Georgia Institute of Technology

Critical Systems Laboratory

Final Report for “Fault Injector for Middleware Applications”

Douglas M. Blough and David Schimmel

School of Electrical and Computer Engineering
Georgia Institute of Technology

Sponsored by Raytheon Company

A. Project Overview

The Fault Injector for Middleware Applications (FIMA) project began in August 2003 and ran through August 2004. The primary goals of this project were to investigate techniques for injecting a wide variety of faults in middleware applications, to evaluate performance overheads associated with these techniques, and to develop a fault injection tool that incorporates the most promising of these techniques. This report summarizes the project accomplishments and details the final status of the project as of August 2004.

B. Project Accomplishments

B.1 Task Completion

The project tasks of the Statement of Work are included verbatim below. All tasks were fully completed as originally specified without modification.

Task Group A

Task A1: Test Bench Development and FIMA Test. Develop distributed object versions of standard vector and matrix libraries to run as applications in the testing process. This task will follow the test methodology developed earlier in the project to run a validation test of the FIMA, to be performed in the presence of the sponsor.

Task A2: Additional FIMA Code. This task will develop code for additional pieces of the FIMA, including basic error detection mechanisms, automatic wrapper insertion tools, external triggering interface, and GUI.

Task A3: FIMA Test Code Modifications. This task will perform any necessary modifications to the FIMA code that are identified during the FIMA validation test. The final FIMA code produced by this task is FIMA v1.0.

Task Group B

Task B1: Performance Evaluation of FIMA Version 1.0. Thorough evaluation of overhead of FIMA mechanisms both with and without faults being injected on the standard vector and matrix library codes. Demonstrate ability to maintain real-time performance with FIMA present.

Task B2: Server-side Communication Fault Injection via Wrappers. Add wrappers to inject communication faults in skeletons on the server. Automate wrapper insertion process.

Task B3: Java Support: Develop and test a Java version of FIMA, which works with the FIMA GUI and includes fault specification interface, communication fault injection capability, log file manager, fault triggering interface, and test application. Develop detailed implementation plan for memory fault injection with Java.

Task B4: FIMA Productization. This task will supply any final updates to documentation and code after the delivery and joint evaluation of FIMA v2.0. Task includes a performance evaluation of FIMA v2.0. Final updates may include product performance improvements deemed important and practical during product evaluation.

Additional Tasks

Project Management. Manage all administrative, financial, schedule and technical aspects of tasks, and hold test reviews for each release of code. Deliver monthly cost/schedule/progress/status reports. Participate in quarterly program meetings with the sponsor.

Startup plan and review. Prepare a startup plan that addresses cost, schedule, technical, quality, risk and customer expectations. Conduct a startup review with Raytheon within 45 days after receipt of contract award.

Product Definition. This task will include a monthly dialogue on proposed FIMA capabilities and uses, to help clarify each during FIMA development. Develop and maintain a requirements document that describes the capabilities to be implemented in FIMA. Share draft and final copies of this document with Raytheon.

User and Porting Support. This task includes a weekly teleconference, regular telephone support hours, and email support to port the FIMA to Raytheon's computing environment. This task also includes the delivery of bug fixes Georgia Tech makes as Raytheon uses v1.0 and v2.0 during the period of performance.

FIMA Documentation. This task will supply documentation on the programming and use of the FIMA. Documentation updates will be delivered with each release of FIMA code. Share draft and final copies of these documents with Raytheon.

B.2 Fault Injection Techniques and their Overhead

Techniques for both communication fault and memory fault injection were developed. The primary technique for communication fault injection was interception of middleware communication calls. This technique was implemented for CORBA within all versions of the FIMA tool described in Section B.3. The technique was found to work equally well with both C++ and Java application programs, and was therefore incorporated in both the C++ and Java versions of FIMA. Memory fault injection was done via a FIMA thread operating in the same address space as the application. This technique was fully developed and evaluated for C++ applications and is present in the C++ versions of FIMA. Memory fault injection in Java applications was the subject of a planning task in this project. We determined that a separate thread operating in the same address space as the application could be invoked using the Java Native Interface (JNI) and then used for memory fault injection. However, since in Java, the same address space is shared by the application and the Java Virtual Machine (JVM) run-time environment, the fault injection experimenter has a somewhat lower degree of control of memory fault injection for Java applications as compared to C++. To be specific, in certain cases, it might not be possible to target fault injection only to the code and data portions of the application without the possibility of affecting the JVM. The details of what can and can not be done with Java memory fault injection are given in the PowerPoint slides delivered to the sponsor as part of the FIMA project review in March 2004.

The various fault injection techniques were implemented and evaluated as part of the FIMA project. Full results detailing the overheads of the various techniques are contained in the March 2004 FIMA review slides. A summary of those results is as follows. FIMA initialization overhead (incurred once at program start time) is approximately 63 msec plus 16 μsec per statically defined fault that must be read from the static fault queue at initialization time. Run-time overhead is approximately 200 μsec for a fault injected into a static partition (code or static data) and about 14 msec for a fault injected into a dynamic partition (heap or stack). The additional overhead for injection into dynamic partitions resulted from the need to dynamically check the memory limits of those partitions to ensure that injection is done to a "live" area. In both cases (static and dynamic partitions), fault injection does not block the main application, because injection is done within a separate thread. Hence, the injection overhead primarily impacts the granularity of fault injection time. In addition, if fault injection is done very frequently (not typically the case for most experimental set-ups), there might be a small impact on CPU utilization.

B.3 Tool Development and Documentation

The studied fault injection techniques were implemented within several versions of the FIMA tool, as part of the project. FIMA 1.4 and FIMA 2.0 were two versions of the tool designed to support C++ applications. J-FIMA 1.0 was developed to provide basic communication fault injection capability for Java applications. All versions of the tool used the architecture developed in an earlier project funded by the sponsor and shown in Figures 1 and 2.

Figure 1 shows the process that is used to instrument an application under test with the FIMA code. The original application must be modified in several ways prior to being used with FIMA. Injection of communication faults is achieved through wrappers that are placed around method invocations on remote objects. These wrappers are inserted in the stubs that connect the local process to the ORB and to the remote object. Hence, part of the FIMA instrumentation process modifies the stubs to insert the wrappers. Some modification to the main application is necessary also. This involves adding a call to FIMA_Init() at the start of the main procedure to create the FIMA thread, initialize the FIMA data structures, and do various other start-up tasks.

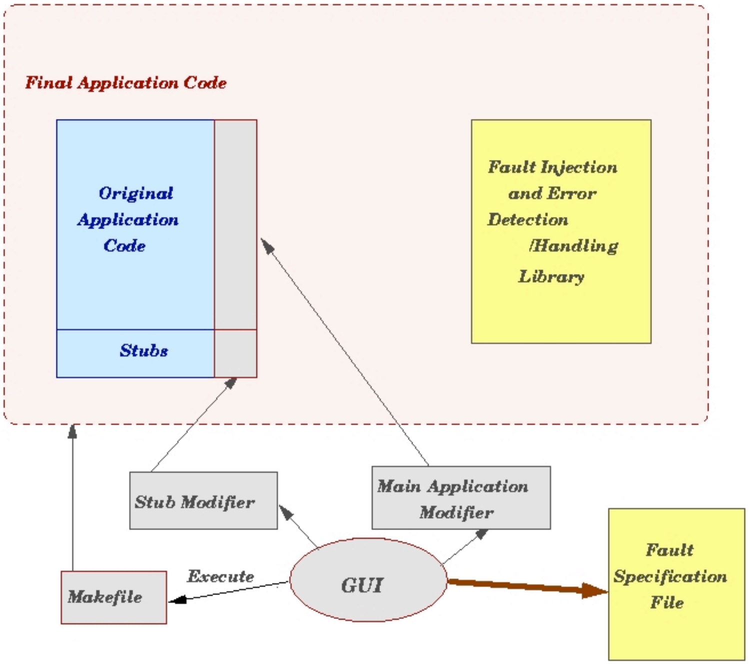
Final Application Code
Original Application Code
Stubs
Fault Injection and Error Detection /Handling Library
Stub Modifier
Main Application Modifier
Makefile
Execute
GUI
Fault Specification File

Figure 1: Application Instrumentation Process for FIMA

Much of the FIMA code, e.g. the memory fault injection module, is contained in libraries that must be compiled with the original application. This requires the application makefile to be modified to include an option to compile in the libraries. FIMA makes use of Posix threads, and if the application is not already using them, an option to compile in the Posix threads' libraries must also be included in the makefile. Once the stubs, application, and makefile have been modified, the test application can be compiled.

Figure 2 illustrates the run-time operation of FIMA. The fault injection and error handling code contained in the FIMA libraries run as a separate thread. When this thread is started via FIMA_init(), it reads a fault specification file to capture statically-specified fault information. Faults can also be specified and triggered dynamically by the application itself through the FIMA API.

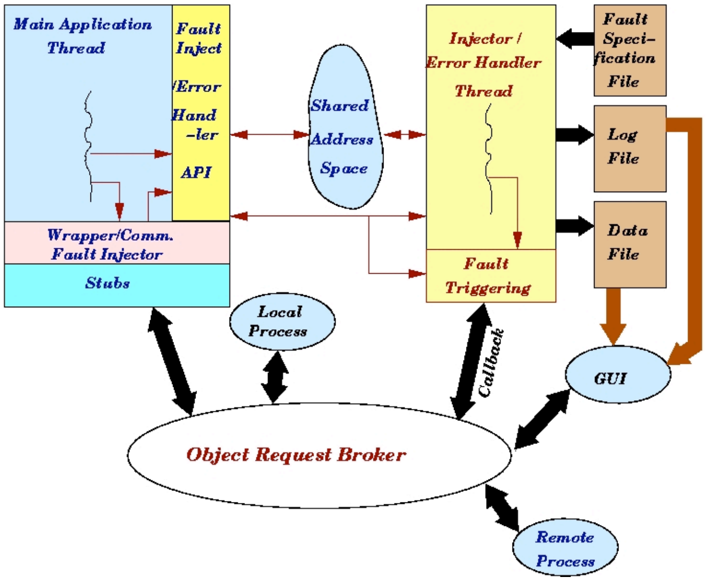

Figure 2: FIMA Run-Time Operation

The application can also call the FIMA thread for various purposes using the API. For example, the application might wish for a fault to be injected at a particular point in the program execution and it can specify and trigger this with API calls. The application can also call the FIMA thread to initiate error detection, e.g. by calling a time-out mechanism, and to handle detected errors. After detecting an error itself or being called by the application after it detects an error, the FIMA thread logs the error event and handles it in one of several ways that can be specified by the user, e.g.:

- flush the experiment log to disk and gracefully terminate the program
- flush the experiment log to disk and return control to the application
- do not flush the log and return control to the application

In addition to implementing time-outs within the FIMA thread, FIMA also contains exception handlers that prevent most of the common operating system and processor exceptions from crashing the application. These events are logged and execution is continued, if so desired.

One of the main components of the FIMA thread is the memory fault injector. This injector has access to the entire virtual address space of the application and is responsible for corrupting different portions of that space when specified. This code has the ability to corrupt instructions, stack, static data, and dynamic data of the application, either at specified or random addresses.

A list of features and code included with the different FIMA releases is as follows:

FIMA 1.4

- vector and matrix libraries for CORBA
- client-side communication fault injection
- memory fault injection
- event-triggered injection
- automatic wrapper insertion tool
- external fault triggering
- GUI

FIMA 2.0

- everything in FIMA 1.4, plus the following
- server-side communication fault injection
- fault-tolerant (triplicated) FFT application

J-FIMA 1.0

- communication fault injection
- event-triggered fault injection
- GUI

Documentation on how to install and build FIMA, and how to use FIMA's API, GUI, and static fault specification format are included with the FIMA releases that were delivered to the sponsor.

C. Conclusion

The FIMA project successfully developed and evaluated a variety of fault injection techniques for middleware applications. These techniques were incorporated into several versions of FIMA tools and delivered to the sponsor.