Using Queries for Distributed Monitoring and Forensics

Atul Singh
Rice University

Petros Maniatis
Intel Research Berkeley

Timothy Roscoe
Intel Research Berkeley

Peter Druschel
Rice University
Max Planck Institute for Software Systems

ABSTRACT

Distributed systems are hard to build, profile, debug, and test. Monitoring a distributed system – to detect and analyze bugs, test for regressions, identify fault-tolerance problems or security compromises – can be difficult and error-prone. In this paper we argue that declarative development of distributed systems is well suited to tackle these tasks. We present an application logging, monitoring, and debugging facility that we have built on top of the P2 system, comprising an introspection model, an execution tracing component, and a distributed query processor. We use this facility to demonstrate a range of on-line distributed diagnosis tools that range from simple, local state assertions to sophisticated global property detectors on consistent snapshots. These tools are small, simple, and can be deployed piecemeal on-line at any point during a system's life cycle. Our evaluation suggests that the overhead of our approach to improving and monitoring running distributed systems continuously is well in tune with its benefits.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.2.2 [**Computer Communication Networks**]: Network Protocols—*protocol architecture, routing protocols*

General Terms

Design, Experimentation, Languages

Keywords

Declarative overlays, distributed monitoring, distributed debugging, invariant checking

1. INTRODUCTION

Finding faults in large-scale distributed systems is hard, be they caused by program bugs, security compromises, unexpected interactions among components, performance anomalies, or infrastructure failures. In this paper, we progressively investigate a methodology and toolset for building distributed systems that can be monitored, debugged, and diagnosed on-line throughout their lifecycle.

Faults in large, widely-distributed systems manifest themselves very differently from those in centralized systems. Faults are often partial, intermittent, and may result in anomalous behavior rather than failure. In addition to the common fault sources of programmer errors, system design flaws and hardware failures, distributed systems are afflicted by complex network failures, emergent (mis) behavior, denial-of-service attacks on the infrastructure, or compromise and subversion of one or more nodes by malicious adversaries. In addition, diagnosing or even recognizing a problem requires identifying and correlating relevant information from many different nodes.

In prior work with the P2 system [19], we demonstrated some of the advantages of building distributed systems by expressing the network-oriented functionality of a distributed application as a set of continuous queries over program and external network state. These queries are translated into an efficient distributed dataflow graph, providing the ability to specify distributed systems behavior concisely while retaining acceptable performance.

The contribution of this paper is an important extension of the P2 approach that was not explored in our earlier work: using query-processing for detecting (and in some cases reacting to) faults, anomalies, and potential security vulnerabilities. To realize this goal, we extend P2 to integrate its distributed continuous *query processor* with a comprehensive *introspection model* and a sophisticated facility for *execution tracing* of P2 programs.

1.1 Diagnostic vs. diagnosable systems

P2, the system we discuss in this paper, blurs the distinction between a *diagnostic system* (that is, a system whose purpose is to monitor and diagnose a distributed system) and what we might call a *diagnosable system*, by which we mean a system designed from the outset to be amenable to both new and existing monitoring and fault-finding techniques. As such, it highlights the fallacy of characterizing systems as either one or the other. Thinking of them as separate inevitably leads to a certain impedance mismatch, because the languages and abstractions used to specify the system and to specify diagnostic queries about the system are not the same.

In P2, distributed algorithms are specified at a high level in a declarative language, which is then translated into a dataflow graph and directly executed. As we will illustrate, by retaining and representing the details of this translation via a reflection model, P2 is a highly diagnosable system. The high-level algorithm description can be automatically instrumented, causing appropriate trac-

EuroSys'06, April 18–21, 2006, Leuven, Belgium.

ing, logging, and checkpointing to occur in the low-level dataflow representation.

However, at the same time P2 is a highly effective diagnostic system. It is at heart a distributed continuous query processor, which provides a concise, powerful, and intuitive way to express the kinds of operations necessary to monitor large networked systems and find faults on-line.

P2 presents a very different view of how to construct distributed systems, when compared to the common approach of defining and implementing low-level protocols: message formats and events in a language like C++ or Java. We advocate the P2 approach *precisely* in order to make the process of detecting faults, bugs, compromises and the like easy and natural, throughout the system's lifetime. This is because we believe that this process currently represents the most involved and costly aspect of designing, implementing, deploying, and operating a widely-distributed system.

1.2 Methodology

Any work on fault-finding techniques, particularly in on-line distributed systems, faces the difficulty of evaluation. Such techniques and facilities are only important for finding non-obvious faults (whether they be bugs, compromises, or failures), and most faults tend to be obvious in retrospect. How, then, should one demonstrate the value of a system feature in finding faults?

User studies work well when assessing the effectiveness of new tools applied to existing systems with real users and operators. However, the user study approach does not work well for evaluating a conceptually new way of constructing and diagnosing distributed systems, like P2. There is a chicken-and-egg problem here: a rigorous user study requires that the system be usefully deployable, with a ready community of users, which is rarely the case with a radically new toolset.

In this paper, we address this dilemma by first illustrating the ease of applying existing fault monitoring and diagnosis techniques on-line to distributed applications built over P2. For instance, we use the Chandy-Lamport algorithm to take consistent snapshots [5], and we show how queries over these snapshots can be easily formulated to verify global invariants and properties.

We then show that the overhead of applying such techniques is sufficiently low that, in many cases, queries to monitor particular conditions in the system can simply be left in place permanently. Thus, the system enables continuous monitoring of important conditions, aiding in the early detection and diagnosis of algorithmic or performance anomalies, as well as the detection and analysis of software bugs that rarely manifest themselves.

1.3 Usage scenarios and motivation

In this paper, we describe P2's fault diagnosis functionality and give examples of its use. Orthogonal to these examples, however, is the usage methodology within which they are deployed. The combination of distributed continuous query processing, introspection, and execution tracing leads to a variety of usage models, which we briefly outline here. This serves as motivation for P2's approach to monitoring and fault diagnosis.

The first scenario is simply *querying program state and logs*. This analysis is best expressed as a query, since a respectable query language can subsume most of the semantics of the ad hoc scripts programmers tend to write at present. Centralized management systems already provide this functionality: for example, IBM's Tivoli console allows the operator to write Prolog programs to perform continuous queries over management state. A scalable distributed query processor enables this approach to be used on-line: logs and state can be queried in place.

This leads naturally to the question of what information should be logged by a program. Developers must typically insert logging statements at compile time, which may be turned on or off later. In contrast, a comprehensive introspection model allows the "what" *to be identified as a continuous query on-line*, while taking care of the how automatically without the need for a programmer to insert "printf"s where they think is best. Combined with execution tracing in P2, this largely obviates the need for manual (and often error-prone) insertions of logging statements and post processing of logs (e.g., to find the causality relation between logged events).

In addition to querying conditions interactively, a continuous query processor allows a developer or operator to install persistent *distributed watchpoints and triggers* in the system, which generate events (as tuples) when a particular distributed condition occurs. Such watchpoints have many uses. Some can function as *intrusion detection measures*, for example to signal the probable compromise or subversion of part of the system. Alternatively, watchpoints installed during debugging can be left permanently in the system as an evolving set of *on-line regression tests*. Furthermore, such watchpoints are not limited to triggers when particular conditions occur. Distributed queries can be installed to perform *continuous on-line performance profiling* of the system.

The results of such watchpoints, derived from program state, logs, and execution traces of the distributed system, are themselves tuples which in turn can be the subject of queries. This leads to *higher-order automatic tracing of distributed execution*, whereby the system can be programmed to react to events by installing new triggers itself, for example to provide more detailed information about a particular area of the system. In this way the query processor provides a powerful building block for autonomic system operation.

In the next section we review the architecture of P2, and describe the aspects of the system new to this paper: the introspection model, and the distributed execution tracing facility. Following this in Section 3, we provide a series of concrete usage examples of this functionality in the context of a P2 implementation of the Chord lookup system. In Section 4 we quantify the performance cost of these facilities. After this, we review related work and conclude.

2. P2 AND SYSTEM MONITORING

P2 is a system for implementing and executing distributed algorithms, particularly for overlay routing and forwarding. We start with a brief overview of P2; for a detailed description see Loo et al. [19].

P2 is based on a relational model. Tuples are used to represent the state of the system in the form of soft-state tables on each participating node. For instance, out-links in a routing table might be represented by rows of the form [*my address, neighbor address, link weight*].

Tuples are also used to represent messages between nodes in the system. On each node, P2 instantiates a software dataflow graph (much like Click [17]) to implement an application's distributed algorithm. This graph is traversed by incoming message tuples, and its elements are C++ objects implementing relational operators (database joins, selections, projections, aggregations) as well as queues, multiplexers, etc. Executing this graph results in tuple transmission over the network, and/or insertion into local tables.

Applications using P2 can create such a dataflow graph explicitly using an embedded language, but P2 provides a higher-level query language to do this. This casts the distributed routing state of an application's overlay network as a database view over the underlying network state, and maintaining this view over time as the execution of a continuous, distributed, relational query. This approach

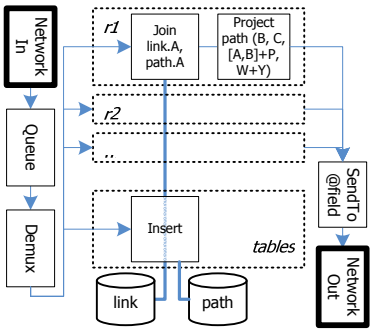

Figure 1: Dataflow for the "all routes" rule. Rectangular boxes are dataflow elements, "drum" boxes denote tables, arrows denote flows, and dashed boxes are for illustration purposes only, delineating the piece of a dataflow that corresponds to a particular rule.

enables the properties of a routing algorithm for an overlay to be specified extremely concisely [19], and it is this mode of usage of P2 that we focus on in this paper.

P2 expresses queries in OverLog, a variant of the Datalog language used in deductive databases. A query is a series of deductive rules of the form

```
[ruleID] result :- preconditions.
```

interpreted as "the result is true when all preconditions are met." For example, in the following:

```
path(B,C,[B,A]+P,W+Y) :- link(A,B,W), path(A,C,P,Y).
```

the precondition is that the tables named `link` and `path` contain entries in which the first respective fields match. When this occurs, tuples for `path` are created for all values matching the input entries. Interpreted logically, this rule says that if node A has a symmetric network link of weight W to B, and node A has a path P to node C with cost Y, then node B also has a path to C formed by prepending the link `[B,A]` to A's path to C, with cost W+Y.

Globally, this rule can (naïvely) build all routing paths from all sources to all destinations. In practice, each individual node manages only some of each table's tuples. We use the convention that the first field of a tuple denotes where the tuple lives. When the rule above triggers, the resulting `path` tuple must be sent to the address in its first field, where it is inserted in the local `path` table. For clarity, OverLog allows `link@A(B,W)` instead of `link(A,B,W)`.

Applications specify the tables they require using `materialize`. The arguments to this construct are the name of the tuple, maximum lifetime of a tuple, maximum number of tuples that can be in the table at any time, and the primary keys of the table in a `keys(...)` argument. The `keys(...)` argument contains, in order, the fields of the tuple that form the primary key of the table, that is, uniquely identify a tuple within the table. For example, the constructs

```
materialize(link, 100, 5, keys(1)).
materialize(path, 100, 5, keys(1,2)).
```

signify that both `link` and `path` tables contain at most 5 tuples at a time, expire tuples after 100 seconds, and uniquely identify their tuples by the first field for table `link` and by the first then second fields in table `path`.

A P2 component called the *planner* translates the rule above to the dataflow in Figure 1, consisting of a network preamble, a number of *rule strands* produced for each OverLog rule, and a network postamble. The preamble is responsible for receiving tuples, unmarshaling them, queuing them for processing, and then demultiplexing them among the rule strands. The postamble marshals output tuples and sends them to the appropriate destination (first tuple field). The rule strands are translations of individual OverLog rules into database query elements such as projection, join, and selection.

2.1 Introspection and Tracing

Since P2 represents application state using a relational model, it is natural to provide introspection on P2's own state in the same way, and so make it available to be queried from OverLog. Most of the state of a running P2 node (tables, rules, dataflow graph, etc.) is reflected back to the system as tables, themselves queryable in OverLog.

We extend this principle further to the logging of system events such as arrival of a tuple or removal of a tuple from a table. Log entries are tuples stored (more precisely, buffered) in P2 tables. This provides a potentially very powerful monitoring facility: OverLog queries can be written that express distributed conditions on P2 state, application state, and logs at the same time. We have found querying P2 logs in P2 itself highly convenient for most system functions.

The representation of both application and P2 state as queryable P2 tables is relatively straightforward, and we do not address its details further. In the rest of this section, we focus on a third level of introspection provided by P2: tracing the execution of individual OverLog rules, and following individual tuples as they flow through the system and across the network.

2.1.1 Tracing tuples and rules

All dataflow element classes in P2 are "tappable": any element can be made to copy the tuples it sends along a particular dataflow arc to an additional element. In the current implementation, a node has a single such element, the *tracer*, which collects tuple handoffs along all instrumented dataflow arcs.

Of course, such tapping of dataflow arcs has to be related to the high-level query rules that generated the dataflow graph. Consequently, the insertion of dataflow taps is performed by the planner when it generates the graph. To capture execution at the rule level, the planner must cause tuples to be traced entering a rule strand (marking the beginning of a rule's execution as its input event arrives), exiting the strand (marking the completion of a successful rule execution that produced a result), and any intermediate tuples fetched from tables (marking the identification of rule preconditions). These three types of tap are identified in Figure 2 by the black boxes, and together are sufficient to reconstruct the chain of computation following the triggering of a rule by an arriving tuple.

The tracer logs the results of these taps in a normalized P2 table called `ruleExec`. Tuples in `ruleExec` capture a causal link between two tuples, one of which caused the creation of the other via the application of a rule. "Cause" tuples are either triggering "event" tuples or precondition tuples used within the rule. A `ruleExec` tuple contains (1) the local node identifier (since it may be queried remotely), (2) the ID of the executed rule, (3) whether the cause was an event tuple or a precondition fetched from a table, (4) the "cause" tuple itself (be it an event or precondition), (5) the corresponding output tuple (the "effect") from the rule execution, and finally (6) the observation times for the two recorded tuples. For example, consider the OverLog rule:

```
r1 head@Z(Y) :- event@N(Y), prec@N(Z).
```

Figure 2 illustrates the dataflow execution of the rule, including tracer taps. Suppose `r1` executes at node `n` in response to a message `event@n(y)` and with precondition `prec@n(z)`, producing a

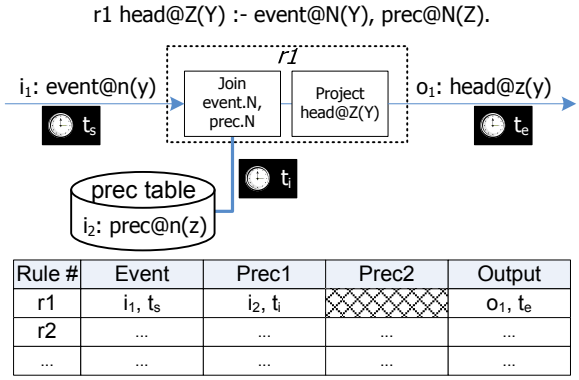

Figure 2: Dataflow detail for the execution tracing example. Black boxes denote times when a particular flow tap yielded a particular result (the pictured tuple). Each shown tuple is preceded by its local tuple ID.

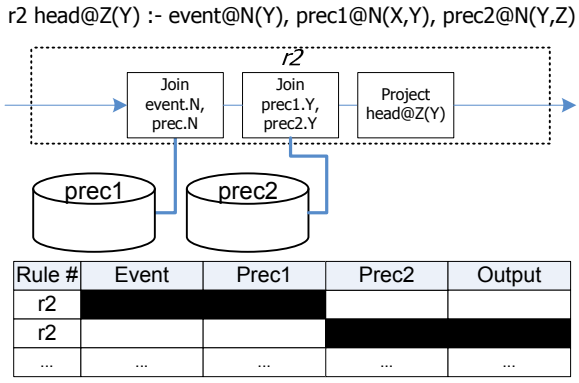

Figure 3: Dataflow detail for pipelined execution tracing. Black-filled fields in tracer records indicate fields that are *not* associated with a particular tracer record. In the example, the top record has only two fields associated with it (the last join and output) while the bottom record for rule `r2` has the first two fields associated with it. This configuration would occur if the first event has finished looking up matches in table `prec1` and is processing remaining matches in table `prec2` while a subsequent event has started processing matches in table `prec1`.

single output tuple `head@z(y)` [1]. Two rows will be added to the `ruleExec` table at node `n`:

`ruleExec@n(r1, event@n(y), head@z(y),` t_s`,` t_e`, true)`
`ruleExec@n(r1, prec@n(z), head@z(y),` t_i`,` t_e`, false)`

where t_s, t_e, and t_i are the wall-clock times at which the rule execution starts, stops, and fetches its precondition in the join element respectively. The former row captures the causal link between the `event` tuple triggering rule `r1` and the resulting tuple `head@z(y)`, while the second row captures the causal link between the precondition `prec@n(z)` – whose existence allowed the rule to be satisfied – and the same resulting tuple `head@z(y)`. In practice, the `head`, `event` and `prec` tuples are not stored directly in `ruleExec`, but memoized using the `tupleTable`, described below.

The tracer correlates observed tuples on the tapped flows to infer, from their ordering in time, when individual rules have completed. To do this, it maintains an array of tracing records, one per rule in the system (illustrated at the bottom of Figure 2). Each such record stores the tuples observed, along with the time of observation, for an entire rule strand. When an input tuple for a rule is observed, any prior contents for the entire record are cleared, and then the input is recorded by its tuple ID. Observed precondition tuples are stored in the appropriate record field, however any filled-in record fields to the right of the newly observed precondition are flushed. This is because tuples flow through a rule strand from left to right (in the figure), so the observation of a precondition in the "middle" of the strand signifies that any prior observed preconditions belonging to its right in the record and the strand are no longer relevant. Finally, when an output tuple is observed, the entire record is packaged into a `ruleExec` tuple and stored into the `ruleExec` table.

2.1.2 *Pipelined execution*

In the preceding description of rule tracing, for simplicity we have assumed that rule strands run sequentially to completion; that is, no new input tuple is let into a rule strand from the left before all tuples in flight within the rule strand have exited to the right or have been dropped. In this section we refine the machinery further to accommodate pipelined execution.

In P2, certain elements access the state stored in tables to process incoming tuples. For instance, a *join* element takes a search argument as input and looks into a table to find matches for that argument, until it has emitted all matches. When it has found all matches, the join element pulls another search argument from its input.

Such elements potentially produce multiple outputs for a single input. For example, consider rule `r2` (see Figure 3):

```
r2 head@Z(Y) :- event@N(X), prec1@N(X, Y),
   prec2@N(Y, Z).
```

When a new `event` arrives, rule `r2` can be satisfied for all combinations of tuples `prec1` and `prec2` for which the first fields of `event` and `prec1` match, and the second field of `prec1` and first field of `prec2` match. For every match found in the `prec1` table, all matches in the `prec2` table are found and `head` tuples emitted. Then another match in `prec1` is found and the process is repeated for `prec2`. When the last match in `prec1` is found, rule `r2` is ready to be executed for a new `event`. In this case, the handling of a new `event` through the join element for `prec1` might be interleaved with the remaining matches of the previous event through the join element for `prec2`.

To handle pipelining, the tracer holds multiple tracer records per rule strand, one for each *stage* in this pipeline (that is, for each stateful element, such as joins). When a stateful element defining an execution stage is complete (as indicated by its seeking a new input), the tracer is signaled of the completion of the stage.

To execute the algorithm in Section 2.1.1, we must match signals received by the tracer to specific tracer records. We do that by associating each tracer record with a sequence of stages in its rule strand, observing that only a single execution record can occupy any one stage. So, when a stage i signals its completion, the record (if any) whose associated stage sequence begins with i abandons that stage, advancing its first associated stage to $i + 1$. If no such tracer record exists, we extend the record with the latest associated stages to contain stage i.

With the above signaling mechanism, we can now match the preconditions and the output to appropriate records. When a new input event is observed, we find a record that does not have any stage as-

[1] Note here that in OverLog terms beginning with an upper-case letter represent variables while those beginning with a lower-case letter represent constants. Hence `N` is a node ID variable in the example, whereas `n` is the ID of a specific node.

sociated with it. If no such record exists, we create a new one and associate it with the first stage. Similarly, when an element at stage i provides a precondition, we find the record with which stage i is currently associated and post the precondition appropriately. Finally, when an output tuple is generated, it is matched to a record with the highest associated stage.

2.1.3 Tracing rules across the network

The tracer as described so far can capture all executions of a particular rule on each node, and make such traces available for distributed querying using OverLog. This is not quite sufficient to implement distributed tracing of execution: what is also needed is a way to relate the output of a rule on one node to the triggering of a rule on another. This amounts to tracing the sending of tuples between nodes.

Each P2 node assigns tuples a node-unique ID when they are first created (tuples are immutable in P2). This ID is used to memoize the tuple, and it is this ID that is stored in the `ruleExec` table rather the tuple itself.

The mapping from tuple IDs to tuples is kept in a second table available to P2 queries: the `tupleTable`, which holds (1) the tuple ID, (2) a source address for tuples that arrived from another node, together with (3) the ID of the tuple on the source node, and (4) the destination address for tuples that have been sent across the network.

At the node `n` in our example, the `tupleTable` would contain the entry `tupleTable@n(`o_1`, n,` o_1`, z)`, indicating that it originated locally (with ID o_1) and was sent to `z`. Assuming it was received, a tuple with the same content would also appear in `z`'s table as `tupleTable@z(`d_1`, n,` o_1`, z)` for some `z`-local tuple ID d_1. `tupleTable` tuples are not themselves described within the `tupleTable`.

We use reference-counting to flush old tuple records from those currently held in `tupleTable`. In practice, this means that an entry is discarded when the last referring entry in `ruleExec` is removed or times out.

Next we discuss various usage scenarios for P2's combination of distributed query processing, introspection, and execution tracing facilities. In Section 4, we evaluate the overhead to these mechanisms when used in practice.

3. APPLICATIONS

We start with a few simple case studies, in which we examine common monitoring tasks that ensure routing consistency (Section 3.1) in overlays. Next, we describe larger, more complex monitoring tasks that may apply to diverse problems, including functional profiling (Section 3.2), and taking consistent snapshots of a running system (Section 3.3). Our objective in this section is not to discover new improvements for overlays, but to cast some of the techniques that researchers have previously employed for the task in the context of P2.

All examples refer to an implementation of Chord [23] on P2. We present the relevant features of the P2 Chord implementation as needed. It is worth pointing out, however, that our features are not specific to Chord in particular or distributed hash tables in general, but apply equally well to other algorithms with distributed state and control.

3.1 Consistent Routing

In a distributed lookup service, routing consistency is the property of answering the same lookup with the same result at the same time, regardless of who is asking. It is an important property to approximate in DHTs since it exemplifies the hash table metaphor: you get what you put in, as if the system were implemented with a centralized hash table.

In the following examples, we give progressively more complex detectors for invariant violations that degrade routing consistency including malformed ring topology (Section 3.1.1), incorrect ID ordering on the ring (Section 3.1.2), and the use of stale routing state during lookups (Section 3.1.3). We conclude the section with an active detector for routing inconsistency itself (Section 3.1.4).

3.1.1 Ring is Well Formed

Inconsistent routing may occur due to a pathological topology. The Chord DHT relies for its correctness on the correct maintenance of a *ring* among all of its members, in which (a) every node has exactly one immediate successor and predecessor, and (b) every node is its successor's predecessor and it's predecessor's successor. If the ring is incorrect, then depending on where a lookup starts, it may return a different response.

Active Probing: To detect incorrect ring maintenance of this type, for example, a node can periodically ask its immediate predecessor for its immediate successor. If the response does not match the probing node itself, then there must be a ring flaw between the prober and the node it considers its immediate predecessor.

In the following OverLog snippet, rule `rp1` periodically[2] (with period `tProbe`) finds its predecessor (in the `pred` tuple), checks that it is non-empty (different from `"-"`), and sends a request for that predecessor's immediate successor. Rule `rp2` handles such requests by finding the current immediate successor (in the `bestSucc` tuple, containing the local address, and the successor's ID and address), and returning it to the requester. Finally, rule `rp3` handles a response to such a probe, checking that it came from the node's predecessor, and raising an alarm tuple called `inconsistentPred` if the successor returned by the predecessor does not match the local node's address.

```
rp1 reqBestSucc@PAddr(NAddr) :- periodic@Naddr(E,
   tProbe), pred@NAddr(PID, PAddr), PAddr != "-".

rp2 respBestSucc@ReqAddr(NAddr, SAddr) :-
   reqBestSucc@NAddr(ReqAddr), bestSucc@NAddr(SID,
   SAddr).

rp3 inconsistentPred@NAddr():-
   respBestSucc@NAddr(PAddr, Successor),
   pred@NAddr(PID, PAddr), Successor != NAddr.
```

Similar rules can also check that a node is its immediate successor's predecessor.

Passive Checks: In contrast to actively sending requests that help ascertain that the ring is well-formed, a designer could also use passive detection rules that take advantage of Chord's message semantics without generating new messages.

For example, Chord employs a periodic process called *stabilization* – interestingly, for the exact purpose of maintaining a well-formed ring – during which a node sends a `stabilizeRequest` message to its immediate successor. Since the semantics of this message is that nodes send it to their immediate successors, the recipient can apply the same logic as rule `rp3` above: "if the sender of this message is not my immediate predecessor, we have an inconsistent ring link." This is what rule `rp4` below does.

```
rp4 inconsistentPred@NAddr() :-
   stabilizeRequest@NAddr(SomeID, SomeAddr),
   pred@NAddr(PID, PAddr), SomeAddr != PAddr.
```

[2] `periodic@N(E, T)` is a built-in event that becomes true every `T` time units at node ID `N` with a random nonce `E`.

The down side of this approach is that detection of inconsistencies occurs at the rate at which `stabilizeRequest` messages arrive, rather than the arbitrary rate `tProbe` used in the example of rules `rp1-rp3`.

3.1.2 Ring ID Ordering is Correct

Even when the ring is well formed topologically, it may be incorrect from the point of view of node ID semantics. Chord requires that nodes are arranged on the ring according to their ID ordering. A node's immediate successor should be the node with the next higher ID in the entire population, and its immediate predecessor should be the node with the next lower ID.

Opportunistic Checks: It is straightforward to write opportunistic passive checks that flag an ID inconsistency whenever a node ID borne by any incoming message falls between the local node's predecessor's ID and the local node's successor's ID. For example, the following rule checks Chord lookup responses, contained in `lookupResults` tuples. These tuples carry the local node's address, the key looked for, the result node ID and address, the lookup request number, and the sender's address.

```
ri1 closerID@NAddr(ResltNodeID, ResltNodeAddr) :-
   lookupResults@NAddr(Key, ResltNodeID,
   ResltNodeAddr, ReqNo, RespAddr), pred@NAddr(PID,
   PAddr), bestSucc@NAddr(SID, SAddr), ResltNodeID
   in (PID, SID).
```

When such a result tuple arrives at a node, the node's immediate successor ID (in a `bestSucc` tuple) and its immediate predecessor ID (in a `pred` tuple) are fetched. If the node ID in the result tuple happens to be between the two immediate neighbors' IDs (checked by the `in` expression at the end of the rule), then a `closerID` event is issued identifying the node and ID that seem to violate the check.

Traversals: In addition to the previous largely localized checks, more sophisticated detection processes are simple to define. Viewed holistically, in a full ring traversal along immediate successor links there should be a single ID "wrap-around" (that is, a drop in the absolute value of the traversed IDs). To check this invariant, a token-passing scheme can be used in which, starting from a random node, a token traverses immediate successor pointers counting wrap-arounds. If at the time of reaching its origin, the traversal has identifier more than one wrap-around (or worse, fewer than 1), then something must be wrong in the ring. The following rules encode this distributed check.

```
ri2 ordering@NAddr(E, NAddr, NID, 0) :-
   orderingEvent@NAddr(E), node@NAddr(NID).

ri3 countWraps@NAddr(SAddr, E, SrcAddr, SID, Wraps)
   :- ordering@NAddr(E, SrcAddr, MyID, Wraps),
   bestSucc@NAddr(SAddr, SID), MyID < SID.

ri4 countWraps@NAddr(SAddr, E, SrcAddr, SID, Wraps
   + 1) :- ordering@NAddr(E, SrcAddr, MyID, Wraps),
   bestSucc@NAddr(SAddr, SID), MyID >= SID.

ri5 ordering@SAddr(E, SrcAddr, SID, Wraps) :-
   countWraps@NAddr(SAddr, E, SrcAddr, SID, Wraps),
   SAddr != SrcAddr.

ri6 orderingProblem@SAddr(E, SrcAddr, SID, Wraps)
   :- countWraps@NAddr(SAddr, E, SAddr, SID,
   Wraps), Wraps != 1.
```

The traversal occurs via a token tuple `ordering`. Token fields are the local node address `NAddr`, the traversal ID `E`, the address of the traversal initiator `SrcAddr`, the ID of the local node `MyID`, and the count of wrap-arounds thus far. Rule `ri2` begins the traversal when the `orderingEvent` appears at a node, by creating the `ordering` token. How a particular traversal initiator is chosen is an orthogonal concern. It could be decided using a leader election algorithm on the ring, or the node responsible for a well-known ID could be the one to start the traversals. Either way, the rule assigns a traversal ID `E` to each traversal, so multiple traversals going on at the same time are allowed.

Rules `ri3` and `ri4` update the token's counter of wrap-arounds by looking up the current node's immediate successor (`bestSucc`) and comparing the two nodes' IDs. `ri3` leaves the number of wrap-arounds unchanged if the current node's ID is lower than its successor's, while `ri4` increments it otherwise. Both produce a `countWraps` tuple with the outcome.

Rules `ri5` and `ri6` decide whether to continue the traversal, based on the updated token `countWraps`. `ri5` forwards the token to the successor, if the successor's ID is different from that of the traversal initiator. `ri6` is triggered when the successor's address and that of the initiator are the same (i.e., the corresponding fields in `countWraps` match), and when the number of wrap-arounds found during the traversal is different from 1. Then an ordering problem is reported to the initiator. No notice is sent if the traversal completes finding exactly a single wrap-around.

3.1.3 State Oscillations

So far, we have addressed correctness invariants in topology, from the point of view of the ring graph and from the point of view of vertex identifiers. In the next set of examples, we shift focus to detectors of potentially pathological execution patterns, namely the proliferation of stale state.

In Chord, as in many stabilizing overlays that use gossip among neighbors, nodes exchange bits of their routing state with their neighbors. In the stabilization process, already mentioned above, a node periodically tells its immediate ring neighbors about its other immediate ring neighbors. Each node sorts through these periodic neighborhood exchanges to obtain an up-to-date view of its vicinity.

An incorrect implementation of the Chord protocol might fall prey to the *recycled dead neighbor* problem. In this pattern, a node finds a neighbor unresponsive and removes it from its routing state, after having gossiped it to its other neighbors. In subsequent gossip exchanges, the node receives the formerly removed neighbor and places it back into its routing state, in the long run oscillating back and forth between removing and reinserting the offending neighbor. Typically, remembering recently "deceased" neighbors for a while may solve this problem. In this section, we focus on the manifestation of the problem and specify detectors for it at three different granularities: local, single-oscillation detection; local, repeated-oscillation detection; and collaborative detection.

Single oscillation: We give an example with Chord successors, held in `succ` tuples, containing the local node address, the successor's ID, and the successor's address. In a simple Chord implementation in P2, if a node pings a successor without response, it remembers that successor as a

```
faultyNode@NAddr(FaultyAddr, Time)
```

tuple and removes it from the `succ` table.

A node opportunistically inserts new successors into its state during periodic stabilization: its immediate successor's predecessor, and its successors' successors. Then it chooses the k closest and discards the rest. For reference, we give below the two P2 Chord rules [19] that perform successor insertions when the two messages (`sendPred` and `returnSucc`) are received for the two cases, respectively.

```
sb4 succ@NAddr(SID, SAddr) :- sendPred@NAddr(SID,
   SAddr).

sb7 succ@NAddr(SID, SAddr) :- returnSucc@NAddr(SID,
   SAddr).
```

To build an oscillation detector for successors, we use the following two rules, which capture the two insertion messages above and, if they contain recently deceased nodes, found in `faultyNode` tuples, signal an oscillation with a timestamp. The `f_now()` built-in gets the current time.

```
os1 oscill@NAddr(SAddr, T) :-
   faultyNode@NAddr(SAddr, T1), sendPred@NAddr(SID,
   SAddr), T := f_now().

os2 oscill@NAddr(SAddr, T) :-
   faultyNode@NAddr(SAddr, T1),
   returnSucc@NAddr(SID, SAddr), T := f_now().
```

Repeat oscillations: A single oscillation may occur naturally once in a while due to transient connectivity disruptions. Here we add two more rules and an extra table to the above example to catch *repeat* oscillations.

```
materialize(oscill, 120, infinity, keys(2,3)).

os3 countOscill@NAddr(OscillAddr, count<*>) :-
   periodic@NAddr(E, 60), oscill@NAddr(OscillAddr,
   Time).

os4 repeatOscill@NAddr(OscillAddr) :-
   countOscill@NAddr(OscillAddr, Count), Count >=
   3.
```

The `materialize` statement creates a table to store `oscill` tuples, for up to 120 seconds each, with a primary key made up of the oscillating node address and time. This means that at any time, the `oscill` table contains all oscillator proclamations from the prior 120 seconds, potentially multiple per node.

Rule `os3` starts a check every 60 seconds and counts the number of oscillations per node in the `oscill` table. Rule `os4` applies a threshold of 3 oscillations within the history of the `oscill` table (120 sec) before declaring a repeat oscillator. When `repeatOscill` is issued, an alarm could be raised to look into this behavior.

Collaborative oscillation detection: We extend the example further by allowing nodes to proclaim oscillations collaboratively within the ring neighborhood. Now each node, after detecting a repeat oscillator, notifies its successors and predecessor that it has registered a repeat oscillator. Since we are currently treating successor oscillations, this set of nodes is precisely the set that would also be experiencing oscillations from the same offender. In the OverLog extension below, a node marks an oscillator `chaotic` if more than three of its neighbors believe it to be a repeat oscillator. Finding a node `chaotic` means that, with high confidence, the system is prone to state oscillations and corrective or palliative action must be taken.

```
materialize(nbrOscill, 120, infinity, keys(2,3)).

os5 nbrOscill@NAddr(OscillAddr, NAddr) :-
   repeatOscill@NAddr(OscillAddr).

os6 nbrOscill@SAddr(OscillAddr, NAddr) :-
   repeatOscill@NAddr(OscillAddr), succ@NAddr(SID,
   SAddr).

os7 nbrOscill@PAddr(OscillAddr, NAddr) :-
   repeatOscill@NAddr(OscillAddr), pred@NAddr(PID,
   PAddr).

os8 nbrOscillCount@NAddr(OscillAddr, count<*>) :-
   nbrOscill@NAddr(OscillAddr, ReporterAddr).

os9 chaotic@NAddr(OscillAddr) :-
   nbrOscillCount@NAddr(OscillAddr, Count), Count >
   3.
```

The `materialize` statement creates another table `nbrOscill` that, for 120 seconds, stores repeat oscillators in my neighborhood and the addresses of the neighbors who told me about them. Rule `os5` places a node's detected oscillators into its own `nbrOscill` table. Rules `os6` and `os7` propagate the same to the node's successors' and predecessor's `nbrOscill` tables, respectively. Rule `os8` counts the number of `nbrOscill` tuples for each oscillator whenever the table is updated. Rule `os9` finally identifies `chaotic` nodes as those with 3 or more reports of repeat oscillations in the neighborhood.

3.1.4 Proactive inconsistency detection

Thus far, we have described checks that can capture contributing factors to routing inconsistency. Here we capture the first-order symptom itself: obtaining differing responses to the same lookup at the same time. We tackle this as an active test, in which we generate a lookup workload and observe the results. Specifically, a node initiates a periodic consistency probe, during which it asks its neighbors to issue lookups for the same key. It checks the results it receives, counting the size of the largest cluster of responses with the same answer. The size of that cluster relative to the number of probe lookups is a consistency metric. Ideally, this metric would be 1.

```
materialize(conLookupTable, 100, 100, keys(1)).

materialize(conRespTable, 100, 100, keys(1)).

materialize(respCluster, 100, 100, keys(1)).

materialize(maxCluster, 100, 100, keys(1)).

materialize(lookupCluster, 100, 100, keys(1)).

cs1 conProbe@NAddr(ProbeID, K, T) :-
   periodic@NAddr(ProbeID, 40), K := f_randID(), T
   := f_now().

cs2 conLookup@NAddr(ProbeID, K, FAddr, ReqID, T) :-
   conProbe@NAddr(ProbeID, K, T),
   uniqueFinger@NAddr(FAddr, FID), ReqID :=
   f_rand().

cs3 conLookupTable@NAddr(ProbeID, ReqID, T) :-
   conLookup@NAddr(ProbeID, K, SrcAddr, ReqID, T).

cs4 lookup@SrcAddr(K, NAddr, ReqID) :-
   conLookup@NAddr(ProbeID, K, SrcAddr, ReqID, T).

cs5 conRespTable@NAddr(ProbeID, ReqID, SAddr) :-
   lookupResults@NAddr(K, SID, SAddr, ReqID,
   Responder), conLookupTable@NAddr(ProbeID, ReqID,
   T).

cs6 respCluster@NAddr(ProbeID, SAddr, count<*>) :-
   conRespTable@NAddr(ProbeID, ReqID, SAddr).

cs7 maxCluster@NAddr(ProbeID, max<Count>) :-
   respCluster@NAddr(ProbeID, SAddr, Count).

cs8 lookupCluster@NAddr(ProbeID, T, count<*>) :-
   conLookupTable@NAddr(ProbeID, ReqID, T).

cs9 consistency@NAddr(ProbeID, RespCount /
```

```
   LookupCount) :- periodic@NAddr(E, 20),
   lookupCluster@NAddr(ProbeID, T, LookupCount), T
   < f_now() - 20, maxCluster@NAddr(ProbeID,
   RespCount).

cs10 delete lookupCluster@NAddr(ProbeID, T, Count)
   :- consistency@NAddr(ProbeID, Consistency).

cs11 delete conLookupTable@NAddr(ProbeID, ReqID, T)
   :- consistency@NAddr(ProbeID, Consistency),
   conLookupTable@NAddr(ProbeID, ReqID, T).
```

This example maintains the following tables: `conLookupTable` holds consistency lookups for the duration of a consistency probe; `conRespTable` holds responses to consistency lookups, `respCluster` clusters together responses that agree with eachother; `maxCluster` keeps the size of the most popular response; and `lookupCluster` counts the consistency lookups that have been sent for a single probe. More details on each table follow.

Rule `cs1` periodically begins a consistency probe of a random key `K` in the Chord ID space, and picks the time of the probe `T`, and a random probe request ID `E` (called `ProbeID` in the remainder). A probe produces one consistency lookup for every faraway neighbor of the node (such are called *fingers* in Chord terminology, and are held in the `uniqueFinger` table here), giving each lookup its own request ID, in rule `cs2`. Each consistency lookup is stored in `conLookupTable` for later perusal (rule `cs3`), and a Chord lookup is issued starting with the chosen finger node in rule `cs4`. A lookup is placed in a `lookup` tuple, which contains the starting node's address, the key looked for, the address of the requester, and a lookup request ID.

Responses are conveyed in a `lookupResults` tuple, containing the requester's address, the key, the outcome of the lookup (that is, the successor ID and address for the sought key), as well as the request ID and the address of the responder. When/if a response arrives back that matches the request ID of a consistency lookup, it is stored in the consistency responses table in rule `cs5`. Rule `cs6` keeps updated a table of response clusters, in which for every successor to the looked-up key `K` returned, an entry is maintained with the number of agreeing responses. Rule `cs7` keeps track of the response cluster with the maximum count, and `cs8` counts the number of consistency lookups sent for a given probe.

To tally the results, periodically a probe's lookup count is found for a probe initiated further than 20 sec in the past, in rule `cs9`. The rule outputs the consistency metric for that probe (tuple `consistency`) by dividing the size of the largest response cluster by the number of consistency lookups issued. After the consistency metric is computed, all consistency lookup state is deleted in rules `cs10` and `cs11`; the `delete` keyword removes from tables any tuples in the rulehead. Remaining state for responses and response clusters expires after a while.

As above, the consistency metric can be used to raise alarms, e.g., with a trigger such as

```
cs12 consAlarm@NAddr(PrID) :-
   consistency@NAddr(PrID, Cons), Cons < 0.5.
```

or in forensic queries, as described in the next section.

3.2 Execution Profiling

A common task of maintainers is to estimate where the system spends its time performing its tasks. For example, an operator may wish to know how P2 Chord spends its time between when a lookup is issued and a response is returned. That would be particularly appropriate for those lookups that turn up inconsistent in the rules of Section 3.1.4.

Here we demonstrate the use of execution tracing to split lookup latencies into time spent executing rules, time spent traversing the network, and time spent in the dataflow between rules. A long time spent between rules might indicate, for instance, unnecessary queuing or execution blocking.

The following rules start from a selected lookup response (indicated in the `traceResp` event) and walk backwards the execution graph of that response, rule by rule, collecting timings in the appropriate bin. When the originating lookup has been reached in the traversal, the results are stored for later perusal.

```
ep1 trav@NAddr(TupleID, TupleID, TupleTime, 0, 0,
   0) :- traceResp@NAddr(TupleID, TupleTime).

ep2 ruleBack@SrcAddr(ID, Curr, LastT, RuleT, NetT,
   LocalT, Local) :- trav@NAddr(ID, Curr, LastT,
   RuleT, NetT, LocalT), tupleTable@NAddr(Curr,
   SrcAddr, SrcTID, LocSpec), Local := (LocSpec ==
   SrcAddr).

ep3 forward@NAddr(ID, In, InT, RuleT + OutT - InT,
   NetT, LocalT + LastT - OutT, Rule) :-
   ruleBack@NAddr(ID, Curr, LastT, RuleT, NetT,
   LocalT, true), ruleExec@NAddr(Rule, In, Curr,
   InT, OutT, true).

ep4 forward@NAddr(ID, In, InT, RuleT + OutT - InT,
   LocalT + LastT - OutT, LocalT, Rule) :-
   ruleBack@NAddr(ID, Curr, LastT, RuleT, NetT,
   LocalT, false), ruleExec@NAddr(Rule, In, Curr,
   InT, OutT, true).

ep5 trav@NAddr(ID, Curr, LastT, RuleT, NetT,
   LocalT) :- forward@NAddr(ID, Curr, LastT, RuleT,
   NetT, LocalT, Rule), Rule != "cs2".

ep6 report@NAddr(ID, RuleT, NetT, LocalT) :-
   forward@NAddr(ID, Curr, LastT, RuleT, NetT,
   LocalT, "cs2").
```

Rule `ep1` is triggered when a `traceResp` event carrying the tuple ID `TupleID` to trace backward. It starts a traversal token `trav`, which contains the local node address, the tuple ID being investigated and the current tuple ID, the latest timestamp observed, and then the three cumulative times: time within rule strands (denoted `RuleT` below), in between rule strands within the same dataflow graph (denoted `LocalT` below), and in between rule strands traversing the network (denoted `NetT` below). The rule starts the traversal with the offending tuple, zeroing out all cumulative times.

Rule `ep2` receives a `trav` token and checks the current tuple ID against the `tupleTable`, figuring out whether the current tuple crossed the network or not, and encapsulating this information into a `ruleBack` tuple.

Rules `ep3` and `ep4` traverse a `ruleExec` tuple and update cumulative times according to whether the currently traversed tuple was found local or remote by `ep2`. Rule `ep3` is triggered if the tuple was local. It finds all `ruleExec` tuples that produced the current tuple as output, and out of those selects only the single `ruleExec` that carries the input event of its rule, ignoring preconditions, since their backtracing is not on the main path of the lookup latency. Then `ep3` subtracts the time when the `ruleExec` execution completed (`OutT`) from the time `LastT` when the current tuple was received by its destination rule. It adds this interval to `LocalT`, since the current tuple was local. Furthermore, the rule adds the time it took for the `ruleExec` rule to complete (`OutT - InT`) to `RuleT`. Finally, `ep3` switches to regarding as current tuple the input tuple `In` of the `ruleExec` entry and sends the state of the computation (in a `forward` tuple) for termination checking below. Rule `ep4` cor-

responds to ep3 for the case where the current tuple did traverse the network, and functions the same way, except it updates NetT instead of LocalT.

Rule ep5 decides whether to stop the traversal or not, based on the outcome of rules ep3 or ep4. It compares the rule ID of the ruleExec tuple just traversed to rule ID cs2. Recall from Section 3.1.4 that cs2 was the rule initiating consistency lookups, therefore that is where this particular execution traversal should conclude accumulating latency components. If in fact the execution traversal has not reached rule cs2 yet, the process continues recursively by issuing a fresh trav tuple. Otherwise, rule ep6 reports (locally) the three cumulative numbers collected from the execution trace.

3.3 Consistent Distributed Snapshots

Distributed system state is hard to capture since, typically, at all times all components of the system are moving forward with their execution tasks. Capturing a "snapshot" of all components' states at the same time may be tricky if for instance some components are lagging behind in their event processing.

A *consistent snapshot* of a distributed computation seeks to capture a state of all components and the communication "channels" among them that is *equivalent* to an actual global state: from any component's point of view, it is indistinguishable from a global state. Consistent snapshots can be invaluable in checkpointing the computation for later restart, or for detecting *stable properties* of the system such as deadlocks, termination of computations, etc. Here we describe an implementation within P2 of the classic Chandy-Lamport algorithm for distributed consistent snapshots [5] for Chord.

Briefly, the algorithm starts with an initiator node, which takes a snapshot of its relevant state (in our context, copies aside the contents of some state tables), and then sends a marker message to all of its neighbors. Every node receiving such a marker for the first time (for a given snapshot) similarly records its state and forwards the marker to all of its own neighbors. A node records all messages it receives from a neighbor between the time when it first snapped its state and until it receives a (subsequent) marker from that neighbor. When a node has snapped its own state and has received markers from all of its neighbors, it terminates the algorithm. The output is the node's snapped state and all recorded messages from each neighbor.

The original algorithm is meant to operate on a FIFO network and when each node knows all its incoming and outgoing links. In contrast, although a Chord node knows its outgoing links, it does not know its incoming links and, in some cases, it might exchange messages with nodes that are not its neighbors in the topology (for instance, when receiving a lookup response). To alleviate the lack of incoming link information, the implementation below creates a *view* of the topology for incoming links in the backPointer table.

```
bp1 backPointer@NAddr(RemoteAddr) :-
   pingReq@NAddr(RemoteAddr).

bp2 numBackPointers@NAddr(count<*>) :-
   backPointer@NAddr(RemoteAddr).
```

P2 Chord nodes ping all their neighbors periodically to check their liveness. Rule bp1 above stores the addresses of those who ping a node (using the pingReq message) to maintain a table of incoming links. bp2 counts the current back pointers.

To handle messages traveling between non-neighboring nodes, we modify the algorithm as follows. If a node taking a snapshot receives such a message from a node that has already snapped its state, it does not record the message in the snapshot, since that message belongs to the *future* of both nodes' snapshots. If the message arrives from a node that has not taken the snapshot yet, it is recorded as per the algorithm, and the sender is added to the local node's incoming links. Finally, if the message arrives at a node that has not started the next snapshot from a node that has, it is regarded as a new marker (which starts the snapshot process) and the sender is added to the incoming links.

An OverLog implementation of this algorithm on top of P2 Chord follows. This implementation assumes that the network guarantees FIFO message delivery. We omit an extension that would use an in-order transport such as TCP underneath to remove this assumption.

```
materialize(snapState, 100, 100, keys(1)).

materialize(snapBestSucc, 100, 50, keys(1, 2)).

materialize(snapFingers, 100, 1600, keys(1, 2)).

materialize(snapPred, 100, 10, keys(1, 2)).

materialize(channelState, 100, 1600, keys(1, 2)).

materialize(channelSendSuccDump, 100, 100,
   keys(1, 4)).

materialize(channelLookupResDump, 100, 100,
   keys(1, 3)).

sr1 snap@NAddr(I + 1) :- periodic@NAddr(E, tSnapFreq)
   snapState@NAddr(I, State).

sr2 snapState@NAddr(I, "Snapping") :-
   snap@NAddr(I).

sr3 currentSnap@NAddr(I) :- snap@NAddr(I).

sr4 snapBestSucc@NAddr(I, SAddr, SID) :-
   snap@NAddr(I), bestSucc@NAddr(SID, SAddr).

sr5 snapFingers@NAddr(I, FPos, FAddr,
   FID) :- snap@NAddr(I), finger@NAddr(FPos, FID,
   FAddr).

sr6 snapPred@NAddr(I, PAddr, PID) :-
   snap@NAddr(I), pred@NAddr(PID, PAddr).

sr7 marker@RemoteAddr(NAddr, I) :- snap@NAddr(I),
   pingNode@NAddr(RemoteAddr).

sr8 haveSnap@NAddr(SrcAddr, I, count<*>) :-
   snapState@NAddr(I, State), marker@NAddr(SrcAddr,
   I).

sr9 snap@NAddr(I) :- haveSnap@NAddr(Src, I, 0).

sr10 channelState@NAddr(Remote + E, Remote, E,
   "Start") :- haveSnap@NAddr(Src, E, 0),
   backPointer@NAddr(Remote), Remote != Src.

sr11 channelState@NAddr(Src, E, "Done") :-
   haveSnap@NAddr(Src, E, C),
   backPointer@NAddr(Remote), (C > 0) || (Src ==
   Remote).

sr12 doneChannels@NAddr(E, count<*>) :-
   channelState@NAddr(E1, Src, E, "Done").

sr13 snapState@NAddr(E, "Done") :-
   snapState@NAddr(E, "Snapping"),
   doneChannels@NAddr(E, C),
   numBackPointers@NAddr(C).
```

```
sr14 snap@NAddr(SrcSnapID) :-
   lookupResults@NAddr(K, SID, SAddr, LookupID,
   Src, SrcSnapID), currentSnap@NAddr(MySnapID),
   SrcSnapID > MySnapID.

sr15 channelSendSuccDump@NAddr(E, LI,
   SID, SAddr, Time) :- returnSuccessor@NAddr(SID,
   SAddr, Src), channelState@NAddr(E2, Src, E,
   "Start"), Time := f_now().

sr16 channelLookupResDump@NAddr(SrcSnapID, E,
   LI, S, SI, C) :- lookupResults@NAddr(K, S, SI,
   E2, LI, SrcSnapID), currentSnap@NAddr(SnapID),
   SrcSnapID < SnapID.
```

Every node maintains its current snapshot state in `snapState` table, each tuple of which contains a snapshot ID (an increasing counter), and the current phase of the snapshot ("Snapping" while it is on-going, "Done" when it is completed). Rule `sr1` periodically advances the snapshot ID by one and begins a new snapshot, storing the new snapshot state in rule `sr2`. Note that `sr1` only executes on the snapshot initiator, while the remainder of the rules execute at all nodes. Rule `sr3` records the current snapshot.

Rules `sr4`, `sr5`, and `sr6` record the relevant state for this snapshot, by copying information from the `bestSucc`, `finger`, and `pred` tables, respectively. The recorded state is stored in separate tables (`snapBestSucc`, `snapFingers`, and `snapPred`) indexed by snapshot ID and input table primary key. Rule `sr7` sends out markers to all outgoing links of the current node. In P2 Chord, the `pingNode` table contains all neighbors that a node pings periodically for liveness, and therefore represents the node's outgoing links.

Rule `sr8` checks the state, if any, of an incoming marker's snapshot ID. The result, `haveSnap`, has a count of snapshot state entries for that ID: 0 if this is a new snapshot, 1 if this is an already seen snapshot. If this is a new snapshot, rule `sr9` begins the snapshot as above. Rule `sr10` begins the recording of messages on all incoming links – other than the one on which the marker arrived. The incoming links are found from the `backPointer` table described above. `sr11` marks instead the recording of the channel as `Done` for the marker's sender or for every incoming link if this is a seen-before snapshot.

Rules `sr12` and `sr13` deal with termination. The former counts the incoming links marked as `Done` – that is, no longer recording incoming messages. The latter compares this count to the number of incoming links and, if the two match, sets the phase of the snapshot to `Done`.

Rule `sr14` deals with the one Chord message that does not necessarily flow over declared topology links, `lookupResults`. As described above, when such a message arrives, it is regarded as a snapshot marker when the sender is already in a snapshot beyond the most recent snapshot on the recipient.

Finally, rule `sr15` is an example of message recording for the `returnSuccessor` message type. Recorded messages for a snapshot are timestamped and stored in a separate table per message type. Similar rules, not shown here, would treat the remaining relevant message types.

A point to note is that our rules will correctly take the consistent snaphsot of the overlay routing structure under two assumptions: (a) snapshots finish within `tSnapFreq` seconds and (b) overlay structure does not change during the snapshot, i.e., no links are added or removed.

Routing Consistency Revisited: The consistency probes of Section 3.1.4 leave room for false positives and negatives. If two of the supposedly concurrent probe lookups are held up *differently* along the way, due to transient control or network stalls, they may experience a different global overlay state from each other. At best, those probes provide hints of problems.

However, with consistent snapshots, this is not the case. One could perform Chord lookups *on the snapped overlay state*, exporting the functionality to the consistency probes, ensuring that all concurrent probe lookups experience the same global state. We review below the three original P2 Chord lookup rules [19] for reference.

```
l1 lookupResults@ReqAddr(K, SID, SAddr, E,
   RespAddr) :- node@NAddr(NID), lookup@NAddr(K,
   ReqAddr, E), bestSucc@NAddr(SAddr, SID), K in
   (NID, SID].

l2 bestLookupDist@NAddr(K, ReqAddr, E, min<D>) :-
   node@NAddr(NID), lookup@NAddr(K, ReqAddr, E),
   finger@NAddr(FPos, FID, FAddr), D := K-FID-1,
   FID in (NID, K).

l3 lookup@FAddr(K, ReqAddr, E) :- node@NAddr(NID),
   bestLookupDist@NAddr(K, ReqAddr, E, D),
   finger@NAddr(FPos, FID, FAddr), D == K-FID-1,
   FID in (NID, K).
```

To run over a consistent snapshot, we can add another set of these three rules, modified to refer to the consistent snapshot instead of the current system state.

```
l1s sLookupResults@ReqAddr(SnapID, K, SID, SAddr,
   E, RespAddr, SnapID) :- node@NAddr(NID),
   sLookup@NAddr(SnapID, K, ReqAddr, E),
   snapBestSucc@NAddr(RecID, SnapID, SAddr, SID), K
   in (NID, SID].

l2s sBestLookupDist@NAddr(SnapID, K, ReqAddr, E,
   min<D>) :- node@NAddr(NID),
   sLookup@NAddr(SnapID, K, ReqAddr, E),
   snapFingers@NAddr(RecID, SnapID, FPos, FID,
   FAddr), D := K-FID-1, FID in (NID, K).

l3s sLookup@FAddr(SnapID, K, ReqAddr, E) :-
   node@NAddr(NID), sBestLookupDist@NAddr(SnapID,
   K,ReqAddr, E, D), snapFingers@NAddr(RecID,
   SnapID, FPos, FID, FAddr), D == K-FID-1, FID in
   (NID, K).
```

Now, the consistency probe rules `cs4` and `cs5` in Section 3.1.4 can be rewritten to use the modified lookup and response rules above for a particular snapshot ID `mysnap`.

```
cs4s sLookup@SrcAddr(mysnap, K, NAddr, E) :-
   conLookup@NAddr(ProbeID, K, SrcAddr, E, T).

cs5s conRespTable@NAddr(LookID, ProbeID, SAddr) :-
   sLookupResults@NAddr(mysnap, K, SID, SAddr, E,
   Responder), conLookupTable@NAddr(LookID, E,
   ProbeID, T).
```

Note that regular lookups – that is, not issued by the consistency probe – proceed using the original rules as normal, while the consistency probe uses the snapshot-specific rules at the same time. The system has had no need to stop and restart for this to happen.

Many properties beyond consistency can be performed on thus obtained consistent snapshots to compute statistics, detect graph properties, identify vulnerabilities, etc.

3.4 Discussion

We have presented examples from a broad design space of "add-ons" that a programmer can apply to a running distributed system implemented in P2: correctness assertions, fault detectors, high-level state inspection, and reactive examination of system execution for forensic analysis. Note that while for convenience we have

presented these examples in the context of a Chord implementation, the general techniques are applicable to the implementations of a wide variety of distributed algorithms, in many cases without significantly changing the OverLog rules. This ability to repurpose rules and mix-and-match techniques on-line is a useful feature of the P2 approach.

For example, the consistency checks of Section 3.1.1 deal with topological soundness, while those in Section 3.1.2 deal with semantic soundness on top of the topology. Neither of these are limited to a simple Chord ring, but instead can be applied to a variety of structured ring-like routing graphs (for example, ones using greedy finger-based routing).

More generally, the traversal algorithms embodied in our examples have wide utility: we have shown ring traversals and chain traversals, for instance. Such traversal algorithms, combined with a per-hop soundness evaluation check, can be applied to other overlay topologies and also to execution graphs (Section 3.2), snapshot graphs (Section 3.3), or even application-defined graphs (for instance, dependencies among stored content within a distributed repository).

Furthermore combining such detectors with execution tracing can help *quantify* system performance in terms of reliability, likelihood of corruption, etc. For example, a traversal of the execution state of a lookup result (as in Section 3.2) can at each step trace back individual preconditions of the execution trace (e.g., specific successor tuples), evaluating whether they may have been dependent on routing oscillators.

An issue which is left unanswered in this paper is the user interface. We simply illustrate with examples our vision of what features might be interesting and leave the interaction details for future work. A visual user interface to represent the results of such monitoring queries (potentially with appropriate aggregation to reduce the data provided to the user) that also supports zooming into specific parts of system would be an invaluable addition to our features.

Logging, execution tracing and monitoring tasks invariably perturb the base system (they consume CPU cycles and memory), so the state accessed by these tasks may not accurately reflect the system state during normal functioning (i.e., without debugging tasks). This is a well known tradeoff between accuracy and perturbation associated with any debugging system. For our system, we currently implement certain optimizations (fixed number of execution records, only store executions that produce a valid output) to reduce the resource consumption of the logging framework. However, optimizing the resource usage of monitoring tasks and execution tracing (e.g., by executing these queries at a lower priority than system queries) to reduce the impact on the base system is left for future work.

We now attempt to quantify the performance cost of the techniques we have described.

4. PERFORMANCE COSTS

This section evaluates the cost of our approach. We quantify the performance impact of 1) making system execution traceable, 2) individual diagnostic rules, and 3) complete diagnostic programs. We use typical space, computation, and communication metrics to measure this cost. Specifically, we track the number of tuples stored in main memory (as well as the process size in bytes) for space measurements, the CPU utilization for computation measurements, and the number of messages sent for communication measurements. In this prototype, we do not export internal P2 state to debugging queries; only application state is exported.

The baseline execution of the system consists of the P2 Chord implementation [19], running on a population of 21 virtual nodes executing as individual processes. Nodes fix fingers every 10 sec, stabilize every 5 sec, and ping neighbors for liveness every 5 sec. 20 virtual nodes start and stabilize for 5 min on a lightly loaded quadruple Intel Xeon 2 GHz compute server with 8 GB of main memory. Then the 21st virtual node starts and stabilizes on an otherwise unloaded dual Intel Xeon 3.2 GHz workstation with 4 GB of main memory. Then we measure this separate 21st node to produce the results below. Unless otherwise noted, each datapoint was produced by three separate runs, and we show average and standard deviation. We do not shape the network topology for these experiments, since none of the measurements involve latencies.

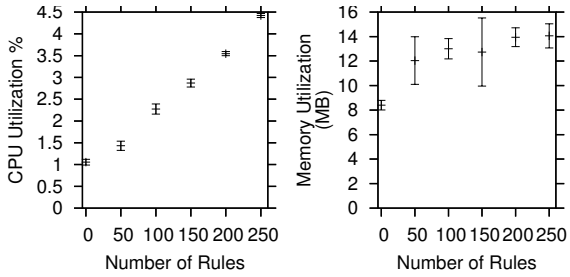

Figure 4: CPU and memory utilization (average, standard deviation) for an increasing number of periodic rules with period 1 sec.

First, we measure the cost of execution logging in the system, while running P2 Chord. We find that execution logging increases CPU utilization on a node running Chord by 40% on average, going from utilization of 0.98 to 1.38. Memory consumption grows by 66% on average, from 8 MB (standard deviation of 1 MB) to 13 MB (standard deviation of 1 MB). Note that the absolute increase in cost is minute.

We turn next to the evaluation of monitoring-rule overheads on top of the running P2 Chord system. We evaluate synthetic rules that exercise two aspects of such rules: the number of concurrently running periodic rules (which install their own timers) and the number of *piggy-backed* rules reading system state.

The first synthetic rule adds to Chord an increasing number of periodic rules (of the form

```
result@NAddr() :- periodic@NAddr(E, 1).
```

all of which trigger with period 1 sec. Figure 4 shows that CPU utilization grows roughly proportionally to the number of rule copies running, going up to 4.5% utilization with 250 copies from the baseline of 1% without extra rules. Memory consumption, mainly due to the increased rates of intermediate tuples generated, stabilizes around 70% over that of Chord itself, which operates at much slower rates than the benchmark. We show no message counts since this was a local benchmark rule producing no messages beyond those generated by Chord.

The second synthetic rule adds to Chord an increasing number of non-periodic, "piggy-back" rules that do not install their own timer. All copies are triggered by a common timer with period 1 sec. The synthetic rule for this benchmark has the form

```
result@NAddr() :- event@NAddr(),
   bestSucc@NAddr(SID, SAddr).
```

and contains a single state lookup (`bestSucc` table) in each rule copy. Figure 5 shows that CPU utilization grows roughly linearly with the number of rule copies, up to about 6% with 250 copies. State lookups are therefore costlier than private timers, as com-

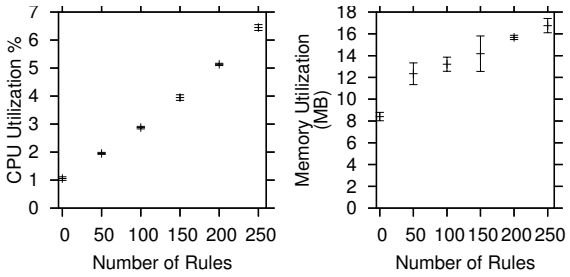

Figure 5: CPU and memory utilization (average, standard deviation) for an increasing number of piggybacked rules on a preexisting periodic event with period 1 sec.

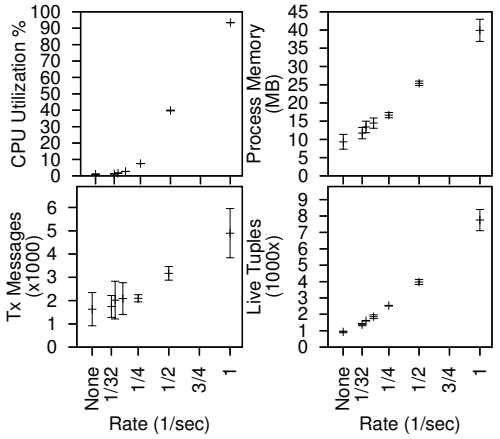

Figure 6: CPU and memory utilization for the proactive inconsistency detector, with a rate (i.e., frequency of initiation) of $1/32$ to 1 sec, alongside Chord without the detector (far left).

paring to Figure 4 shows. Memory consumption is similar to the periodic-rule benchmark.

Next we evaluate the performance overhead of two usage examples from Section 3: proactive consistency probes and consistent snapshots. Figure 6 plots overhead measurements for the proactive consistency probes, running at increasing rates ranging from once every 32 sec to once every sec. The "None" point on the x axis denotes running Chord without the consistency probes. The figure indicates that memory consumption and messages transmitted grow linearly with the rate of the probe. CPU utilization, however, grows superlinearly with the rate, as frequent probes (and their multiple lookups) contend for cycles on the initiator and all nodes in the testbed.

Figure 7 similarly plots the overheads caused by consistent snapshots taken at rates from $1/32$ to 1 snapshot per sec. Linear growth of memory consumption is much lower than with consistency probes, and so is the superlinear rate at which CPU utilization grows. Note, however, that consistent snapshots are much less taxing on the system than the many parallel lookups initiated by consistency probes for the same rates. Note also that the high rates that we measure here for consistency probes and consistent snapshots are for evaluation purposes; usually an operator would take snapshots at much lower rates than these.

5. RELATED WORK

Finding faults (bugs, anomalies, etc.) in networked systems is a large and burgeoning field. We limit ourselves here to representative examples of specific related areas to provide a context for our work.

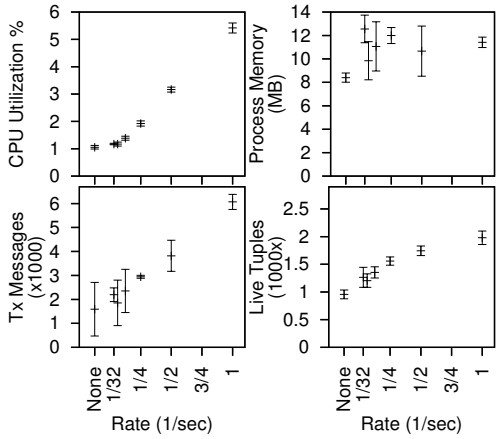

Figure 7: Consistent snapshots

Monitoring and debugging with databases. Management interfaces to networked systems often have a more-or-less relational flavor, and database techniques have been used to externally monitor, debug, and manage distributed systems [7, 8, 22, 26]. Logs are typically stored in centralized or clustered databases, and subsequently queried. P2 takes a different approach: the query processor is deeply embedded in the system, and has access to much more detailed data on execution in realtime.

Performance debugging. Recent work [1, 2, 4, 6] provides mechanisms to profile existing networked systems on-line. These approaches track the life cycle of events as they pass through different system components (e.g., an HTTP request causing a disk access, a page fault, etc.); the information gathered is then mined to find performance anomalies or bottlenecks in the system. Much of the achievement of these systems is reconstructing meaningful data- and control-flow from low level monitoring information. P2 avoids this challenge by constructing the system in the first place such that high-level structural information is retained and can be related to low-level tracing in a natural way on-line by the query processor.

Debugging configurations or intrusions. Several systems trace implications of configuration errors by inferring causality relationships [16, 20, 24] or mining for events correlated with changes in system behavior [25]. Kiciman and Subramanian [15] provide a model that constitutes a diagnosable system in the same context. P2 is narrower in scope in this respect: our examples are at present rule-based tests and metrics using detailed execution information and system reflection, rather than large-scale statistical measures and machine-learning techniques.

Distributed debuggers. Early work by Bates et al. [3] proposed a high-level debugger that compares the expected behavior of a network to the observed behavior of the implementation. The challenge lies in inferring system behavior from observable information. In P2, the high-level specification of the system facilitates this by making explicit the preconditions and expected output of each algorithmic step.

Harris [11] proposed sandboxing distributed components in a virtual machine monitor (VMM) to capture and replay external factors affecting system execution (e.g., processor status, scheduling, etc.). We achieve something close to this on-line by logging and analyzing the system's execution, since we can observe the system at a level of abstraction higher than processor flags and interrupts.

Closest to our vision is work by Lin et al. [18] on an integrated toolkit for optimizing the implementation and testing of distributed systems. The authors plan to generate code from a high level specification that can run in both simulation and real networks. It is not yet clear what high-level abstraction they will use. In P2, we compile a logic language to a dataflow graph, providing for on-line debugging of the system at multiple abstraction levels.

Recently, Geels et al. [10] proposed a technique for debugging distributed systems by logging the execution of deployed systems and replaying them deterministically for offline analysis. It also integrates *gdb* to allow source level debugging. However, due to its inherent requirement of replay at one site, it needs to ship logs to one place which is costly. Secondly, this technique is designed to find non-deterministic bugs or race conditions, rather than violations of high-level correctness conditions.

Pip [21] is a new methodology for debugging distributed systems. It works by comparing actual behavior and expected behavior to expose bugs. Programmers express expectations about a system's structure, timing and other properties. Pip logs actual behavior and provides query and visual interface for exploring the expected and unexpected behavior. It has been shown to be useful in finding bugs in existing systems. However, Pip debugging happens at a central place where all the system logs are collected and it is offline.

Debugging Languages. Crawford et al. [9] present a framework for the design of imperative debugging languages. They construct a generic debugging language, GDL, to capture the main required features for any such language. Many of the inspectional language constructs of GDL are present in OverLog, although we do not yet provide support for program *control* such as stepping or breakpointing. It is unclear what the implications of this might be in an on-line networked environment.

Deep embedded monitoring. IBM Websphere XD provides a health monitoring subsystem for the IBM Websphere [14]. A set of rules or conditions are provided which define the good health of the system. These conditions are monitored and certain actions are taken when these conditions are violated. For example, if memory usage of an application hovers above the specified threshold for some specified period, an event may trigger the restart the application. Similarly, if a server is overloaded for a specified period of time, some of its work is offloaded to an underloaded server. Typically, the conditions are performance oriented and lack the functional aspect of debugging (e.g., backtracking the causality chain to find the cause) as supported by our system.

Hollingsworth et al. [12] provides a mechanism for performance monitoring of parallel programs by guiding the search of bottlenecks in the program execution. It tries to answer three questions: why, where and when does the bottleneck appear. The system starts with a given set of hypotheses (e.g., synchronization is the bottleneck) provided by the user and depending on the execution, a hypothesis is tested and automatically refined. This helps in reducing the trace data to be collected and at the same time zoom to the specific area plagued by bottlenecks. The application needs to be recompiled to enable instrumentation at appropriate places. The focus of this work is to find only the performance bottlenecks, however we focus on finding arbitrary bugs and their root causes without need for recompilation.

Huang et al. [13] propose an architecture for adapting applications to changing runtime environments using the event-action based rules provided by the application designers. These event-action rules explain what event to monitor and what action to take when that particular event happens. This paper explains the difficulty in building an architecture which can support this feature, as there might be different adaptations which might conflict with each other, both in terms of action as well as intent and how to identify these conflicts and resolve them. Our work does not focus on adaptations but on finding problems and their causes.

6. CONCLUSIONS

The objective of this paper has been to argue that combining features of "diagnosable" systems, such as exposed state and execution transparency, with features of "diagnostic" systems, such as the ability to process distributed queries, it is possible to build distributed systems that evolve along with their fault-finding tasks in an organic and natural way. We have proposed a candidate system based on P2 that combines declarative query processing, execution logging, and on-line execution tracing. Finally, we have demonstrated and measured the performance of a broad range of fault-finding tasks, both local and distributed in scope, some simple and others as sophisticated as complex distributed algorithms, to explore the power and flexibility of our approach.

Many challenges remain ahead for our work. We have extended our execution logging facilities to the high-level declarative rule-based execution abstraction of OverLog, the logic language in which P2 applications are specified. However, faults can frequently be found at lower-level abstractions, such as the dataflow graph abstraction on which P2 applications are executed in practice. Extending the language to enable diagnostic specifications at lower-level abstractions is work in progress and may inch away from the current logic language. Furthermore, unlike rule-level diagnostics, dataflow-level diagnostics must be specified with regards to a "moving target" lower representation, which may be subject to transparent optimizations. Exposing useful state for diagnosis while still allowing optimization clearly requires careful engineering.

Perhaps more importantly, this work will have to face a grander challenge of scope. Beyond the mechanistic applications we have described here, fault finding typically extends to the very large (anomaly detection in large distributed populations), the very small (scheduling decisions or timing issues), and the very complex (intrusion detection and misbehavior). P2's query-processing pedigree suggests that large-scale statistical processing may be possible. Prioritized execution of debugging rules may allow the unperturbed observation of sensitive, low-level scheduling artifacts. And the fine-grained dataflow element building blocks of P2 may lend themselves well to verifiable audit trails that can be inspected for undeniable inconsistencies leading to detection of misbehavior. Pursuing all of these exciting possibilities is the subject of our ongoing work.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd Karsten Schwan and the anonymous reviewers for their comments and suggestions. We are also indebted to Boon Thau Loo and Tyson Condie for their help with the P2 codebase and experimental harness, as well as Joseph M. Hellerstein and Lakshminarayanan Subramanian for their thoughtful comments on drafts of this paper.

8. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, Oct. 2003.

[2] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, Dec. 2004.

[3] P. Bates, J. Wileden, and V. Lesser. A Debugging Tool for Distributed Systems. In *Proceedings of the Second Annual Phoenix Conference on Computers and Communications*, Phoenix, AZ, USA, 1983.

[4] A. Chanda, K. Elmeleegy, A. Cox, and W. Zwaenepoel. Causeway: System Support for Controlling and Analyzing the Execution of Distributed Programs. In *Proceedings of USENIX Hot Topics in Operating System (HotOS)*, Santa Fe, NM, USA, June 2005.

[5] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[6] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, Mar. 2004.

[7] L. Conradie and M.-A. Mountzia. A Relational Model for Distributed Systems Monitoring using Flexible agents. In *Proceedings of IEEE Workshop on Services in Distributed and Networked Environments (SDNE)*, Hong Kong, 1996.

[8] M. Consens, M. Hasan, and A. Mendelzon. Using Hy+ for Network Management and Distributed Debugging. In *Proceedings of Centre for Advanced Studies on Collaborative research: software engineering*, pages 450–471, Toronto, Ontario, Canada, Nov. 1993.

[9] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee. Semantic Issues in the Design of Languages for Debugging. *Computer Languages*, 21(1):17–37, 1995.

[10] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, USA, May 2006.

[11] T. L. Harris. Dependable Software Needs Pervasive Debugging (Extended Abstract). In *Proceedings of ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[12] J. Hollingsworth and B. Miller. Dynamic Control of Performance Monitoring of Large Scale Parallel Systems. In *Proceedings of Super Computing (SC)*, Tokyo, Japan, July 1993.

[13] A.-C. Huang and P. Steenkiste. Building Self-adapting Services Using Service-specific Knowledge. In *Proceedings of IEEE High Performance Distributed Computing (HPDC)*, Research Triangle Park, NC, USA, July 2005.

[14] IBM Websphere XD. `http://www-306.ibm.com/common/ssi/fcgi-bin/ssialias?infotype=an&subtype=ca&htmlfid=897/ENUS206-010`, Jan. 2006.

[15] E. Kiciman and L. Subramanian. A Root Cause Localization Model for Large Scale Systems. In *Proceedings of USENIX Hot Topics On Dependability (HotDep)*, Yokohama, Japan, June 2005.

[16] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, Oct. 2003.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transaction on Computer Systems*, 18(3), 2000.

[18] S. Lin, A. Pan, and Z. Zhang. WiDS: an Integrated Toolkit for Distributed System Developement. In *Proceedings of USENIX Hot Topics in Operating System (HotOS)*, Santa Fe, NM, USA, June 2005.

[19] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, Oct. 2005.

[20] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of ACM Special Interest Group On Data Communications (SIGCOMM)*, Pittsburg, PA, USA, Aug. 2002.

[21] P. Reynolds, J. L. Biener, J. C. Mogul, M. A. Shah, C. Killian, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, May 2006.

[22] R. Snodgrass. A Relations Approach to Monitoring Complex Systems. *IEEE Transactions on Computer Systems*, 6(2):157–196, 1988.

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions of Networking*, 11(1):17–32, 2003.

[24] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, Dec. 2004.

[25] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, Dec. 2004.

[26] O. Wolfson, S. Sengupta, and Y. Yemini. Managing Communication Networks by Monitoring Databases. *IEEE Transactions on Software Engineering*, 17(9):944–953, 1991.