Automated Testing of Distributed Systems

Nathan Boy Jared Casper Carlos Pacheco Amy Williams

Abstract

We present a technique to test servers that interact with clients using the Sun RPC protocol. The technique requires the user to provide two things: a list of RPC calls for the server being tested, and a set of invariants that are required to hold over the RPC communications trace between a set of clients and the server. The technique works by generating random sequences of RPC calls and checking that the invariants holds over the traces. If an invariant is violated, the violating sequence of RPC calls is reported to the user. We report the results of our testing a block server and a lock server.

1 Introduction

Testing network applications is complex and time-consuming, often even as laborious as writing the application itself. The test suite must be large and varied, and it must exercise all the capabilities of the protocol being implemented. Generating a sufficient test suite can therefore be a very daunting task.

A complication unique testing network applications is the fact that network traffic is often unpredictable, so applications run non-deterministically. Bugs in non-deterministic systems can be very difficult to expose: doing so essentially requires some luck during testing. In addition, once a bug is discovered, it can be problematic to replicate, since the exact system state (including the wider area network traffic) is often not known and/or not easily controlled. This issue calls for a rich testing suite, and one in which testing results are logged.

We address the specific problem of testing servers that interact with clients using the Sun RPC protocol. Our proposed solution is a system that automatically generates and performs tests on a such a server. Given a specification of RPCs handled by the server, we generate tests that send random sequences of RPCs to the server and generate a trace of the RPC activity between client and server. After obtaining the trace, a set of user-specified invariants are verified. These invariants check for particular patterns in the traces. If any of these invariants fail, the trace that failed the invariant is presented to the user.

2 Related Work

In the area of test generation for distributed systems, previous work has involved inspecting the code of the program to guide in the generation of good tests cases [3, 6, 8, 5]. Random testing has been used with differing degrees of success [7, 4]. Our approach aims to combine the benefits of random testing—automation, and generation of tests that a tester might not have thought of—while allowing the user to specify the general shape of tests, to avoid generating a larger number of meaningless inputs.

Much of the work on testing and verification of distributed systems focuses on testing internal correctness properties of distributed programs. In this area, model checking [1] has enjoyed wide acceptance as an effective testing technique. The disadvantage of model checking is that the system first has to be abstracted, and in this abstraction, important details may be lost or mistakes made. The benefits of model checking are that once an appropriate abstraction exists, the system can be exhaustively tested for a small model.

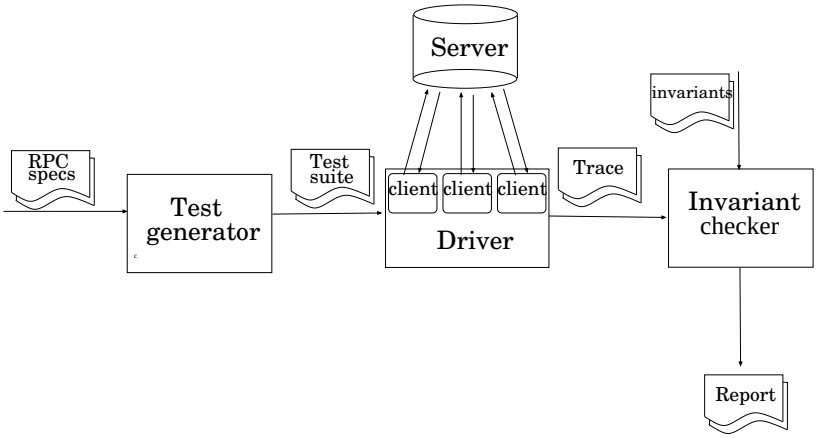

Figure 1: Layout of the testing system.

Our approach is different: we do not require the user to abstract a system, and our tests are by no means exhaustive. However, they test an actual system, not an abstraction.

3 Design

Our testing system is divided into three components. The first generates the test code. The second is the driver which uses this code to perform the tests and generates a trace file. The third checks that the user-specified invariants hold for the trace file. The layout of the system is shown in Figure 1.

3.1 Test Generation

Tests are generated by creating sequences of random RPC calls—one sequence for each client taking part in the test. The user supplies a list of RPC signatures that the server accepts and may also specify a set of possible values for each RPC argument. Each time the tester is run, the test generator selects calls from this list and produces valid input values for each call. These complete RPC calls are packaged together into a complete test which is used by the test driver to directly send the calls to the server.

A simple example of a test for a lock server is shown below.

```
client 1: ACQUIRE("lock_a")
client 2: ACQUIRE("lock_b")
client 1: ACQUIRE("lock_c")
client 1: RELEASE("lock_b")
client 2: RELEASE("lock_b")
```

Because the RPCs are generated completely randomly, the client calls may not correspond to "normal" behavior. In the code above, for example, client 1 releases `lock_b`, which it does not own. Our assumption in this style of test generation is that a server should behave correctly regardless of any client behavior. This type of random testing has two main benefits: it generates code that a tester may not think of, and it can (and does) generate large numbers of such tests which exposes to a broad spectrum of possible usage patterns.

RPCs are specified in the same format as the `.x` file used by arpc's `rpcc` program. This allows our testing tool to use the arpc library to perform the tests and, if the server already used arpc, saves the programmer time in writing the RPC specifications for our tool. The programmer may optionally specify arguments for our tester to use when performing the RPC calls in the `.x` file. These *value specifications* are given in

ValueSpec	::==	`test` *ValueList*	
ValueList	::==	*StringList*	
	::==	*IntList*	
	::==	*BoolList*	
StringList	::==	*String*	
	::==	*String*, *StringList*	
String	::==	`"*"`	
	::==	`"*`*NumRange*`*"`	[A string containing a varying integer value inside it]
	::==	`"?`*NumRange*`"`	[A varying length string containing random text]
IntList	::==	*integer*	
	::==	*NumRange*	
	::==	*IntList*, *IntList*	
NumRange	::==	[*integer*:*integer*]	
BoolList	::==	`T`	
	::==	`F`	
	::==	`T,F`	

Figure 2: Language for specifying values that RPC arguments may take.

```
struct put_args {
  opaque key<>; /* test "key[1:5]" */
  opaque value<>; /* test "?[1:10]" */
};
```

Figure 3: An example of the two random string specifications. This example defines the arguments for a `PUT` RPC call on a block server.

specially formatted comments on the same line as the argument in the `.x` file and may describe strings, integers, or boolean values (extending the language to handle other types of values such as floating point numbers is trivial). If more than one value is given, the tester randomly picks a value from that set when generating RPCs. The format of these comments is given in Figure 2. Note that if value specifications are not given, default values are used.

Two of the *String* specifications are worth explaining. In the first, a range of numerical values is given in a *NumRange*. For this specification, our tester inserts a random value in the given range into the string. In the second, a *NumRange* is preceded by a question mark. In this case, our tester generates a string of random characters with a length in the specified range. Examples of both of these string specifications is given in Figure 3. The possible values for the `key` argument are `"key1"`...`"key5"`; `value` is any string of length between 1 and 10, inclusive.

3.2 Test Driver

After the tests have been generated, the test driver performs the sequence of RPC calls that were created. In order to perform these calls, a set of clients are instantiated and connections opened to the server. As the RPC calls are made, the communication between the server and the clients are logged—we refer to this as the communications trace or simply the trace. This trace is later inspected to determine whether the results

conform to the user-specified invariants. Below is an example trace for a lock server. The trace is simply a log of all the RPC calls sent and received by each of the clients.

```
client-server: client 1 ACQUIRE("lock_a")
client-server: client 2 ACQUIRE("lock_b")
server-client: client 1 GRANT("lock_a")
client-server: client 1 ACQUIRE("lock_c")
server-client: client 2 GRANT("lock_b")
server-client: client 1 GRANT("lock_c")
client-server: client 1 RELEASE("lock_b")
client-server: client 2 RELEASE("lock_b")
server-client: client 1 GRANT("lock_c")
```

Clients wait a random amount of time after sending each RPC, and after a timeout expires, the test terminates, with the resulting trace passed to the invariant checker (Section 3.3).

3.3 Invariant Checking

In order to verify the accuracy of a server's behavior, user-specified trace pattern invariants are checked. These invariants are written in the language specified in Figure 4. This language's function is primarily pattern matching, and it provides primitives useful for writing the desired guarantees.

As an example, consider checking a lock server. The lock server's author may wish to check that the server never gives a lock to more than one client at the same time. One way to express such an invariant on the RPC trace file is to ensure that for a given lock l, after a LOCK_GRANT RPC is sent by the server, no other LOCK_GRANT RPC is issued before a LOCK_RELEASE RPC is received for l. This invariant can be expressed using our language as shown in Figure 5. Another example invariant for a block server is shown in Figure 6.

A description of the functionality of each of the language's statements is given below. Note that invariants that are not found to be valid are ignored. If an invariant fails, the test associated with the trace is assumed to have discovered a bug, and the offending trace is presented to the user.

match RPC A pattern matching statement. All **match** statements must be matched in the trace in order for the invariant to be valid.

require RPC A pattern matching statement. If the invariant is valid (i.e., if all **match** statements are matched), and a **require** statement does not match, the invariant fails.

require $Conditional$ Ensures that the values of the variables bound during pattern matching obey the specified condition. The invariant fails if it (the invariant) is valid and the specified condition does not hold.

any Matches an arbitrary number of trace entries, similar to the regular expression pattern /*/. This statement is only useful when it is followed by a **match** or **require** pattern. It does not match the pattern given in the next **match** or **require** statement—such patterns must go with those statements.

any_except RPC Similar to **any**, but causes the rule to be invalid if the given pattern is matched. To allow an arbitrary number of constraints, multiple **any_except** or **any_except_fail** statements may appear before the next **match** or **require** statement.

any_except_fail RPC Similar **any_except**, but if the given pattern matches, the rule fails (instead of becoming invalid). This statement is enforced—the invariant fails—even when there are subsequent match statements that have not yet been satisfied.

$Invariants$	::==	Invariant { $Statements$ }
	::==	Invariant { $Statements$ } $Invariants$
$Statements$	::==	$Statement$
	::==	$Statement$ $Statements$
$Statement$	::==	**match** RPC
	::==	**require** RPC
	::==	**require** $Conditional$
	::==	**any**
	::==	**any_except** RPC
	::==	**any_except_fail** RPC
RPC	::==	$Type$ $Client$ RPC_NAME($Arguments$)
RPC	::==	$Type$ $Client$ RPC_NAME($Arguments$) = $Retval$
$Type$	::==	send
	::==	recv
$Client$	::==	Arg
$Retval$	::==	Arg
$Arguments$	::==	ϵ
	::==	$ArgsList$
$ArgsList$	::==	Arg
	::==	Arg, $ArgsList$
Arg	::==	$Identifier$
	::==	*
$Conditional$	::==	$Identifier$ == $Identifier$
	::==	$Identifier$!= $Identifier$
	::==	$Identifier$ <= $Identifier$
	::==	$Identifier$ < $Identifier$

Figure 4: Language for specifying trace pattern invariants.

```
Invariant {
  match            send client   LOCK_GRANT(lock)     [lock and client are variable names, bound
                                                       during checking]
  any_except_fail  send *        LOCK_GRANT(lock)
  match            recv client   LOCK_RELEASE(lock)
}
```

Figure 5: Lock server invariant.

```
Invariant {
  match          recv *   BLOCK_PUT(key, value)
  any_except     recv *   BLOCK_PUT(key, *)
  any_except     recv *   BLOCK_REMOVE(key)
  match          recv *   BLOCK_GET(key) = value2
  require                 value == value2
}
```

Figure 6: Block server invariant.

4 Implementation

Our tester is mainly implemented in C++, but also contains some Perl scripts. The C++ class hierarchy begins with `test_suite` at the top, which drives the testing. `test_suite` contains pointers to multiple `test` objects which are our randomly generated tests (see Section 3.1).

Each `test` object contains a list of randomly generated `RPC_call` objects (Section 4.1) and a pointer to a `tester_client`. The `tester_client` object is responsible to send and receive RPC calls to and from the server (Section 4.2). As the RPC calls are sent/received by the `tester_client`, those calls are logged in a `Trace` object. `Trace` objects are generated for every `test`, and are used to run the invariant checker (i.e., the invariants are defined over a `Trace` object) (Section 4.3).

In addition to the main tester application, our design also includes a utility that compiles .x files and invariant definitions into functionality needed by the tester. Due to timing constraints, we will not be implementing a compiler, but instead describe how the translation might occur. We have manually performed the translation for the programs we are testing.

4.1 Test Generation

When the .x file for a given protocol is compiled, a C++ file is generated that contains functionality necessary to generate tests for that protocol. First, the .x file is parsed and a list of RPCs and arguments for each call is determined, along with possible values for each argument. With this information, four functions are generated for each call:

void *get_args(svccb sbp) If `sbp` is NULL, this function randomly generates valid arguments and returns a pointer to them in a memory block ready to be passed to the server as the arguments to the RPC call. If *sbp* is not NULL, it demarshalls the arguments contained in the svccb structure and returns a pointer to the resulting structure.

void print_args(FILE *f, void *arg) Writes a formatted string of the arguments in the structure pointed to by `arg` to the file `f`.

void *get_reply() Allocates enough memory to hold a reply to the associated call and fills it with the default reply values.

void print_reply(FILE *f, void *rep) Writes a formatted string of the reply in the structure pointed to by `rep` to the file `f`.

Function pointers to each of the above functions for a given RPC call are bundled into a `RPC_call_defn` object. An array containing one of these objects for each call is created and included in a `RPC_proto_defn` object.

Test generation occurs when a `test_suite` object is instantiated. The `test_suite` constructor is passed the number of tests to create, the protocol definition, and a random seed. The random number generator is seeded and the appropriate number of `test` classes are created. The `test` constructor is given the number of RPC calls to generate and the protocol definition and generates a `RPC_call` object for each call. The `RPC_call` constructor takes in the protocol definition. It randomly selects a valid RPC call id and calls the `get_args` and `get_reply` functions contained in the corresponding `RPC_call_defn` object to obtain the input and output pointers that will be given to the server. Once each RPC call in each test and each test in the test suite has been created, test generation is complete and the actual testing is ready to be executed.

4.2 Driver

The top-level driver is a Perl script that takes as input the number of tests to be run, as well as test-generation parameters such as the length of each client's RPC call sequence, the number of clients per test, etc. The script runs the number of tests requested, instantiating a new server and a new set of clients for each test.

Each test is identified by the random seed used to create it. The default operation of the script is to create tests with seeds $0, 1, 2, ...$ up to the number of tests specified by the user. For a test i, the top-level driver performs the following steps.

1. Start a new instance of the server on any unused port number.
2. Call the `test` program with seed i. The `test` program is written in `C++` and uses `libasync`, an asynchronous socket library [2]. The `test` program creates a test sequence to be performed by multiple clients, connects to the server, performs the RPC calls, and after waiting a predetermined amount of time for remaining server responses, disconnects from the server and checks the invariant on the trace generated by the client-server communication.
3. If `test` returns `0`, the test is successful. Otherwise, the test has failed. Terminate the top-level driver loop and show the user the output generate by the `test` program (reason for failure).

At step 1 above, a new instance of the server is created. The reason for this creation is that the driver cannot easily reset the entire server state—most servers lack an explicit method to do this. In general, we cannot undo the effects of the RPC calls previously made, and the server is likely to end up in a peculiar state after the random RPCs. (A lock server, for example, may have granted all of the locks that another set of tests may request and not receive.) For clarity's sake, each test stands on its own, starting from a fresh server state.

4.2.1 Reproducibility

As we experimented, we observed that some aspects of the testing are beyond our control. In particular, we cannot control the time at which the server receives the RPC calls we specify because we cannot control the network traffic or libasync. The effect of this is such that it is difficult to exactly reproduce the results of a given test.

Even though it is difficult to reproduce a particular test, repeated instances of different tests consistently yield some of the same results. If incorrect behavior is exhibited by a certain test once, a different test is likely to uncover the fault later.

4.3 Invariant Checker

To check an invariant specified in the language shown in Figure 4, we begin by tracking a current position in the Trace object. The pattern matching statements—**match** and **require**—apply only to the current Trace entry. The **any*** statements—**any**, **any_except**, or **any_except_fail**—allow the pattern matching statements to match trace entries beyond the current one. That is, if a pattern matching statement does not match the current trace entry, and an **any*** statement appears before it, the next trace entry is checked. Note that invariants which do not begin with an **any** statement have one implicitly prepended. Note also that before the pattern matching statement is checked, the pattern specified by an **any_except** or **any_except_fail** statement—the **except** pattern—is checked. If an **except** pattern is found to match, the invariant is ruled either invalid (in the case of an **any_except** statement) or the invariant fails (in the case of an **any_except_fail** statement).

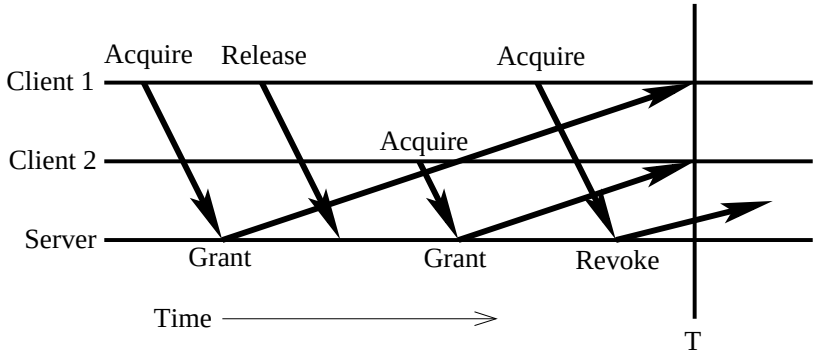

Figure 7: Timing diagram of a bug found in the lock server tested. Note that all RPC calls are performing on the same lock. At time T, both Client 1 and Client 2 have the lock.

In order for an invariant to be valid, all of the **match** statements that appear in it must match some distinct location in the trace. If a rule is valid, each **require** statement must also match or the rule fails. The **require** *Conditional* statement imposes conditions on variables that are bound during the pattern matching process. If any of those conditions are not true, the invariant fails. As well, if any of the **require** *RPC* statements do not match, but all the **match** statements do, the rule fails.

5 Experiments

5.1 Subject Programs

To validate our ideas, we used our tool to test a lock server and a block server. Both servers were implemented to support several programming assignments for a graduate distributed systems course at MIT. The servers were written by the professor.

From the lock server, a client can `REQUEST` a lock, or `RELEASE` a lock that it holds. A server can `GRANT` a lock that a client requested, or it can `REVOKE` a client's currently held lock. Figure 5 shows the trace invariant property that our tool checked: only one `LOCK_GRANT` RPC for a given the same lock should appear before a `LOCK_RELEASE` on that lock.

The block server maintains a mapping of keys to values; a client may `PUT` a value with a specific key onto the server, `GET` a value with a given key back, or `REMOVE` a key from the server. The server does not send any RPCs to the client. Figure 6 shows the invariant we checked for the block server. It verifies that if value is `PUT` on the server at some key, a later `GET` on that key retrieves the same value.

5.2 Setup

We evaluated our tool on each server using a wide variety of parameter values from 2-10 clients, 5-100 RPC calls per client, and 1-10000 tests per run. RPC calls were delivered to the server as quickly as possible. The tester ran successfully for both the lock and block servers.

5.3 Results

The block server passes all the tests. The lock server, on the other hand, violates its invariant. Figure 5 shows the bug that our tester found. Below is a trace from a test which fails the invariant.

```
Final Trace:
```

```
CLIENT-SERVER client_id=0, acquire("lock5") reply: (true)
CLIENT-SERVER client_id=0, release("lock5") reply: (true)
CLIENT-SERVER client_id=1, acquire("lock5") reply: (true)
SERVER-CLIENT client_id=0, grant("lock5") reply: (true)
SERVER-CLIENT client_id=1, grant("lock5") reply: (true)
CLIENT-SERVER client_id=0, acquire(‘‘lock5’’) reply: (true)
SERVER-CLIENT client_id=1, revoke(‘‘lock5’’) reply: (true)
TEST FAILED: invariant did not hold.
```

In this test, clients 0 and 1 both receive a grant from the lock server, resulting in both clients holding the lock, and violating a lock server's crucial guarantee that a client's ownership of a lock is unique.

Unfortunately, this bug cannot be reproduced each time the same RPC calls are made (see Section 4.2.1) —the test above causes no invariant violation on a second try. However, the bug is rediscovered on some similar test case each time we run our tool. On average, our tool catches the bug after 80 tests (average computed over 10 runs of our tool running with two clients).

We located the source of the faulty behavior by running the tool on successively smaller examples—decreasing the number of clients and RPC calls per tests—until we reproduced the fault using two clients and five RPC calls per client. At this point, we looked at the lock server's code to analyze the cause of the fault.

The bug in the lock server turns out to be a subtle timing bug that arises when a client sends an unexpected RPC call to the server. Figure 7 shows a timing diagram of the events that lead to two clients being granted the same lock. Client 1 requests a lock, and the server sends back a `grant` RPC. While the grant is in transit, client 1 releases the same lock, client 2 requests it, and client 1 requests it again. For this sequence of RPC calls, the server will send a `grant` of the same lock to client 2, without checking first that its previous grant of the lock (to client 1) was received and acknowledged. So at time T both clients have the lock.

The bug went undetected through heavy use of the lock server (by dozens of students) because the clients interacting with the server never happened to make this sequence of calls. Moreover, the precise timing of events is crucial, so it would have been difficult to write a manual test that detects the fault. This bug illustrates the potential power of random testing.

There is a simple fix for this bug: before sending a grant of a lock, the server can check that there is no grant in transit for the same lock. This is easy to check, since a client that receives a grant sends back an acknowledgment to the server by replying to the grant RPC call. The server shouldn't send a grant or respond to a release RPC unless its last grant of the same lock has been acknowledged.

6 Conclusion

Random testing is not a widely adopted testing practice. This is due in part to the lack of tools that automate the process. Our solution aims to reduce the work of the tester as much as possible by automating both the generating and evaluation of tests, and by providing a simple, intuitive language in which to express correctness properties.

Our technique has two main limitations. The user of our tool may want to check properties that are not easily (or at all) expressible with our invariant language. In our tool, invariants must be expressed over linear traces of client-server communication, at the level of granularity provided by the RPC interface. The linear nature of the traces, and their level of granularity, may be insufficient for some correctness properties. The second drawback is the lack of control the user has over test generation. The user can only specify the values that are used as arguments of the RPC calls; the specific sequences of calls are beyond their control.

On the other hand, these limitations provide a user with a great advantage: automation. The user need not write any code to generate tests or to test the invariant. If the invariant is expressible using our language very little is required from the user—the tool can produce thousands of tests within seconds. As well, our experiments indicate that our language quite naturally expresses interesting and relevant properties of servers.

A second benefit of our approach is that the user chooses how much or how little to specify. Our tool works just as effectively with a partial specifications as it does with a complete specification. In the extreme, the tool is useful even if no specification is present by checking that the server does not crash under long or bizarre sequences of RPC calls from the clients.

Random testing is a blunt, yet highly effective tool that complements more manual, precise approaches to testing. The key to using random testing effectively is to keep it simple; more constraints on the testing might easily rob a user of its benefits. These benefits come from generating myriads of simple, cheap tests that reveal bugs not easily envisioned through other testing methodologies.

References

[1] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[2] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[3] T. Katayama Z. Furukawa and K. Ushijima. Design and implementation of testcase generation for concurrent programs. In *Proeedings 1998 Asia-Pacific Softw. Eng. Conf. (APSEC '98)*, pages 262–269, 1998.

[4] Dick Hamlet. Random testing. In *Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[5] R. N. Taylor D. L. Levine and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

[6] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of 11th International Conference on Software Engineering (CSE.89)*, pages 44–52, 1989.

[7] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[8] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, 1983.