D7.2: Report on Debugging solution (isolation and replay): Reports on the outcomes of task 7.2

Project no. 257574

FITTEST

Future Internet Testing

Specific Targeted Research Project

Information and Communication Technologies

Due date of deliverable: 29-02-2012

Actual submission date: 02-05-2012

Start date of project: 01-09-2010 Duration: 36 months

Organisation name of lead contractor for this deliverable: IBM

Document Manager: Onn Shehory

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission	
CO	Confidential, only for members of the consortium (including the Commission	

FITTEST Consortium

This document is part of a research project partially funded by the ICT Programme of the Commission of the European Communities under FP7 as project number FP7-ICT-2009-5 257574.

UNIVERSIDAD POLITÉCNICA DE VALENCIA
Camino de Vera s/n
46026 Valencia, SPAIN
Tel: +34 690 917 971
Fax: +34 963 877 359
Contact person: Tanja E.J. Vos
E-mail: tanja@pros.upv.es

BERNER & MATTNER SYSTEMTECHNIK GMBH.
Gutenbergstr. 15
10587 Berlin, GERMANY
Tel: +49 30 688385240
Fax: +49 30688385259
Contact person: Joachim Wegener
E-mail: joachim.wegener@berner-mattner.com

UNIVERSITY COLLEGE LONDON
Malet Place, London
WC1E 6BT, UK
Tel: + 44 20 7679 1067
Contact Person: Mark Harman
E-mail: m.harman@cs.ucl.ac.uk

IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD
IBM Haifa Labs, Haifa University Campus, Mount Carmel
Haifa, ISRAEL
Tel. +97 2 48296154
Fax. +97 2 48296114
Contact person: Eitan Farchi
E-mail: farchi@il.ibm.com

FONDAZIONE BRUNO KESSLER
Via Sommarive, 18
38123 Povo – Trento, ITALY
Tel. +39-0461-314524
Fax +39-0461-302040
Contact person: Paolo Tonella
E-mail: tonella@fbk.eu

SOFTEAM
Avenue Victor Hugo, 21
75016 Paris, FRANCE
Tel. +33 1 30 12 18 57
Fax. +33 1 53 96 84 01
Contact person: Andrey Sadovykh
E-mail: andrey.sadovykh@softeam.fr

UNIVERSITEIT UTRECHT
PO Box 80.089
3508TB Utrecht, THE NETHERLANDS
Tel. +31 30 2534090
Fax +31 30 253 2804
Contact person: Wishnu Prasetya
E-mail: wishnu@cs.uu.nl

SULAKE CORPORATION OY
Korkeavuorenkatu, 35
00130 Helsinki, FINLAND
Tel. +358 10 656 7000
Fax. +358 10 656 7010
Contact person: Osma Ahvenlampi
Email: osma.ahvenlampi@sulake.com

Abstract

This deliverable presents the work done in Task 2 of WP7: Debugging FI applications for concurrency bugs. It presents the concurrency debugging solution developed in this task. The solution comprises two main functions. The first one is isolation, which allows learning which component or components of the system contain the bug. The second function is a high-level record-and-replay functionality, which enables debugging the "suspicious" component in isolation while maintaining the communication patterns that manifest the bug. These two functionalities extend the ConTest tool (of which details are found in D7.1).

Contents

1. Introduction

1.1 Purpose and scope

The purpose of this document is to describe the concurrency debugging technologies of IBM's ConcurrencyTesting tool, AKA ConTest, to support concurrency debugging of Future Internet (FI) applications. For details on the ConTest tool the reader is referred to D7.1.

1.2 Document overview

Chapter 2 discusses briefly the type of bugs which are typical to FI applications. These are the bugs which the IBM ConcurencyTesting tool aims to expose (as discussed in Chapter 3). Debugging support technologies for removing such bugs are discussed in Chapter 4. It refers to deadlock analysis, to recording at critical points, and to reply for debugging. Chapter 5 proceeds with details on deadlock prevention.

1.3 Definitions and glossary

For completeness and comprehension, before proceeding into the technical details, this reports (re-)presents definitions of a few terms. The terms concurrency, concurrency testing and concurrency bugs, are often used in reference to multi-threading, and possibly with distributed computing as well. In this report, concurrency is interpreted in its wider sense. That is, it refers to the concurrent execution of multiple interacting computing units, separated either physically or logically. Referring to FI applications, we focus on physical, cross-platform concurrency, where different components of an application execute across a network.

FI is used a shorthand to Future Internet.

Communication, in the context of this document, means passing of data between components, in particular over the network.

2. Concurrency issues in FI applications

To debug an internet applications' concurrency bugs, we first need to understand what typical bugs in these environments are, and under what circumstances and setting are they likely to manifest. In similarity to bugs in multithreaded systems, FI applications face timing scenarios which are vulnerable to concurrency bugs. In multithreading, these are usually referred to as "interleavings". In the FI context, two (or more) events may have an interleaving if their components communicate. A concurrency bug manifests if an interleaving occurs that is not correctly handled by the applications. Components of internet applications are typically developed and supplied by different vendors. As a result chances of unhandled interleavings (and therefore concurrency bugs) increase significantly.

An FI application integrator may use components in the test environment which may be later replaced by other components with different implementations in the field. This may occur either at the first deployment of the application, or over time as components are replaced by new releases of the same components or alternate components. Additionally, hardware platforms on which components execute in the production environment may differ from those available in the test environment. Further, even if initially the platforms are identical, over time there may be platform upgrades in the production environments from which differences may arise. Difference in implementation and in hardware platforms can influence performance, and in particular timings, of FI application. Consequently, timing observed in the test environment will commonly differ from timings in production. In such cases, traditional testing does not provide a good means for exposing concurrency bugs.

Another source of FI applications' concurrency bugs may arise from usage patterns. Component usage in the production environment may differ significantly from usage in test. A typical case for such difference is the rate of communication. Common consequent concurrency problems include queues filling up, deadlocks and data corruption, to name a few. Yet another source of complexity (and concurrency issues) is real-time constraints. In such cases components may have strict constraints on the time they are allowed to allocate to their operations. This may lead to operation timeout and delays at one component, but may be seen as a failure by other components, resulting in a failure of the integrated FI application.

The issues referred to above may be exposed and then debugged in late test stages of the composed FI application. In earlier work of WP7, we have developed a classification of such bugs and provided means for testing and exposing them, at least in part. In this document we report on techniques we have developed for debugging such FI concurrency issues. We do not claim to have fully addressed all issues, but we surely address well some of them, as will be described below.

2.1 Exemplifying FI application bugs

We have demonstrated major FI bug patterns via a toy networked application which we have developed as part of Task 7.1. An overview of the demo application was reported in D7.1. Although the demo application is a toy application, it includes the elements needed to exemplify the bug patterns in a realistic way relevant to FI applications. We do not repeat here the report on that application, however we do present the patterns and where necessary add supporting information. The demo application is a distributed internet shopping application, including an electronic shop, a payment server, as database, and customer endpoints. Interaction scenarios among these are standard in web shopping. Communication between the components is done via TCP sockets using uniform message formats. The application has several intentional bugs of the types described below.

2.2 FI concurrency bug patterns

2.2.1 Queues fill-up

Queues may fill up when some components are much faster than others, or clients generate heavy traffic. The component containing the queue may not have code to handle queue fill-up. It may also have such code, but since this scenario is hard to reach in test, this code is likely not tested.

Example 1 – well-known in client-server architectures. Consider a server that handles clients' requests. Requests are placed in a queue and server's threads remove requests from the queue and handle them. If requests arrival rate in higher than the handling rate, the queue may fill up. If queue fill-up is not well addressed we have a bug. The well-known Denial of Service (DOS, DDOS) attack is based on this kind of problem; a well-designed server can avoid address the attack (with some delays in response), but if there is a bug the attack may cause damage.

Example 2: Consider a component that generates data items and sends them to other component(s). The sender holds a queue for outgoing items. It also has threads that generate items and enqueues them, and threads dequeue items and send them. Network delays may cause queue fill-up, resulting in possible errors.

2.2.2 Deadlocks

Deadlocks can occur when there is a set of components, each waiting for another to perform some action. For example, in the demo application, assume that a customer sent a purchase request, received the acknowledgement, continued shopping, and after a while requested another purchase. The payment service's interactions are typically quick, relatively to the rate in which the customer is likely to submit requests. Therefore the purchase confirmation is given back to the user long before it requests another purchase.

Suppose now that at some occasion the query from the shop to the payment service was slow, due to a delay in the shop server or in the payment server or in the network in between. Before the payment service sent payment request to the user's component, the user sent its second purchase request. Now the server is designed not to give acknowledgement while there's a pending payment confirmation request for the same user. While the user's component is waiting for the acknowledgement, the payment service asks the user's component for the confirmation. Suppose, however, that the user's component is designed or implemented to "do one thing at a time", e.g. to make multithreading data protection easier. In particular, it is blocked while waiting for the acknowledgement, and doesn't handle the payment confirmation in the meantime (in the demo, this policy is implemented by using a semaphore). So each of the three components is waiting for the other in a cycle, and we have a deadlock.

2.2.3 Races

In multithreaded applications, races occur when two threads read and write from/to a shared memory location without proper synchronization, resulting in data corruption or wrong data read. In FI applications, similar races may manifest when applications read and write from/to a shared network data repository. For example, in the demo application, the database update operations are implemented as static service methods which write messages to the socket connected with the database server. Some of the updates are composite data updates (modifying more than one data item). In such cases it is necessary to ensure atomicity of data updates, but in our case the database server does not include code to ensure that. This is of course a bug as some updates may take place in-between two elements of a supposedly atomic composite data update, exhibiting a data race.

2.2.4 Timeouts

FI applications' components may have real-time constraints, and these may be violated by actual communication patterns. Such timeouts are in times invalid. For example, a service request has its timeout defined to be more stringent than the maximal time defined by service's SLA. A timeout may also be valid, but not correctly addressed by the application code.

3. ConTest for FI

Many concurrency bugs originate from the need for shared memory – be it local memory for multithreaded application or network storage for FI applications. Shared memory requires access protection. Inadequate protection results in data corruption or invalid data read (races). The protection mechanisms themselves can lead to further bugs, notably deadlocks. Other bugs result from broken assumptions about order of actions, or about completion time of actions.

IBM's ConcurrencyTesting tool [EFG+03], ConTest for short, tackles these bugs. The observation underlying it is that while different concurrent bugs have different causes, different descriptions and different results, at various levels of descriptions, almost all of them *can* be described, and construed, in terms of relative order of events. Therefore, if we exercise many different orders of events, we may expose concurrency bugs. To this end, only the order of *concurrent events* matters.

Trying all possible interleavings (relative order of concurrent events) is not feasible for any but the most trivial cases, because as the size of the application increases the interleaving space quickly becomes intractable. Indeed, ConTest does not try to do that. It just causes *many* interleavings to be exercised – orders of magnitude more than would be exercised without it.

The way this is done is by injecting *noise* before and after concurrent events. Noise means operations that cause some delay (or promote context switch in multithreaded applications). This is done randomly, and therefore the order of these events is varied from one execution of the events to another. As more interleavings are explored, more concurrent bugs can be revealed, and this can be done early in testing.

3.1 Low-level network noise

The ConTest tool implements several types of noise. The pre-FITTEST tool implemented mainly low-level noise, focusing on UDP and TCP/IP. Support for high-level protocols was added mostly in FITTEST. In ConTest for Java we implemented "noise for UDP", where the tool causes packet omission, duplication, and change of order. This is done by overriding the methods for packets send and receive. This is described in [FKN04]. We also implemented noise for TCP/IP. Data is guaranteed to arrive in order, in a stream. There is indeterminacy, however, in when different chunks of the stream will arrive. These lower-level types of noise do not fully address the needs of FI applications. This is discussed later in this report.

3.2 Moving from multithreading to FI

In Task 7.1, ConTest for concurrency in the multithreading domain was extended to address concurrency bugs of FI applications. This extension was reported in D7.1. The underlying concept of that extension is that concurrency of multi-threaded applications and concurrency of FI application both exhibit bugs as a result of their interleavings. Hence, in a nutshell, ConTest for FI works as follows: a given test is augmented by making it run many more interleavings, and this is achieved by adding *random noise* to the events that matter. This helps exposing concurrency bugs of the types described above. The debugging process is supported by debugging aids as specified in Chapter 4 below.

Note that it is sufficient to apply the noise at each component independently of the others. At a point where component *A* sends a message to component *B*, if we delay the sending on the *A* side then we are likely to cause a change of order of events in *B* – if *B* has other imminent events. It is useful to apply noise in *B*'s receiving method too, because from *B*'s point of view the noise was not necessarily generated on the sender side.

In the multithreaded domain, too, ConTest's noise is applied to each thread ignorantly to what other threads are doing. In that domain ConTest competes against other tools for finding concurrent bugs, tools that look at all the threads together, coordinating events between them (e.g. [MQB+08], [PLZ09]). Such coordination is relatively easy because the coordinating tool can run within the process under test, together with the threads it handles, and use the same shared memory and process info about the threads. In the internet application case there is no such luxury: coordinating processes on different nodes requires a whole communication protocol to be induced on all the application's components. The random approach of ConTest allows for a much simpler solution.

The details of the internet applications noise are different from multithreading noise in two respects as summarized below.

3.3 Noise for FI applications

Noise for FI applications should target calls to methods that coordinate processes: sending and receiving of messages, as well as writing and reading of shared data (from a DB or a remote file system). In the ConTest for FI tool, delays are added in the following places:

1. Before and after a message-sending method
2. Before and after reading and writing from/to shared data

Extending ConTest for multithreading, ConTest for FI application can add such noise. Some methods may implement both sending and receiving of data. We can treat them as *send*s, and add delay before their call. If they return received data to the code, as opposed to just using received data internally, we can add delay before the returning.

As discussed in D7.1, it is not sufficient to add noise only at the TCP and UDP *send* and *receive* methods. Internet applications today already use higher level protocols above TCP/IP, starting from HTTP and going much further. Proliferation of such protocols renders implementation of noise only at the lower level impractical, as a single event at a high-level protocol may be composed of multiple TCP send/receive calls.

Therefore, FI application system testers need noise to be applied at the level of abstraction that their application uses, and in which the bugs of their application are described – the level of the methods in their code under test. To address the diversity at the higher level of abstraction, we opted to asking the tester what the relevant methods are for her application. To this end, we enhanced ConTest with the capability of obtaining from the user a list of target methods.

3.4 FI noise application

In ConTest for multithreading, noise is applied using calls to *yield*, lock taking, *wait* or short *sleep*. These add delay, but also encourage the system's thread scheduler to exercise context switches. In communication protocols, we do not have a central scheduler. The only generic way to affect relative order of events is to add delay before and after them.

Following our insights from ConTest for multithreading, we apply noise using a "noise frequency" parameter, which specifies the probability of noise application. For each concurrent event, ConTest for FI chooses randomly, at this probability, whether or not to apply noise. For FI applications, however, the probability of noise is higher than it is for multithreading. Another noise parameter is its intensity. This roughly refers to the length of the delay imposed. Here, small values are good for multithreading, but larger ones are needed to expose FI bugs.

4. Debugging support

The ConTest tool aims to simplify debugging of multi-threaded and FI applications. To allow debugging support, ConTest was enhanced with several debugging facilities, including the callback option, the deadlock option, and the "orange box" option. The ConTest callback option allows a tester to query an application as it is executing. Thus, a test executed on a client can query the server regarding the part of the application which currently executes. The deadlock option allows a tester to acquire information regarding deadlocks that were encountered. The orange box facility can be used to collect information about the values and location of variable assignments of interest (for example in null pointer exception).

A sample program using the deadlock and "orange box" debugging features was implemented but cannot be disclosed in this public deliverable due to confidentiality issues. Its concepts are explained below.

4.1 Deadlock analysis support

ConTest provides two types of reports which are useful for analyzing deadlocks:

- **Lock status report**: this report provides information on the component or thread which is locking (holding a resource) and which is waiting for each lock
- **Last location report**: this report provides the current location within the code of a component or a thread

To enable these reports, the user of ConTest has to set the tool's preferences of `lockStatus` and/or `Location`, respectively. The reports can be obtained through a callback , or through ConTest report APIs (class `Report`).

4.2 Orange box

When an application fails, it is often useful to know something about its behaviour in the last segments of its execution prior to failure (in similarity to the role of a black box in an airplane). The ConTest orange box keeps a record of the last read or write accesses (two by default) to each nonlocal variable or data record (if the last two were reads, it additionally keeps the last write). For example, if the program execution was terminated due to an exception caused by variable `foo` having the value `null`, one can use the orange box feature to check where this value was set, and with what read/write operation is it associated. To enable the orange box, the user has to set the ConTest preference `orange_box`. This report, too, can be obtained through a callback or through the ConTest report APIs (class `Report`).

If the user in interested in viewing more than the last two operations, she may set the preference `last_values_size` to the desired number of operations.

Clearing debug information (in Java applications)

By default, if an object is no longer referenced and is collected by the garbage collector, ConTest will remove it from the orange box as well. In some cases this may remove some valuable debug information, for example, in the case where during an application failure the JVM was running the garbage collector. To prevent such loss of valuable debug information, a tester can set the ConTest preference `forgetting_orange_box`. This prevents objects with recorded variable information from being garbage-collected.

Note that this may apply high load on the JVM memory. The problem is especially severe with iterative tests where one JVM repeatedly runs tests (e.g., when a server handles client requests). To alleviate this problem, a tester can force ConTest to "forget" the current orange-box information it holds. For instance, if it is known that some test ended without a bug, the tester probably does not need the debug information and get disposed of it. Clearing the debug information is done either through callback or through the ConTest report API `Report.resetTest()`.

4.3 Record and replay

In many cases, the fact that a synchronization bug was identified is not sufficient for debugging, and it is necessary to expose the bug several times instead. To this end, we have enhanced ConTest with the replay feature. Once a synchronization bug occurs, the tester can use the replay feature to increase the likelihood that the same synchronization bug will reoccur. This is achieved by saving all ConTest initialization properties and restoring them back on the next run. Of course, because of non-determinism, we cannot guarantee that the bug will reoccur, but we increase the chance of this happening.

For recoding, the tester should specify the preference `replay = true`. In each execution a file will be saved in the ConTest output directory, under a name such as `KingProperties_1030542514554`, where the number is the ID of the specific execution.

To replay an execution, the tester should specify ID of the desired execution as the value of the replay preference. For example, `replay = 1030542514554`
ConTest will use this value to perform its current execution in exactly the same way as in the specified execution.

Timings of application operations are determined by many factors which are not controlled by ConTest. Such factors include the scheduler algorithm, platform architectures, network delays, and system loads. As a result, it is not guaranteed that a synchronization bug will reoccur when the replay feature is used. To increase the chance that the synchronization bug will reoccur during replay, the tester may attempt to minimize the effect of factors that impact the timing. For example, to neutralize the system load factor, the tester may execute some application components under test on a dedicated machine.

If ConTest debug reports (orange box, lock status, etc.) were enabled in the original ConTest execution, they are enabled in the replay execution as well. If they were not enabled in the first execution, if desired, the tested may enable them in the replay execution by changing the appropriate value in the `replay` file, rather than in the regular `KingProperties` file. It is important to note that the replay execution is be more likely to succeed in exposing the bug if the debug reports are enabled in the original execution, since the collection of data involved may itself affect the scheduling.

Testers are sometimes interested in debug print statements to in support of the debugging process. This may also be sought by testers who run ConTest iteratively, using the reply facility. This printing capability should be used with caution; the addition of debug print statements between replay executions might affect the application's timing and hence interfere with the replay, many times in a negative manner. To minimize this risk, the tester can prevent instrumentation of such a print statement. This is supported by the ConTest interface.

4.4 Lock History Report

Another facility of ConTest to support debugging is its function of writing a trace file with the complete "lock history" of the tested application. The trace includes events of taking a lock (e.g., on a data record of a networked DB), releasing a lock and waiting on a lock. To activate this facility, the tester should set the property `lockHistory = true`. A directory `lockHistoryTraces` will be created under the ConTest output directory. For each execution, a trace file will be generated and placed in that directory. The trace includes a data line for each event. The line specifies the name of the executing component, the description of the event, which lock object was involved, and the code location (when this data is available). In using this facility, a tester should keep in mind that, for many applications, traces may be rather large.

The debugging support facilities added to ConTest, and in particular the support they provide to debugging FI applications, are useful but have limitations. One should keep in mind that FI applications may be multi-vendor, distributed applications. Access to their code may be limited, and in many cases all that one can access is their APIs. This imposes testing difficulties, as it may be difficult or even impossible to isolate the exact locations of bugs. Nevertheless, with the facilities described in this report, one can focus on specific components and operations of an FI application under test, thus guiding the tester to elements of the application that are more likely the source of the bug.

5. Bibliography

[ABF+10]	R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S.D. Stoller, S. Ur, L. Wang, Detection of Deadlock Potential in Multithreaded Programs, IBM Journal of Research and Development, September 2010.
[BFM+05]	A. Bron, E. Farchi, Y. Magid, Y. Nir, S. Ur, Applications of synchronization coverage. PPoPP 2005.
[EFG+03]	O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur, Framework for testing multithreaded Java programs. Concurrency and Computation: Practice & Experience, 2003.
[FKN04]	E. Farchi, Y. Krasny, Y. Nir, Automatic Simulation of Network Problems in UDP-Based Java Programs. IPDPS 2004
[MQB+08]	M. Musuvathi, S. Qadeer, T. Ball, P.A. Nainar, I. Neamtiu, Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.
[PLZ09]	S. Park, S. Lu, Y. Zhou, Ctrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. ASPLOS 2009.