Fault Management in Distributed Systems

WPE-II Written Report

Wenchao Zhou

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
wenchaoz@seas.upenn.edu

January 05, 2010

Abstract

In the past decade, distributed systems have rapidly evolved, from simple client/server applications in local area networks, to Internet-scale peer-to-peer networks and large-scale cloud platforms deployed on tens of thousands of nodes across multiple administrative domains and geographical areas. Despite of the growing popularity and interests, designing and implementing distributed systems remains challenging, due to their ever-increasing scales and the complexity and unpredictability of the system executions.

Fault management strengthens the robustness and security of distributed systems, by detecting malfunctions or violations of desired properties, diagnosing the root causes and maintaining verifiable evidences to demonstrate the diagnosis results. While its importance is well recognized, fault management in distributed systems, on the other hand, is notoriously difficult. To address the problem, various mechanisms and systems have been proposed in the past few years. In this report, we present a survey of these mechanisms and systems, and taxonomize them according to the techniques adopted and their application domains. Based on four representative systems (Pip, Friday, PeerReview and TrInc), we discuss various aspects of fault management, including fault detection, fault diagnosis and evidence generation. Their strength, limitation and application domains are evaluated and compared in detail.

Contents

1 Introduction

In the past decade, distributed systems have been rapidly evolved, from simple client/server applications in local area networks, to large-scale cloud platforms and Internet-scale P2P systems deployed on tens of thousands of nodes across administrative domains and geographical areas. Despite of the growing popularity and interests, designing and implementing these systems remains challenging, due to their ever-increasing scales as well as the complexity and unpredictability of the system executions. For instance, participating nodes may have heterogeneous performance and be connected by communication networks with unpredictable delays and losses; nodes may join and leave the system arbitrarily. Moreover, distributed systems, especially those Internet-scale P2P systems, are known to be vulnerable to security threats, where malicious adversaries may intentionally deviate the systems from their expected behaviors.

Fault management provides avenues to detect malfunctions or violations of desired properties, and diagnose the root causes which may be bugs in system designs and implementation or malicious behaviors from compromised nodes. In addition, verifiable evidences are also maintained to attest the diagnosis. It is well recognized that fault management is an integral part for developing robust and secure distributed systems. While its importance is recognized, fault management of distributed systems is notoriously difficult, due to the complexity and unpredictability of target systems.

A common practice for debugging distributed systems relies on logs generated by manually inserting *prinf* statements at various implementation points. Systems designers analyze the logs by inspecting them manually or with ad-hoc application-specific programs. This approach is feasible when the scale of the target system is small and bugs are apparent. However, with the ever-increasing scales and the complexity of distributed systems, logs may become overwhelmingly large, which makes this approach labor-intensive and error-prone.

Over the past few years, there have been intensive research activities exploring effective and efficient automation of fault management. Various techniques have been proposed for detecting faults in distributed systems. Based on how the expected behavior of a target system is defined, one may check the system against specified properties (or invariants) [22, 19, 18], its reference implementation [12], or abstract state machine [20, 14, 27]. Once the faults are detected, one may further diagnose the system to track the root causes using deterministic replay [9, 19, 14, 27], log-based causality analysis [22, 18, 8] or statistical inference [2, 5]. In addition, references [12, 16] also propose mechanisms, particularly in untrusted environments, for generating evidences. In this paper, we are going to provide a survey of the mechanisms and systems proposed for fault management of distributed systems.

The rest of the paper is organized as follows. We start with problem statement in Section 2, by defining faults in distributed systems and introducing the progressing steps of fault management. Section 3 provides an taxonomy of the mechanisms and systems proposed for fault management, according to the techniques adopted and their application domains. We then present four representative systems that cover different aspects of fault management: we discuss Pip in Section 4, Friday in Section 5, PeerReview in Section 6, and TrInc in Section 7. Finally, Section 8 reviews all the mechanisms and discusses challenges.

2 Problem Statement

To scope our survey paper, we begin first with a problem statement that provides a concise definition of the types of faults in distributed systems that we are focusing on, followed by an overview of fault management that detects, diagnoses and generates evidences for these faults.

2.1 Faults in Distributed Systems

Generally, system faults can be defined as the deviation from the expected behavior of a given system, which may affect the function, performance or both of the system. A *functional fault* is an incompliance to the specification of the system's functionality, whereas a *performance fault* manifests itself as abnormal consumption of important resources. For example, failing to organize nodes in a well-formed ring in Chord DHT [24] is a classic functional fault; high latency for Chord lookup request is a performance fault, which may indicate bugs in the system design or implementation.

Faults in distributed systems may result from two types of causes. In some situations, the running environment of a distributed system is fully trusted, i.e. all the participating nodes are trusted. In this scenario, faults are derived from bugs in the system design, implementation or configurations. We refer to this type of faults as *software errors*. On the other hand, users may have full confidence of the system design and implementation, and the faults manifest themselves as *malicious behaviors*, as a result of part of the nodes being compromised by adversaries. Compared to software errors, malicious behaviors are more difficult to be detected, as the compromised nodes may cheat about their behaviors and collude with each other. Moreover, the nodes not only behave erroneously, but also fail to behave consistently when interacting with multiple other peers (known as *equivocation* [6]).

2.2 Fault Management

There are two general techniques for developing robust and secure systems against system faults. The first technique, *fault tolerance*, aims to tolerate either hardware or software faults and continue the system operation with, perhaps, a reduction in throughput or an increase in latency. Various techniques are introduced for fault tolerance, including voting-based consensus and replications. It offers users the illusion of interacting with a single, reliable server, as exemplified by PBFT [3], Ivy [21], SUNDR [17], and Zyzzyva [15]. The second technique, *fault management*, provides users with information of existing faults as accurate and informative as possible, to enable detection of malfunctions or violations of desired properties, diagnosis of the root causes and maintenance of verifiable evidences that demonstrate the diagnosis.

While fault tolerance focuses on improving the availability and reliability of distributed systems; fault management complements it by enabling users to (1) fix design or implementation bugs to strengthen the robustness of distributed systems; and 2) detect and analyze malicious behaviors to minimize the impact on the systems.

Being a challenging and rich research topic, fault management in distributed

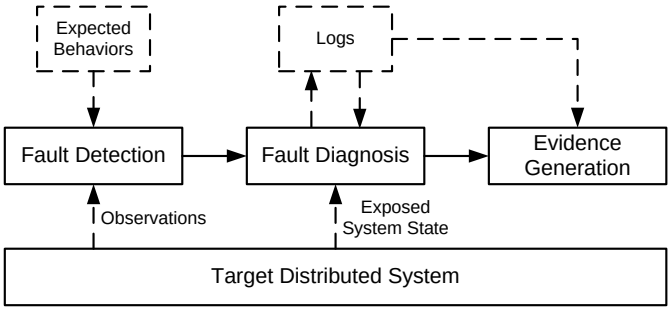

Figure 1: Progressing Steps of Fault Management in Distributed Systems

systems can be split into three progressing steps, i.e. *fault detection*, *fault diagnosis* and *evidence generation* as shown in Figure 1:

- **Fault Detection:** The first step is to monitor execution of a distributed system and check the observations against its expected behaviors, which may be encoded in the form of desired properties (or invariants), state machine model, or reference implementation. The fault is reported whenever an deviation from the expected behavior is discovered. Instead of manual inspection, automated processes are introduced.

- **Fault Diagnosis:** Once a fault is detected, additional mechanism is utilized to diagnose the system to identify the nature of the fault and track the root causes. To enable fault diagnosis, log-based mechanisms are generally required to reproduce the fault, and generate the causality paths [4]. As the embodiment of control flows and state transitions associated with executions, causality paths connect abstract heterogeneous components or system state, including values of monitored variable, read/write of files, send/receive of messages, etc. Starting from the detected fault, users can tracking back along causality paths to the root causes.

- **Evidence Generation:** Evidences can be broadly defined as a set of processed information that demonstrate the assertions drawn from fault diagnosis. After the cause of the fault is detected, the final step provides the evidence that convinces system administrators or other peers about the diagnosis results. Evidences might be used for debugging purposes, to convince software owners to make modifications accordingly. In addition, evidences enable developing accountability or reputation in sysstems where faults may be derived from malicious behaviors. Accountability, by itself, can reduce the incidence of certain faults. For instance, knowing malicious behaviors would be detected and recorded will certainly discourage adversaries to compromise the system.

  Generating evidences is trivial in a fully trusted environment, where the causality path used in fault diagnosis for tracking root cause is directly applicable to serve as the evidence. However, in an environment where multiple adversaries may compromise the system in concert, the integrity of the evidence must be enforced during its maintenance and validated when it is used for demonstration.

3 Taxonomy of Fault Management

Having presented the definition and the three progressing steps of trust management. We present a brief overview of the mechanisms and systems proposed, and taxonomize them according to the techniques adopted and their application domains (shown in Figure 2).

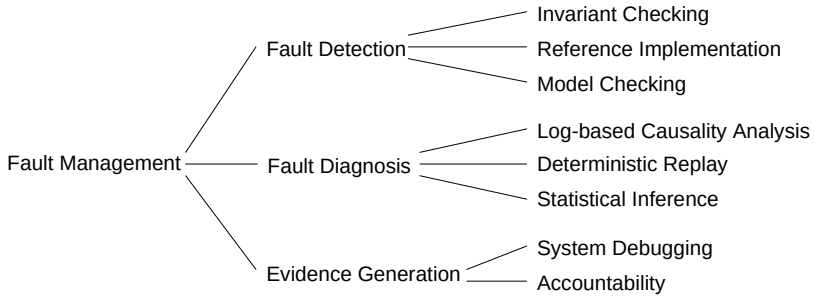

Figure 2: Classification of Fault Management

3.1 Fault Detection

According to how the expected behavior of a distributed system is encoded, different techniques are appliable for fault detection. Based on that, the mechanisms can be classified into three categories, i.e. *invariant checking*, *reference implementation*, and *model checking*.

- **Invariant Checking:** The expected behavior of a distributed system is specified as a set of desired properties (or invariant). For instance, in Chord DHT, a desired property (well-formed ring) is that each node is the predecessor of its successor. To check whether the properties hold, relevant system state are acquired by inserting additional statement (e.g. Pip [22] and P2Monitor [23]) or modifying the underlying operating system to automatically expose them (e.g. WiDS [19] and D3S [18]).

  The recorded data of system state are checked against the properties using online assertions or offline analysis. To facilitate users expressing the properties, in some fault management systems, e.g. Pip, declarative domain-specific languages are provided.

- **Reference Implementation:** Given a reference implementation, users are able to detect faults by comparing the behavior of the actual system and the one of the reference implementation[1], as exemplified by PeerReview [12]. Once all non-deterministic events, such as read/write of files and send/receive of messages, are recorded (i.e. fixed), the reference implementation's behavior exhibited in deterministic replay should be identical to the one observed in the actual system.

[1]Note that, the prerequisite of applying this technique is that the behavior of the system should be deterministic.

- **Model Checking:** The complexity of distributed systems often leads to bugs in corner cases, which can only be exposed through a particular sequence of state transitions. Model checking enables comprehensive checking that covers these corner cases. Example systems include CMC [20], MaceMC [14], and MODIST [27].

  In model checking, the behavior of a distributed system is modeled as a state machine. Starting from an initial state, a model checker performs exhaustive search by systematically enumerating all possible execution paths. The target system is steered to follow each of the execution paths to check whether it behaves correctly. Due to the state explosion problem, the scale of the exhaustive search is limited to 15-20 steps of state transitions. Random walk is introduced as a tradeoff between exhaustiveness and performance.

3.2 Fault Diagnosis

According to the techniques used for fault diagnosis, the mechanisms can be classified into three categories, i.e. *deterministic replay*, *statistical inference*, and *log-based causality analysis*.

- **Log-based Causality Analysis:** Users may insert additional statements in the source code of a target system to expose desired system state. For instance, Pip [22] logs path instances started from outside inputs; XTrace [8] maintains task trees. On the other hand, D3S [18], instead of annotating source code, modifies operating systems to allow automatic injection of state exposers and predicate checkers. Based on the logs and snapshots of system state, with the support of visualization tools and query engines, users are enabled to reason about the causality paths that traceback to the root causes.

- **Deterministic Replay:** At runtime, distributed systems record all non-deterministic events. Once a system fault is detected, users can perform deterministic replay to faithfully reproduce the fault (e.g. liblog [10], Friday [9], WiDS [19], and MaceMC [14]). Diagnosis is performed by inspecting how system state progress towards the fault, similar to how GDB is utilized for single-node applications.

  To facilitate diagnosis, users are enabled to monitor events in the replayed system based on data and control flow, leveraged by watchpoints and breakpoints respectively. In some systems, e.g. Friday, on top of symbolic low-level debugging, diagnosis of high-level faults, such as violation of global distributed properties, is also supported, by enabling users to attach arbitrary python commands to distributed watchpoints and breakpoints.

- **Statistical Inference:** A target system is modeled as a set of system components, where no knowledge of the components is required (i.e. the components are treated as black-boxes). As execution profiles, the system state recorded at runtime are typically in the form of path instances consisting of the used system components. Data mining techniques, such as clustering, are used to correlate the detected faults and correct executions to determine which components are most likely to be faulty. Magpie [2]

and Pinpoint [5] are representative examples of the systems that adopt such mechanism.

3.3 Evidence Generation

Based on the application domains, the mechanisms for evidence generation can be classified into two categories, i.e. *software debugging*, and *accountability*.

- **Software Debugging:** When targeting software errors, as the runtime environment is considered fully trusted, there is no need to question the integrity of logs. The evidence generation is thus straightforward: the causality paths that traceback to the root causes of the faults are directly applicable to serve as the evidence. Such systems include Pip [22], D3S [18], XTrace [8], WiDS [19] and MaceMC [14].
- **Accountability:** Performed in totally untrusted environments, the integrity of evidences has to be enforced. Tamper-evident logs are introduced to prevent modifications on history from un-authorized peers. In addition, equivocation, i.e. making conflicting statements to different nodes, should also be prevented by introducing additional protocols to allow users to compare received statements (e.g. the consistency protocol in PeerReview [12]).

  Enhanced with attestation-based trusted hardware, e.g. TrInc [16], a simplified version of Attested Append-only Memory (A2M) [6], equivocation can be eliminated in distributed system, resulting in a significant reduction in performance overhead.

3.4 Summary

Steps	Categories	Systems
Detection	Invariant Checking	**Pip** [22], WiDS [19], D3S [18], P2Monitor [23]
	Reference Implementation	**PeerReview** [12]
	Model Checking	CMC [20], MaceMC [14], MODIST [27]
Diagnosis	Log-based Causality Analysis	**Pip** [22], D3S [18], XTrace [8]
	Deterministic Replay	liglog [10], **Friday** [9], WiDS [19], MaceMC [14]
	Statistical Inference	Magpie [2], Pinpoint [5]
Evidence	Software Debugging	**Pip** [22], D3S [18], XTrace [8], MaceMC [14]
	Accountability	**PeerReview** [12], **TrInc** [16]

Table 1: Taxonomy of Fault Management

In Table 1, we conclude the classification of the mechanisms and systems according to our proposed taxonomy. It is not feasible to cover all the related literatures in fault management of distributed systems. We are going to selectively focus on some representative ones (shown in bold in Table 1), which include: (1) Pip, an infrastructure for comparing actual and expected behaviors to expose faults; (2) Friday, a system for debugging distributed applications based on deterministic replay; (3) PeerReview, a comprehensive system that provides accountability in distributed systems; and (4) TrInc, a trusted hardware for preventing equivocations. These systems cover most of the categories in the taxonomy.

4 Pip: Actual Behavior against Expectations

Pip provides automatic checking of the behavior of a distributed system against a programmer's expectation about the system's communication structure, timing and resource consumption. It generally targets three broad types of users, including original developers for debugging their own system; secondary developers for learning about an existing system; and system maintainers for monitoring a system.

4.1 Overview

Behavior Model: Pip models application behaviors as a collection of path instances, each of which consists of an ordered series of timestamped events on one or more hosts. These path instances encode the causalities during the execution of the system. Pip classifies the events into three types: i.e. *tasks* – intervals of processing with a start and an end; *messages* – communication between hosts or threads; and *notices* – strings with a timestamp which are essentially logs entries. To illustrate, Figure 3 shows a example path instance which reflects the execution of an http request. The behavior model adopted in Pip is naturally suited to a wide range of distributed applications, especially event-based applications.

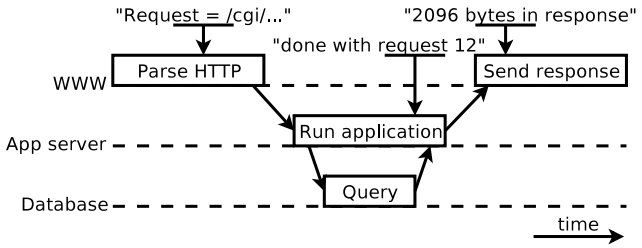

Figure 3: An Example Path Instance in Pip

Workflow: Pip is a suite of programs that gather, check, and display behavior of distributed systems. Figure 4 shows the architectural overview of Pip. Applications are annotated with log events and resource measurements, which will be reconciled into path instances. The path instances are then checked against the programmers' expectations, which are written in declarative languages, to detect unexpected behaviors.

4.1.1 Annotation

To annotate an application, the access of the source code of the application is required. A number of source code annotations will manually injected to the source code, to indicate which path is being handled, the beginning and end of interesting tasks, and the transmission and receipt of messages. The annotated applications are then linked against Pip's annotation library, to generate events and resource measurements during there execution, which will be logged into local trace files. After the execution terminates, these trace files are gathered

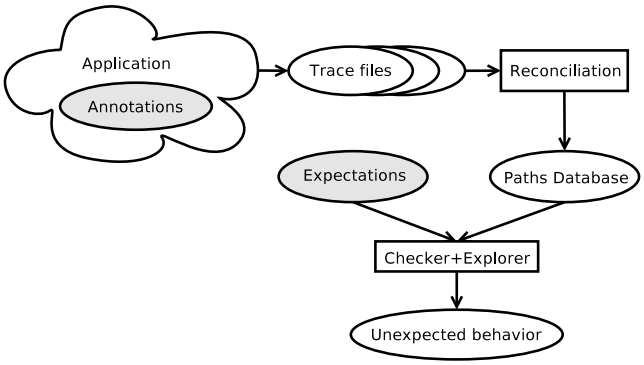

Figure 4: Architectural Overview of Pip

to a centralized location. These trace files are then processed by a *reconciler*, which pairs start- and end-task and message-send and message-receive events. The reconciled events are written into a database as path instances, which will be used as input for expectation checks.

4.1.2 Invariant Checking

To enable easy expression of the expected behavior of a distributed system, Pip allows programmers to express their expectations in a domain-specific declarative language. Expectations are categorized into two types, i.e. *recognizers* and *aggregates*.

Recognizers are descriptions of expected structural or performance behaviors of path instances. They can be further classified into *validators* and *invalidators* which define valid and invalid sequences of events respectively. A path instance is regarded valid if it matches at least one validator and no invalidator. Formally, the user-specified recognizers define a non-deterministic finite-state machine (NFS). For each path instance, the expectation checker verifies whether the path instance can be accepted by the NFS.

The results of checking path instances against recognizers are fed to the aggregates, which are essentially assertions about the properties of sets of path instances. For instance, an aggregate might state expected properties related some specific quantities. Any violation of the assertions indicates a fault.

4.1.3 Log-based Diagnosis

Once faults are detected, fault diagnosis is leveraged to identify their root causes, by analyzing path instance logged in Pip which reveals a series of causal relationship. Pip provides an interactive GUI environment that displays causal structure, sets of validated and invalidated path instances, and resource graphs for tasks or paths. This visualization toolkit enables programmers to study most aspects of application behaviors and track back to root causes of faults. In addition, Pip stores all of the path instances in an SQL database, so that programmers may further explore the logs using queries, to facilitate the debugging of the target system.

4.1.4 Evidence Generation

As the runtime environment is regarded as fully trusted. Evidence generation in Pip is straightforward. The path instances that trace to the root causes are sufficient to demonstrate the diagnosis results, for the purpose of software debugging.

4.2 Evaluation

According to the taxonomy presented in Section 3. Pip adopts invariant checking for fault detection, and log-based causality analysis for fault diagnosis. We summarize the strengths and limitations of Pip as follows:

Strengths:

- Programmers are enabled to specify expectations in a clean and declarative language. The expectations essentially define an NFS, where checking the validity of a path instance can be reduced to classic problem of deciding whether a word is in a regular language. This also makes possible to automatically generate the description of actual behaviors based on a set of path instances.

- Pip provides APIs in its annotation library for annotating resource measurements, thus enabling programmers to reason about performance behaviors. This allows Pip to detect performance faults, in additional of structural faults.

Limitations:

- Since the mechanism used for fault detection in Pip is based on invariant checking, programmers have to apriori define the events and other system state to be logged. Thus prior knowledge of the target system is presumed. Improper settings may leads to false negatives or false positives.

- Enabled by the path instances collected at runtime, Pip uses log-based causality analysis for fault diagnosis, it intrinsically limits the flexibility of diagnosis, as the diagnosis largely depends on the comprehensiveness of the logs. Logs of overwhelmingly large sizes may severely affect the performance of the target system. On the other hand, if a certain system state needed in diagnosis is not included in the logs, modification of annotations and fresh runs of the target system will be needed.

- Extracting path instances requires access to the source code of the target system, which might not be always available. In addition, understanding of the source code is critical for annotating source code. For complex systems, this process is non-trivial, and may become erroneous.

- The behavior model adopted in Pip is based on individual path instances. So is the fault detection mechanism. It lacks of a global perspective: the ability of reasoning about a global condition of system state at a given time is missing in Pip. Therefore, it is hard to check high-level global distributed properties, such verifying properties defined for mutual exclusion.

5 Friday: Global Comprehension for Distributed Replay

Unlike the prior system Pip that is primarily based on log-based analysis, the second system Friday, adopts a different approach for fault diagnosis based deterministic replay. Friday debugs distributed systems with low-level symbolic debuggers complemented with distributed watchpoints and breakpoints. To enable reasoning about high-level distributed conditions and actions, Friday further allows programmers to view and manipulate system state at any replayed node using arbitrary python commands.

5.1 Overview

Deterministic Replay: In Friday, liblog [10] is leveraged to deterministically and consistently replay the execution of a distributed application. To achieve this, each application should records all non-deterministic system calls, messages, and node failures to a local log. These logged information will be sufficient to replay the execution following the same code path.

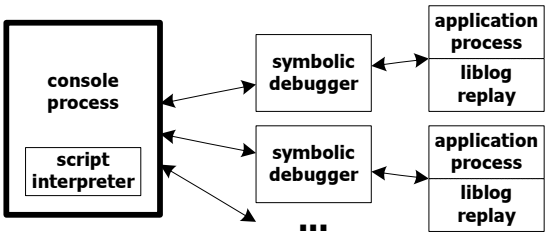

Figure 5: Architectural Overview of Friday

Architectural Overview: Figure 5 presents the architectural overview of Friday. As the interface for users interacting with Friday, a central debugging console is provided, which is connected to replay processes, each of which runs an instance of a traditional symbolic debugger such as GDB. The information provided by liblog may be overwhelmingly large, making difficult to track down the root cause of a particular problem. To mitigate this problem, Friday introduces distributed watchpoints and breakpoints to ease the search. On top of it, users are enabled to write arbitrary commands to view and manipulate system state.

5.1.1 Distributed Watchpoints and Breakpoints

An important feature supported by Friday is distributed watchpoints and breakpoints. Friday extends the functionality of watchpoints to monitor variable and expressions from multiple nodes in the replayed distributed application. It allows users to refer to a variable on all nodes (indicated by `watch <variable>`), a single node (indicated by `watch <node ID> <variable>`) or a specific set of nodes (indicated by `watch [<node ID, ...>] <variable>`). For instance, `watch [@4] srv.successor` indicates a watchpoint should be added

for the variable `srv.successor` located at node 4. Similarly, breakpoints are also extended. Programmers can install breakpoints on one, several or all replayed nodes.

Distributed watchpoints and breakpoints are implemented by setting local instances on each replay process and mapping their addresses to a global identifier. While distributed breakpoints are implemented directly based on GDB breakpoints, in contrast, Friday implements its own mechanism for local watchpoints. The memory page that maintains the watched variable is set to write-protected. When the variable is updated, the ensuing SEGV signal is intercepted, leading Friday to unprotect the memory page and complete the update.

5.1.2 Command Support

Another crucial feature of Friday is the ability to view and manipulate the distributed state of replayed application using python commands in an interactive or automated manner. Interactive commands are passed directly to the debugger processes, whereas automated commands are triggered by watchpoints or breakpoints.

In particular, the commands may involve four types of system state: 1) Friday's own debugging state, for gathering statistics or modeling global application state; 2) state inside replay processes, for access to the state of the target system; 3) the metavariable that maintains the node ID where a watchpoint or breakpoint is triggered; and 4) the logical time kept by the Lamport clock or real time recorded in logs.

```
watch srv.successor
  command
    if srv.node.addr !=
      @nodesbyID[srv.successor](srv.predecessor) :
      print "Node srv.node.addrs succesor "
            "link is not symmetric."
    end

watch srv.predecessor
  command
    if srv.node.addr !=
      @nodesByID[srv.predecessor](srv.successor) :
      print "Node srv.node.addrs predecessor "
            "link is not symmetric."
    end
```

Figure 6: Commands for Checking Whether a Chord Ring is Well-formed

As an example, the two sets of commands shown in Figure 6 define a high-level distributed property, namely a Chord should be well-formed, by checking two conditions: (1) each node should be the predecessor of its successor; and (2) each node should be the successor of its predecessor.

5.2 Evaluation

According to the taxonomy presented in Section 3. Friday adopts deterministic replay for fault diagnosis. We summarize the strength and limitation of Friday of follows:

Strengths:

- Friday is the first replay-based debugging system for unmodified distributed applications at the fine granularity of source symbols. In addition, it provides the ability to write arbitrary commands in an interactive or automated approach for tracking high-level distributed conditions and actions.
- The replay-based nature of Friday allows programmers to refine checks after repeated replays without recompilation and fresh log extractions. Programmers can dynamically select the system state to be monitored at runtime of the replay.

Limitations:

- Friday has large storage requirement for logs, since all non-deterministic events need to be recorded. The increase in the complexity of target systems and their scales will lead to significant increase of the log size. Furthermore, the logs are collected at a centralized location, the aggregate size of the logs could be prohibitively large.
- Performing replay starting from an intermediate state is not supported. Friday have to replay the application from the beginning. This limitation might be addressed by introducing static snapshot of the application. However, defining snapshot of an application is difficult by itself for complex systems.
- The deterministic replay is performed at a centralized location, which requires all information to be transmitted to that node. In addition, centralized replay also demands significant amount of CPU, memory and storage resources, leading to limits on the scalability.

6 PeerReview: Practical Accountability for Distributed Systems

Pip and Friday manage faults derived from bugs in software design or implementation, where the runtime environment is assumed to be completely trusted. In contrary, PeerReview is deployed in an untrusted environment, and focuses on faults caused by malicious behaviors from compromised nodes.

PeerReview provides accountability in distributed systems. It maintains a tamper-evident record that provides irrefutable evidence for all nodes' actions. Based on the logs, PeerReview ensures that an observable fault is eventually detected and that a correct node can defend itself against any false accusations.

PeerReview is widely applicable to various applications, ranging from detecting unexpected in interdomain routing [11], developing secure network coordinates [25], to auditing P2P distributed virtual environments.

6.1 Overview

System Model: In PeerReview, each node i is modeled as a deterministic state machine S_i, a detector module D_i and an application A_i, as shown in Figure 7. The state machine S_i represents the behaviors that need to be checked; whereas the application A_i represents the other functionalities, and the detector D_i implements PeerReview.

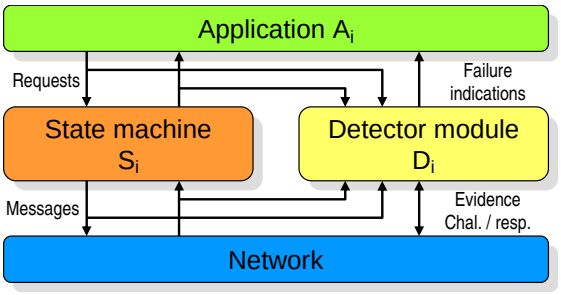

Figure 7: Architectural Model of Node i

PeerReview Design: PeerReview assigns each node i a set of *witnesses* (annotated as $w(i)$), nodes that periodically check the correctness of node i and collect verifiable evidences. Each node should keep a log that records all the messages it has exchanged with other nodes, and report this to its witnesses. Then the witnesses use a reference implementation of that node to detect faulty behaviors.

Assumptions: PeerReview makes the following assumptions: (1) Any state machine S_i is deterministic; (2) A message sent from a correct node to another is eventually received; (3) Each node can authenticate messages with signatures that cannot be forged by other nodes; and (4) Each node i has the reference implementation of all the state machines; (5) For each node i, and its witness set $w(i)$, it is assured $\{i\} \cup w(i)$ contains at least one correct node.

6.1.1 Detect Tampering and Inconsistency in Logs

Tamper-evident Logs: A log can be modeled as a linear append-only list that contains all inputs and outputs of a node. Each log entry $e_k = (s_k, t_k, c_k)$ has a sequence number s_k, a type (`SEND` or `RECV`) t_k, and the content c_k, where the sequence numbers must be strictly increasing. Additionally, each entry is attached with a recursively defined hash value $h_k = H(h_{k-1}||s_k||H(c_k))$, as shown in Figure 8(a), where H is a hash function and $||$ represents concatenation.

An *authenticator* $\alpha_k^j = \sigma_j(s_k||h_k)$ is a signed statement by node j that the log entry with sequence number s_k has hash value h_k, where σ_j means the statement is signed using j's private key. The hash chain, along with a set of authenticators, makes the log *tamper-evident*:

Once receiving α_k^j fro node j, node i can challenge j for log entries preceding e_k. If j cannot response with the corresponding entries, then i can use α_k^j as the verifiable evidence that j has tampered with its log.

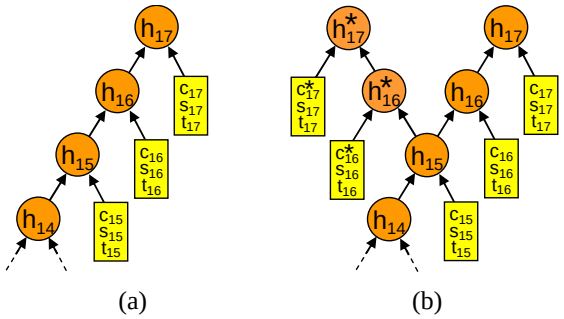

Figure 8: (a) A linear log and its hash chain, and (b) a forked log

Commitment Protocol: To ensure the log of each node is consistent with the set of messages it has exchanged with other nodes, an authenticator is included in the each message and its acknowledgement. Therefore, the sender (respectively the receiver) of a message m will obtain verifiable evidence that the receiver (respectively the sender) has logged the transmission.

Consistency Protocol: A faulty node might try to cheat by keeping multiple logs (e.g., by forking its log, as shown in Figure 8b), and sending different copies of logs to different nodes. To prevent this, when a node i receives a message m from another node j, it forwards m to j's witnesses, so they can ensure m actually appears in j's log. In practice, noting the situation where i could be a faulty accomplice of j, each witness $\omega \in w(i)$ needs to extract the message m from i's log and send it to the witness set of j. The complexity of this step is $O(|w(i)| \cdot |w(j)|)$.

6.1.2 Detect Faulty Behaviors

Each witness ω of a node i periodically checks i's most recent authenticator (say α_k^i) and challenges i to return all log entries since its last audit. The returned the new log entries are appended to ω's local copy of i's log.

Now ω has acquired logs of all the inputs and outputs of i, which enables it to replay the inputs on the reference implementation of S_i. As S_i is deterministic, the replayed outputs should be identical to the actual outputs of i. Any deviation found, ω can use α_k^i as the verifiable evidence against i, this evidence can be checked by any correct node.

6.1.3 Optimizations

When performing deterministic replay for node i, one can start from an intermediate snapshot of S_i, instead of always running from the beginning. PeerReview assumes the reference implementation can be initialized to a given snapshot of a state machine.

PeerReview has the assumption that the state machine of each node should be deterministic, however, randomization is needed in some scenarios. A recent work [1] bridges this gap by providing techniques to generate a pseudo-random

sequence. The elements of the sequence up to a given point can be proved to be correctly generated, while future values remains unpredictable.

One of the most frequently used operation in PeerReview is the access and modification of the tamper-evident log. [7] introduces tree-based tamper-evident logs to improve the efficiency of appending and querying log entries.

6.2 Evaluation

Strengths:

- Unlike Pip and Friday which handle faults derived from software errors, PeerReview targets on general Byzantine faults including those resulting from malicious behavior by compromised nodes. PeerReview provides a approach to make strong guarantees on eventual Byzantine fault detection.
- PeerReview provides general and practical accountability. Most of the assumptions made in PeerReview are realistic and easy to met. With little presumption on the target system, it is generally applicable to a wide range of protocols, e.g., network filesystems, peer-to-peer system and overlay multicast systems.

Limitations:

- PeerReview assumes the availability of the reference implementation of all other nodes, which may not be necessarily true. In addition, as a performance optimization (See Section 6.1.3), PeerReview also assumes reference implementations can be initialized to an arbitrary intermediate snapshot of system state. As discussed in Section 5.2, this process is non-trivial or may not be feasible for all target systems.
- As described in Section 6.1.1, to prevent inconsistent logs, when a node i receives a message from another node j, each of the i's witness needs to forward the message to all the witnesses of j. The complexity of this process is roughly $O(W^2)$, where W is the average size of the witness sets. This may limit the scalability of PeerReview. We note that PeerReview has introduced probabilistic guarantees to mitigate this problem, which proves to be notably effective.

7 TrInc: Trusted Hardware against Equivocation

Performed in an untrusted environment, PeerReview employs several mechanisms to ensure the integrity of logs. One of them (the consistency protocol) is used to prevent nodes sending inconsistent logs to other nodes. In general, making inconsistent statements (e.g. logs in the case of PeerReview) to others is known as *equivocation*. Consisting fundamentally of only a non-decreasing counter and a key, TrInc (trusted incrementer) provides a new primitive – unique, once-in-a-lifetime attestations to combating equivocations.

7.1 Overview

Equivocation: Equivocation is a necessary property for Byzantine faults and many other forms of fraud is equivocation. A common approach to prevent equivocation requires communication with each other or with a third party [12, 13], so they can learn about all the distinct statements. However, this additional communication can become bottleneck for P2P systems, as we discussed in Section 6.1.1. TrInc minimizes both communication overhead and the number of non-faulty nodes required. In addition, it is designed to be small, simple and cheap for deployment in distributed systems.

Architectural Overview: TrInc introduces *trinket* a trusted hardware to generate and verify attestations. For each data m, an attestation is attached, which binds m to a certain value of a counter, and ensures no other data will be bound to the same value.

7.1.1 TrInc State

Figure 9 describes the state of a trinket. Each trinket is bound to a unique identifier I and a public/private key pair (K_{pub}, K_{priv}), which are configured by its manufacturer. An attestation A is also provided to prove the validity of the trinket. These four state are permanently maintain in the trinket.

A trinket may include several counters; a meta-variable M maintains the number of the counters. Another global state is Q, a limited-size FIFO queue that contains the most recently generated counter attestations. When encountered a power failure, a trinket uses Q to recover its state.

A final part of a trinket's state is an array of counters, each of which maintains three state, i.e. the identifier of the counter i, the current value of the counter c, and the key to use for attestations K.

Global state:

Notation	Meaning
K_{priv}	Unique private key of this trinket
K_{pub}	Public key corresponding to K_{priv}
I	ID of this trinket, the hash of K_{pub}
$\mathcal{A}$	Attestation of this trinket's validity
M	Meta-counter: the number of counters this trinket has created so far
Q	Limited-size FIFO queue containing the most recent few counter attestations generated by this trinket

Per-counter state:

Notation	Meaning
i	Identity of this counter, i.e., the value of M when it was created
c	Current value of the counter (starts at 0, monotonically non-decreasing)
K	Key to use for attestations, or 0 if K_{priv} should be used instead

Figure 9: State of a trinket

7.1.2 TrInc Operations

Generating Attestation: The core of TrInc's operations is `Attest`, which takes three parameters: i, c', and h. Here i is the identifier of the counter; c' is the requested new value for that counter; and h is a hash of the message m to which the user wishes to bind the counter value.

Since the counter is meant to be non-decreasing, `Attest` first examines the validity of the requested new value, by check whether it is not smaller than the current value of the counter, i.e. $c \leq c'$. If the new value is valid, the attestation a is generated as $a \leftarrow \langle I, i, c, c', h \rangle_K$, where $\langle X \rangle_K$ indicates using key K to encrypt the content X. Then the value of the counter is updated to c'. `Attest` finishes by returning the generated attestation a.

Verifying Attestation: To verify an attestation $a = \langle I, i, c, c', h \rangle_K$, a trinket verifies the authenticity of a with the corresponding key. If the attestation is signed using asymmetric cryptographic scheme, the corresponding public key is used; otherwise, for symmetric cryptographic scheme, the symmetric key K should be exchanged aprioi and is used to verify the authenticity of a.

7.2 Applications

Trusted Logs with TrInc: A trusted log are maintained by augmenting an append-only list of logs with an attestation attached to each of the log entries. When a log entry e with a sequence number s is to be appended, an attestation a is generated as $a = \mathit{Attest}(i, s, h(e))$, where i is the identifier of the trinket. The triplet (s, e, a) is appended to the end of the log. Given a trusted log, any trinket can check whether it is tampered, by verifying the attestations attached to the log entries. Without knowing the key of the counter, it is practically impossible for a malicious user/node to tamper log entries without being noticed. In addition, when users lookup for the log entry with a particular sequence number s, only one log entry may match s, thus equivocation is protected.

Simplifying PeerReview with TrInc: As described above, TrInc can easily supply a trust log without the assistance of a witness set. TrInc-augmented PeerReview includes such trusted logs. Whenever a message is sent or received, the node should log that message with an attestation from its trinket. On the receiving end, a node only processes a received message if it is accompanied by an attestation that the message has been logged by the sender's trinket.

Enabled with TrInc, PeerReview nodes no longer need to verify a hash chain of log entries. The fact that TrInc signs the messages is sufficient. Furthermore, the expensive consistency protocol for preventing equivocations can also be removed. In PeerReview, if a node i sends an authenticator to another node j, then j's witnesses should forward it to i's witnesses. This is not necessary in TrInc-augmented PeerReview, because equivocation is make impossible by using the TrInc-enabled trusted log. Therefore, the communication overhead is significantly reduced.

7.3 Evaluation

Strengths:

- Fundamentally, TrInc consists only of a non-decreasing counter and a key.

The small size and simple semantics make it easy to deploy, as demonstrated by implementing it on currently available trusted hardware.

- With TrInc eliminating equivocations, a wide range of applications can be significantly simplified. For instance, the heavy consistency protocol used in PeerReview is avoided, which results in a significant reduction in communication overhead. Without the power of equivocation, the number of participants to tolerate f Byzantine faults is reduced from $3f + 1$ to $2f + 1$.

Limitations:

- There lacks a suitable trusted hardware to deploy TrInc. Though the core functional elements of TrInc are included in the Trusted Platform Module (TPM), it proves TPM is not a good option for deploying TrInc. TrInc uses the trusted hardware in a fundamentally different way than what these hardware are designed for. While the trusted hardware are designed to perform few operations during a machine's boot cycle, TrInc uses them much more frequently during operation. The limited performance of trusted hardware results in significant overhead.

8 Discussion

In this section, we conclude our survey paper with comparisons of the different techniques proposed for fault management of distributed systems, followed by a discussion of the challenges.

8.1 Comparison

In previous sections, we have discussed representative mechanisms and systems for fault management of distributed systems, including Pip in Section 4, Friday in Section 5, PeerReview in Section 6 and TrInc in Section 7. These systems, together with the taxonomy presented in Section 3, show that programmers indeed have a variety of options for fault detection, fault diagnosis, and evidence generation.

Incorporated with the evaluations of individual systems presented in previous sections, we present a comparison of the different techniques for fault detection and fault diagnosis. We omit the comparison of the techniques proposed for evidence generation, as these techniques are applied to disjoining application domains, rendering the comparison less interesting.

8.1.1 Comparison of Fault Detection Techniques

- **Invariant Checking:** For invariant checking, programmers are allowed to specify the system state to be exposed, thus they have the ability to control of the size of logs. On the other hand, due to this configuration is performed aprioi, prior knowledge of the target system is presumed. In addition, improper settings may leads to false positives and false negatives.

  Though Pip is not suitable for checking high-level distributed properties, invariant checking, in general, is a promising technique for this purpose.

- **Reference Implementation:** Fault detection with reference implementation is straightforward and easy to use. However, this approach is limited to detecting faults resulting from malicious behaviors. For software error, it might be difficult to obtain a reference implementation that operates fully correct.

- **Model Checking:** Invariant checking and reference implementation only detect faults existed in running executions, yet cannot guarantee the correctness of a system in all possible execution paths. Model checking performs more comprehensive detections using exhaustive search. In addition, leveraged by heuristic random walk, model checking is applicable to detect liveness faults with high confidence.

  State explosion problem, though may be mitigated by adopting optimizations and random walk, is a daunting problem of model checking. In addition, the difficulty in developing the state machine of a given system raises barrier to the wide application of model checking.

8.1.2 Comparison of Fault Diagnosis Techniques

- **Log-based Causality Analysis:** While log-based causality analysis requires less comprehensive logs than deterministic replay, the flexibility of diagnosis is limited by the comprehensiveness of the logs. If a certain system state needed for diagnosis is not logged, it can only be acquired by refining the specification of the state to be exposed and rerunning the target system.

- **Deterministic Replay:** Unlike log-based causality analysis, programmers are essentially able to acquire any system state during the replay. Thus programmers can refine checks after repeated replays without recompilation and refresh log extraction.

  The drawback of deterministic replay is also obvious. The logs for enabling deterministic replay are usually significantly large. Therefore, it is expensive in terms of CPU, memory and storage resources to maintain logs and perform replays.

- **Statistical Inference:** Based on machine learning techniques such as clustering, statistical inference inevitably encounters false positive and false negatives. Thus, careful balancing between them is required.

8.2 Challenges

- **Usability:** The first challenge involves improving the usability of fault management, specifically, in the following three aspects: (1) allowing mechanisms to be applied to application written in arbitrary language (unlike MaceMC, WiDS and P2Monitor that require target system to be based on Mace, WiDS and NDlog respectively); (2) enabling fault management without manual modification of source code (unlike Pip and XTrace that inject annotations to source code); and (3) allowing users to specify expectation of system behavior in more approachable means.

- **Reasoning about Time:** The second challenge resides in improving the ability of system operators to reason about time, e.g. optimizing virtual clocks to trace real time more closely even when the distributed clocks are poorly synchronized. This could be helpful to reason about properties related concurrency and race conditions.
- **Utilization of Distributed Resources:** Most detection and diagnosis mechanisms adopt centralized approaches. For instance, Pip gathers trace files to a central location, where the reconciled path instances are maintained at a centralized database; Friday performs replay at a centralized node; model checking systems, MaceMC and MODIST, steer systems locally. To leverage better utilization of distributed resources, programmers may face a series of challenges, including consistency maintenance and failure handling in parallel analysis (e.g. DS3) and distributed replay.
- **Impact Analysis and Repair:** Given detected faults, and their root causes, a challenging research topic is how to accurately estimate their impact on the current system, and how to repair the system online without recompilation and rerun. As an alternative approach to minimize the impact of faults on a running system, a recent work [26] explores model checking techniques to predict possible system state in the future and steer systems to avoid faults.
- **Trusted Hardware Design:** A2M [6] and TrInc [16] explore a novel direction to implement Byzantine fault tolerance or detection system, by adopting the support of trusted hardware. However, today's trusted hardware is designed mainly for bootstraping software, which has few performance requirements. Developing high-performance trusted hardware that are more suitable for distributed systems is a valuable area for future work.

9 Acknowledgements

I would like to thank Prof. Insup Lee for chairing my WPE-II committee, as well as Prof. Boon Thau Loo and Andreas Haeberlen for sitting on the committee.

References

[1] M. Backes, P. Druschel, A. Haevberlen, and D. Unruh. CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS'09)*, February 2009.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[4] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI'04)*, March 2004.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, June 2002.

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

[7] S. A. Crosby and D. S. Wallach. Efficient Data Structures for Tamper-Evident Logging. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.

[9] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.

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

[11] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, April 2009.

[12] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical Accountablitiy for Distributed Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

[13] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, April 2008.

[14] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.

[15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

[16] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, April 2009.

[17] J. Li, M. N. Krohn, D. Mazieres, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, December 2004.

[18] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, April 2008.

[19] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.

[20] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[21] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[22] 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 *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[23] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using Queries for Distributed Monitoring and Forensics. In *Proceedings of the 2006 EuroSys Conference*, April 2006.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable P2P Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, August 2001.

[25] G. Wang and T. S. E. Ng. Distributed Algorithms for Stable and Secure Network Coordinates. In *Proceedings of ACM SIGMETRICS/USENIX Internet Measurement Conference (IMC'08)*, October 2008.

[26] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, April 2009.

[27] J. Yang, T. Chen, M. Wu, Z. Wu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, April 2009.