFIRM are not discarded and can reach the receiving end.

3.4 Creating a fault scenario

Rulesets described in Section 3.2 should map to a specific instance of the supported fault model. These instances are the fault scenarios of a fault injection campaign. For instance, discarding messages emulates crash and omission faults. Delaying messages emulates timing faults and when the contents of the messages are modified, or messages are duplicated, byzantine faults can be emulated.

To illustrate how rules can be created to test implementations of network protocols and applications, here we present how ComFIRM can be applied to test specific conditions or events.

For example, a test operator may be interested in watching the behavior of a TCP implementation when requests to close a connection are lost. To do that, a rule must be specified that, for each message: (i) checks if its transport level protocol is TCP, value 6, (ii) verifies if it contains a request to close a connection (FIN bit set and ACK bit cleared) and (iii) drops the message. This rule would discard every close connection request. The operator may add a counter to the rule, to discard, for instance, only the first 3 requests processed. Such a rule would, after matching request to close a connection, increment a counter and check if its value is less than '3', dropping the message only if so.

Another rule may be set to test how a target system behaves when IP packets with low-precedence Type of Service (ToS) bits (in the range 0-3) are delivered with a higher delay. For that, this rule would check the priority bits and, if they are found to have a low value, introduce and artificial delay of, say, 10 ms. This is done testing the most significant bit of the ToS field, bit #8, and adding a delay of 10 ms. Additionally, the rule can increment a counter to monitor how many times this action was performed. The increment should happen before the

delay instruction is executed, since that instruction causes instruction evaluation to stop.

The representation as bytecode of the ruleset containing the two rules described above can be seen in Figure 1.

4 Implementation details

The current version of ComFIRM is a major rewrite of an original architecture which was very intrusive to the Linux kernel. It required modifications in key functions of the network subsystem and the timer handler, since Netfilter hooks were not available. Wrappers were used in the packet sending and receiving functions and the periodic timer handler had to be adapted as well. This modifications had to be made in the original kernel source files, and so recompilation of the full kernel was necessary to instrument it. Also, the tool had to be constantly updated whenever a new kernel version was released. While this was functional, it was not a clean solution. The changed kernel could behave in an unanticipated way, out of its original specification, invalidating the results of any tests performed over it. The rewrite of ComFIRM using the Netfilter framework, significantly minimizes this probe effect.

Netfilter [7] is a kernel framework for packet manipulation that defines specific points, or hooks, in the protocol stack where callback functions can be registered. When packets reach these points, they are passed to the functions which have full access to their contents. The functions can read the packets and decide if they should continue the traversal of the protocol stack or not. Netfilter supports callback functions for the IPv4, IPv6 and DECnet protocol families. Hooks are available on many locations: where local packets arrive and leave the host and where packets are forwarded, both before and after routing.

ComFIRM is implemented as a Netfilter module for the IPv4 protocol family. For incoming packets, it registers a callback function in the `NF_IP_PRE_ROUTING` hook, which processes packets identified as IP before routing occurs. For outgoing packets, a function is registered at the `NF_IP_POST_ROUTING` hook, where all processing of IP packets is already done and they are just about to be sent to the transmission media. These callbacks parse the fault injection rules, repeatedly for every packet sent and/or received. ComFIRM's evaluation of a rule can return to the protocol stack how the packet should be handled. The Netfilter supported actions used by ComFIRM are `NF_ACCEPT`, `NF_DROP` and `NF_STOLEN`. `NF_ACCEPT` is used when a selection rule does not match the packet, that is, it should be processed normally, or when a packet is matched but the only action rules are content, flag or counter manipulation. `NF_DROP`, as the name implies, is used when an action indicates that a packet must be dropped. `NF_STOLEN` is a special case of drop, where the packet does not continue the traversal of the protocol stack as well, but its allocated resources are not freed. This is used for delayed messages, which are queued for later processing inside the periodic timer handler. For message duplication, the message is sent or delivered during rule interpretation

and when the full rule evaluation is finished a `NF_ACCEPT` guarantees that the sending or delivery will be performed again.

The other core component of ComFIRM is the processing of the virtual files created on the `proc` filesystem, which are used for inputting rules, control commands and reading the fault injection event log. To each of these virtual files, functions are associated to the `open()`, `close()`, `read()` and `write()` manipulation primitives. This gives the test operator access to the ComFIRM functions and structures, which are internal to the kernel, using standard system utilities.

The controlling virtual files and the callback functions are dynamic resources which are registered on module load and unregistered on module removal giving ComFIRM the characteristics of a *plugin* that can be easily added or removed from a running system, without reboot.

5 ComFIRM demonstration

Here, we present the application of ComFIRM to a target system to observe how the communication faults it creates get manifested on the upper layers of a distributed application. For our testbed, we chose JGroups [14], an open source toolkit which provides reliable multicast communication for applications developed in Java.

JGroups is flexible, in that it allows its protocol stack to be adapted to different development requirements and/or network characteristics, through the inclusion of specific protocol layers. Also, the toolkit provides a group membership service (GMS), making it possible to create group views, consisting of nodes which can be spread across multiple LANs or WANs. Members can join or leave such groups, and a detection mechanism may notify each member whenever others join, leave, or even detect node crashes. Detected crashed members can then be removed from the group view, maintaining a consistent state. These features turn JGroups into an attractive alternative to the development of distributed systems and applications.

For this example, we started with a very simple protocol stack consisting of a multicast protocol at its base. Next, a loss-less transmission layer was added, through the addition of a NAKACK (negative acknowledgement) protocol. To provide membership change notifications (i.e. node join, leave or crash detection) the group membership service was added, by including the GMS protocol. This set of layers became the starting point for our experiments.

The behavior of the GMS layer was tested under three different scenarios: (i) without a fault detection mechanism, (iii) with a fault detection mechanism based on a heartbeat protocol (FD) and (iii) with a fault detection mechanism based on sockets (FD_SOCK). The modularity of JGroups protocol stack makes it easy to modify the scenarios, since it is only necessary to alter the stack configuration. It is not even necessary to recompile the target application.

On top of our target, JGroups, we developed an application which uses its services by joining a specific group and notifying the user when the membership changes. These view alterations are provided by the GMS. Using the fault

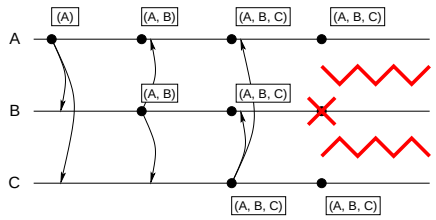

Fig. 2. JGroups' GMS used without a fault detector

injector under the JGroups middleware and our application above it, it is then possible to observe fault propagation from the network to the application. Three instances of our JGroups' application were started, in three different machines: A, B and C. Each machine joins the group at a different time, and multicast messages notify group joins and leaves. ComFIRM was applied in host B to break connectivity of this machine with the other members of the group (A and C), that is, emulating a link crash.

The first scenario is illustrated in Figure 2. As expected, after B gets out of reach (when ComFIRM is activated), members A and C still consider that the machine is part of the group. Since the protocol stack uses a negative acknowledgement protocol – in which it is assumed by default that all messages reach all destinations, and only a transmission fault generates a retransmission request – members A and C do not receive any message from B and assume that no error has occurred. The lack of fault detection mechanisms causes an inconsistent group view.

Next, the application was configured to use JGroups' heartbeat based fault detection mechanism (FD). In this case, each member sends an *are-you-alive* message periodically to its neighbor. If a response is not received within a time interval (3 seconds by default), and for a fixed number of tries (2 by default), the member is suspected, and after a second round, excluded from the group view. Figure 3 corresponds to this situation. At left, it is shown a common case where the machine B simply fails (process failure). Since no responses are sent to the other machines' inquiries, they remove machine B from the group view. At right, ComFIRM is applied to break connectivity between the machines (link failure). While the application is still running, it cannot be reached by the other members, which update the view. This is not instantaneous. After the injector is activated, it takes a variable time until the other nodes acknowledge that the machine is unreachable, which depends on the time interval and number of tries described above, being typically in the order of 12 seconds ($3\times2\times2$).

Finally, the third scenario represents the situation in which a fault detector based on a socket mechanism (FD_SOCK) is used. The protocol establishes TCP connections between each member of the group and its neighbor. The advantage of this method is that no *are-you-alive* messages need to be sent,

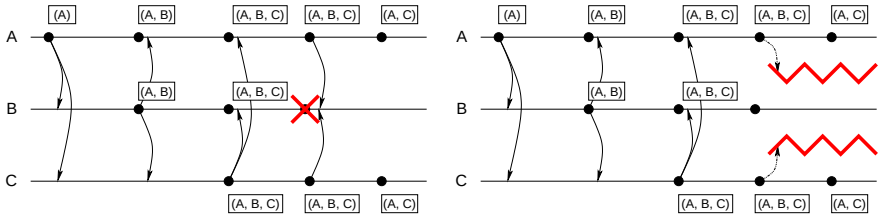

Fig. 3. JGroups' heartbeat based fault detector

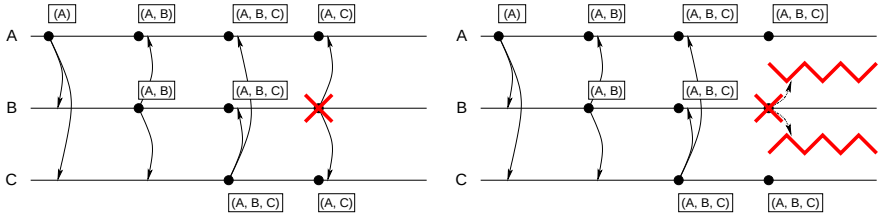

Fig. 4. JGroups' socket based fault detector

and so traffic is reduced. When a process is terminated, the socket is closed by the virtual machine and operating system. This causes TCP FIN requests to be sent, as illustrated in Figure 4. At left, it is shown that when these requests reach machines A and C, and the sockets are locally closed, they update their views to exclude B. At right, however, ComFIRM does not let the requests reach A and C, who keep their TCP connections in the established state, maintaining an inconsistent view. Since no data is sent through these TCP connections, only the TCP *keep alive* mechanism may detect the fault, but this mechanism is seldom used, and if so, it defines that at least 120 minutes should have elapsed to send the *keep alive* messages which could trigger the link crash detection. This demonstrates the unsuitability of the FD_SOCK protocol to detect router and connectivity faults.

This example does not aim to provide a complete example of a dependability evaluation of the JGroups' group membership service, but exemplify how ComFIRM can be applied to such kind of evaluation. Link crash can be emulated through many ways, but ComFIRM's instructions allow the selection of specific messages, which helps in a more precise setting of the experiments. The advantage of inspecting and selecting messages deterministically, instead of randomly selecting messages, is that it is possible to test the fault tolerance mechanisms in a shorter time, since it is not necessary to wait for the random selection of a message which would trigger the mechanisms. Random selection, however, is

also supported by ComFIRM (through the 'Test random byte' instruction, which can be used to build more complex random-based selection scenarios). By experimenting with the targets early in the development process, new bugs can be found and a greater confidence can be had on the correct operation of the fault tolerance mechanisms of distributed systems and protocols, building more reliable systems.

6 Concluding Remarks

Fault injection is a powerful technique to evaluate how protocols and distributed systems behave when faults occur. A fault injection tool allows the designer to measure the efficiency of detection, correction and error recovery mechanisms of a system before it is put into effective operation.

Several approaches to implement fault injectors are available. ComFIRM is implemented as a Linux kernel module using the Netfilter framework for communication message processing. This gives the tool full access to both the incoming and outgoing message flows, in a very clean and non-intrusive way. The bytecode instructions supported by ComFIRM allow messages to be inspected and selected in a deterministic or statistical way, and provides many actions to be performed with packets, which mimic the behavior of real faults: message drop and duplication, late delivery and content manipulation. The tool is fully operational and can be easily added to a running Linux system to perform experiments with distributed systems, as shown in Section 5.

In the future, ComFIRM may be adapted to work with other protocol families, like IPv6. Since this is an emerging infrastructure standard, many new protocols and applications are being developed on top of it. Their testing and validation are necessary and so a fault injection tool which supports IPv6 can be valuable. For such adaptation, ComFIRM only needs to be registered in the IPv6 Netfilter hooks, which already exist. Optimally, new instructions can be implemented which support larger variable sizes (which hold 128 bit addresses, for instance), or even which permits that the contents read from a given offset in a packet can be used as a parameter to the next instructions. This would help to process the variable size message headers concatenated through the *next header* field used in IPv6.

References

1. Echtle, K., Leu, M.: The EFA fault injector for fault-tolerant distributed system testing. In: Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems. (1992) 28–35
2. Rosenberg, H.A., Shin, K.G.: Software fault injection and its application in distributed systems. In: Proc. of the 23rd Intl. Symposium on Fault Tolerant Computing. (1993) 208–217
3. Han, S., Shin, K.G., Rosenberg, H.: DOCTOR: An integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems. In: Proc. of the Intl. Computer Performance and Dependability Symposium. (1995) 204–213

4. Dawson, S., Jahanian, F., Mitton, T., Tung, T.L.: Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In: Proc. of the 26th Intl. Symposium on Fault Tolerant Computing. (1996) 404–414
5. Jacques-Silva, G., Drebes, R.J., Gerchman, J., Weber, T.S.: FIONA: A fault injector for dependability evaluation of Java-based network applications. In: Proc. of the 3rd IEEE Intl. Symposium on Network Computing and Applications. (2004) 303–308
6. Farchi, E., Krasny, Y., Nir, Y.: Automatic simulation of network problems in UDP-based Java programs. In: Proc. of IEEE International Parallel & Distributed Processing Symposium 2004, Santa Fe, New Mexico (2004)
7. Russell, R., Welte, H.: Linux netfilter hacking HOWTO (2002) Available at: <`http://www.netfilter.org/documentation/`>.
8. Beck, M.: Linux Kernel Internals. 2nd edn. Addison Wesley (1998)
9. Cristian, F.: Understanding fault-tolerant distributed systems. Communications of the ACM **34** (1991) 56–78
10. Dawson, S., Jahanian, F.: Probing and fault injection of protocol implementations. Technical Report CSE-TR-217-94, University of Michigan (1994)
11. Carson, M., Santay, D.: NIST Net: a Linux-based network emulation tool. SIGCOMM Computer Communication Review **33** (2003) 111–126
12. Rizzo, L.: Dummynet: a simple approach to the evaluation of network protocols. SIGCOMM Computer Communication Review **27** (1997) 31–41
13. Wiesmann, M., Défago, X., Schiper, A.: Group communication based on standard interfaces. In: Proc. of the 2nd IEEE Intl. Symposium on Network Computing and Applications, Cambridge, MA (2003) 140–147
14. Ban, B.: JavaGroups - Group communication patterns in Java. Technical report, Department of Computer Science, Cornell University (1998)