Robustness Testing of a Distributed Simulation Backplane

Kimberly M. Fernsler
IBM, Austin TX
kimfern@austin.ibm.com - (512) 733-5283

Philip Koopman
Carnegie Mellon University
koopman@cmu.edu - (412) 268-5225
http://www.ices.cmu.edu/koopman
Abstract

- This research is built upon and extends the **Ballista** project.
  - High level testing done using the API to perform fault injection
    - Send exceptional values into a system through the API
    - Requires no modification to code -- only linkable object files needed
  - Each test is a specific function call with a specific set of parameters
    - Combinations of valid and invalid parameters tried in turn

- Yes, Ballista can be extended and it turned out to be easy!

- Applied Ballista to a general-purpose distributed system software used for military simulations
  - Specifically engineered for robustness
  - They weren’t perfect -- but usually they weren’t too bad either
  - Ballista found the weak spots that they should concentrate on (a “profiling” tool for robustness!)

- Porting exception handling code seems to be a problem
Overview: Testing the RTI

- **System Robustness**
  - Motivation
  - Ballista Automatic Robustness Testing Tool

- **Current Application:** High-Level Architecture Run-Time Infrastructure (HLA RTI)

- **Enhancements for RTI Testing**
  - Results of Ballista testing on 4 versions of HLA RTI (86 functions)
  - Data analysis of results (77,000 tests)
  - Comparison to Operating Systems
  - Segmentation faults vs. RTI Internal Error exception

- **Issues and Future Direction**
  - Extending Ballista to other application areas
  - Creating a general-purpose, scalable testing framework
Why do we care?

- Dozens of vendors, hundreds of users...can’t afford a system crash
System Robustness

- **Graceful behavior in the presence of exceptional conditions**
  - Unexpected operating conditions
  - Activation of latent design defects

- **Robustness definition also includes operation in overloads**
  - Not in current research, but is set as an eventual goal
  - We conjecture overload robustness also hinges on exception handling

- **Our research goal: improved system robustness**
Background and Related Work

- Ballista gets its roots from both the Software Testing and Fault-Injection communities.

- Predecessors include:
  - FIAT
  - Crashme
  - FAUST
  - Fuzz
  - FERRARI
  - FTAPE
  - CMU-Crashme

- Previous Ballista Work: Testing POSIX Operating Systems

- Questions left unanswered:
  - Ballista worked on OS testing – would it work at all elsewhere?
  - How painful would it be to test Object-Oriented, C++ code with callbacks?
  - Are non-OS APIs/architectures be better suited to robust implementations?
RTI - Some Terms

- **HLA**: a general-purpose architecture for creating distributed simulations

- **RTI**: the RTI software is the actual implementation of services to coordinate operations and data exchange during a runtime execution

- **Federation**: a set of simulations and supporting RTI that are used together to form a larger model or simulation

- **Federate**: a member of a federation; one simulation
  - Could represent one platform, like a cockpit simulator
  - Could represent an aggregate, like an entire national simulation of air traffic flow

- **Federation Execution**: a session of a federation executing together
rtiAmb.subscribeObjectClassAttributes(
    ObjectClassHandle theClass,
    AttributeHandleSet& theAttributes,
    Boolean active)

RTI::Ulong
TEST OBJECT
INT_SAMPLE
INT_ZERO
INT_ONE
INT_TWO
INT_FOUR
INT_15
...
INT_129
INT_255
INT_257
INT_2POW31
INT2POW31_1
INT_NEG_ONE

RTI::AttributeHandleSet
TEST OBJECT
AHS_VALID
AHS_NO_CREATE
AHS_CREATE_ONE
AHS_CREATE_ALOT
AHS_CREATE_NEG
AHS_DELETE
AHS_MEMB_EMPTY

RTI::Boolean
TEST OBJECT
BOOL_ZERO
BOOL_ONE
BOOL_TWO
BOOL_THREE
BOOL_FOUR
BOOL_FIVE
BOOL_SIX
BOOL_SEVEN
BOOL_EIGHT
BOOL_NINE
BOOL_TEN
BOOL_NEG_ONE

rtiAmb.subscribeObjectClassAttributes(
    INT_2POW31, AHS_NO_CREATE, BOOL_ZERO)
RTI Testing Approach

For every test run, the following steps were performed:

1. Ensure that the RTI server (RtiExec) is running
2. Create a federation:
   Registers task with the RtiExec and starts up FedExec process
3. Join the federation:
   Testing task is a federate
4. Perform “scaffolding” setup functions
5. Test the actual function
6. Free any memory allocated by the setup functions
7. Resign from the joined federation
8. Destroy the federation:
   De-register from the RtiExec
9. Shut down the RtiExec if last test or error occurred
Enhancing Ballista for RTI - *Obstacles*

- **Exception based error reporting models**
  - Previous Ballista testing - any call which resulted in a signal being thrown was considered a robustness failure
  - The RTI throws an RTI-defined exception (rather than using the POSIX strategy of return codes).

- **RTI is a distributed system**
  - Certain amount of setup code must be executed to set the state before a test
  - In the OS testing all such “scaffolding” was incorporated into constructors and destructors for each test value instance
    - *e.g.*, creating a file for a read or write operation
  - In the RTI there were some function-specific operations required to create reasonable test starting points
  - Distinct scaffolding required to test each and every RTI function?

- **Testing object-oriented software structures**
  - Callbacks, passing objects by reference, class data types, and constructors
In general, the solutions turned out to be simpler than anticipated:

- **Exception based error reporting models:**
  - Included user-defined exception handling code in the general Ballista testing harness.
  - Any user-defined, “thrown” exception was considered a “pass”…
    … except for the “unknown” RTI exception, which indicated an internal RTI exception handling software defect.

- **User-configurable test scaffolding code:**
  - Used for clean setup and shutdown of the RTI environment
  - Only 10 different scaffolding code variants sufficed for 86 functions
  - Amount of code and development effort was relatively small

- **Object-oriented software structures:**
  - Ballista framework is flexible enough to support callbacks, passing objects by reference, class data types, and constructors – with only minor syntax changes
Evaluating Test Results

The results for RTI testing fall into the following categories, loosely ranked from best to worst in terms of robustness:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass</td>
<td>Function call executed and returned normally, no indication of error</td>
</tr>
<tr>
<td>Pass with Exception</td>
<td>Valid, HLA-defined exception was thrown, indicating a gracefully caught and handled exceptional condition</td>
</tr>
<tr>
<td>RTI Internal Error exception</td>
<td>RTI encountered a supposedly impossible error condition, RTI managed to free memory and resign from the federation cleanly</td>
</tr>
<tr>
<td>Unknown Exception</td>
<td>Unknown exception was thrown &amp; caught internally to the RTI by a catch-all condition (as opposed to a hardware signal)</td>
</tr>
<tr>
<td>Abort</td>
<td>Error occurred that was not caught, code exited immediately (“core dump”), no memory cleanup, required manual restart of federation</td>
</tr>
<tr>
<td>Restart</td>
<td>The function call did not return after an ample period of time (a “hang”)</td>
</tr>
<tr>
<td>Catastrophic</td>
<td>System was left in a state requiring rebooting the operating system to resume testing</td>
</tr>
</tbody>
</table>
Robustness Failures of RTI 1.0.3

- Testing was performed on 4 different versions of the RTI
  - RTI 1.0.3 for Digital Unix 4.0
  - RTI 1.3.5 for Digital Unix 4.0
  - RTI 1.3.5 for SunOS 5.6
  - RTI 1.3 NG for SunOS 5.6

- Over 77,000 data points collected
  - RTI developers’ goal was ZERO Aborts and ZERO RTI Internal Errors.
RTI 1.3.5 Failures - Different Platforms

Digital Unix

SunOS

Per Function Failure Rate

RTI Functions

Restart
Abort
Unknown exception
RTI Internal Error exception
**HW Signal vs. RTI Internal Error**

- Robustness failure rates from both RTI 1.3.5 versions are essentially the same

- **BUT**, different manifestations of robustness failures:
  - **SunOS port** - any unanticipated signal apparently leaked through and was seen as a segmentation fault
    - Code immediately aborts – no graceful shutdown
    - Could significantly disrupt the currently running federation execution
      - Other federates will not be informed properly that one federate has left
      - What will happen to the data that federate was sharing?
      - What are the consequent effects on the rest of the distributed simulation?
  - **Digital Unix port** - unanticipated signal was caught and converted to an RTI Internal Error
    - This allows recovery and cleanup; code is not aborted

- Illustrates possible problems in porting robust applications across platforms with different exception handling support.
Same Platform, Different Design Teams
Some Interesting Robustness Failures

- Client process randomly crashing through an RTI 1.0.3 service function call, requiring machine reboot to continue testing

- RTI 1.3.5 Digital Unix port - one test Aborted after producing:
  "Exception system exiting dues[sic] to multiple internal errors:
  exception dispatch or unwind stuck in infinite loop
  exception dispatch or unwind stuck in infinite loop".

- RTI 1.3.5 SunOS port – one Abort failure resulted in a “zombie” federate each time it occurred

- RTI 1.3.5 SunOS port – one Abort terminated after displaying:
  "Run-time exception error; current exception: RTI internal error
  Unexpected exception thrown."
  • Appears to indicate an incomplete implementation of an RTI Internal Error
Normalized Failure Rates

- **Directly measured robustness failure rates:**
  - RTI 1.0.3 for Digital Unix: 6.41%
  - RTI 1.3.5 for Digital Unix: 10.20%
  - RTI 1.3.5 for SunOS: 10.05%
  - RTI 1.3 NG for SunOS: 8.44%

  Computed by:
  - Determining the proportion of robustness failures across tests for each function within each system being tested
  - Producing a uniformly weighted average across all the functions

- **Operational Profiling**
  - Test with a specific simulation program running
  - Weightings would be used to reflect the dynamic frequency of calling each function to give an exposure metric that is potentially more accurate
  - Common-sense check on these results shows that functions with high robustness failure rates do in fact include commonly used features
Comparison to OS Results

Comparing Ballista Robustness Results of RTI with Typical Operating Systems

- RTI 1.3NG for SunOS
- RTI 1.3.5 for Sun OS
- RTI 1.3.5 for Digital Unix
- RTI 1.0.3 for Digital Unix
- Typical Operating Systems

Robustness Failure rates

- 0% 5% 10% 15% 20% 25%

- RTI is more robust than POSIX operating systems
  - RTI robustness = 6.4% - 10.2%
  - POSIX OS robustness = 10.0% - 22.7%

- This was expected – RTI was designed specifically to be robust
  - Fewer Catastrophic and Restart Failures as well

- Newer software version does not necessarily indicate increased robustness (as seen with OS results)
Future Work

- Working to make Ballista part of the standard verification suite for RTI development
- Explore issues of concurrent testing to find potentially more subtle bugs related to timing and resource sharing
- Pattern analysis software for better test selection & result analysis

- Generalized testing now available as WWW testing service
  - [http://www.ices.cmu.edu/ballista](http://www.ices.cmu.edu/ballista)
Conclusions

- **Ballista robustness testing approach**
  - Scalable, portable, reproducible
  - Demonstrated to find exception handling problems in software specifically written to be highly robust

- **Testing the RTI led to scalable extensions of the Ballista architecture**
  - Exception-based error reporting models
  - Object-oriented software structures (callbacks, pass by reference, constructors)
  - Operating in a state-rich, distributed system environment
    - All were easily integrated into the existing Ballista framework!

- **As expected, RTI is much more robust than POSIX operating systems. BUT several weak spots found:**
  - Non-robust testing responses included exception handling errors, hardware segmentation violations, "unknown" exceptions, and task hangs.
  - Difficulties in providing comparable exception handling coverage across platforms
  - Results illustrate common robustness failures that programmers can overlook