Unified Debugging of Distributed Systems with Recon

Kyu Hyung Lee Nick Sumner Xiangyu Zhang Patrick Eugster
Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA
{kyuhlee,wnsumner,xyzhang,p}@cs.purdue.edu

***Abstract*—To scale to today's complex distributed software systems, debugging and replaying techniques mostly focus on single facets of software, e.g., local concurrency, distributed messaging, or data representation. This forces developers to tediously combine different technologies such as instruction-level dynamic tracing, event log analysis, or global state reconstruction to gradually explain non-trivial defects.**

This paper proposes Recon, a debugging system that provides iterative and interactive homogeneous debugging services. As related systems, Recon promotes SQL-like queries for debugging distributed systems. Unlike other approaches, however, Recon allows for *all* system artifacts including nodes, communication channels, events, or instructions to be *uniformly* described by relations. Also, an application in Recon originally runs with a *lightweight logger* that only collects replay logs for individual nodes. Developers debug a complete program by replaying the execution with *fine-grained instrumentation* that is capable of exposing instruction-level information.

We illustrate the effectiveness of Recon on programs as diverse as BerkeleyDB, i3/Chord, RandTree, and Pastry. Our evaluation includes executions in local clusters as well as in Amazon EC2 and exhibits an unreported bug in RandTree.

***Keywords*-Software reliability, distributed systems, debugging, replay, instrumentation**

I. Introduction

As computing infrastructures continue to become more powerful and pervasive, software for these systems continues to increase in complexity. As a consequence, debugging and replaying techniques are facing increasing scalability challenges, in particular for distributed systems. A natural response to these challenges is to focus on *subsets* of the equation, that is, to focus on an individual abstraction such as local concurrency, distributed messaging, or data representation. The downside of this approach is that it fails to capture the oftentimes subtle interactions of the different abstractions. Developers are thus commonly forced to manually combine technologies such as instruction level dynamic tracing, event log analysis, or global state reconstruction to gradually explain non-trivial defects that cross the boundaries of individual abstractions.

This paper proposes Recon, a *homogeneous* debugging system that provides iterative and interactive debugging services through a *unified* relational view of pervasive distributed systems. Figure 1 presents an overview of our approach. Originally, the distributed application runs with a

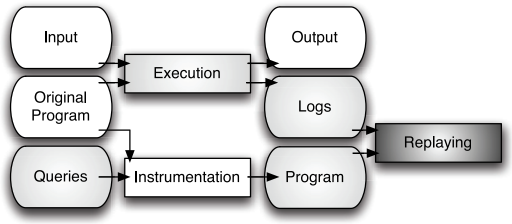


Figure 1: Overview of the Recon approach. The original program executes in a lightweight logger. The replay occurs with heavyweight instrumentation for fine-grained inspection.

lightweight logger that collects minimal logs for deterministic replay of individual nodes. Developers then debug a complete program by *replaying* the execution with heavyweight instrumentation that is capable of exposing instruction-level information. Like related systems (e.g., [25], [18], [11]), Recon supports SQL-like queries for describing system artifacts that are of interest. Unlike these systems, Recon supports the *uniform* description by relations of *all* system artifacts, from the highest level to the lowest level, including nodes, communication channels, messages, event causality and dependencies, method invocations, instructions, and dependencies between instructions; also, Recon does not impose a specific programming language, and supports online analysis as well as replay. Due to resource constraints, most relations are populated in a demand-driven way. In other words, large tables are not explicitly maintained. Rather, queries are compiled into program instrumentation that is executed during replay to answer the query. Developers iteratively refine queries based on previous results. Such a unified view is highly beneficial because the onus is no longer on the developer but on the Recon query system to decide what level of instrumentation to employ to efficiently collect the queried information.

In summary, this paper makes the following contributions:

- Debugging of distributed systems based on *unified views*. Recon enables a unified view that overcomes the heterogeneous requirements for various technologies in debugging distributed systems. The key idea is to formulate all system artifacts, system wide or node local, event level or instruction level, static or temporal, as relations. Debugging is carried out by writing SQL-like queries over any number of such relations.

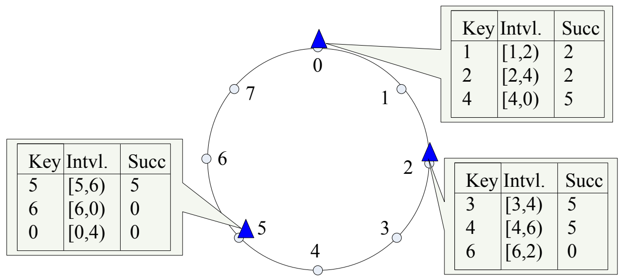
- Compilation of queries to *various instrumentation levels*. Due to resource constraints, most relations are not explicitly maintained by Recon. We describe how Recon's query compiler translates queries into event log filtering components and code instrumentation. Executing these evaluates queries on demand.
- Separation of *lightweight logging* and *fine-grained replay*. Most existing logging and replay techniques are monolithic. That is, they are integrated into one tool that is either lightweight, and hence adequate for production runs but not sufficiently powerful for debugging, or relies on expensive infrastructures such as virtual machines that favor functionality over efficiency. Recon separates logging from replay to avoid expensive instrumentation during original executions. More powerful heavyweight instrumentation is only activated during replay.
- Evaluation of efficiency and effectiveness. We evaluate the efficiency and effectiveness of Reconon a diverse set of distributed programs and their bugs. The results conducted in a local cluster as well as in Amazon Elastic Compute Cloud (EC2) show that our recording overhead is only about 3% and that instrumented replay causes a constant factor slowdown, depending on the queries. 8 bugs, including a previously unreported bug in RandTree, are effectively resolved by querying various levels of information.

Roadmap. This paper presents Recon in a top-down manner: Section II uses real examples to illustrate the use of Recon. Section III presents a design overview of Recon's architecture. Section IV introduces Recon's debugging view. Section V presents the separation of logging from instrumented replay in Recon. Experimental results are presented in Section VI. Section VII discusses related work. Section VIII concludes with final remarks.

II. Overview by Examples

In this section, we present an overview of our technique through examples. The sample bugs are from a Macedon [2] implementation of the Chord [27] distributed hash table protocol. Chord maps a key to a node so that the corresponding data item can be stored and maintained in that node. Chord has been extensively used in pervasive computing environments [4]. The protocol is able to find the host for a key in $O(\log N)$ within N nodes, and to handle nodes joining and leaving at a cost of $O(\log^2 N)$.

The Macedon Chord was found to have a number of bugs in [14]. Some of them cause failures that are local to the node, i.e., they do not manifest directly in the interaction with other nodes. Let us first consider an uninitialized-pointer-dereference bug that crashes the local execution. In the beginning, the observable symptom is merely the crash. The developer may have no idea what caused the crash. It may be an incorrect local assignment or faulty



Key	Intvl.	Succ
1	[1,2)	2
2	[2,4)	2
4	[4,0)	5

Key	Intvl.	Succ
5	[5,6)	5
6	[6,0)	0
0	[0,4)	0

Key	Intvl.	Succ
3	[3,4)	5
4	[4,6)	5
6	[6,2)	0

Figure 2: Triangles represent nodes. Each table denotes a partial image of global key distribution. More particularly, a `key` value is stored in node `succ`; queries for key values in `intvl.` should be relayed to node `succ`.

interaction with other nodes. With our technique, the system is originally running with a lightweight logger. Upon the crash, replay logs are collected from nodes for debugging. The distributed execution is described by a set of *conceptual* relations that contain information such as values defined or used at a given execution point, data and control dependences of a given statement execution, and node interactions. The developer can write queries about such relations as though the relations were fully populated. In the above crash, suppose the failed node n crashed at execution point t when accessing variable d. The simplest way to proceed is to query when d was defined by writing the following query.

```
SELECT definition FROM Data_Dependence
 WHERE use = t AND variable = d AND host = n
```

The query compiler generates instrumentation that collects data dependences. Then the execution of n is replayed with the instrumentation. The query returns an empty table, indicating the pointer was never initialized.

Consider a more subtle Chord-related bug [14]. In the protocol, keys are distributed to nodes and each node has a partial image of the global distribution. In Figure 2, the key domain [0,7] is represented in a ring. Three nodes (triangles) have ids 0, 2, and 5. A key is stored in a node whose id is the key value or the first clockwise node successor of the key value in the ring, e.g., key 6 is stored in node 0. The tables at each node show the partial images. For example, the first row of the table at node 0 says that it knows keys 1, 2, and 4 are stored at nodes 2, 2, and 5, respectively. Furthermore, keys falling in the range of [1,2), [2,4), and [4,0) are relayed to the nodes specified in the `succ` column.

In these tables, a pair of $(\texttt{key}, \texttt{succ})$ is a *perfect pair* if `key=succ`. That means the key is stored in the node with the key value as its id. For example, (5,5), denoting that key 5 is stored at node 5, constitutes a perfect pair. According to the Chord protocol, upon the joining of a new node with id x, if x falls into the range of $(\texttt{key}, \texttt{succ})$, indicating the key has a new successor that is closer to being perfect, the `succ` entry will be updated to x, and x becomes the new host of the key. The bug occurs when a pair is perfect; upon the joining of any node x, the Macedon implementation updates the `succ` entry to x.

In Figure 2, the failure occurs if a new node 6 joins. The table at node 5 is updated by setting the `succ` of key 5 to 6 due to the fault. Assume the request for key 5 is issued at node 2. Node 2 relays the request to node 5 according to its table, which returns 6 as the node that hosts key 5. The attempt from node 2 to access the key at node 6 returns `null`, which is the failure. The programmer observes the `null` return at node 2. A local query performed at node 2 reveals that node 6 was returned as the successor by the previous request sent to node 5. The programmer now can write another query to retrieve, at node 5, who defined the returned value. The answer to that query reveals that the successor was updated to 6 in the method that handles the joining of new nodes. Note that these queries operate at various levels including communication across nodes. How to answer them is completely handled by our technique and hence transparent to the developer.

III. Design Overview

Recon consists of several components, as shown in Figure 1. During normal execution, a light-weight logger records a minimal application log for deterministic replay, containing non-deterministic effects on the execution such as system calls, CPU instructions, or signals. Recon provides an SQL-based interface to the user for debugging operations; a query compiler generates program instrumentation that is executed during replay.

A. Logging and Replay System

One unique feature of Recon is the separation of logging and replay to allow for *fine-grained replay* with necessary heavyweight instrumentation only upon this replay, whilst only requiring *lightweight logging* at runtime. Most existing logging and replay systems are monolithic, collapsing both the recording and replay capabilities. If *low recording overhead* is the target, typical logging and replay architectures (e.g. [9]) change the table for system calls (*syscalls*) or rewriting instructions around syscalls in the executable to redirect them to wrappers. The perturbations imposed on the application execution are usually very limited. During replay, the same interception is performed to read values from the log file without executing the actual syscalls. *Jockey* [24] is an example of such a system. Jockey provides low-overhead logging and deterministic replaying for interactive or distributed programs. Jockey records invocations of non-deterministic system calls and CPU instructions for deterministic replay.

Alas, such OS-level interception is not able to expose fine-grained execution information such as values and dependences of individual instructions necessary for debugging distributed systems. The necessary information is obtained with *heavyweight* instruction-level instrumentation through dynamic binary instrumentation tools like *Pin* [17] or *Valgrind* [20], but at a high cost in terms of runtime overhead. For example, [21] shows the overhead of Pin is up to 60% for intercepting syscalls. An alternative is to realize the logging and replay functionality in a full-fledged virtual machine that allows intercepting both fine-grained instruction execution and coarse-grained system calls. However, this requires that applications run on the virtual machine.

Recon, in short, uses the Jockey recording/replaying tool as a logger at recording time, yet we implemented a replaying tool which is combined with a dynamic instrumentation tool in Pin. Our replay tools also provide guarantees on deterministic replay of individual nodes by trapping system calls and CPU instructions of application and retrieving events from the log file without actually passing syscall requests on to the OS. We will discuss our implementation in Section V in detail.

B. User Interface

Recon provides a declarative interface based on SQL to the developer. Various levels of system artifacts are encoded uniformly into relations. Debugging operations can be composed as queries on one or more such relations. For example, if the developer is concerned about the local instruction-level status and global network status, he/she can easily write a join query of a local table and a global table on conditions about time, etc. Relations are solely conceptual, i.e., they are not explicitly populated and stored. Instead the query compiler translates queries into binary instrumentation at selected program points to evaluate queries online during replay. Note that queries can vary between replays, thus supporting iterative debugging. In the past, SQL interfaces have been proposed for logging and profiling related applications, such as in [25], [16], [11], [18]. Existing work mostly focuses on querying event logs. In other words, an event log is collected beforehand and serves as a database; events are often at a coarse level, otherwise the space consumption would be prohibitive. Some [11], [18] feature online query evaluation. However, they do not support replay and their schema designs are not debugging oriented. For example, they do not support data and control dependence. In contrast, under the hood of a SQL interface, we support iterative and interactive debugging through replay and expose critical execution artifacts through various relations.

The query language supports standard relational operations including *selection*, *projection*, *join*, *anti join*, and *aggregation*, i.e., the **`GROUPBY`** operation. The aggregation operation can be useful in finding anomalous behavior. The semantics of the query language also largely follow the standard SQL semantics. The precise information that queries may work with and how the queries can be effectively answered are discussed in the following sections.

State (ST)		
Field	Type	Description
`host`	**`INT`**	host id
`location`	**`EXE_PNT`**	exec. point
`variable`	**`STRING`**	variable
`value`	**`BYTE[]`**	untyped value

Control_Dependence (CD)		
Field	Type	Description
`host`	**`INT`**	host id
`branch`	**`EXE_PNT`**	branch point
`location`	**`EXE_PNT`**	current exec. point

Data_Dependence (DD)		
Field	Type	Description
`host`	**`INT`**	host id
`definition`	**`EXE_PNT`**	definition point
`use`	**`EXE_PNT`**	use point
`variable`	**`STRING`**	variable of dependence

Output (OUT)		
Field	Type	Description
`host`	**`INT`**	host id
`location`	**`EXE_PNT`**	output point
`value`	**`BYTE[]`**	output value

Control_Flow (CF)		
Field	Type	Description
`host`	**`INT`**	host id
`location`	**`EXE_PNT`**	instruction point
`pc`	**`INT`**	program counter
`instance`	**`INT`**	instance
`file`	**`STRING`**	source file
`line`	**`INT`**	line number

Communication (COM)		
Field	Type	Description
`sender`	**`INT`**	sender id
`send_point`	**`EXE_PNT`**	send point
`receiver`	**`INT`**	receiver id
`recv_point`	**`EXE_PNT`**	receive point
`message`	**`BYTE[]`**	message

Figure 3: Schema for system properties that may be queried.

IV. SQL INTERFACE

Recon's logging system collects minimal information for deterministic replay *at runtime*. Queries are evaluated by executing corresponding instrumentation generated by the query compiler *at replay*. This section describes schemas, instrumentation generation, and our query compiler.

A. Schemas

A set of relations that describe common aspects of distributed systems is predefined. These relations include **State**, **Data_Dependence**, **Control_Dependence**, **Control_Flow**, **Output**, and **Communication**. The detailed schemas are shown in Figure 3. In particular, type **`EXE_PNT`** is used to describe an execution point. Intuitively, one can interpret it as a triple (`source file`, `line number`, `instance`) that pinpoints a specific instance of a source code statement[1]. In our implementation, we use a more precise representation, which will be discussed in the next section. Variables are represented by their symbolic names.

Relation **State** records variable information observed within the system. It describes variable values on any host at any point over the duration of the system's execution. Recall that the relation is conceptual and only populated on demand, driven by queries. For example, if a variable value is queried at a specific location l for a node n, only the execution of n is replayed, and only the variable value at l is reported. Variables are not necessarily those that are defined or used at a `location` but rather those that are live.

The **Data_Dependence** (**DD**) relation describes which definition (assignment) of a variable is used at a given execution point. For example, executing the code snippet in Figure 4(a) on node 0 conceptually inserts a tuple (`host`=0, `definition`=(`f1.c`, 1, 1), `use`=(`f1.c`, 2, 1), `variable`=`x`) in the table, meaning that the use of variable `x` at the first instance of statement 2 is defined at the first instance of 1 in file `f1.c`.

Similarly, **Control_Dependence** (**CD**) describes the `branch` point that directly determines whether or not an instruction at `location` on `host` is executed. Assume an execution of the code snippet in Figure 4(b) takes the true branch and the host is 0. The **CD** table conceptually contains two tuples (`host`=0, `branch`=(`f2.c`, 1, 1), `location`=(`f2.c`, 2, 1)) and (`host`=0, `branch`=(`f2.c`, 1, 1), `location`=(`f2.c`, 3, 1)). Relation **Control_Flow** (**CF**) associates each execution point with a binary program counter, execution instance, and symbolic information.

[1]A statement can get executed multiple times, leading to many instances.

f1.c

```
1. x = ...;
2. y = x+1;
```

(a) Data dependence

f2.c

```
1. if (P) {
2.   s1;
3.   s2;
4. }
5. s3;
```

(b) Control dependence.

Figure 4: Data\control dependence examples.

Besides the above relations describing fine-grained information, we also have relations for coarser-grained information. The output of a host is described by the **Output** (**OUT**) relation. The interaction between hosts is captured by the **Communication** (**COM**) relation. This relation observes all messages sent between hosts along with the hosts involved and when they sent or received the message in question.

B. Instrumentation Primitives

We now discuss how we leverage instrumentation to populate the aforementioned relations. The population is controlled by our query compiler.

Representing Execution Point. As discussed earlier, an execution point can be conceptually considered a triple of file name, source code line, and execution instance. However, this is not sufficiently accurate because it is common that a source code line may be composed of multiple statements and even different parts of a statement may lead to different instructions. For example, if a predicate has a conjunctive condition, the different clauses may have different execution instances at runtime. In Recon, we use the instruction count as the internal representation of an execution point (**`EXE_PNT`**), i.e., the number of executed instructions up to the point in the same process. This is because replay is deterministic (we will discuss how we handle concurrency in Section V-A) and thus the same execution point always has the same instruction count. While instruction count is not a user-friendly representation, the symbolic information can be easily acquired by joining a query with the **CF** relation. For example, the following query identifies the source code location that outputs a string "Hello".

```
SELECT file, line, instance FROM CF
  INNER JOIN OUT
    ON CF.location = OUT.location
WHERE OUT.value="Hello"
```

Resolving Variables. For usability, we allow developers to specify variables in their queries using symbolic names. Because the instrumentation to answer queries has to be at the binary level, we have to resolve variables to their addresses. We leverage a tool called *dwarfdump* [1] to establish the mappings between symbolic names and addresses. For global variables, the tool can directly yield their addresses. The offset of a local variable in the stack frame of its enclosing function can also be identified by the same tool. At runtime — combined with the stack base address which can be acquired from the binary instrumentation tool Pin — Recon can compute the stack addresses of local variables. For heap variables, their reference paths are followed to identify their addresses. Note that it must be true that the reference path of a heap variable starts with a global or stack variable, whose address we have already resolved.

Primitives for Relations. To answer a query on the **State** relation, we first replay the specified host up to the requested execution point; then we resolve the addresses of the specified variables and retrieve their values. For the **OUT** and **COM** relations, as both output emission and message sending have to go through system calls, queries can be answered by simple processing of the replay log. Queries of the **CF** relation are straightforward to answer as the Pin infrastructure allows identifying the precise PC (program counter) of each instruction and its symbolic information.

The remaining two relations (**DD** and **CD**) require more effort. Consider the **DD** relation first. We detect data dependences through a data structure called *shadow memory* – a small piece of memory allocated inside Pin for each program variable. This memory can be considered the shadow of the respective variable. Upon the definition of the variable, the `location` that defines the variable is recorded in its shadow memory. Later, when the variable is used, a dependence is detected. Let $SM(x)$ be the shadow memory of x. Consider the example in Figure 4(a). When line 1 is executed, $SM(x)$ is set to 1. When line 2 is executed, since x is used, a dependence is detected between 2 and $SM(x) \equiv 1$ on variable x. We developed an instrumentation module for Pin to carry out the detection.

Control dependences in the **CD** table are detected as follows. Immediately after a branch point, the linear control flow from that point is dependent upon the branch point because whether or not an instruction is executed is directly determined by the branch. At the postdominator [8] of the branch, i.e, the join point of the branch, execution is no longer determined by the branch point. This implies that control dependence is organized into nested regions much like function calls, and similarly, control dependence can be efficiently observed throughout an execution by pushing a branch point onto a stack called the *control dependence stack*, and popping once its postdominator is encountered [28]. Thus, in the same way that the function call on the top of the call stack is the currently executed function, the control dependence for the current execution point is observed at the top of the control dependence stack. By maintaining this stack during replay with Pin, control dependences can be detected on the fly. Consider the example in Figure 4(b). Upon executing the branch point at line 1, line 1 is pushed onto the control dependence stack. The following executions of lines 2 and 3 are control dependent upon the top entry on the stack, which is line 1. The entry is popped at the execution of line 5, the postdominator of 1, indicating that the following executions are no longer decided by the branch at 1.

C. Query Compiler

Above, we discussed how to detect information defined in the relations in Figure 3 through instrumentation primitives. However, we cannot afford first populating the relations and then answering queries. We rely on the *query compiler* to collect as little information as possible to answer queries. The compiler takes a query provided by the user, performs instrumented replay of the relevant hosts using Pin to collect the specific information needed to answer the query, and then performs the post-processing or aggregation necessary to present the results in the format requested by the user.

The compiler front end is a query parser. Because we support queries with a grammar structured like SQL, we leverage an existing SQL parser [3] to build an abstract syntax tree representing the query structure. Answering a single query then involves replaying precisely the desired subset of hosts with instrumentation that collects the information requested in the query. Determining which relations should be populated can be done by looking at the relations referenced in the query itself. The corresponding instrumentation primitives are enabled through command line options of Pin. A primitive can be easily attached to (or detached from) Pin by providing (removing) the corresponding instrumentation module name via the command line.

To avoid collecting too much information, it is necessary to control the elements or rows of the relations to be populated. That is, we never want to return results for the entire duration of any given host's execution, e.g., answering a query like "**SELECT** * **FROM** CF **WHERE** host=0", as it entails recording the entire control flow trace of a process. Hence, we only want to observe the relations at specific times or intervals. This is ensured through two different aspects of the system: **WHERE** clause filtering and **join** operations. First, let us consider simple queries over a single relation. In this case, the filters over which elements should be recorded are explicitly provided by the user in the form of the **WHERE** clause. Note that we require the **WHERE** clause to restrict at least one more field other than the host id when querying the **State**, **DD**, or **CD** relations because these relations reflect execution properties that may be viewed as continuous over the entire execution, yielding an intractable quantity of information.

When performing a join operation, or more specifically an inner join, results from a query over one relation are combined with results from another relation based on some common attributes. For example,

```
SELECT use FROM DD
  INNER JOIN COM
    ON DD.definition = COM.recv_point
```

is a query that collects the execution points for all uses of variables defined at the reception of a message. Conceptually, it requires retrieving the **DD** and **COM** relations and then finding pairs of results that satisfy the join predicate as specified by **ON**. In practice, however, implementing the query this way would be intractable, requiring not only the explicit capture of both already large relations, but also some filtering over their Cartesian product. To deal with this, we pipeline the data collection of the relations used within a join operation, such that the concrete results from one relation are explicitly used in the collection of another. For example, in the above join operation, the query over **COM**, returning all recv_points observed in the system, can be executed immediately. If that query is executed first, and the results from that query are used as the basis for restricting the query over **DD**, then finding a solution to the join can become practical because the number of communication events is expected to be tractable in practice.

Precedence	Relations
(High) 1	**Output**
2	**Communication**
3	**CD**, **DD**, **CF**
(Low) 4	**State**

Table I: Relation precedence in joins.

To generalize the above observation, we thus roughly use the expected approximate size of the implicit relations to prioritize which information should be collected first. The priorities are given in Table I. For example, output is expected to be a relatively rare occurrence on any particular node in the system, so evaluating queries on the **Output** relation is often directly feasible. Thus, it has the highest priority, and the results from an output query will be collected first in a join operation. Communication between nodes is not a rare event, but it does not happen at every instruction across all hosts, so **COM** has the next highest priority. The **CD**, **DD**, and **CF** relations all contain a bounded number elements at every instruction executed within the system, so they are all larger relations than **COM** and should only be evaluated afterward in order to winnow their results. Finally, **State** has a theoretically unbounded number of elements for every instruction executed on every host, so it clearly has the lowest priority when joining. In the event that the size of a result set still exceeds a maximum limit, we simply stop data collection and return the partial result set. The user can then refine the query further.

V. Replay with Instrumentation

This section describes the separation of logging from replay in Recon and the implementation of its replay system.

Low overhead is important during logging execution. In contrast, replay is often conducted during the debugging phase, in which overhead is not a major concern and more functionality is highly desirable. Hence, we separate Recon into two subsystems. One is the logging tool that is very lightweight for normal runs; the other is the replayer built on a powerful dynamic instrumentation infrastructure.

Achieving such a separation is, however, far from trivial. Ideally, we could have combined existing industrial strength tools such as Jockey to record an execution and Pin for dynamic instrumentation. Unfortunately, the nature and the complexity of the tools don't allow this to happen straightforwardly as we explain in the following.

Jockey runs in user space. When an application is run on Jockey, Jockey calls an initialization method before the application gains control, in which Jockey scans through the binary, including the libraries loaded by the application, looking for any syscall sites. Those syscalls are redirected to Jockey by overwriting the instructions at the syscall sites.

In contrast, the dynamic instrumentation engine Pin takes a binary. During execution, before executing any new (never instrumented) code regions, it calls a provided instrumenting function. This function instruments the given code regions and returns a new code region that contains both the original semantics and the instrumentation semantics that expose execution details. Pin executes the instrumented code instead of the original code. The instrumented code region is also copied to a new code space and thus it can be reused without calling the instrumentor again during the same execution. Note that such instrumentation is dynamic, meaning that it occurs during execution of the program.

Simple aggregation of the two tools can occur in two ways: one is *Jockey+Pin+application*, meaning that both Pin and the application run on top of Jockey; the other is *Pin+Jockey+application*. Alas, neither works. In the former case, Jockey gains control first and the subject program of

Jockey is Pin instead of the application, which is itself the subject of Pin. Recall that Pin instruments the application on the fly during execution, and only the instrumented version gets executed. The instrumented version does not exist at the beginning, and hence its syscalls are not visible to Jockey. In fact, Jockey will search for syscalls in Pin and patch them. Thus, during Pin execution, all those syscalls will be undesirably redirected to Jockey.

Inversely, if Jockey and the application run on top of Pin, Jockey first intercepts all the syscalls of the application. The patches by Jockey and then the logging/replay code become part of the application. When the patched application executes inside Pin, Pin will try to instrument the patches and the Jockey code along with the application code. Such instrumentation is not only undesirable but often destructive because Jockey patches and code are at a very low level. Also, both tools reserve an overlapping virtual space region for their own purposes, which causes intricate conflicts. Finally, the threading models of the tools are incompatible. As a result, even after solving the compatibility problem, the integration would still fail on threading programs.

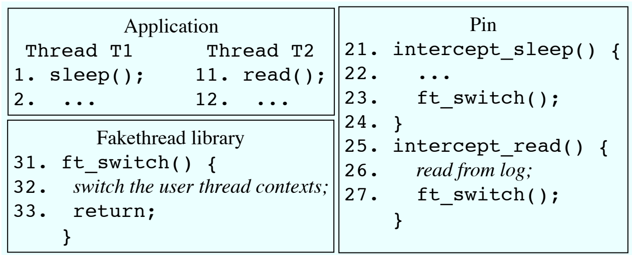
Our solution is to avoid using Jockey for replay. Instead, we implement the replay functionality inside Pin. The insight is that since Pin can potentially trap each instruction in the application, we can easily trap all the syscalls in the application. In particular, we implement an event log parser that can parse the logs generated by Jockey. Then we trap all the syscalls of the application inside Pin. Upon the execution of a syscall, we retrieve the corresponding event from the log through the parser. Note that Pin allows us to compose and attach multiple instrumentation modules exposing various aspects of application execution. We still benefit from the low logging overhead by using Jockey to generate logs.

A. Handling Threading

Jockey supports multithreaded applications using its emulation library called `fakethread` to allow maximal control over recording thread execution. Fakethread support is composed as a library that can be linked with threaded applications, providing the same interface as `pthreads`. It is completely inside user space and threads are opaque to the kernel, which is different from pthreads. Fakethread allows one thread to execute at a time. A thread is never preempted. Instead, it runs continuously until a blocking syscall, such as a blocking read or write, or until it fails to acquire a lock. In such cases, fakethread performs a (user space) context switch to the next ready thread. Jockey records all thread creation, join, termination, and context switch events.

In contrast, Pin's threading model is based on `pthreads`. It relies on the kernel to manage threads. In particular, the existence of a user thread is relayed to Pin by the kernel so that Pin can allocate space dedicated to the thread. Such space maintains thread local information related to analysis modules attached to Pin, e.g., a stack inside Pin for the thread in addition to the stack in user space. Pin does not interfere with thread scheduling. Context switches and synchronizations are managed by pthreads independently.

The differences between the two threading models make replaying Jockey logs inside Pin challenging. The problem stems from Pin not being aware of threads created through the user level fakethreads. As a result, thread specific space is not allocated inside Pin, leading to various problems.



```
 Thread T1        Thread T2
1. sleep();     11. read();
2.  ...         12.  ...
```

```
31. ft_switch() {
32.   switch the user thread contexts;
33.   return;
    }
```

```
21. intercept_sleep() {
22.    ...
23.    ft_switch();
24. }
25. intercept_read() {
26.    read from log;
27.    ft_switch();
    }
```

Figure 5: Example for fakethreads in Pin.

B. Achieving Separation in Recon

Consider an example in Figure 5, in which fakethreads are used. The application has two threads, **T1** and **T2**. Their respective calls to `sleep()` and `read()` lead to blocking syscalls. The **Pin** box shows the functions (inside Pin) that intercept the two syscalls, which in turn call the fakethread context switch function. The **Fakethread** box shows the context switch function inside the fakethread library. Upon a syscall from the application, Pin traps the call and dispatches the call to the right handler inside Pin, e.g., `intercept_read()` for the `read()` call in **T2**. The handler first reads the event from the log file. If the syscall is blocking, the fakethread context switch is called.

Figure 6 shows a sample execution that executes **T1** to the blocking syscall `sleep()` at line 1, then executes **T2** to the blocking call `read()` at line 11, and finally switches back to **T1**, ending with a segmentation fault. The reason is the invisibility of fakethreads inside Pin. The stacks shown beneath the trace inside Figure 6 explain the problem. Figure 6(a) shows the stacks after line 32 and before the context switch. Upon the switch, the user stack is changed from **T1** to **T2**. The problem lies in the trap of `read()` by Pin. Since Pin does not know that there are two threads, it uses the same internal stack such that the activation record of the `intercept_read()` invocation is pushed on top of that of `intercept_sleep()` (from **T1**), as shown in (b). As a result, in (c), when the fakethread scheduler switches back to **T1**, the user stack is correctly recovered but not the Pin stack. In particular, the return at line 33 would direct the control flow to `intercept_read()` while it should return to `intercept_sleep()`.

Our solution is to use real pthreads during replay so that Pin is aware of the user threads. In particular, we create pthreads during replay at places where the log indicates that fakethreads were created during the original run; pthread

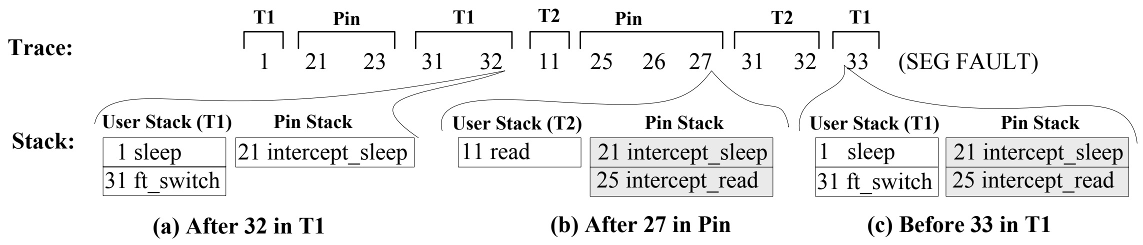


Figure 6: Fakethreads are problematic. The highlighted Pin stacks cause the problem.

joins and exits also strictly follow the log. The challenge lies in forcing the pthread scheduler to respect the schedule specified by the fakethread log. More specifically, fakethreads are not preemptive, but pthreads' default scheduling policy is preemptive, e.g., a thread's time slice may expire, such that undesirable context switches may occur. Furthermore, the pthread scheduler is external to Pin; thus forcing a context switch between two specific threads is not straightforward.

We use a global pthread lock to enforce the recorded schedule. More specifically, a thread has to own the global lock to start or resume its execution. A thread yields the lock only when the fakethread log indicates a context switch. Hence, a pthread originated context switch during replay does not lead to the execution of a different thread if the log does not say so.

VI. Evaluation

To establish the practicality of Recon, we examined the runtime overhead it incurs for logging and replay. We also evaluated Recon on several real world bugs, including one unreported bug, from four distributed applications.

A. *Efficiency*

We first examined the runtime overhead incurred for both the logging that occurs in original runs and for the instrumented replays in the debugging phase. Overhead was measured on a set of four distributed programs including BerkeleyDB, i3/Chord, Pastry, and RandTree. BerkeleyDB is a popular open-source database engine that provides data replication capability to enable a group of processes to service the workload. The Pastry and RandTree implementations are those from the Mace infrastructure [12]. All tests were performed on an Intel dual core 1.66GHz machine with 3GB memory. Individual nodes in the network were executed in independent virtual machines, each running Linux 2.6.11. We also measured the logging overhead of BerkeleyDB on a real distributed environment, Amazon EC2. Each node had 2.66GHz CPU with 1.7GB memory.

Let us first consider the runtime overhead during logging and uninstrumented replay (on Pin), as presented in Figure 7. Here we deploy 4 nodes for all benchmarks. We use the insertion of 10,000 key/value pairs as the input to BerkeleyDB. For I3/Chord, we use the test suite provided by I3. We use the `appmacedon` tool, included in the Mace

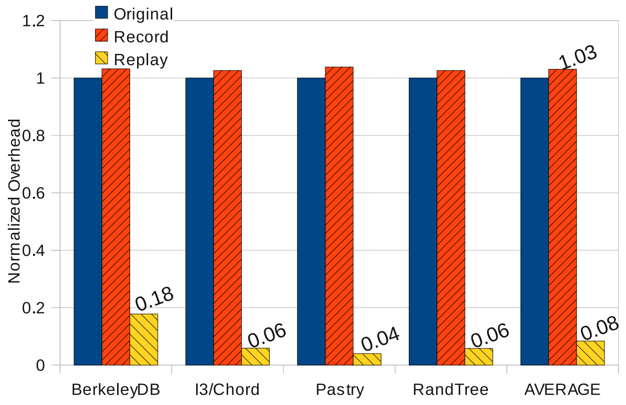


Figure 7: Logging/replay overhead.

release, as a driver for multicast and randomly generated streaming data to measure overhead of RandTree and Pastry. All times are normalized against the original execution time. *Original* marks the unmodified runtime of the original program. *Record* denotes the logging time. It was measured from the starting time of the first action of the system to the finishing time of the last action. *Replay* denotes the most significant overhead for replaying a *single node* in the system. Replay time is normalized against the original time for that same node. Note that our system allows multiple node replay, controlled by the query compiler.

Observe first that the logging overhead is both consistent and tolerable. The maximum overhead is only 4%, for Pastry, and all other cases have 3% overhead, which is tolerable for realistic runs. For replay, the overhead results appear surprising at first. Indeed, the time taken for replay is significantly *less than the original run* and only 8% of the original run on average. This results from the time saved by emulating all system calls during replay. When no waiting time is incurred during replayed system calls, the overall execution is, in fact, much faster than the original.

We also vary the size of our deployment to observe overhead changes for BerkeleyDB. We use the same input and vary the number of nodes from 1 to 8. We executed the same set of experiments on both the local machine and Amazon EC2 cluster. Note that the same set of data is replicated to all nodes in the system. The results are presented in Table II. Observe that the logging overhead is high for one node but less for other settings. The reason is that the communication delay masks some overhead when

# of nodes	exec. time (s)		logging overhead		log size
	Local	EC2	Local	EC2	
1	180.5	25.58	7.19%	2.88%	6.79 MB
2	414.4	396.33	3.14%	0.54%	9.13 MB
4	512.0	212.74	3.19%	1.11%	12.28 MB
8	594.7	313.4	3.04%	1.05%	17.07 MB

Table II: Logging costs vs. # Nodes for BerkeleyDB.

multiple nodes are involved. The execution time with 2 nodes is a lot slower than the 1 node execution because the data has to be replicated to the second node. When the node count goes higher, the overhead does not increase as substantially because the replication is done through broadcasting. In the EC2 cluster, execution time with 2 nodes is slower than the 4 node execution because the master node must wait until receiving at least one acknowledgment message from clients for each input data. When we have 2 nodes, one master and one client, the only client needs to store all replicas and sends ack messages to the master, but in the 4 node case, any available client can send ack to the master. When we have 8 nodes, the master node needs at least 3 ack messages from clients, so the execution time is slower than with 4 nodes but still faster than the 2 node case. Local machine execution does not show this symptom because increasing the number of nodes causes CPU and memory congestion. The log file sizes are also presented in the last column.

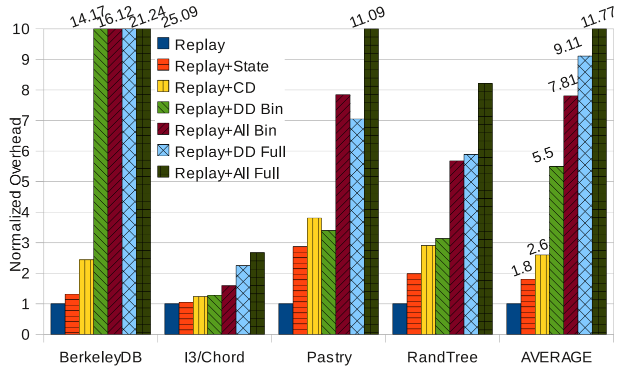


Figure 8: Instrumentation overhead.

The instrumentation overhead for collecting several different execution artifacts is presented in Figure 8. Once again, all runtimes are normalized, this time against the replay time for each program. *Replay* presents the base replay time, i.e., replay inside Pin without instrumentation. *Replay+State* gives the time necessary to locate a particular point in the replay of a node such that the requested state can be extracted with the help of `dwarfdump`. Note that populating the **Output** and **Communication** relations demands the same online instrumentation and hence the same overhead applies to those primitives as well. *Replay+CD* gives the time necessary for detecting the control dependence for each execution point within a node. *Replay+DD Bin* presents the time for detecting the data dependence for each execution point in the application binary while *Replay+DD Full* covers both the application binary and libraries. We separate these two options because instrumenting libraries may not be necessary for all programs. BerkeleyDB builds some of its own libraries as well, so those are included in its *Bin* results. *Replay+All Bin* presents the overhead of all instrumentation primitives for the application binary while *Replay+All Full* does so for both the binary and the libraries.

In all cases, tracking the execution position to collect state information takes less than 3 times the replay runtime, and in most cases, it takes less than 2× as long to execute, 1.8× on average. Tracking control dependence alone usually has less than 3× overhead with respect to the base replay runtime, 2.6× on average. For most benchmarks, data dependence is the most expensive execution artifact to detect, with full data dependence detection, including through library calls, taking an average of 9.11× to execute. Note that the overhead from BerkeleyDB is the dominant contributor to this average. The BerkeleyDB tests involve significantly more data than those for the other programs, resulting in a higher number of data dependencies being detected. Sometimes, it may be beneficial to exclude data dependencies that occur inside external library calls. When we do this, tracking data dependence can be done with 5.5× the base runtime on average. Having all primitives activated incurs more slowdown. Overall, considering that replay is much faster than the original run (0.08×), the instrumented runs are even faster or comparable to the original runs. Such overhead is acceptable in the debugging phase.

B. *Effectiveness*

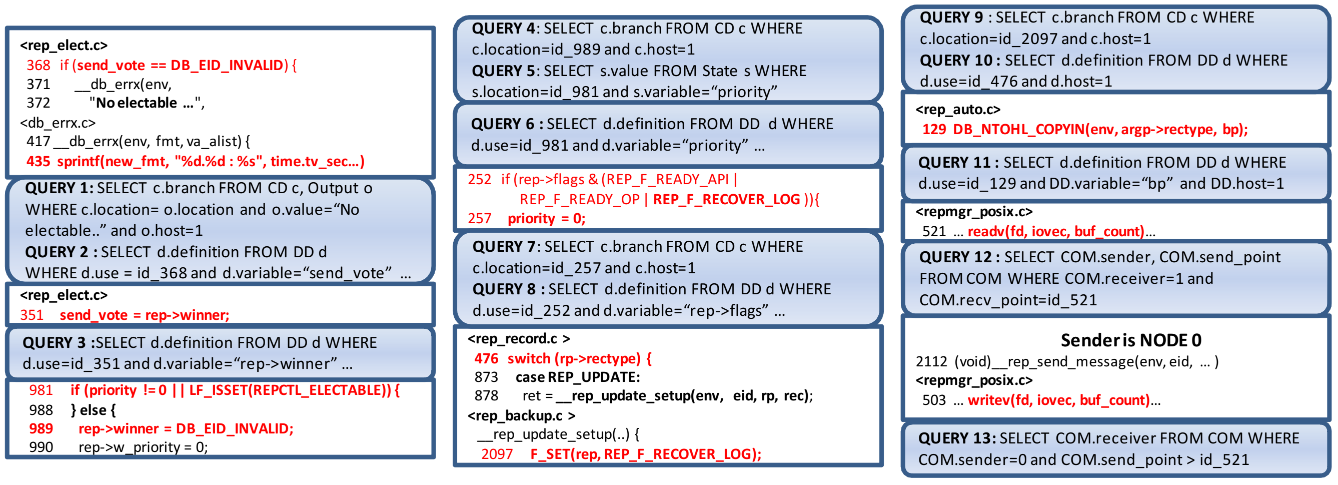
We tested our tool with eight bugs from four distributed applications. One of them (RandTree #1) is an unreported bug and we can easily generate queries to find a root cause. We will discuss details in the Case Study #2. Figure 9 presents the total number of queries generated for debugging, and also the frequency with which they involved each relation from Figure 3. The right-most column represents the number of nodes we needed to replay in order to find the cause of each bug. From the table, we can observe that all these real bugs can be resolved in a few queries with 14 being the maximum, showing the effectiveness of our technique. Queries at different levels, from variable state to node interactions, are needed, and most of these bugs require reasoning across node boundaries, illustrating the necessity of a unified debugging view. Next, we will present a few case studies.

Case Study #1: The first case is a failure of a leader election scheme in BerkeleyDB 4.7.25 that supports the single master/multiple clients model. This bug was originally analyzed in [32].

When a master crashes, all remaining nodes in the system automatically start a new round of leader election. However, in the case of this bug, the system permanently fails to elect a new master because all nodes in the system believe

Bug	Summary	Total Queries	STATE	CD	DD	OUTPUT	CF	COM	# of nodes
DB#1	Permanent election failure [32]	13	1	4	6	1	0	2	2
DB#2	Master node panic [32]	14	1	3	8	1	0	1	2
Chord#1	Packet handling failure [9]	7	3	2	3	1	0	0	1
Chord#2	Inconsistent ring [10]	7	0	2	1	1	2	2	2
RandTree#1	Permanent join fail	9	0	2	4	1	0	3	3
RandTree#2	Disjoint tree [10]	7	0	3	2	1	1	1	2
RandTree#3	Fail to find peer [13]	5	1	1	3	1	0	1	2
Pastry#1	Buffer overflow [13]	5	1	2	1	1	0	1	2

Figure 9: Analyzed bugs with the number of queries used and relation usage distributions.



```
<rep_elect.c>
 368  if (send_vote == DB_EID_INVALID) {
 371      __db_errx(env,
 372         "No electable ...",
<db_errx.c>
 417 __db_errx(env, fmt, va_alist) {
 435  sprintf(new_fmt, "%d.%d : %s", time.tv_sec...)
```

QUERY 1: SELECT c.branch FROM CD c, Output o WHERE c.location= o.location and o.value="No electable.." and o.host=1
QUERY 2 : SELECT d.definition FROM DD d WHERE d.use = id_368 and d.variable="send_vote" ...

```
<rep_elect.c>
351   send_vote = rep->winner;
```

QUERY 3 :SELECT d.definition FROM DD d WHERE d.use=id_351 and d.variable="rep->winner" ...

```
 981    if (priority != 0 || LF_ISSET(REPCTL_ELECTABLE)) {
 988    } else {
 989      rep->winner = DB_EID_INVALID;
 990      rep->w_priority = 0;
```

QUERY 4: SELECT c.branch FROM CD c WHERE c.location=id_989 and c.host=1
QUERY 5: SELECT s.value FROM State s WHERE s.location=id_981 and s.variable="priority"

QUERY 6 : SELECT d.definition FROM DD d WHERE d.use=id_981 and d.variable="priority" ...

```
252  if (rep->flags & (REP_F_READY_API |
          REP_F_READY_OP | REP_F_RECOVER_LOG )){
257    priority = 0;
```

QUERY 7: SELECT c.branch FROM CD c WHERE c.location=id_257 and c.host=1
QUERY 8 : SELECT d.definition FROM DD d WHERE d.use=id_252 and d.variable="rep->flags" ...

```
<rep_record.c >
 476  switch (rp->rectype) {
 873    case REP_UPDATE:
 878      ret = __rep_update_setup(env,  eid, rp, rec);
<rep_backup.c >
  __rep_update_setup(..) {
  2097    F_SET(rep, REP_F_RECOVER_LOG);
```

QUERY 9 : SELECT c.branch FROM CD c WHERE c.location=id_2097 and c.host=1
QUERY 10 : SELECT d.definition FROM DD d WHERE d.use=id_476 and d.host=1

```
<rep_auto.c>
 129  DB_NTOHL_COPYIN(env, argp->rectype, bp);
```

QUERY 11 : SELECT d.definition FROM DD d WHERE d.use=id_129 and DD.variable="bp" and DD.host=1

```
<repmgr_posix.c>
 521  ... readv(fd, iovec, buf_count)...
```

QUERY 12 : SELECT COM.sender, COM.send_point FROM COM WHERE COM.receiver=1 and COM.recv_point=id_521

Sender is NODE 0

```
2112  (void)__rep_send_message(env, eid,  ... )
<repmgr_posix.c>
 503  ... writev(fd, iovec, buf_count)...
```

QUERY 13: SELECT COM.receiver FROM COM WHERE COM.sender=0 and COM.send_point > id_521

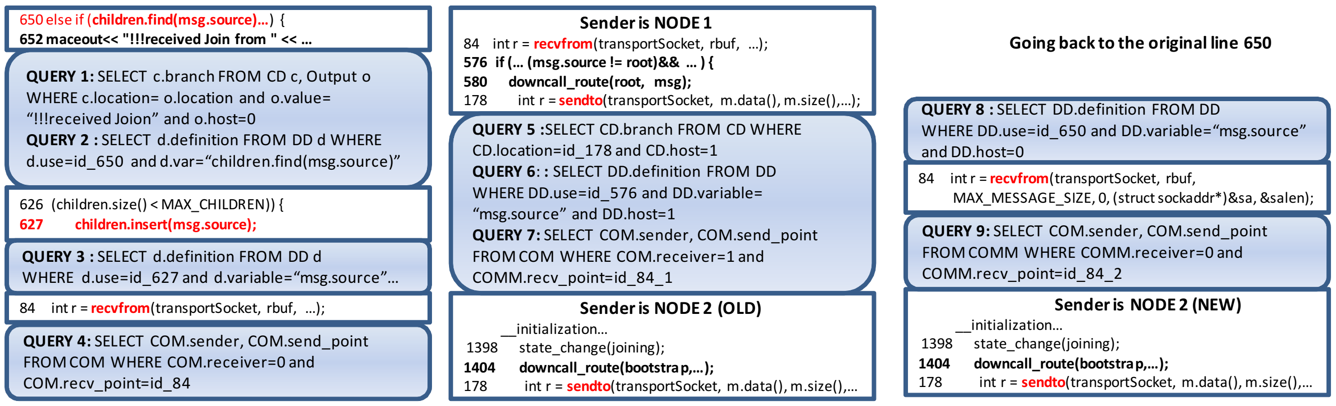
Figure 10: Case study 1: BerkeleyDB - failure in election.

they cannot be the master. Suppose node 0 is the original master node and is synchronizing data with all other client nodes, nodes 1 and 2. The steps of synchronization are as follows. First, when node 0 receives modified data, it sends a `REP_PAGE` message with the modified data to all clients. All clients who receive this message change their state to `LOG_RECOVERY` by setting a flag to `REP_F_RECOVERY_LOG`, meaning they expect the corresponding log update in the next step. When this flag is set, the node cannot be a master. The clients send a `REP_LOG_REQ` message to the master node, requesting the updated log. In the next step, when node 0 receives the `REP_LOG_REQ` message, it sends a `REP_LOG` message with the updated log records. When clients receive the `REP_LOG` message from the master, they restore to the normal state. The bug happens when node 0 is down after it sends a `REP_PAGE` message but before sending the `REP_LOG` message. When the master is down, all clients start election but permanently fail to elect a new master. The symptom is an error message on all client nodes – "`No electable site found : ...`"

We generated 13 queries. Figure 10 presents the code snippets related to the failure appended with the corresponding queries. We started debugging with node 1, which is one of the client nodes that printed the error message. From QUERY1 (Q1), we can obtain the execution point where the message was printed, line 435, and we can also reach the branch point controlling the execution there, which is line 368. From the first query, we find that this error happens because variable `send_vote` has the value `DB_EID_INVALID`. Now we want to know where the value of `send_vote` was defined. In Q2, we query the data dependence of `send_vote` at line 368. Here, we use `id_368` to denote the execution point of line 368 for readability. Q2 returns the execution point of line 351, and now we need to know where the variable `rep→winner` was defined. From Q3, we reach line 989, where `rep→winner` was given the constant value `DB_EID_INVALID`. Now we need to obtain the control dependence of the execution point at line 989. Q4 shows that line 981 controls line 989. In order to decide which condition caused the wrong branch outcome, we query the value of `priority` at 981 (Q5), which reveals that it has an undesirable value of 0. Through Q6 and Q7, we reach line 252, the branch controlling the definition of `priority` to 0 at line 257.

From queries Q8-11, we can reach the line 521, where the message was received from another node. At this point, Q12 reveals that that node 1 has received the message from node 0 at line 503. Node 0 was supposed to send `REP_LOG` after the execution of line 503 (sending `REP_PAGE`). However, according to Q13, whose result is an empty set, node 0 did not send any further message to node 1. From these combined queries, we thus discover the reason for the election failure.

Case Study #2: The second case study is for a previously unreported bug in RandTree. This bug exhibits as a permanent failure when a node joins the tree. In the experiment, we



```
650 else if (children.find(msg.source)...) {
652 maceout<< "!!!received Join from " << ...
```

```
QUERY 1: SELECT c.branch FROM CD c, Output o
WHERE c.location= o.location and o.value=
“!!!received Joion” and o.host=0
QUERY 2 : SELECT d.definition FROM DD d WHERE
d.use=id_650 and d.var=“children.find(msg.source)”
```

```
626 (children.size() < MAX_CHILDREN)) {
627       children.insert(msg.source);
```

```
QUERY 3 : SELECT d.definition FROM DD d
WHERE d.use=id_627 and d.variable=“msg.source”...
```

```
84   int r = recvfrom(transportSocket, rbuf, ...);
```

```
QUERY 4: SELECT COM.sender, COM.send_point
FROM COM WHERE COM.receiver=0 and
COM.recv_point=id_84
```

Sender is NODE 1

```
84   int r = recvfrom(transportSocket, rbuf, ...);
576  if (... (msg.source != root)&& ... ) {
580    downcall_route(root, msg);
178      int r = sendto(transportSocket, m.data(), m.size(),...);
```

```
QUERY 5 :SELECT CD.branch FROM CD WHERE
CD.location=id_178 and CD.host=1
QUERY 6: : SELECT DD.definition FROM DD
WHERE DD.use=id_576 and DD.variable=
“msg.source” and DD.host=1
QUERY 7: SELECT COM.sender, COM.send_point
FROM COM WHERE COM.receiver=1 and
COMM.recv_point=id_84_1
```

Sender is NODE 2 (OLD)

```
      __initialization...
1398    state_change(joining);
1404    downcall_route(bootstrap,...);
178      int r = sendto(transportSocket, m.data(), m.size(),...
```

Going back to the original line 650

```
QUERY 8 : SELECT DD.definition FROM DD
WHERE DD.use=id_650 and DD.variable=“msg.source”
and DD.host=0
```

```
84   int r = recvfrom(transportSocket, rbuf,
      MAX_MESSAGE_SIZE, 0, (struct sockaddr*)&sa, &salen);
```

```
QUERY 9: SELECT COM.sender, COM.send_point
FROM COMM WHERE COMM.receiver=0 and
COMM.recv_point=id_84_2
```

Sender is NODE 2 (NEW)

```
      __initialization...
1398    state_change(joining);
1404    downcall_route(bootstrap,...);
178      int r = sendto(transportSocket, m.data(), m.size(),...
```

Figure 11: Case study 2: RandTree – permanent failures to join tree.

have two nodes in the tree, nodes 0 and 1, in the initial phase. Later, node 2 starts and sends a join message to node 1. The message is relayed to the root node 0. Node 0 sends a join reply message to node 2 and updates its own child list, which now includes node 2, but node 2 resets right after sending the join message. After node 2 is reset, it ignores the join reply message from node 0 because it is in the initialization phase. After node 2 finishes initialization, it sends a join message to node 0. Node 0 ignores that message because node 2 is already its child. Node 2 keeps sending join messages to node 0, but all of them are ignored and thus node 2 cannot join the tree. When node 0 ignores the join message from node 2, it prints the error message “`!!!received Join from XX, already child.`”

We used 9 queries to debug this failure, all presented in Figure 11. We started from the error message. After Q1, we reach line 650, meaning node 2 is found in the child list. From this point, we seek to know why node 2 is already in the list. Using Q2, we find that node 2 is added at line 627, as the result of a message received at line 84 (Q3). Q4 discloses that the message was sent from node 1, as the result of a message received at line 84 in node 1 (Q5-6). Q7 identifies that the message was from the *previous* process of node 2 expressing its intention of joining the tree. Node 1 simply relayed the join message. Then, we go back to line 650 and try to understand why `msg.source` has the value of node 2. Through Q8 and Q9, we know that this join is from the *new* node 2 process.

The full queries for the remaining bugs in Figure 9 are omitted due to space restrictions.

VII. Related Work

Debugging pervasive systems is becoming increasingly important. D3S [15] and WiDS [14] are projects that aim to detect runtime errors in distributed systems. They are based on runtime property checking at the event level. These systems assume the user knows exactly what properties to check. In contrast, our system provides an interface that exposes artifacts at various levels, allowing the user to examine them regardless of foresight, starting from the observed symptoms. Another thread of work for debugging distributed systems uses model checking [19], [31]. MaceMC [12] is an explicit state model checker that checks liveness properties. In general, model checking also requires the user to have prior knowledge about the property to check and generally scales poorly. CrystalBall [30] is a tool that checks *predefined* properties on the fly.

In [25], Singh et al. propose a declarative language based debugging interface that is similar to our SQL interface. However, their system is specialized to their own declarative pervasive system programming language [16]. Our system is much more general, working on arbitrary binaries. `PTQL` [11] and `PQL` [18] propose query languages for single process execution. These techniques are not combined with replay and their schema design does not focus on debugging. Magpie [5], Pinpoint [6], and Pip [23] are projects based on log mining. In other words, they try to identify problems by looking at event logs. These techniques are quite effective in debugging performance problems, but less so for faults. In [29], Xin *et al.* present a technique to analyze distributed systems by building task graphs from event log files. Pothier *et al.* [22] present a portable Trace-Oriented Debugger for Java which uses efficient instrumentation techniques for event generation and a scalable storage system for completeness and efficient querying. All attributes of all events are logged at all times which is considered feasible in the targeted centralized setting targeted and with the assumption of a wireline connection to a dedicated scalable backend. Logging and replay [7], [9], [26], [21] is an important strategy for pervasive debugging. Existing work focuses on single node replay, which is insufficient for pervasive debugging. Friday’s replay system [10] supports replay with a GDB-like interface, but it cannot handle fine-grained instrumentation.

VIII. Conclusion

We have presented Recon, a system for debugging distributed systems based on a novel architecture, providing a consistent view of salient system properties. This view exposes properties via a relational framework that can be queried with a simple language based on SQL. Information collection is performed on demand to answer the queries, using filtering and prioritization to avoid collecting data unnecessary to formulating the answer.

We qualitatively evaluated Recon on several bugs in popular distributed programs. Furthermore, we have evaluated our design that separates a logging infrastructure from heavyweight analyses during replay, showing that it allows Recon to be used to record realistic runs with acceptable overhead (3%) and debug the runs later by replaying them.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research is supported in part by the National Science Foundation (NSF) under grants 0845870 and 0834529. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] *dwarfdump, http://reality.sgiweb.org/davea/dwarf.html*.

[2] *Macedon, http://www.macesystems.org/macedon*.

[3] *python-sqlparse, http://code.google.com/p/python-sqlparse*.

[4] M. Balazinska, H. Balakrishnan, and D. Karger, "Ins/twine: a scalable peer-to-peer architecture for intentional resource discovery," in *Pervasive*, 2002.

[5] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: online modelling and performance-aware systems," in *HotOS*, 2003.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *DSN*, 2002.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in *OSDI*, 2002.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM TOPLAS*, vol. 9, no. 3, 1987.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *USENIX*, 2006.

[10] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: global comprehension for distributed replay," in *NSDI*, 2007.

[11] S. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *OOPSLA*, 2005.

[12] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: finding liveness bugs in systems code," in *NSDI*, 2007.

[13] C. E. Killian, "Systems and language support for building correct, high performance distributed systems," in *Ph.D. dissertation*. University of California, San Diego, 2008.

[14] X. Liu, W. Lin, A. Pan, and Z. Zhang, "WiDS checker: combating bugs in distributed systems," in *NSDI*, 2007.

[15] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. Kaashoek, and Z. Zhang, "D3S: debugging deployed distributed systems," in *NSDI*, 2008.

[16] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *SOSP*, 2005.

[17] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[18] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA*, 2005.

[19] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *NSDI*, 2004.

[20] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[21] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. "Pres: probabilistic replay with execution sketching on multiprocessors," in *SOSP*, 2009.

[22] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *OOPSLA*, 2007.

[23] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: detecting the unexpected in distributed systems," in *NSDI*, 2006.

[24] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *AADEBUG*, 2005.

[25] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel, "Using queries for distributed monitoring and forensics," in *EuroSys*, 2006.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *ATEC*, 2004.

[27] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001.

[28] B. Xin and X. Zhang, "Efficient online detection of dynamic control dependence," in *ISSTA*, 2007.

[29] B. Xin, P. Eugster, X. Zhang, and J. Yang, "Lightweight task graph inference for distributed applications," in *SRDS*, 2010.

[30] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "Crystalball: predicting and preventing inconsistencies in deployed distributed systems," in *NSDI*, 2009.

[31] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM TOCS*, vol. 24, no. 4, 2006.

[32] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: transparent model checking of unmodified distributed systems," in *NSDI*, 2009.