D^3S: Debugging Deployed Distributed Systems

Xuezheng Liu[†] Zhenyu Guo[†] Xi Wang[‡] Feibo Chen[¶]
Xiaochen Lian[§] Jian Tang[†] Ming Wu[†] M. Frans Kaashoek[*] Zheng Zhang[†]

[†]*Microsoft Research Asia* [‡]*Tsinghua University*
[¶]*Fudan University* [§]*Shanghai Jiaotong University* [*]*MIT CSAIL*

Abstract

Testing large-scale distributed systems is a challenge, because some errors manifest themselves only after a distributed sequence of events that involves machine and network failures. D^3S is a checker that allows developers to specify predicates on distributed properties of a deployed system, and that checks these predicates while the system is running. When D^3S finds a problem it produces the sequence of state changes that led to the problem, allowing developers to quickly find the root cause.

Developers write predicates in a simple and sequential programming style, while D^3S checks these predicates in a distributed and parallel manner to allow checking to be scalable to large systems and fault tolerant. By using binary instrumentation, D^3S works transparently with legacy systems and can change predicates to be checked at runtime. An evaluation with 5 deployed systems shows that D^3S can detect non-trivial correctness and performance bugs at runtime and with low performance overhead (less than 8%).

1 Introduction

Distributed systems are evolving rapidly from simple client/server applications to systems that are spread over many machines, and these systems are at the heart of today's Internet services. Because of their scale these systems are difficult to develop, test, and debug. These systems often have bugs that are difficult to track down, because the bugs exhibit themselves only after a certain sequence of events, typically involving machine or network failures, which are often difficult to reproduce.

The approach to debugging used in practice is for developers to insert print statements to expose local state, buffer the exposed state, and periodically send the buffers to a central machine. The developer then writes a script to parse the buffers, to order the state of each machine in a global snapshot, and to check for incorrect behavior.

This approach is effective both during development and deployment, but has some disadvantages for a developer: the developer must write code to record the state of each machine and order these states into a globally-consistent snapshot. The developer must anticipate what state to record; an implementation monitoring too much state may slow down the deployed system, while monitoring too little may miss detection of incorrect behavior. The developer may need to distribute the checking across several machines, because a central checker may be unable to keep up with a system deployed on many machines—an application we worked with produced 500$\sim$1000 KB/s of monitoring data per machine, which is a small fraction (1$\sim$2%) of the total data handled by the application, but enough monitoring data as a whole that a single machine could not keep up. Finally, the developer should have a plan to approximate a globally-consistent snapshot when some processes that are being checked fail, and should make the checking itself fault tolerant.

Although many tools have been proposed for simplifying debugging of distributed or parallel applications (see Section 7), we are unaware of a tool that removes these disadvantages. To fill that need, we propose D^3S, a tool for debugging deployed distributed systems, which automates many aspects of the manual approach, allows runtime checking to scale to large systems, and makes the checking fault tolerant.

Using D^3S, a developer writes functions that check distributed predicates. A predicate is often a distributed invariant that must hold for a component of the system or the system as a whole (e.g., "no two machines should hold the same lock exclusively"). D^3S compiles the predicates and dynamically injects the compiled libraries to the running processes of the system and additional verifier processes that check the system. After injection, the processes of the system expose their states as tuples (e.g., the locks a process holds), and stream the tuples to the verifier processes for checking. When the checking identifies a problem (e.g., two processes that hold the same lock), D^{3}S reports the problem and the sequence of state changes that led to the problem. By using binary instrumentation, D^{3}S can transparently monitor a deployed, legacy system, and developers can change predicates at runtime.

A key challenge in the design of D^{3}S is to allow the developer to express easily what properties to check, yet allow the checking to use several machines so that the developer can check large systems at runtime. A second challenge is that D^{3}S should handle failures of checking machines. A third challenge is that D^{3}S should handle failures of processes being checked—the checkers should continue running without unnecessary false negatives or positives in the checking results. Using the lock example, suppose a client acquires a lock with a lease for a certain period but then fails before releasing the lock, and after the lease expires another client acquires the lock. The predicate that checks for double acquires should not flag this case as an error. To avoid this problem, D^{3}S must handle machine failures when computing snapshots.

D^{3}S's design addresses these challenges as follows. For the first challenge, D^{3}S allows developers to organize checkers in a directed-acyclic graph, inspired by Dryad [21]. For each vertex, which represents a computation stage, developers can write a sequential C++ function for checkers of this stage; the function can reuse type declarations from the program being checked. The state tuples output by the checkers flow to the downstream vertices in the graph. During the checking, a vertex can be mapped to several verifier processes that run the checkers in parallel; in this way D^{3}S can use multiple machines to scale runtime checking to large systems. Within this framework, D^{3}S also incorporates sampling of the state being checked, and incremental checking. These features can make the checking more lightweight.

For the second challenge, D^{3}S monitors verifier processes. When one fails, D^{3}S starts a new verifier process and feeds it the input of the failed process. Because checkers are deterministic, D^{3}S can re-execute the checkers with the same input.

For the third challenge, the verifier processes remove failed processes from globally-consistent snapshots before checking the snapshots. D^{3}S uses a logical clock [24] to order the exposed state tuples, and has a well-defined notion of which processes are in the snapshot at each timestamp. For the previous lock example, D^{3}S will not report a false positive, because the lock state acquired by the first client is removed from the snapshot at the time when the second client acquires the lock.

We have implemented D^{3}S on Windows and used it to check several distributed systems. We were able to quickly find several intricate bugs in a semi-structured storage system [26], a Paxos [25] implementation, a Web search engine [37], a Chord implementation [35, 1], and a BitTorrent client [2]. We also found that the burden of writing predicate checkers for these systems was small (the largest predicate we used has 210 lines of code; others are around 100 lines), and that the overhead of runtime checking was small compared to the system being checked (the largest overhead is 8% but for most cases it is less than 1%).

The main contributions of this paper are: the model for writing predicate checkers; the runtime that allows real-time checking to scale to large systems, that constructs global snapshots to avoid unnecessary false negatives and positives, and that handles failures of checking machines; and, the evaluation with 5 distributed systems.

The rest of this paper is organized as follows. Section 2 details D^{3}S's design. Section 3 explains how D^{3}S computes global snapshots. Section 4 describes our implementation of D^{3}S on Windows. Section 5 presents an evaluation with several distributed systems. Sections 6 reports on the performance of D^{3}S. Section 7 relates D^{3}S to previous work. Section 8 summarizes our conclusions.

2 Design

We describe the design of D^{3}S (see Figure 1). D^{3}S compiles a predicate into a state-exposer library and a checking library. Using binary instrumentation, D^{3}S dynamically injects the state exposer into the running processes of the system. The state exposers produce tuples describing the current state of interest, and partitions the stream of tuples among the verifier processes. The verifying processes either run on dedicated machines or on the same machines that the system runs on. The verifying processes order tuples globally, and evaluate the predicate on snapshots of tuples. If a predicate fails, D^{3}S reports to the developer the problem and the sequence of state changes that led to the problem.

The rest of this section describes the design of D^{3}S in more detail: how developers write predicates, how D^{3}S inserts them into a deployed system, how D^{3}S allows for parallel checking and stream processing for checking predicates in a scalable and efficient manner.

2.1 Writing predicates

To illustrate the ease with which a developer can write a predicate, we will describe a predicate written in C++ that we have used to check the distributed lock service in Boxwood [29]. This distributed lock service allows clients to acquire multiple-reader single-writer locks. A lock can be held in either `Shared` or `Exclusive` mode. A critical property of the service is that the lock holders must be consistent, i.e., either there is one

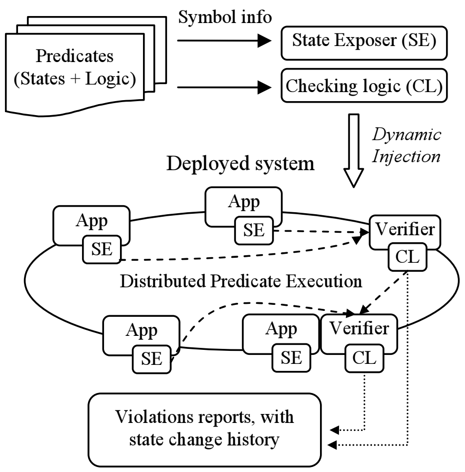


Figure 1: Overview of D^{3}S.

`Exclusive` holder and no `Shared` holders, or there is no `Exclusive` holders. Because clients cache locks locally (to reduce traffic between the clients and the lock server), only the clients know the current state of a lock.

Figure 2 shows the code that the developer writes to monitor and check the properties of Boxwood's distributed lock service. The developer organizes the predicate checking in several stages and expresses how the stages are connected in an acyclic graph; the developer describes this graph with the script part of the code. In the example there are only two stages that form a single edge with two vertices (V_0 and V_1). (Later examples in this paper have more stages.)

The vertex V_0 represents the system and the state it generates. The developer describes the state after a change as a set of tuples; in the example, each tuple has three fields of types: *client:ClientID*, *lock:LockID* and *mode:LockMode*. These types come from the header file of the lock service code, and the developer can reuse them in the script and C++ code. The tuples together express the locks and their state that a lock client is holding.

The vertex V_1 represents the computation of the lock predicate. As the script shows, V_1 takes as input the output of V_0 and generates a set of tuples, each of which has one field *conflict:LockID*. This vertex is marked as **final** to indicate it is the final stage of the checker.

The developer specifies the computation to check the predicate at vertex V_1 by writing C++ code, again reusing the types of the system being checked. In the example, the computation is the class *LockVerifier*, which is derived from the *Vertex* class and which the developer ties to V_1 using a template argument. The developer must write a method *Execute*. The D^{3}S runtime invokes this method each time it constructs a global snapshot of tuples of the type that V_0 produces for a timestamp t; how the runtime produces sequences of global snapshots is

```
# scripts   1. Describe computation graph with output type in each stage
V0: exposer → { (client: ClientID,  lock: LockID,  mode: LockMode) }
V1: V0        → { (conflict: LockID) }  as  final
#           2. Correlate state changes with monitored functions in app's code
after (ClientNode::OnLockAcquired) addtuple ($0->m_NodeID, $1, $2)
after (ClientNode::OnLockReleased) deltuple ($0->m_NodeID, $1, $2)

// C++ code for predicate in V1.
class LockVerifer : public Vertex< V1 > {
    virtual void Execute(const V0::Collection & snapshot) {
        std::map< LockID, int >   exclusive, shared;  // count the lock holders
        while ( ! snapshot.eof() ) {
            // V0::Tuple is V0's output type, i.e., (ClientID, LockID, LockMode)
            V0::Tuple t = snapshot.get_next();
            If ( t.mode == EXCLUSIVE )
                exclusive[t.lock]++;
            else   shared[t.lock]++;
        }
        // check conflicts and add to "output" member of Vertex.
        for (Iterator it = exclusive.begin();  it != exclusive.end();  ++ it)
            if ( it->value > 1 || (it->value == 1 && exist(shared, it->key) )
                output.add( V1::Tuple(it->key) );
    }
    static Key Mapping(const V0::Tuple & t) {  // map states to key space
        return t.lock;
    }
};
```

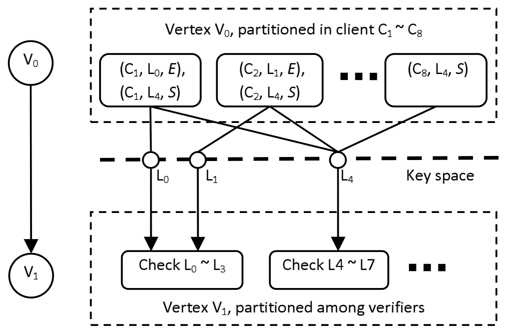


Figure 2: (a) Checking code. (b) graph and checker execution.

the topic of Section 3. In the example, *Execute* enumerates all tuples in the snapshot and tracks the number of clients holding an `Exclusive` and `Shared` lock for each lock ID. It outputs the IDs of locks that are in conflict at timestamp t.

As shown, the developer can check distributed properties by writing just sequential code that processes states in a centralized manner and reuses types from the system being checked. How the runtime transmits the state of multiple clients, collects the state of the clients into a globally-consistent snapshot, and checks them in parallel is hidden from the developer. This design achieves D^{3}S's design goals of expressiveness and simplicity.

2.2 Inserting predicates

To change what properties of a system to check, a developer can insert predicates when the system is running. The developer uses D^{3}S's compiler to generate

C++ code from the predicates for a state exposer and a checking logic module. The output of the compiler is two dynamically-linked libraries, one for each module, that can be attached to a running process.

D^{3}S then disseminates and attaches the generated state exposer to every process of the system. When loaded, the state exposer rewrites the binary modules of the process, so as to add new functions that will execute either before or after the functions to be monitored (Section 4 explains the details). These new functions construct tuples in V_0's output from memory states. For the script in Figure 2, the state exposer adds two functions after `ClientNode::OnLockAcquired` and `ClientNode::OnLockReleased`, respectively, to obtain the acquired and released lock states. These functions construct tuples of the form ($0→m_NodeID, $1, $2), in which $\$i$ is the i^{th} parameter to the original function (for member functions in C++, $0 is the "this" pointer). The state exposer will add or delete the constructed tuples in V_0's output, instructed by the **addtuple** and **deltuple** keywords. The developer is allowed to embed C++ code in the script to construct tuples for V_0 in case the script needs more than the function parameters. However, we find that in most cases (all systems we checked), exposing function parameters is sufficient to monitor state changes.

The state exposer can start outputting tuples immediately after it adds all monitoring functions. Alternatively, it can start on a certain time instructed by the developer. Due to network delay, different instances of state exposers may not start at exactly the same time. This causes no problems, because the D^{3}S verifiers will wait with running the checkers until they can construct a global snapshot.

The checking library contains the programs for vertices other than V_0. D^{3}S attaches them to all verifiers, and the verifiers start to process incoming tuples, run the checkers when a global snapshot is ready, and stream outputs to their next stages.

When a developer inserts a new predicate checker while the system is running, the checker may miss violations that are related to previous unmonitored history. For instance, in the lock example, if the verifier starts after a client has acquired a lock, the verifier does not know that the client already has the lock and a related violation may go undetected.

2.3 Dataflow graphs

More complex predicates than the lock example will have more complex dataflow graphs of verifiers. D^{3}S runs vertices to process timestamp t when the input data from upstreaming vertices are ready to construct a consistent snapshot for t. After the snapshot is processed, the output data is also labeled with t and transmitted to all downstream vertices. When all vertices has executed for t, the predicate is evaluated for t, and D^{3}S produces the checking result from the output of the *final vertex*. Vertices can work on different timestamps simultaneously in a pipeline fashion, transparently exploiting the parallelism in the predicate.

Predicates are deterministically calculated from the exposed states. When failures happen in intermediate vertices, after recovery D^{3}S can re-execute the same timestamp from the original exposed states in V_0 (V_0 can buffer exposed states of the timestamp, until the final stage finishes). This scheme allows D^{3}S to handle verifier failures easily.

2.4 Partitioned execution

The D^{3}S runtime can partition the predicate computation across multiple machines, as in Figure 2(b), with minimal guidance from the developer. Using the lock service example, to guarantee the correctness of predicate checking when the runtime partitions the computation, the tuples describing the same lock should be checked together. Similar to the Map phase in MapReduce [12], the developer expresses this requirement through the *Mapping* method, which maps output tuples to a virtual key space. The runtime then partitions the key space dynamically over several machines and runs the computation for different key ranges in parallel. Tuples mapped to a key are checked by the verifier that takes that key as input. In the example, the first and the second machine will run *Execute* for lock $0 \sim 5$ and $6 \sim 10$, respectively. Each vertex can have an independent mapping function. D^{3}S uses a default hash function when there is no mapping function in a vertex.

A notification mechanism (details in Section 4) tells verifiers the current key assignments of their downstream vertices so that verifiers can transmit outputs to the verifiers that depend on them. If a verifier fails, its responsible input range will be taken over by other remaining verifiers. By changing the assignment of key spaces to verifiers on demand, D^{3}S is free to add and remove verifiers, or re-balance the jobs on verifiers. This design achieves D^{3}S's design goals of scalability and failure tolerance.

2.5 Stream processing and sampling

Often, there are only minor variations in state between consecutive timestamps. In such cases it is inefficient to transmit and process the entire snapshots at every timestamp. For this reason D^{3}S supports stream processing, in which vertices only transmit the difference in their output compared to the last timestamp, and check the state

```
V0: exposer → { (pred: chordID,  self:  chordID,  succ: chordID) }
V1: V0       → { (sum_range_size: int) }
V2: V1       → { (range_coverage: float) } as final
before  (stabilize)  deltuple  ($0->leftID,  $0->node.id,  $0->rightID)
after   (stabilize)  addtuple  ($0->leftID,  $0->node.id,  $0->rightID)

class RangeSum : public Vertex< V1 > {
    virtual void Execute(const V0::Collection & snapshot) {
        // calculate size of key range
        int sum_range_size = 0;
        while ( ! snapshot.eof() ) {
            V0::Tuple t = snapshot.get_next();
            sum_range_size += circle_distance(t.pred, t.self);
        }
        output.add( V1::Tuple( sum_range_size ) );
    }
    // incremental evaluation on delta data
    virtual void ExecuteChange(const V0::Delta & delta) {
        // calculate delta of key range size
        int delta_range_size = 0;
        while ( ! delta.eof() ) {
            V0::Tuple t;  DeltaFlag flag;
            delta.get_next( t, flag );
            int  sign =  ( flag == DELTA_FLAG_DELETED ) ? -1 : 1;
            delta_range_size += sign * circle_distance(t.pred, t.self);
        }
        output.add( V1::Tuple( last_output + delta_range_size ) );
    }
};
class Aggregation : public Vertex< V2 > {
    virtual void Execute(const V1::Collection & snapshot) {
        // aggregate range sizes from previous vertex
        int sum = 0;
        while ( ! snapshot.eof() ) {
            sum += snapshot.get_next().sum_range_size;
        }
        output.add( V2::Tuple( sum / CIRCLE_SIZE ) );
    }
    static Key Mapping(const V1::Tuple & t) {
        return 0;   // make sure all state are transmitted to single verifier
    }
};
```

Figure 3: The predicate that checks the key range coverage among Chord nodes.

incrementally. There is an optional *ExecuteChange* function to specify the logic for incremental processing.

To illustrate the use of *ExecuteChange* and dataflow graph with more vertices, we will use Chord DHT in $i3$ service [35] as another example (Figure 3). Section 5.4 presents the checking results of this example.

We check the consistency of its key ranges among Chord nodes. Every Chord node exposes the ring information as tuples with three fields: *pred*, *self* and *succ*, which indicate the *chordID* of the node's predecessor, itself and the successor, respectively. According to the $i3$-Chord implementation, the key range assigned to the node is $[pred, self)$. The key range predicate computes the aggregate key range held by current nodes, relative to the entire ID space. In ideal case where the Chord ring is correct, this value should be 100%. Below 100% indicates "holes" while above 100% indicates overlaps in key ranges.

For the key range predicate in Chord, we use three vertices $V_0 \rightarrow V_1 \rightarrow V_2$. V_0 represents state exposer that outputs states from Chord nodes. Every state represents the neighborhood of a Chord node in the ring. The second vertex V_1 calculates the sum of range sizes in *Execute* from received states from all Chord nodes.

ExecuteChange shows incremental execution. It receives only the difference of two consecutive snapshots, and uses its last output as the base of execution. This avoids most of the redundant transmission and processing on unchanged states, reducing the overhead in both state exposers and verifiers.

To make the Chord checker scalable, we partition the execution of V_1 to multiple verifiers, each verifier taking only a subset of the states. Therefore, we need a third vertex V_2 to aggregate the outputs from all verifiers in V_1, i.e., the sum of key ranges in each partition. It calculates the relative range coverage as final output of the predicate. We use one verifier in the final vertex, and the verifier communicates with verifiers in V_1. This algorithm is essentially a 2-level aggregation tree; more levels will further improve scalability and pipelining.

Beside stream processing, developers can use sampling to further reduce overhead. Developers can check only sampled states in each vertex. To achieve this, D^{3}S allows verifiers to take as input only a portion of the key space for some vertices. These vertices process only the states that are mapped to covered keys. Tuples mapped to uncovered key space are dropped at the producer side. In addition, developers can check only sampled timestamps. This can be done because D^{3}S can stop checking in the middle of system execution and restart predicate checking at later global snapshots.

With sampling, D^{3}S can use a few verifiers to check a large-scale system in probabilistic manner. For instance, to check the consistency of Chord key range, D^{3}S can randomly sample a number of keys at different time and check that each sampled key has exact one holder (see Section 5.4), instead of checking the entire key space all the time. This approach makes the checking more lightweight, at the risk of having false negatives (i.e., missed violations).

2.6 Discussion

The two examples check predicates for safety properties only. For liveness properties, which should *eventually* be true, a violation often implies only that the system is in fluctuating status, rather than a bug. Similar to our previous work [27], a D^{3}S user can specify a timeout threshold plus stability measures in the predicate to filter out false alarms for liveness violations.

In theory, D^{3}S is capable of checking any property specified on a finite length of consecutive snapshots. To be used in practice, D^{3}S should impose negligible overhead on the system being checked, and be capable of checking large-scale systems. Based on our experience in Section 6, the overhead of the system is small in most cases, because we need to expose only states that are relevant to the predicate, and can omit other states. As a result, a state exposer consumes a tiny fraction of CPU and I/O compared with actual payloads of the system.

The scalability of checking depends on the predicate logic. When the predicate can be partitioned (as shown in our examples), a developer can add more verifiers to check larger systems. For such cases, the dataflow programming model effectively exploits parallelism within and among stages. However, some properties cannot be partitioned easily (e.g., deadlock detection that looks for cycles in the lock dependency graph). In such cases, the developer must write more sophisticated predicates to improve the scalability. For instance, the developer can add an early stage to filter locks that changed state recently, and have the final stage check only the locks that have been in the acquired state for a long time. By this means the final verifier avoids checking most correct locks, while still catching all deadlocks.

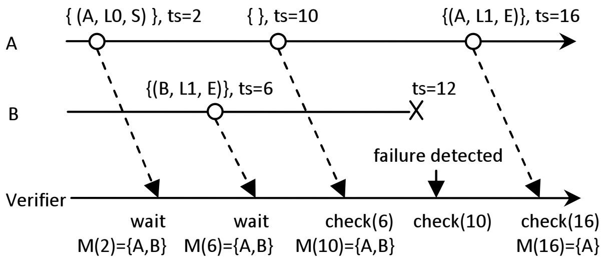
3 Global Snapshots

This section explains how D^{3}S constructs global snapshots and how accurate the predicates are under failures. Because of machine failures, snapshots might be incomplete, missing the state of failed machines, which may lead to false positives and false negatives when the system is reconfiguring.

3.1 Snapshots

We model the execution of the system being checked as a sequential state machine that traverses a sequence of consistent snapshots with global timestamps. Assume we have an increasing timestamp sequence $\mathcal{T} = \{t_0, t_1, ...\}$, where $t_i \in \mathcal{T}$ is a timestamp for $i \in \mathbb{N}$. The *membership* at timestamp t is the set of live processes at t, denoted by $M(t)$. For a process $p \in M(t)$, we use $S_p(t)$ to denote its local state at timestamp t. A *consistent snapshot* at t, denoted by $\pi(t)$, is the collection of states from all live processes at t, i.e., $\pi(t) = \bigcup_{p \in M(t)} S_p(t)$. With this notation, the system goes through a sequence of consistent snapshots, denoted by $\Pi = \{\pi(t_i), i = 0, 1, ...\}$. D^{3}S checks properties defined over these global snapshots.

To construct a global snapshot we need a global timestamp, which D^{3}S provides in a standard manner using a logical clock [24]. Each process maintains a logical clock, which is an integer initialized to 0. Each time



$\boldsymbol{T}$ = { 2, 6, 10, 16, … }
$\pi(2)$ = { (A, L0, S) }, $\pi(6)$ = { (A, L0, S), (B, L1, E) },
$\pi(10)$ = { (B, L1, E) }, $\pi(16)$ = { (A, L1, E) }

Figure 4: Predicate checking for consistency of distributed locks. Two processes A and B expose their states in the form of $\{$(ClientID, LockID, Mode)$\}$ (E for `Exclusive` and S for `Shared`). $\mathcal{T}$ is the sequence of timestamps and $\pi(t)$ is the snapshot for timestamp t. Given a failure detector that outputs membership for every timestamp, the verifier can decide whether a complete snapshot is obtained for checking.

a process reads its logical clock (e.g., to timestamp its state on a state change), it increases the logical clock by 1. Each time the process sends a message, it attaches its logical clock to the message. On receiving a message, the receiving processes sets its logical clock to the maximum of its local logical clock and the clock attached to the message. This way the D^{3}S runtime preserves happens-before relationships, can order all tuples in a consistent total order, and can construct snapshots.

Figure 4 illustrates the memberships and snapshots of the lock checking example. Process A and B are lock clients being checked, and they expose their state changes. Every state change produces a set of (*ClientID, LockID, Mode*) tuples that represent all current locks the process holds. The state changes happen at disjoint logical times $\{2, 10, 16\}$ and $\{6\}$, respectively. In addition, process B crashes at logical time 12.

If process p exposes two set of tuples at timestamp t_1 and t_2, for any timestamp t between t_1 and t_2, $S_p(t) = S_p(t_1)$. For example, $S_A(6) = S_A(2) = \{A, L0, Shared\}$. Therefore, given $M(6) = \{A, B\}$, the snapshot $\pi(6) = S_A(6) \cup S_B(6) = S_A(2) \cup S_B(6)$.

3.2 Predicates

We model a predicate as *a function defined over a finite number of consecutive snapshots*. The number of consecutive snapshots needed is called the *window size* of a predicate. Specifically, a predicate P with window size n is a function evaluated for every timestamp in $\mathcal{T}$, $P(t_i) = F(\pi(t_{i-n+1}), \pi(t_{i-n+2}), ..., \pi(t_i))$ for some $n \geq 1$, where F is a user-specified function. With this definition, a predicate can depend only on a recent time window of snapshots, and thus can be checked in the middle of system running. In our experience, all useful properties can be checked with only a recent time window of snapshots.

In the lock example, the checked property is that at any time t_i, there is no conflict between read and write locks. This property is checked by a predicate over the current snapshot, i.e., $LockConsistency(t_i) = F(\pi(t_i))$ in which F checks whether $\forall l \in LockID$, the set $\{(c, l', m) \in \pi(t_i) | l' = l, m = Exclusive\}$ contains at most one element (Figure 2 (a) implements this function). So $LockConsistency$ is a predicate with window size 1. Predicates with multiple consecutive snapshots are useful when specifying historical properties.

3.3 Correctness of predicate checking

To verify a predicate correctly, D^{3}S needs a *complete* snapshot, which contains the state of all processes that constitute the system at a timestamp. To construct a *complete* snapshot $\pi(t)$, D^{3}S must know the membership $M(t)$, and the local states $S_p(t)$ for all p in $M(t)$, but $M(t)$ can change due to failures.

D^{3}S relies on a *failure detector* to establish $M(t)$. We model the failure detector with a query interface, similar to most failure detector specifications [8]. A verifier can query for any timestamp t in $\mathcal{T}$, and the failure detector will return a *guess* on $M(t)$, denoted by $M'(t)$, which can be incorrect.

The verifier uses the failure detector as follows. It queries the failure detector and receives $M'(t)$. Then, the verifier waits until local states $S_p(t)$ for all $p \in M'(t)$ have been received. Then, it constructs snapshot $\pi(t)$ as $\bigcup_{p \in M'(t)} S_p(t)$. The verifier knows it has received $S_p(t)$ either when it receives it directly or when it receives two consecutive states $S_p(t_1)$ and $S_p(t_2)$ ($t_1 < t < t_2$). In the latter case the verifier infers that $S_p(t) = S_p(t_1)$.

If we would use this procedure unmodified, then D^{3}S has a problem when p does not expose any state for a long time (i.e. $t_2 \gg t_1$). In that case, D^{3}S is unable to construct $\pi(t)$ for any t between t_1 and t_2, because it doesn't know if $S_p(t_1)$ is the latest state from p. There are several ways to deal with this problem; we describe the solution we have implemented. The state exposer injected to p sends periodically p's current timestamp to the verifier. D^{3}S uses this heartbeat as both failure detector and the notification of p's progress. Thus the verifier receives a train of timestamps of heartbeats intermixed with the exposed state from p. When computing $\pi(t)$, D^{3}S uses the latest received $S_p(t_1)$ as long as the largest timestamp received from p exceeds t. If the failure detector declares that p has crashed at t_2 through a heartbeat timeout, for all t between t_1 and t_2, $\pi(t)$ uses $S_p(t_1)$. From t larger than t_2, D^{3}S excludes all p's state.

Figure 4 provides an example. B exposes its latest state at 6 and then crashes at 12. Thus, $\pi(10)$ is $S_A(10) \cup S_B(6)$ (after waiting for more than the timeout threshold for new state update from B). $\pi(16)$, however, will exclude B, since D^{3}S will decided that B has departed from the system.

3.4 Practical implications

D^{3}S guarantees that as long as $M'(t) = M(t)$. In other words, if the failure detector outputs correctly for timestamp t, the corresponding snapshot will be complete. If a snapshot is incomplete, then a checker can produce false positives and false negatives. In practice, there is a tradeoff between quick error alerts and accuracy.

To handle process failures that can lead to incomplete snapshots, D^{3}S must wait before constructing a snapshot. A larger waiting time T_{buf} results in larger buffer size to buffer state and delays violation detection. A too small T_{buf}, however, can result in imprecise results due to incorrect membership information. T_{buf} thus yields a knob to control the tradeoff between performance and accuracy.

The appropriate value of T_{buf} depends the failure detector D^{3}S uses, and should be larger than the failure detector's timeout T_{out}. We derive T_{out} as follows. For machine-room systems, there is usually a separate membership service that monitor the machine status using lease mechanisms [18]. We can use the membership service (or our own heartbeats) in failure detector. T_{out} is set as the grace period of the lease (resp. heartbeat interval) plus message delay. For wide-area applications such as DHT, T_{out} is determined by the specification of the application that declares the status of a node from its neighborhood. For predicates that does not rely on strict event orders, e.g. some runtime statistics, T_{buf} can be any values that is practically reasonable.

As an example, consider cluster storage system that we have checked (see details in Section 5.1). The system is designed for in machine-room environment, and the largest observed message delay between state exposers and verifiers is less than 350ms. The keep-alive message in the system runs every 1000ms. Thus, T_{buf} should be at least 1000ms + 350ms, and we use 2000ms. This guarantees that the verifiers always check on consistent snapshots.

4 Implementation

We have implemented D^{3}S on Windows. When compiling a predicate script, D^{3}S extracts the types of the tuples and the actions of monitoring functions. It then generates the corresponding C++ source code, which contains the definitions of the tuple types, vertex classes, monitoring functions, and the checking code in the predicates. D^{3}S compiles the source code into a state exposer DLL and a checking DLL.

The state exposer and the logical clock in D^{3}S uses WiDS BOX [19] to instrument processes being monitored. BOX is a toolkit for binary instrumentation [20]; it identifies the function addresses through symbol information, and rewrites the entries of functions in code module to redirect function calls. When injecting a DLL into a process, BOX loads the DLL into the process's address space, and redirects function calls that are interposed on to callbacks in the DLL. Because BOX provides rewrites code atomically, it does not need to suspend the threads in the process during the instrumentation.

Through the callbacks for application-level functions, the state exposer copies the exposed states into an internal buffer. States that are used by predicates are emitted to the verifiers, whereas all others are omitted. D^{3}S buffers the states to batch data transmission. It waits until the collected states exceed 500 bytes, or 500 ms has elapsed after the last transmission.

The D^{3}S runtime also interposes on the socket APIs for sending and receiving messages. In those callbacks, D^{3}S updates the logical clock. D^{3}S also adds 8 additional bytes to each message for the logical clock and some bookkeeping information, same as [27, 15].

To make the logical clock relate to real time, D^{3}S divides a second in 1,000 logical ticks. Every second D^{3}S checks the value of the logical clock, and if hasn't been updated a 1,000 times, D^{3}S bumps the clock to 1,000. This gives a convenient way to specify timeout of the monitored processes (i.e., T_{out} in Section 3.4).

D^{3}S uses reliable network transmission between state exposers and verifiers, and also among verifiers when computation graph has multiple levels.

D^{3}S uses a central master machine to manage a partitioned key space. Each verifier periodically reports its recently verified timestamp to the master. A verifier is considered to be failed when it doesn't report within a timeout period. In such case the master re-arranges the partition of key space to make sure that every key is appropriately covered. The new partition is then broadcast to all related state exposers and verifiers. By this means the appropriate states will arrive at the new destinations.

5 Experience with Using D^{3}S

To evaluate the effectiveness of D^{3}S, we apply it to five complete, deployed systems, including systems that are based on production-quality code. Table 1 summarizes these checked systems, line of code in predicates (LoP), and the results, which are obtained within one month. For each of these systems, we will give sufficient descriptions of their logic and properties, and then report the effectiveness of D^{3}S with our debugging experience, in terms of both the benefits and lessons we gained.

Unless otherwise specified, experiments are performed on machines with dual 2 GHz Intel Xeon CPU and 4 GB memory, connected with 1 Gb Ethernet and run Windows Server 2003. For some systems we injected failures to improve testing coverage.

5.1 PacificA

PacificA [26] is a semi-structured distributed storage system. It shares with BigTable [9] the property that large tables are segmented into small pieces, which are replicated and distributed across a cluster of commodity servers. The goal of this project is to explore different design choices (e.g., semantics-agnostic vs. semantics-aware in replication), and develop a semi-structured storage system with better combination of these trade-offs.

PacificA has been developed for more than one year, and is fairly complete with around 70,000 lines of code. It is a good example of a complex system that builds on top of well-specified components, including: two-phase commit for consistent replica updates, perfect failure detector on storage nodes, replica group reconfiguration to handle node failures, and replica reconciliation to rejoin a replica. When combining these components , the whole system can have complex message sequences with bugs that are hard to detect and analyze. However, each component must be working correctly with predictable behaviors and invariants. Therefore, we use D^{3}S to check individual components against their specifications.

Based on specifications, there are several invariants for every major component. These invariants include consistency among replicas, data integrity after recovery, and consistent membership view across a replica group. Violating an invariant may eventually leads to data loss or replica inconsistency. We specify these invariants as predicates in D^{3}S, and check a daily used deployed instance of PacificA. The deployment stores a social graph data for a social network computing platform. It uses 8 servers as storage nodes, and another server as frontend for clients. Clients running in 4 other machines frequently query and update edge data stored in PacificA tables to complete graph computation tasks, which usually last from a day to a week. The size of original social graph data is 38 GB, while during the execution the intermediate tables exceed 1 TB in size. A PacificA machine can have 40 MB/s throughput at the peak time. We use another 3 machines to run verifiers, and have detected 3 bugs. We will explain in detail one bug in replica group reconfiguration. The bug violates the *primary invariant* (see page 3 of [26]) during failures.

In PacificA, the basic unit of data is a slice (100 MB data chunk). A slice is replicated in three storage nodes

Application	LoC	LoP	Predicates	Results
PacificA (Structured storage)	67,263	118	membership group consistency; replica consistency	3 correctness bugs
MPS (Paxos implementation)	6,993	50	consistency in consensus outputs; leader election	2 correctness bugs
Web search engine	26,036	81	unbalanced response time of indexing servers	1 performance problem
$i3$-Chord (DHT)	7,640	72	aggregate key range coverage; conflicting key holders	availability and consistency
libtorrent (BitTorrent client)	36,117	210	neighbor set; downloaded pieces; peer contribution rank	2 performance bugs; free riders

Table 1: Benchmarks and results

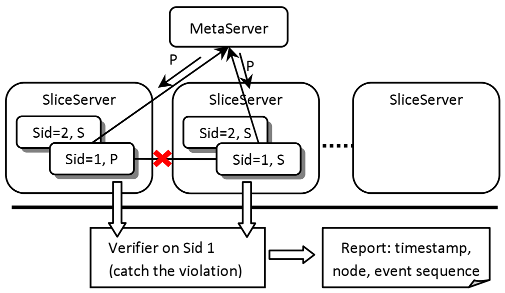


Figure 5: PacificA architecture and the bug we found.

(SliceServers), one replica being the primary and the other two being secondaries. A simplified architecture is shown in Figure 5. The primary (labeled with P) answers queries and forwards updates to the secondaries (labeled with S). These replicas monitor each other with heartbeats. When noticing a failure, the remaining replicas will issue requests to a MetaServer (master of SliceServers) for reconfiguration, and may ask to promote themselves as the primary if they think the primary is dead. The primary invariant states that at any given timestamp, there cannot be more than one primary for a slice. This is because multiple primaries can cause potential replica inconsistency during updates.

We expose replica states with tuples of the form (Sid, MachineID, $\{P / S\}$) (Figure 5), and check the number of P's for every Sid (slice identifier). Because replicas of the same slice should be checked together, we map tuples to their Sid.

As expected, the predicate found no violations in normal cases, since the system reconfigures only after a failure. To expand test coverage, we randomly injected failures to SliceServers and MetaServer and then recovered them. After dozens of tries, D^{3}S detected a violation in a slice group, which had two primary replicas. By studying the sequence of states and the events that led to the violation, we determined that, before MetaServer crashed, it accepted a request and promote the replica to be the primary. After crash and recovery, the MetaServer forgot the previous response and accepted the second replica's request, which resulted in the second primary. This violation should have been avoided by MetaServer's failure tolerance mechanism, which logs accepted requests to disk, and restores them at next start. However, the code uses a background thread to do batch logging, so the on-disk log may not have the last accepted request. To do it correctly, the MetaServer must flush the log *before* it sends a response.

D^{3}S helped in detecting the bug in the following ways. First, the bug violates a distributed property that cannot be checked locally. D^{3}S provides globally-consistent snapshots that make checking distributed properties easy. Second, contrast to postmortem log verification, D^{3}S enforces always-on component-level specification checking, and catches rare-case bugs with enough information to determine the root cause. Without this predicate, we may still have noticed the bug when conflicting writes from two primaries would have corrupted the data. However, from this corruption it would have been difficult to determine the root cause.

We also applied D^{3}S to an old PacificA version which has two data races. It took the developers several days to resolve them. With D^{3}S, these bugs were caught in several hours of normal use, and state sequences provided much better hints of the root cause.

5.2 The MPS Paxos Implementation

Paxos [25] is a widely used consensus algorithm for building fault-tolerant services [29, 6]. Despite the existing literature on Paxos, implementing a complete Paxos-based service is non-trivial [7]. We checked MPS[1], a Paxos-based replicated state machine service used in production. MPS provides many features over the base Paxos protocol, e.g., it uses leader election for liveness, and perform state transfers to allow a new node to catch up with others. These additional features are necessary for ensuring progress of the service, because Paxos itself guarantees only safety but not liveness of agreement.

MPS provides an API that application code can invoke to execute arbitrary commands, with the guarantee that all nodes execute the same sequence of commands. We deploy MPS on 5 nodes, the typical configuration in cluster environment, and make each node invoke the API independently. For checking consensus protocols, testing

[1]MPS is an anonymous name of a Paxos-based service system developed by Microsoft product team.

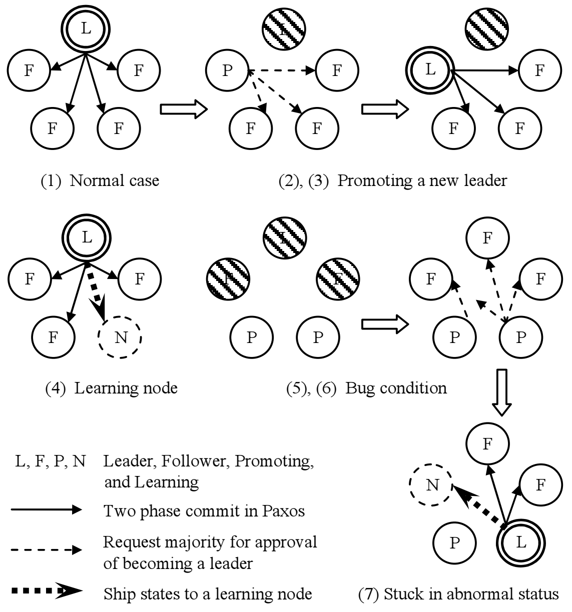

L
F
P
N
(1) Normal case
(2), (3) Promoting a new leader
(4) Learning node
(5), (6) Bug condition
(7) Stuck in abnormal status
L, F, P, N Leader, Follower, Promoting, and Learning
Two phase commit in Paxos
Request majority for approval of becoming a leader
Ship states to a learning node

Figure 6: Leader election in Paxos implementation and the snapshots of states that lead to the bug.

with failure cases is important. Therefore we inject a failure module along with the state exposer library, which can fail the MPS process.

An important invariant is *safety and liveness of agreement*: all nodes execute identical sequence of commands with sequence number always increasing by 1 (safety), and the system can make progress as long as there exist majority of nodes (liveness). To check this invariant, D^{3}S exposes every executed command in each node with its sequence number. After days of running, D^{3}S detected no violations. We did find a careless pointer error in a rare code path, thanks to failure injection.

Then we turned our attention to the leader election implementation of MPS—this part of the service was less well specified and proven by the development team, compared to their implementation of the Paxos protocol. A node can be in one of the four status: `Leader`, `Follower`, `Promoting` and `Learning`. In normal case, one `Leader` broadcasts incoming commands to `Followers` using the Paxos protocol (Figure 6.(1)). `Promoting` and `Learning` are only transient status. `Promoting` is used when leader is absent, and the promoting node needs to get majority's approval to become the new leader (Figure 6.(2) and (3)). A `Learning` node is out-of-date in execution of commands, and is actively transferring state from another node (Figure 6.(4)). Leader election should make the system always go back to the normal case after failures. Therefore, we wrote a predicate to check the status of nodes and catch persistent deviation from the normal case.

After running for hours with randomly injected failures, this predicate detected a persistent abnormal status, as shown in (5) $\sim$ (7). We name the five nodes as $A \sim E$ in counterclockwise order. After A, B, E crash and recover, both C and D are promoting themselves (in (5)). Only D gets majority's approval (A, E and itself). C is approved by B but the messages to A and E are lost (in (6)). Later when D's promoting request arrives at B, B happens to be out-of-date compared with D, so B switches to `Learning` (in (7)). By design, C will become `Follower` when it learns the new leader. But now C considers A, E are dead and only keeps requesting B, which is a learning node and will just ignore any requests except state transfers. Therefore, C can be stuck in `Promoting` for a long time. Although this bug does not corrupt the Paxos properties right away, it silently excludes C from the group and makes the system vulnerable to future failures. This bug was confirmed by people working on the code.

This is a fairly complicated bug. Although the system has correct external behavior, D^{3}S helps us discover the bug with always-on checking of internal states. This bug had been missed by log verification. We guess the reason might be that, when the abnormality shows up for a short time during a long execution, it is hard to detect bugs with postmortem analysis. Another lesson is that less rigorously specified components (e.g., leader election) are harder to get right. Therefore, it is productive for testers to write and install "ad-hoc" predicates on-the-fly to probe the suspected components, as D^{3}S enables.

5.3 Web search engine

Another research group in our lab has developed an experimental search engine that powers an online vertical search [37]. It has a frontend server which distributes queries to a set of backend index servers and collects results. The index is partitioned among the index servers using the partition-by-document scheme. Each index server further partitions its responsible index into five tiers, according to document popularity. Only the first tier is kept in memory. Given a query of terms, an index server first retrieves the documents from the first tier. If there are 100 hits, the index server returns, otherwise it tries the next tier, and so on. The system is deployed in 250 servers, each having 8 GB memory and 1 TB disk. We use a dedicated server to run as a verifier, which communicates with all index servers and the frontend. The frontend keeps the index servers by replaying query logs from a commercial search engine.

Developers of the system were not satisfied about the performance. They had statistics on total latency for queries, but they wanted to know what contributed to the latency of the critical path, and how much time was spent

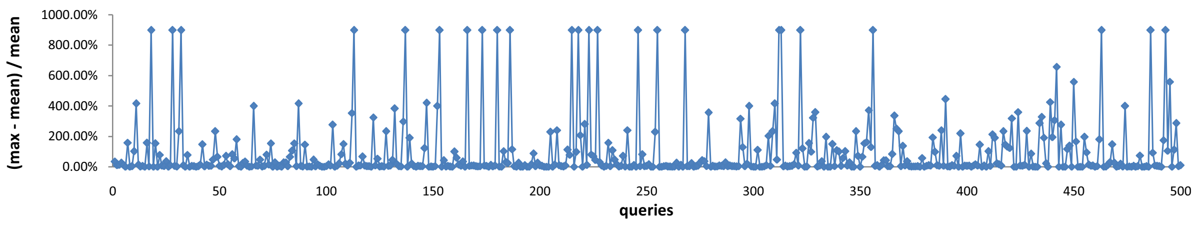


Figure 7: Load imbalance in the web search engine.

on each tier. We used D^{3}S to interpose on every function in the critical path (e.g., fetching documents from each tier, intersecting lists of terms, and sorting the results), and exposed the execution time and the related terms in the query. This provided detailed performance data. We wrote predicates to probe different aspects of the latency to catch abnormal queries.

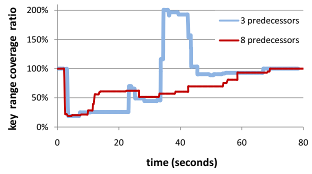
One predicate we wrote checked load balance. For each index tier, the predicate calculates $Pr = (max - mean)/mean$, using maximum and mean of the latency in the tier among index servers. Larger Pr means less balance of the load. Since the frontend needs to collect results from all the index servers, unbalanced load may result in a performance degradation, since the slowest index server determines the final response time.

Figure 7 shows the values of Pr of 500 selected queries over 10 index servers. For the queries whose Pr is greater than one (i.e., $max > 2 \times mean$), there are always one or two index servers running much slower than others. Further diving into the latency measures with other predicates (e.g., the number of visited tiers in different index servers), we found that simple hashing does not balance the load across the index server uniformly, and the imbalance degrades performance significantly when the query has more than one terms.

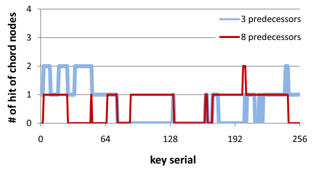
D^{3}S is useful in terms of flexibility and transparency. Adding logs to the search engine code and performing postmortem analysis of these logs will yield the same insights. However, it is more work and doesn't allow refining predicates on-the-fly.

5.4 Chord

DHTs are by now a well-known concept. There are availability measures of DHTs, in terms of convergence time and lookup success rate in case of failures. However, the routing consistency in DHTs is less well understood. Routing consistency means that a lookup returns the *correct* node holding the key, which may not happen when nodes fail or the network partitions [11]. For example, a DHT put for a key may store the key at a node that after healing of the network isn't the successor or isn't even in the successor list, and the key becomes unavailable. Similarly, it may happen that several nodes think



(a) Aggregate key ranges.



(b) Holders of random sampled keys.

Figure 8: $i3$-Chord experimental results.

they are *the* successor of a key, which could lead to inconsistencies. Because availability and consistency are tradeoffs in distributed systems [17], a better study on DHTs should measure these two issues altogether.

We ran an experiment with 85 Chord nodes on 7 servers. We ran the $i3$ [1] Chord implementation, which we ported to Windows. We used the predicate in section 2 (Figure 3) to check the aggregate key ranges held by the nodes, with 4 verifiers for the first stage and 1 verifier for the final stage. This predicate checks a necessary but insufficient condition for the integrity of the DHT logical space. If the output value is below 100%, then the system must have "holes" (i.e., no node in the system believes it is the successor of a key range). This indicates unavailability. On the other hand, an output value above 100% indicates overlaps (i.e., several nodes believe they are the successor).

In $i3$, a Chord node maintains several predecessors and successors, and periodically pings them for stabilizing the ring. We used two configurations for the number of predecessors, one is 3 and the other is 8. The number of successors is set to the same value as the number of predecessors in either configuration. We crashed 70% of nodes and then monitored the change of the aggregated key ranges.

Figure 8a shows the results of the two configurations. The 3-predecessor case has unpredictable behavior with respect to both consistency and availability. The total key range value oscillates around 100% before converging. The 8-predecessor case does better, since it never exceeds 100%. The swing behavior in the 3-predecessor case results from nodes that lose their predecessors and successors. Such nodes use the finger table to rejoin the ring. This case is likely to result in overlapping key ranges. With more predecessors and successors, this case happens rarely.

With the total key range metric, the "holes" and "overlaps" can offset each other, and therefore when the metric is below 100%, there could be overlaps as well. To observe such offsets, we added a predicate to sample 256 random points in the key space, and observed the number of nodes holding these points. Figure 8b shows a snapshot taken at the 25^{th} second. Despite that the total key range never exceeds 100% for the 8-predecessor configuration, overlaps occur (see key serial number 200), which indicate inconsistency. The figure also shows a snapshot when both inconsistency and unavailability occur in the 3-predecessor configuration.

D^3S allows us to quantitatively measure the tradeoffs between availability and consistency in a scalable manner using D^3S's support for a tree of verifiers and for sampling, and reveals the system behavior at real time. Although we did these experiments in a test-lab, repeating them in a wide-area large-scale deployment is practical. For a wide-area online monitoring system, D^3S's scalable and fault-tolerant design should be important.

5.5 BitTorrent client

We applied D^3S to libtorrent [2] release 0.12, with 57 peers running on 8 machines. The peers download a file of 52 MB. The upload bandwidth limit varies from 10 KB/s to 200 KB/s in different experiments, and peers finish downloading in 15 minutes to 2 hours. The bugs we found (and acknowledged by the developers) illustrate the use of real time monitoring with D^3S predicates.

Similar to Chord, BitTorrent exhibits complex behavior which is difficult to observe in detail. We start with investigating key properties of the running system with predicates in order to gain some insights. The first property we looked at was the peer graph, because the structure of graph is important to dissemination of data.

We exposed the neighbor list from every peer, and used a two-stage predicate to form an aggregation tree, similar to the total key range predicate in Chord. At the final stage, we obtained the entire graph in real time. We printed the graph to a console and kept refreshing it, in order to observe the graph changes. we found an abnormality right away: sometimes a peer had more than 300 neighbors (recall that we used only 57 peers). We inspected the code and found that, when adding a new IP address into the neighbor list, libtorrent did not check if the same IP address already existed. This resulted in duplicated IP addresses in neighbor list, and degraded performance because peers may had less choices when downloading from neighbors. We fixed the bug and continued with subsequent experiments.

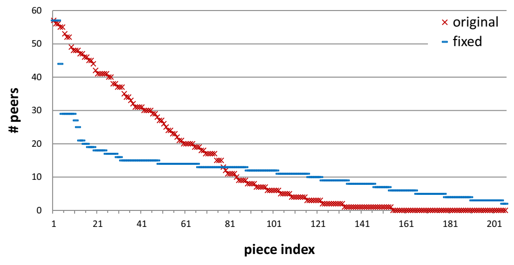


Figure 9: Piece distributions over peers when finishing 30% downloading.

After the peer graph looked correct, we started another predicate to compute how pieces are distributed over peers. Every time when a peer downloads a piece, the predicate exposes the piece number along with its own ID. The predicate aggregates the vector of available pieces in every peer, again using a two stage aggregating tree. With simple visualization (i.e., printing out the distribution of pieces at real time), we observed that some pieces spread much faster than the others. Figure 9 illustrates a snapshot of the numbers of pieces over peers. Some pieces are distributed on all peers in a short time, while other pieces make less progress. This was not what we expected: pieces should have been selected randomly and therefore the progresses should *not* have differed that dramatically. Thus, we suspected that peers are converging on same pieces. We examined the code and found that the implementation *deterministically* chose which pieces to download: for pieces with the same number of replicas, all clients chose replicas in the same order. We fixed the bug and collected the data again. As shown in Figure 9, the progress of pieces is much closer to random selection.

After fixing these bugs, we implemented algorithms that detect free riders who disproportionally download compared to upload. Our purpose was to further expand the use of D^3S from debugging-focused scenarios to online monitoring. Of the 57 peers, the upload bandwidths of Peer 46∼56 were limited to 20 KB/s, while other peers' upload bandwidths remained 200 KB/S. All peers had unlimited download bandwidth. Peer 46∼56 were more likely to be free riders, compared to the other peers [33]. We compute contributions of the peers using predicates that implement the EigenTrust algorithm [22]. The predicates collect the source of every downloaded piece in each peer, and calculate EigenTrust in a central server. As shown in Figure 10, the predicates successfully distinguished peers via their contributions.

5.6 Summary of debugging experience

Using examples we have shown that D^3S helps find nontrivial bugs in deployed, running systems. The effective-

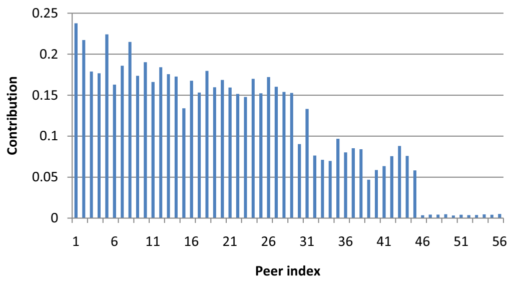
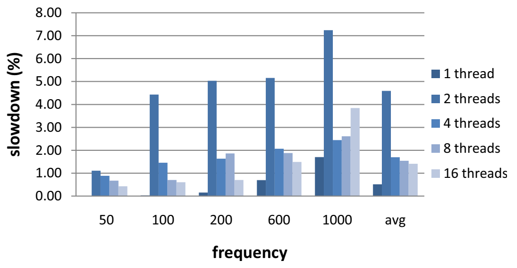
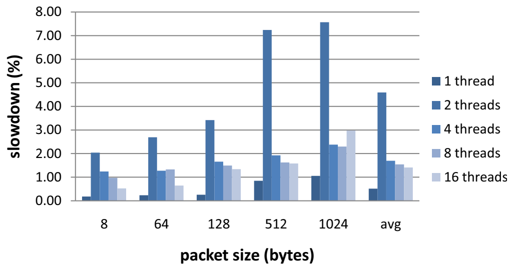


Figure 10: The contributions of peers (free riders are 46∼56).



(a) Slowdown with average packet size 390 bytes and different exposing frequencies.



(b) Slowdown with average frequency 347 /s and different exposing packet sizes.

Figure 11: Performance overhead on system being checked.

ness of D^{3}S depends on whether or not we have useful predicates to check. When a system already has specifications and invariants (e.g., at the component level), which is common for complex, well designed systems, D^{3}S is effective, because the predicates can check the invariants. Writing the predicates is mostly an easy task for developers, because they are allowed to use sequential programs on global snapshots. When a system doesn't have a clear specification (e.g., in performance debugging), D^{3}S is more like a dynamic log-collecting and processing tool, which can help zooming into specific state without stopping the system. This helps developers probing the system quickly, and eventually identify useful predicates.

D^{3}S is not a panacea. Component-level predicates are effective for debugging a single system with a good specification. However, when debugging large-scale web applications running in data centers, this approach is sometimes insufficient. First, data center applications often involve a number of collaborative systems that interact with each other. When unexpected interactions happen that lead to problems (e.g., performance degradation), developers have little information about which system they should inspect for the problem. Second, these systems evolve on daily basis, and sometimes there are no up-to-date specifications to check. These issues are what our on-going research on D^{3}S aims to address.

6 Performance Evaluation

This section studies the performance of D^{3}S, using the machine configuration described at the beginning of Section 5.

We first evaluate overhead of checking on a running system. This overhead is caused by the cost of exposing state, and depends on two factors: the frequency of exposing state and the average size of the state exposed. To test the overhead under different conditions, we use a micro benchmark in which the checked process starts various number of threads. Each thread does intensive computation to push CPU utilization close to 100%. Figure 11 shows the overhead. We can see that the state exposer is lightweight and in general the overhead is around 2%. The largest overhead happens when the process has 2 threads of its own, which maps perfectly to the dual-core CPU. State exposer brings one additional thread, and thus increases the thread scheduling overhead. In this case the overhead is still less than 8%.

These results are consistent with all the systems checked. Systems that are neither I/O nor CPU intensive (e.g., Chord and Paxos) have negligible overhead; BitTorrent and Web search have less than ($< 2\%$) overhead. The impact to PacificA varies according to system load (Figure 12). We created 100 slices and we vary the number of concurrent clients, each sends 1000 random reads and writes per second with average size 32KB per second. The overhead is less than 8%. A PacificA machine generates in average 1,500 snapshots per second, and consumes at the peak time less than 1000 KB/s additional bandwidth for exposing states to verifier. On average, exposing states uses less than 0.5% of the total I/O consumption. These results encourage adopting D^{3}S as an always-on facility.

Second, we evaluate the impact on performance of PacificA when we start new predicates. We start checking all predicates in Section 5.1 at the 60th second. Before that there is no state exposer injected to PacificA. Figure 13 shows the total throughput seen by clients. Given that PacificA itself has fluctuating throughput due to reorganizing disk layout (see [26]), there is no visible impact on performance when starting new predicates.

In addition, we evaluate the failure handling of D^{3}S.

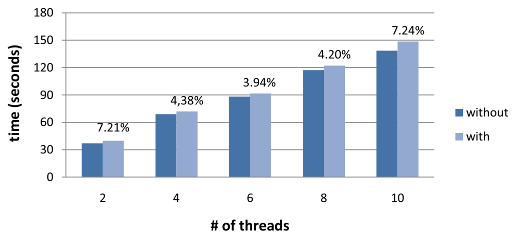


Figure 12: Performance overhead on PacificA with different throughput.

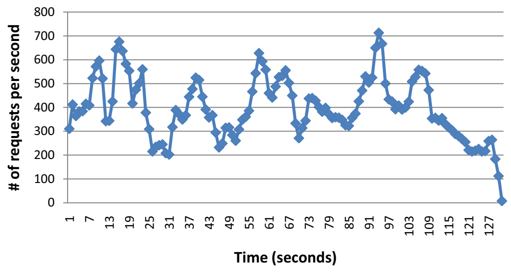


Figure 13: Throughput of PacificA when a predicate starts.

In the above PacificA testing, we start 3 verifiers and kill one at the 30^{th} second. After the failure is detected, the uncovered key range are repartitioned. Figure 14 shows how the load of the failed verifier is taken over by the remaining verifiers. The fluctuation reflects the nature of PacificA, which periodically swaps bulk data between memory and disk.

7 Related Work

Replay-based predicate checking. People have proposed to check replayed instances of a distributed system for detecting non-trivial bugs that appear only when the system is deployed [27, 16]. D^{3}S addresses one critical weakness in that replaying the entire execution of a large-scale system is prohibitively expensive. The advantage that replay brings is to repeatedly reproduce the execution to aid debugging, and the role of the online checking is to accurately position and scope the replay once a bug site is reported. We see replay as a key complementary technology to online predicate checking. The ultimate vision is to use online checking to catch violations, and then enable time-travel debug of a recent history with bounded-time replay.

Online monitoring. P2 monitor [34] is designed for online monitoring of distributed properties, which is close to ours in spirit. However, D^{3}S differs in a number of ways. First, D^{3}S allows expressing predicates on global snapshots of states, while P2 monitor requires its user to take care of collecting and ordering the monitored states. Second, D^{3}S works on legacy systems, while P2 monitor

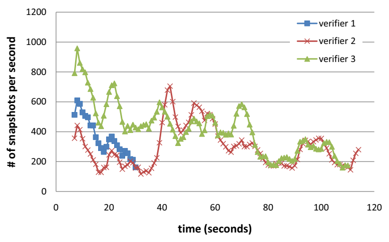


Figure 14: Load when verifier 1 fails at 30^{th} second.

is confined to systems built with OverLog language [28]. Finally, D^{3}S can tolerate failures from both the system being checked and itself.

Log analysis. There is a collection of literature that relies on logs for postmortem analysis, including analyzing statistics of resource usage [3], correlating events [5], tracking dependency among components [10, 14] and checking causal paths [32]. Using a specific and compact format, the work in [36] can scale event logging to the order of thousands of nodes. Fundamentally, logging and online predicate checking all impose runtime overhead to expose information for analysis. D^{3}S is, at a minimum, an intelligent logger that collects states in a scalable and fault-tolerant way. However, the D^{3}S's advantage in exposing new states on-the-fly and allowing a simple programming model can significantly improve debugging productivity of developers in practice.

Large-scale parallel applications. Parallel applications generally consist of many identical processes collectively for a single computation. Existing tools can check thousands of such processes at runtime and detect certain kinds of abnormalities, for example, by comparing stack traces among the processes [4, 30], or checking message logs [13]. In contrast, D^{3}S works for both identical and heterogeneous processes, and is a general-purpose predicate checker.

Model checking. Model checkers [23, 31, 38] virtualize the environments to systematically explore the system space to spot a bug site. The problem of state explosion often limits the testing scale to small systems compared to the size of deployed system. Similarly, the testing environment is also virtual, making it hard to identify performance bugs, which require a realistic environment and load. D^{3}S addresses the two problems by checking the deployed system directly.

8 Conclusion and Future Work

Debugging and testing large-scale distributed systems is a challenge. This paper presented a tool to make debugging and testing of such systems easier. D^{3}S is a flexible and versatile online predicate checker that has shown its promise by detecting non-trivial correctness and performance bugs in running systems.

As future work, we are exploring several directions. We are pushing forward our vision of combining online predicate checking and offline replay. We are also exploring tools to debug data center applications that are composed of many systems so that a developer can easily find bugs due to unexpected interactions between the systems.

Acknowledgments

We would like to thank our shepherd, Amin Vahdat, and the anonymous reviewers for their insightful comments. Thanks to our colleagues Haohui Mai, Zhilei Xu, Yuan Yu, Lidong Zhou, and Lintao Zhang for valuable feedback.

References

[1] i3 implementation: `http://i3.cs.berkeley.edu/`.

[2] libtorrent: `http://libtorrent.sourceforge.net/`.

[3] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[4] Arnold, D. C., Ahn, D. H., de Supinski, B. R., Lee, G., Miller, B. P., and Schulz, M. Stack trace analysis for large scale debugging. In *IPDPS* (2007).

[5] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004).

[6] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *OSDI* (2006).

[7] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos made live: an engineering perspective. In *PODC* (2007).

[8] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. In *JACM* (1996).

[9] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[10] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic, internet services. In *DSN* (2002).

[11] Chen, W., and Liu, X. Enforcing routing consistency in structured peertopeer overlays: Should we and could we? In *IPTPS* (2006).

[12] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[13] DeSouza, J., Kuhn, B., and de Supinski, B. R. Automated, scalable debugging of MPI programs with Intel message checker. In *SE-HPCS* (2005).

[14] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-Trace: A pervasive network tracing framework. In *NSDI* (2007).

[15] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay debugging for distributed applications. In *USENIX* (2006).

[16] Geels, D., Altekarz, G., Maniatis, P., Roscoey, T., and Stoicaz, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[17] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT. ACM 33*, 2 (2002).

[18] Gray, C. G., and Cheriton, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989).

[19] Guo, Z., Wang, X., Liu, X., Lin, W., and Zhang, Z. BOX: Icing the APIs. MSR-TR-2008-03.

[20] Hunt, G., and Brubacher, D. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium* (1999).

[21] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[22] Kamvar, S. D., Schlosser, M. T., and Garcia-Molina, H. The EigenTrust algorithm for reputation management in P2P networks. In *WWW* (2003).

[23] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978).

[25] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[26] Lin, W., Yang, M., Zhang, L., and Zhou, L. PacificA: Replication in log-based distributed storage systems. MSR-TR-2008-25.

[27] Liu, X., Lin, W., Pan, A., and Zhang, Z. WiDS checker: Combating bugs in distributed systems. In *NSDI* (2007).

[28] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing declarative overlays. In *SOSP* (2005).

[29] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004).

[30] Mirgorodskiy, A. V., Maruyama, N., and Miller, B. P. Problem diagnosis in large-scale computing environments. In *SC* (2006).

[31] Musuvathi, M., and Engler, D. Model checking large network protocol implementations. In *NSDI* (2004).

[32] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[33] Saroiu, S., Gummadi, P. K., and Gribble, S. D. A measurement study of peer-to-peer file sharing systems. In *MMCN* (2002).

[34] Singh, A., Roscoe, T., Maniatis, P., and Druschel, P. Using queries for distributed monitoring and forensics. In *EuroSys* (2006).

[35] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet indirection infrastructure. In *Sigcomm* (2002).

[36] Verbowski, C., Kiciman, E., Kumar, A., Daniels, B., Lu, S., Lee, J., Wang, Y.-M., and Roussev, R. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI* (2006).

[37] Wen, J.-R., and Ma, W.-Y. Webstudio: building infrastructure for web data management. In *SIGMOD* (2007).

[38] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. In *OSDI* (2004).