DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs

Haipeng Cai
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA, USA
haipeng.cai@wsu.edu

Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA
dthain@nd.edu

ABSTRACT

Dynamic impact analysis is a fundamental technique for understanding the impact of specific program entities, or changes to them, on the rest of the program for concrete executions. However, existing techniques are either inapplicable or of very limited utility for distributed programs running in multiple concurrent processes. This paper presents DistIA, a dynamic analysis of distributed systems that predicts impacts propagated both within and across process boundaries by partially ordering distributed method-execution events, inferring causality from the ordered events, and exploiting message-passing semantics. We applied DistIA to large distributed systems of various architectures and sizes, for which it on average finishes the entire analysis within one minute and safely reduces impact-set sizes by over 43% relative to existing options with run-time overhead less than 8%. Moreover, two case studies initially demonstrated the precision of DistIA and its utility in distributed system understanding. While conservative thus subject to false positives, DistIA balances precision and efficiency to offer cost-effective options for evolving distributed programs.

CCS Concepts

•**Software and its engineering** → **Software evolution;**

Keywords

Impact analysis; distributed systems; dynamic partial ordering

1. INTRODUCTION

Program changes drive the evolution of software systems, yet also pose threats to their quality and reliability [55]. Thus, it is crucial to understand potential consequences of those changes even *before* applying them to candidate program locations. To accomplish this task, developers need to perform impact analysis [9, 57, 62] with respect to those locations, an integral step of modern software development process [56]. In particular, for developers working with concrete executions of the program, *dynamic* impact analysis [9,40] is an attractive option as it narrows down the search space of such impacts to the specific context of those executions.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970352

Dynamic impact analysis has been extensively investigated [40], resulting in a rich and diverse set of relevant techniques and tools (e.g., [4, 11, 13, 39]). However, most existing approaches address sequential programs only, with fewer targeting concurrent yet centralized software [25, 27, 35, 65] and very few applicable to distributed systems. To accommodate the increasingly demanding performance and scalability needs of today's computation tasks, more distributed systems are being deployed than centralized ones, raising an urgent call for techniques, including impact analysis, for cost-effective evolution of those systems [8, 24, 30].

Dynamic code analyses for distributed systems also have been explored early on [18, 31, 34] and more in recent years [7, 44, 52], largely focusing on detailed analysis of program dependencies. However, the majority of these approaches were designed only for procedural programs [7]. For distributed object-oriented programs, *backward* dynamic slicing algorithms have been developed, yet it is still unclear whether they can work with (and scale to) real-world systems [7, 44]. And for impact analysis, *forward* slicing would be needed. Nevertheless, the fine-grained (statement-level) analysis used by slicing would be overly heavyweight for impact analysis which is commonly adopted at method level [4, 9, 11, 39, 40].

Unfortunately, developing a cost-effective dynamic impact analysis for real-world distributed systems remains challenging. One major difficulty lies in the lack of explicit invocations or references among decoupled components of those systems [27, 30, 64], whereas traditional approaches often rely on such explicit information to compute dependencies for impact prediction. Lately, various analyses other than slicing have also been proposed [24, 53, 64]. While efficient for *static* impact analysis, these approaches are limited to systems of special types such as distributed *event-based* systems (DEBS) [45], or rely on specialized language extensions like EventJava [19]. Other approaches are potentially applicable in a wider scope yet depend on information that is not always available, such as execution logs of particular patterns [41], or suffer from overly-coarse granularity (e.g., class-level) [24, 41, 53] and/or unsoundness [46], in addition to imprecision, of analysis results.

We have developed DistIA, a dynamic impact analysis for commonly deployed distributed systems where components communicate via socket-based message passing. We define a component as the code that runs in a separate process from all other parts of the system. By exploiting the happens-before relation [38] between method-execution events and the semantics of message passing among distributed components, DistIA predicts impacts of one method on others of a given system both within and across its concurrent processes. Our approach offers *rapid* results that are safe [29] relative to the analyzed executions, while relying on neither well-defined inter-component interfaces nor message-type specifications needed by peer approaches (e.g., those for DEBS).

```
1   public class S {
2       Socket ssock = null;
3       public S(int port) { ssock = new Socket(port); ssock.accept(); }
4       char getMax(String s) {...}
5       void serve() { String s = ssock.readLine();
6           char r = getMax(s); ssock.writeChar(r); }
7       public static int main(String[] a) {
8           S s = new S(33); s.serve(); return 0; } }
9   public class C {
10      Socket csock = null;
11      public C(String host,int port) { csock = new Socket(host,port); }
12      void shuffle(String s) {...}
13      char compute(String s) { shuffle(s); csock.writeChars(s);
14          return csock.readChar(); }
15      public static int main(String[] a) { C c = new C('localhost',33);
16          System.out.println( c.compute(a[0]) ); return 0; } }
```

Figure 1: An example distributed program E consisting of two components: S (server) and C (client).

We evaluate DISTIA on six distributed Java programs, including four enterprise-scale systems, and demonstrate that it is able to work with large, complicated distributed systems using blocking and/or non-blocking (e.g., selector-based [5]) communication. In the absence of more advanced techniques, we compare with a coverage-based approach [51], which reports as impacted all methods covered in the utilized executions, as a safe baseline alternative, and measure the effectiveness of DISTIA against it. The results show that DISTIA can greatly reduce the size of potential impacts to be inspected by over 43% on average, relative to the baseline, at the mean cost of one minute to finish the one-time instrumentation and 65ms to query impacts, with run-time overhead less than 8%.

Since there is no automatic approach available for computing ground-truth impacts, we manually evaluated the precision of DISTIA on randomly selected cases. We regard a method having any statement data and/or control dependent on any statement of the queried method (i.e., the *query*) as a *true positive*, and the *precision* as the fraction of predicted impacts that are true positives. Also, we explored the usefulness of DISTIA in program comprehension in a second case study. Our results suggest that developers using DISTIA may expect an average precision of around 70% and considerable benefits in understanding distributed programs and executions.

The main contributions of this work include:

- The *first* dynamic impact analysis, DISTIA, for distributed systems where components run in concurrent processes and communicate only via socket-based message passing (Section 3).
- An *open-source* implementation of DISTIA working with diverse, large enterprise distributed systems that use either or both of blocking and non-blocking communication (Section 4).
- An evaluation of DISTIA showing its promising effectiveness and scalability, and demonstrating its expected accuracy as well as utility in understanding distributed programs (Section 5).

2. MOTIVATION AND BACKGROUND

2.1 Motivating Example

When maintaining and evolving a distributed program which consists of multiple components, the developer needs to understand potential change effects not only in the component where the change is proposed, but also in all other components. To achieve better flexibility and scalability, the components that constitute a distributed system are usually loosely coupled or entirely decoupled as a result of implicit invocations among them realized via a socket-based message passing. This design paradigm, however, greatly reduces the utility of existing impact-analysis techniques.

Consider the example program E of Figure 1, which consists of two components: a server and a client, implemented in classes `S` and `C`, respectively. The client simply retrieves the largest character in a given string by sending the task to the server (line 13), which finishes the task and sends the result back to the client (line 6). Suppose now the developer proposes to apply a new algorithm in the `S::getMax` method as part of an upgrade plan for the server and, thus, needs to determine which other parts of the program may have to be changed as well. Having an available set I of inputs, the developer wants to perform a dynamic impact analysis to get a quick but safe estimation of the potential impacts of the candidate change with respect to the program executions on I. Note that static approaches would be largely disabled by the implicit communication via socket between these two components.

Table 1: A full method event sequence of program E

Server process		Client process	
Method Event	Timestamp	Method Event	Timestamp
`S::main`$_e$	0	`C::main`$_e$	0
`S::init`$_e$	1	`C::init`$_e$	1
`S::init`$_i$	2	`C::init`$_i$	2
`S::init`$_x$	3	`C::init`$_x$	3
`S::main`$_i$	4	`C::main`$_i$	4
`S::serve`$_e$	5	`C::compute`$_e$	5
E_c(C,S)	-	`C::shuffle`$_e$	6
`S::getMax`$_e$	**10**	`C::shuffle`$_i$	7
`S::getMax`$_i$	11	`C::shuffle`$_x$	8
`S::getMax`$_x$	12	`C::compute`$_i$	9
`S::serve`$_i$	13	E_c(C,S)	-
E_c(S,C)	-	E_c(S,C)	-
`S::serve`$_x$	14	`C::compute`$_x$	**14**
`S::main`$_i$	15	`C::main`$_i$	15
`S::main`$_x$	16	`C::main`$_x$	16

To accomplish this task, method-level dynamic impact-prediction techniques of various cost-effectiveness tradeoffs (e.g., [4, 11, 13]) seem to be able to offer the developer many options. However, since there is no explicit dependencies between `S` and `C`, existing approaches would predict impacts within the *local* component (i.e., where the changes are located; S in this case) only. In consequence, the developer would have to ignore impacts in *remote* components (C in this case), or make a worst-case assumption that all methods in remote components are to be impacted.

Scope. As illustrated by E, the distributed system we address is one in which components located at networked computers communicate and coordinate their actions *only* by passing messages [17]: The components run concurrently in multiple processes *without a global clock*, making it hard to infer impacts across components.

2.2 Dynamic Impact Analysis

Typically, a dynamic impact analysis technique inputs a program P, an input set I, and a *query set* M (the set of methods for which impacts are to be queried), and outputs an *impact set* (the set of methods in P potentially impacted) of M when running I. One representative such technique is based on the execute-after-sequences (EAS) [4], which computes impacts from the execution order of methods. Given a query c, EAS considers all methods that execute after c as potentially affected by c or by any changes to it.

To find the execution order, EAS records two main method events using two integers for each method m: the first time m is entered and the last time program control is returned into m. Then, the analysis infers the execute-after relations according to the occurrence time of those events. In presence of multi-threaded executions, EAS monitors also method returns and treats them as returned-into events. For the concrete set of executions, no methods that never executed after the query c can be impacted by it; thus, the results produced by EAS are safe (i.e., of 100% recall) *relative* *to those executions* [29]. However, the execute-after relation does not always lead to impact relation since a method may execute after the query yet has no any dependence on that query; thus, EAS is imprecise due to its conservative nature [12].

On the other hand, the need for maintaining only little information (i.e., the two integers per method) enables the high efficiency of EAS. Therefore, despite of its known imprecision, impact analysis using execute-after relations like EAS remains a viable option, especially for users who desire getting a safe approximation of impacts quickly, such as the developer in the above example scenario. In fact, to the best of our knowledge, EAS is still the most efficient dynamic impact analysis to date [11,13,14]. Thus, as the first attempt exploring efficient dynamic impact analysis for distributed systems, we start with an EAS-based approach in this work.

2.3 Timing in Distributed Systems

Adopting analyses like EAS for distributed systems is challenging, though, because the execution order of methods is not observed across multiple processes in such systems due to the lack of a global clock. A lightweight approach to clock synchronization, the *Lamport timestamps* (LTS) algorithm [38] employs logical clocks to partially order distributed events over concurrent processes. For each process P_i, the LTS approach first defines a logical clock C_i which is a function that assigns a number $C_i\langle a\rangle$ to an event a in P_i. Then, an event a happens before another event b implies the number assigned to a is less than that assigned to b, or formally

$$a \longrightarrow b \implies C\langle a\rangle < C\langle b\rangle \qquad (1)$$

which is called the *clock condition*. To maintain this condition, the following rules [38] must be observed during the system execution:

- Each process P_i increments C_i between any two successive events that happen in P_i.
- If event a is that process P_i sends a message m, then the message contains a timestamp T_m=$C_i\langle a\rangle$.
- When a process P_j receives a message m, it sets C_j greater than or equal to its current value and greater than T_m.

For our solution to impact analysis for distributed systems, the LTS algorithm can be utilized to preserve the partial ordering of distributed events across multiple processes running on separated machines. Further, this ordering would enable inferring causality between methods hence the computation of impacts of one method on others both within and across system components.

3. APPROACH

To achieve an efficient dynamic impact analysis for distributed programs, DistIA utilizes only lightweight run-time information such as method execution order. We first present the fundamentals underlying our approach, including the definition of method events used by DistIA and its rationale for impact prediction. Then, we give an overview and illustration of the inner workings, followed by details on the analysis algorithms, of the DistIA approach.

3.1 Fundamentals

3.1.1 Method-Execution Events

In the general context of distributed systems, an event is defined as any happening of interest observable from within a computer [38]. More specifically, events in a DEBS are often expressed as messages transferred among system components and defined by a set of attributes [24,45]. While it also deals with message passing in distributed systems, DistIA neither makes any assumption nor reasons about the structure or content of the messages. Particularly for dynamic impact analysis, DistIA monitors and utilizes two major classes of events as defined below:

- **Communication Event.** A communication event E_C is the occurrence of a message transfer between two components $c1$ and $c2$, denoted as $E_C(c1,c2)$ if $c1$ initiates E_C which attempts to reach $c2$. Further, according to the direction of message flow, we distinguish two major subcategories of such events: *sending a message* to a component and *receiving a message* from a component. We refer to the process running the component that sends and receives the message as the *sender process* and *receiver process*, respectively.
- **Internal Event.** An internal event E_I is an occurrence of method execution within a component c, denoted as $E_I(c)$. Further, we differentiate three subcategories of internal events: *entering a method*, *returning from a method*, and *returning into a method*, denoted as m_e, m_x, and m_i, respectively, for the relevant method m.

For internal events, we capture both the return and returned-into events for each method. However, we distinguish them during instrumentation only and treat them equally in the monitoring algorithm (Section 3.3.2). In sequential program executions, a method m is potentially affected by any changes in the query c if m, or part of it, executes after c, and monitoring method entry and returned-into events suffices for retrieving such execute-after relations; in the case of concurrent (single-process) programs, however, m is potentially affected by the changes also if m, or part of it, executes in parallel with c, and method return events need to be monitored as well in order to correctly identify the impact relations from interleaving method executions in multiple threads, as shown in [4].

3.1.2 Basic Impact Inference

One challenge to developing DistIA is to infer impact relations based on execution order in the presence of asynchronous events over concurrent multiprocess executions. Fortunately, maintaining a logical notion of time per process to discover just a partial ordering of method-execution events suffices for that inference required in DistIA. The impact relation between any two methods can be semantically over-approximated by the happens-before relation between relevant internal events of corresponding methods; and the partial ordering of the internal events reveals such happens-before relations [38]. Formally, given two methods $m1$ and $m2$, we have

$$m1_e \prec m2_x \bigvee m1_e \prec m2_i \Longrightarrow m1 \textit{ impacts } m2 \qquad (2)$$

where $\prec$ denotes the *happens-before* relation. Without loss of generality, $m1_e \prec m2_x$ and $m1_e \prec m2_i$ imply that "$m2$ executes after or in parallel with $m1$, thus $m2$ may be affected by (any changes in) $m1$", hence the impact relation between $m1$ and $m2$.

Based on the above inference, for a given query c, computing the impact set $IS(c)$ of c is reduced to retrieving methods, from multi-process method-execution event sequences, that satisfy the partial ordering of internal events of candidate methods as follows:

$$IS(c) = \{m \mid c_e \prec m_i \vee c_e \prec m_x\} \qquad (3)$$

Note that only internal events are directly used for impact inference, while communication events are utilized to maintain the partial ordering of internal events in all processes. Also, while the impact inference in DistIA is conservative, it is sound [29] and only requires lightweight dynamic analysis and computation.

3.1.3 Exploiting Message-Passing Semantics

The above basic inference leads to a safe yet possibly *overly* conservative approximation of dynamic impacts, because it is based purely on happens-before relations between method events. However, since message passing is the only communication channel between processes in the systems we address (Section 2.1), a method ${P_1}^m$ in process P_1 would not be impacted by a method ${P_2}^m$ in process P_2 if P_1 never received a message from P_2 *before* the last

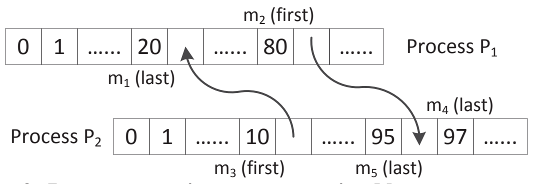

Figure 2: Inter-process impact constrained by message passing.

execution event of ${P_1}^m$, even if ${P_1}^m$ is ever executed after ${P_2}^m$ in the whole-system method event partial ordering.

Figure 2 illustrates how message passing could constrain the impact relation defined by Equation 2. Each numbered cell represents the first or last execution event of a method with the number indicating the time (stamp) when that event occurs, and each empty cell represents a message receiving or sending event. The left and right arrowed lines indicate the *first* message passing from P_2 to P_1 and P_1 to P_2, respectively. m_1 in P_1 was executed after, but would not be impacted by, m_3 in P_2 since P_2 sent the first message to P_1 after the last time m_1 was executed. Similarly, m_5 would not be impacted by m_2. m_4, however, is potentially impacted by m_2 because m_4 was executed after P_2 received a message from P_1.

More generally, considering the message-passing semantics, ${P_i}^m$ is potentially impacted by ${P_j}^{m'}$ only if (1) the first execution event of ${P_j}^{m'}$ happens before the last execution event of ${P_i}^m$, (2) P_j sends at least one message to P_i, and (3) the last execution event of ${P_i}^m$ happens after the first message receiving event in P_i from P_j. Formally, let $T_F(m)$ and $T_L(m)$ denote the time (stamp) of the first and last execution event of a method m, respectively, and $T_S(P)$ denote the time (stamp) of the event of receiving the first message from process P, we define a customized form of partial order relation between two methods ${P_i}^m$ and ${P_j}^{m'}$ as follows:

$$
{P_j}^{m'} \prec {P_i}^m := \begin{cases} T_L({P_i}^m) \geq T_F({P_j}^{m'}), & \text{if } i = j \\ T_S(P_j) \neq \text{null} \wedge T_L({P_i}^m) \geq \\ \quad \max(T_F({P_j}^{m'}), T_S(P_j)), & \text{if } i \neq j \end{cases} \tag{4}
$$

With this constrained definition of $\prec$, a more precise impact inference is obtained from Equation 3 since the potentially impacted methods identified by the basic inference that do not satisfy the additional constraints (2) and (3) will be pruned. Again, given the scope of distributed systems we address, this pruning does not reduce the safety of resulting impact sets. To facilitate our discussion and evaluation later on, hereafter we refer to DistIA with and without the pruning as its *basic* and *enhanced* version, respectively.

3.2 Technique Overview

3.2.1 Workflow

The overall workflow of our technique is depicted in Figure 3, where the three primary inputs are the program D under analysis, a set I of program inputs for D, and a query set M. An *optional* input, a message-passing API list L can also be specified to help DistIA identify program locations where probes for communication events should be instrumented (as detailed in Section 4). The output of DistIA is a set of potential impacts of M computed from the given inputs in four steps as annotated in the figure.

The *first step* instruments the input program D for both monitoring method-execution and message-passing events, and for synchronizing logical clocks among concurrent processes. This step produces the instrumented version D' of D. The *second step* executes D' on the given input set I, during which internal events are produced and time-stamped by means of communication events such that the partial order for all internal events is preserved. Meanwhile, a *message-receiving map* is produced, where each entry indi-

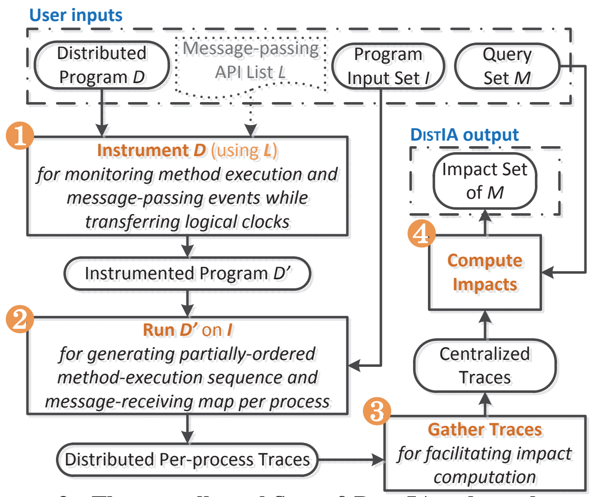
User inputs
Distributed Program D
Message-passing API List L
Program Input Set I
Query Set M
1
Instrument D (using L)
for monitoring method execution and message-passing events while transferring logical clocks
Instrumented Program D'
2
Run D' on I
for generating partially-ordered method-execution sequence and message-receiving map per process
Distributed Per-process Traces
DistIA output
Impact Set of M
4
Compute Impacts
Centralized Traces
3
Gather Traces
for facilitating impact computation

Figure 3: The overall workflow of DistIA, where the numbered steps are detailed in Section 3.2.1.

cates the sending process and time of a message-receiving event—for each unique sending process only the first such event is recorded. In the *third step*, event traces generated in all processes are gathered to the machine where the impact computation is to be performed. The *fourth step* takes the query set M and centralized traces to compute the impact set of M using the impact inference.

3.2.2 Illustration

To illustrate the above process flow, consider program E of Figure 1. DistIA first instruments E and produces the instrumented server and client components S' and C'. Then, suppose the components are deployed on two distributed machines, and S' starts before C'. When running concurrently, S' and C' generate two method-event sequences in two separate processes, as listed *in full* in the first two and last two columns in Table 1, respectively. As shown, logical clocks are updated upon communication events. For instance, the logical clock of the server process is first updated to 10 upon the event E_c(C,S) originated in the client process, which is greater by 1 than the current logical clock of the client process. Later, the client logical clock is updated to 14 upon E_c(S,C). The internal events are time-stamped by these logical clocks while communication events are not, as marked by '-' (i.e., not applicable).

Next, suppose the query set M={`S::serve`}, DistIA gathers traces of the two processes and, by inferring impact relations from the time-stamped events, the basic version gives {`S::getMax`, `S::serve`, `S::main`, `C::shuffle`, `C::compute`, `C::main`} as the impact set of M. By exploiting message-passing semantics, the enhanced version prunes `C::shuffle` from this impact set according to Equation 4. As demonstrated, DistIA can predict impacts across distributed components (processes). For example, if the developer plans for a change to method `serve` in the server, the methods `compute` and `main` in the client, in addition to the other two server methods, are potentially affected and, thus, need impact inspection by the developer before applying that change.

3.3 Analysis Algorithms

3.3.1 Partial Ordering of Internal Events

Preserving the partial ordering of internal events is at the core of the basic DistIA, for which we adopt the LTS approach [38] as described before based on a logical notion of time.

Algorithm 1 summarizes in pseudo code the DistIA algorithm for partially ordering internal events based on the original LTS. The logical clock of the current process C is initialized to 0 upon process start, as is the global variable `remaining`, which tracks the

Algorithm 1 Monitoring communication events

```
let $C$ be the logical clock of the current process
remaining = 0 // remained length of data to read
 1: function SENDMESSAGE($msg$)// on sending a message $msg$
 2:     $sz$ = length of $sz$ + length of $C$ + length of $msg$
 3:     if using the enhanced version then
 4:         $sz$ += length of the sender process id $sid$
 5:         pack $sz$, $C$, $sid$, and $msg$, in order, to $d$
 6:     else
 7:         pack $sz$, $C$, and $msg$, in order, to $d$
 8:     write $d$
 9: function RECVMESSAGE($msg$)// on receiving a message $msg$
10:     read data of length $l$ into $d$ from $msg$
11:     if remaining $> 0$ then
12:         remaining -= $l$
13:         return $d$
14:     retrieve and remove data length $k$ from $d$
15:     retrieve and remove logical clock $ts$ from $d$
16:     remaining = $k$ - length of $k$ - length of $ts$ - $l$
17:     if $ts > C$ then
18:         $C = ts$
19:     increment $C$ by 1
20:     if using the enhanced version then
21:         retrieve and remove sender process id $sid$ from $d$
22:         add ($sid$, $C$) to the message-receiving map if $sid \notin$ its key set
23:         remaining -= length of $sid$
24:     return $d$
```

remaining length of data most recently sent by the *sender* process. The rest of this algorithm consists of two parts, as are trigged upon the occurrence of communication events during system executions.

The first part is the run-time monitor SENDMESSAGE trigged online upon each message-sending event. The monitor piggybacks (prepends) two extra data items to the original message: the total length sz of the data to send, and the present value of the local logical clock C (of this *sender* process) (lines 2–7); then, it sends out the packed data (line 8). The sender process id is also packed in the enhanced version (lines 3–5).

The second part is the other monitor RECVMESSAGE, which is trigged online upon each message-receiving event. After reading the incoming message into a local buffer d (line 10), the monitor decides whether to simply update the size of remaining data and return (lines 11–13), or to extract two more items of data first: the new total data length to read, and the logical clock of the peer *sender* process (lines 14–15). In the latter case, the two items are retrieved, and then removed also, from the entire incoming message. Next, the remaining data length is reduced by the length of the data already read in this event (line 16), and the local logical clock (of this *receiver* process) is compared to the received one, updated to the greater, and incremented by 1 (lines 17–19). In the enhanced version, the sender process id is retrieved as well (line 21) and a new entry is added to the message-receiving map if this is the first message received from that sender process (line 22). Lastly, the monitor returns the message as originally sent in the system (i.e., with the prepended data taken away).

To avoid interfering with the message-passing semantics of the original system, DISTIA keeps the length of remaining data (with the variable `remaining` in the algorithm) to determine the right timing for logical-clock (and sender id) retrieval. In real-world distributed programs (e.g., Zookeeper [3]), it is common that a *receiver* process may obtain, through several reads, the entire data sent in a single write by its peer *sender* process. For example, a first read just retrieves the data length so that an appropriate size of memory can be allocated to take the actual data content in a second read. Therefore, not only is it unnecessary to attempt retrieving the prepended data items (data length and logical clock) in the second read since the first one should have already done so, but also such

Algorithm 2 Computing impact sets

```
let $P_1, P_2, ..., P_n$ be the $n$ concurrent processes of the system
let $c$ be the query method
 1: locIS = $\emptyset$, extIS = $\emptyset$, comIS = $\emptyset$
 2: for $i$=1 to $n$ do
 3:     $ts_q$ = computeLocalImpact($c$, locIS, tr($P_i$))
 4:     if $ts_q$==null then continue
 5:     for $j$=1 to $n$ do
 6:         if $i == j$ then continue
 7:         if using the enhanced version $\wedge$ S(tr($P_j$))[$P_i$]==null then
 8:             continue
 9:         for each method $m \in$ keyset(R(tr($P_j$))) do
10:             if R(tr($P_j$))[$m$] >= $ts_q$ then
11:                 if using the enhanced version then
12:                     if R(tr($P_j$))[$m$] >= S(tr($P_j$))[$P_i$] then
13:                         extIS $\cup$= $\{m\}$
14:                 else
15:                     extIS $\cup$= $\{m\}$
16: comIS = locIS $\cap$ extIS
17: return locIS, extIS, comIS
```

attempts can break the original network I/O protocols. DISTIA addresses this issue by piggybacking the length of data to send and tracking the remaining length of data to receive.

3.3.2 Monitoring Internal Events

The *basic* impact inference in DISTIA relies on the execution order of methods that is deduced from the timestamps attached to all internal events, for which DISTIA monitors the occurrence of each internal event. However, as proved in [4], recording the *first* entry and *last* returned-into (or return) events only is equivalent to tracing the full sequence of those events for the dynamic impact analysis. Similarly, this equivalence also applies in DISTIA. Thus, instead of keeping the timestamp for every internal-event occurrence (as shown in Table 1), DISTIA only records two key timestamps for each method m: the one for the first instance of m_e, and the one for the last instance of m_i or m_x, whichever occurs later.

Accordingly, the online algorithm for monitoring internal events uses two counters to record the two key timestamps for each method, similar to what EAS does but different in that it does so *in each process* rather than in just one process. Also, we use the per-process logical clock, instead of a global integer as used by EAS, to update the per-method counters during runtime. In the meanwhile, the logical clock C_i of each process P_i is maintained as follows:

- Initialize C_i to 0 upon the start of P_i.
- Increase C_i by 1 upon each internal event occurred in P_i.
- Update C_i upon each communication event occurred in P_i via the two online monitors shown in Algorithm 1.

Finally, for the offline impact computation in DISTIA, the online algorithm here also dumps per-process internal-event sequences (i.e., the two timestamps for each executed method) as traces upon the event of program termination (of each component). Additionally, in the enhanced version, the message-receiving map is also serialized as part of the per-process traces.

3.3.3 Impact Computation

During system executions, the online internal-event monitoring algorithm generates event traces concurrently (and usually on distributed machines). Since it computes impacts offline, DISTIA gathers these traces after they are produced to one machine, and computes impact sets there as outlined in Algorithm 2.

For a detailed analysis, we refer to the process where the query is first executed as *local process* versus all other processes as *remote process*, and impacts in local and remote processes as *local impacts* and *remote impacts*, respectively. For a given query c, DISTIA computes its impact set as three subsets of interest: *local impact set*, *remote impact set*, and their intersection referred to as *common impact set* (denoted as *locIS*, *extIS*, and *comIS*, respectively).

The algorithm takes c and n per-process traces as input, and outputs the three subsets (line 17) all initialized as empty sets (line 1). It traverses the n processes (loop 2–15) taking each as the local process (line 2) against all others as remote processes (lines 5 and 6) to first obtain the local impact set (line 3) and then the remote impact set (loop 7–15) based on Equation 4. *tr(P)* denotes the trace of process P, *R(t)* denotes the hashmap from each executed method to its last execution event time in trace t, *S(t)* denotes the hashmap from the id of each sender process to the event time when the remote process receives the first message from the sender, and *keyset(.)* returns the key set of a hashmap. In the enhanced version, if the (remote) process P_j never received any message from the (local) process P_i, no remote impacts would be found in P_j (lines 7–8). The subroutine `computeLocalImpact` identifies methods whose last execution is not earlier than the first execution of c in *tr(P_i)* as in EAS [4], and unionizes the local impact set in that trace with *locIS*. It returns the first execution event time ts_q of c if c is found in the input trace and `null` otherwise (line 4). With respect to a single test, the impact-computation algorithm computes the impact set of one method at a time; for multiple methods in the query set, the result is the union of all the one-method impact sets computed separately (e.g., in parallel). Similarly, the impact set for multiple tests is the union of all per-test impact sets.

With the enhanced version, we focus on safely reducing only the remote impact sets produced by the basic version. A complementary approach would be to reduce local impacts, such as through static-dependence-based pruning as in [11]. It is straightforward to incorporate such approaches in DistIA to further improve its effectiveness, but that will largely increase the cost as well [13].

4. IMPLEMENTATION

DistIA[1] consists of three main modules: an instrumenter, two sets of run-time monitors, and a post-processor. Careful treatments are crucial for a non-interfering implementation as recaped below.

4.1 Instrumenter

DistIA instruments the input program such that all relevant events are monitored accurately, which is crucial to the soundness [29] and precision of DistIA. We used Soot [37] for the instrumentation in two main steps. First, DistIA inserts probes for the three types of internal events in each method, for which we reused relevant modules of Diver [11], a hybrid impact analysis that is built on Soot and uses method-execution traces also. The second step is to insert probes for communication events, for which DistIA uses the list L of specified message-passing APIs to identify probe points based on string matching: L includes the prototype of each API used in the input system for network I/Os. If L is not specified, a list of basic Java network I/O APIs is used covering two common means of blocking and non-blocking communication: Java Socket I/O [50] and Java NIO [49] (both are *socket-based*).

4.2 Run-time Monitors

The two sets of run-time monitors implement the two online algorithms: the first focuses on monitoring internal events and the second is dedicated to preserving the partial ordering of them. The first set again reuses relevant parts of Diver [11]. For the second set, instead of invoking additional network I/O API calls to transfer logical clocks, the monitors *take over* the original message passing so that they can piggyback the three extra data items (i.e., the data length, logical clock, and sender id) to the original message.

Our experience comparing different instrumentation schemes suggested that the piggyback strategy is more viable than inserting additional network I/O API calls, especially when dealing with systems using selector-based non-blocking communications [5]. For instance, the ShiVector tool [1, 8], which adopts the latter, works with only two of our six subject programs (MultiChat and Voldemort). One reason as we verified is that, for a pair of an original call and the corresponding additional call, the two messages may not be read in the same order by the *receiver* process as in which they are sent by the *sender* process. As a result, an original message-receiving call may encounter unexpected data in the message hence causing network I/O protocol violations even system failures.

4.3 Post-processor

The post-processer is the module that actually answers impact-set queries. To that end, it starts by gathering distributed traces with a helper script which passes per-process traces to the offline impact-computation algorithm. To compute the impact set following Algorithm 2, the post-processor retrieves the partial ordering of internal events by comparing the associated timestamps.

5. EVALUATION

Our evaluation was guided by the following research questions:

- **RQ1** How effective is DistIA compared to existing options?
- **RQ2** How are the impacts distributed across process boundaries in the impact sets given by DistIA?
- **RQ3** How efficient and scalable is DistIA?

The main goal of this evaluation was to investigate the effectiveness (RQ1) and efficiency (RQ3) of DistIA. We also intended to examine the composition of DistIA impact sets concerning how impacts propagate within and across constituent components (processes) in distributed systems (executions) (RQ2).

5.1 Experiment Setup

We evaluated DistIA on six distributed Java programs, as summarized in Table 2. The size of each subject is measured by the number of non-comment non-blank Java source lines of code (*#SLOC*) and number of methods defined in the subject (*#Methods*) that we actually analyzed. The last two columns list the type of test input used in our study (one test case per type) and the number of methods executed at least once in the respective test (*#Cov.M.*) which we all used as impact-set queries. The third and fifth columns together also give the method-level coverage of the test inputs.

MultiChat [26] is a chat application where multiple clients exchange messages via a server broadcasting the message sent by one client to all others. NioEcho [61] is an echo service via which the client just gets back the same message as it sends to the server. Open Chord is a peer-to-peer lookup service based on distributed hash table [6]. ZooKeeper [3, 28] is a coordination service for distributed systems to achieve consistency and synchronization. Voldemort [2] is a distributed key-value storage system used at LinkedIn. Freenet [63] is a peer-to-peer data-sharing platform offering anonymous communication. Some of these systems use Socket I/O or Java NIO only, while others use both mechanisms, for message passing among their components. For all subjects, we checked out from their official online repositories the latest versions or revisions as shown in (the parentheses of) Table 2.

We chose these subjects such that varied system scales and architectures, application domains, and uses of either and both of blocking and non-blocking I/Os are all considered. We chose test inputs to cover different types of inputs when possible, including system

[1]Code and study results are at https://chapering.github.io/distea/.

Table 2: Statistics of experimental subjects

Subject (version)	#SLOC	#Methods	Test type	#Cov.M.
MultiChat (r5)	470	37	integration	25
NIOEcho (r69)	412	27	integration	26
Open Chord (v1.0.5)	38,084	736	integration	354
ZooKeeper (v3.4.6)	62,450	4,813	integration	749
			system	817
			load	798
Voldemort (v1.9.6)	163,601	17,843	integration	2,048
			system	1,242
			load	1,323
Freenet (v0.7.0)	196,281	16,673	integration	2,477

test, integration test, and load test, which we assume exercise typical, overall system behaviour instead of a small specific area of the source code. The integration tests were created manually, while the other types of inputs come with the subjects in their repository.

In each integration test, we started two to five server and client nodes on different machines and performed client operations that cover main system services—specially for peer-to-peer systems, we operated on all nodes, and for ZooKeeper we started a container node in addition. For MultiChat and NioEcho, the client requests were sending random text messages. For ZooKeeper, the ordered client operations were: create two nodes, look up for them, check their attributes, change their data association, and delete them. For Voldemort, the operations were: add a key-value pair, query the key for its value, delete the key, and retrieve the pair again. For Open Chord, the operations were in order: create an overlay network on machine (node) A, join the network on machines B and C, insert a new data entry to the network on C, look up and then delete the data entry on A, and list all data entries on B. For Freenet, we first uploaded a file to the network with a note on node A, shared it to nodes B and C, then on B and C accepted the sharing request and the note, followed by downloading the file and replying with a note.

5.2 Experimental Methodology

As mentioned earlier, we did not have existing dynamic impact analysis to compare with DistIA. Similarly, we know of no available (dynamic) slicer working with real-world distributed systems like our subjects. A possible alternative to DistIA is a coverage-based approach (referred to as *MCov*) which takes all methods executed in remote processes as potentially impacted. In fact, *MCov* is a realistic solution adapted for distributed systems from CoverageImpact [51], a major existing option for centralized programs. Thus, we consider *MCov* as the baseline technique and the covered (executed) sets of methods as baseline impact sets.

We considered every method of each subject as a query, yet we report results only for queries executed at least once in one process—only such queries have a non-empty impact set. For methods executed in more than one processes, we took them as separate queries each per process. For each query, we measure the effectiveness of DistIA by comparing the impacts it predicted to those given by *MCov* in terms of impact-set size ratios, and examine the composition of the impact set concerning its two subsets: *local impact set* and *remote impact set*, while also analyzing the *common impact set*. Accordingly, we also measure the effectiveness of DistIA with respect to these subsets relative to the corresponding *MCov* results. Finally, besides the impact querying time, we report the instrumentation and run-time costs of DistIA, and storage costs of event traces, all as efficiency metrics. We also compare the effectiveness and efficiency of the basic versus the enhanced version. The machines used in our studies were all Linux workstations with an Intel i5-2400 3.10GHz CPU and 8GB DDR2 RAM.

5.3 Results and Analysis

5.3.1 RQ1: Effectiveness

Figure 4 shows the effectiveness of the basic DistIA, with one plot depicting the data distribution for each subject and input type, shown as the plot title (hereafter, the input type is omitted for subjects for which only an integration test is available and utilized). Each plot includes three box plots showing that data distribution for one of three categories (on x axis): the holistic impact set (*all*) and its two subsets (*local* and *remote*), with each underlying data item indicating the effectiveness metric (on y axis) for one query. The total numbers of queries involved are shown in the parentheses.

The results indicate that even the basic version is much more effective than *MCov*, reducing the impact sets of the latter by 10% up to over 45% in most cases (see the 75% quartiles). Compared to the small subjects, the four large subjects see noticeably better effectiveness. Also, the ratios with respect to *all* (rightmost boxplots) impact sets are mostly higher than those to the two subsets. The reason is that the two subsets from both approaches have substantial intersections, while the ones from DistIA are consistently smaller than those from *MCov*.

Complementary to the five-number summaries of Figure 4, Table 3 (left seven columns) gives the mean effectiveness. Overall, the basic DistIA reports on average only 69% of the impacts produced by *MCov*. In particular, the reductions in remote impact sets are even higher, by 38% on average and well above 35% mostly for the three largest subjects. This implies that, relative to the baseline, developers can save the time that would be spent on inspecting over a third of the impacts propagated to remote processes. Together with the impact size ratios, the sizes of DistIA impact sets (left four columns) indirectly give the baseline impact-set sizes, implying the significance of the large impact-set reductions by our technique. The bottom row shows the mean for all individual queries across the six subjects rather than the average of the means above.

The effectiveness of the enhanced version of DistIA is highlighted against the basic version in Figure 5, where the height of each bar indicates the mean (holistic-) impact-set size ratio to the same baseline *MCov* with the associated variation indicated by the cap above the bar. The means of the basic version correspond to the seventh column of Table 3. For a fair comparison, both versions took exactly the same traces to compute the impact sets of same queries, with the basic version simply ignoring the message-receiving maps in the traces during the impact-computation phase.

The results reveal that the pruning based on message-passing semantics further reduced the baseline impact sets considerably beyond the basic version, by around 5% to over 50% on average for individual subjects/inputs and 13.2% overall (the rightmost bars). The improvements also tend to be more significant on larger subjects than on smaller ones. For each query, both versions constantly produced the same local impact sets, which is expected as the enhancement prunes remote impacts only. In all, the enhanced DistIA reported in its impact sets only 57% of the methods identified by the baseline as potentially impacted, implying even greater savings in developers' impact-inspection effort than the basic version.

5.3.2 RQ2: Impact-Set Composition

Figure 6 plots the impact-set composition for each individual query numbered on the y axis, where the x axis indicates the percentage of three complementary sets, *local*, *remote*, and *common*, for each subject and input type. The common sets have been removed from the local and remote subsets, in this figure only, to help clarify the composition. A first observation is that remote impacts constantly account for as large portions as local impacts in corresponding holistic impact sets. This finding suggests that impacts

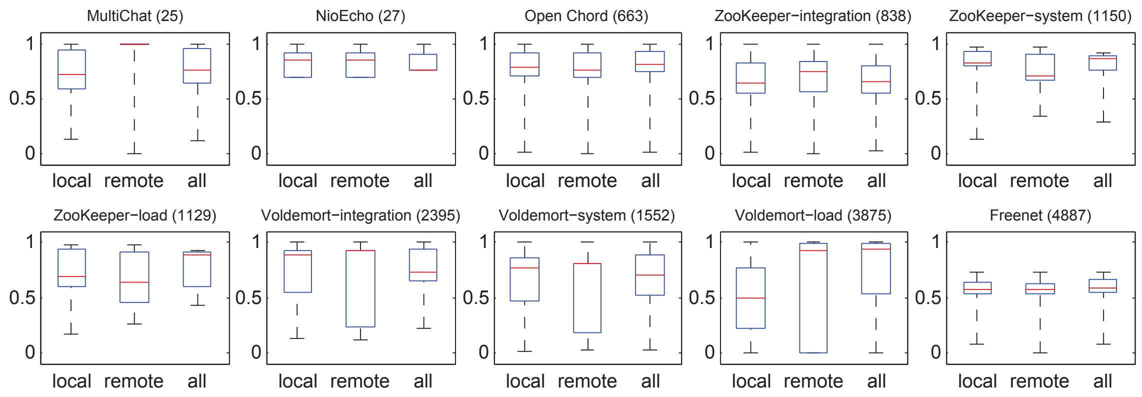

Figure 4: Effectiveness of the basic DISTIA expressed by the ratios (y axes, the lower the better) of its per-query impact-set sizes, including those of the local and remote subsets (x axes), to the baseline, per subject and input type (with #queries in parentheses).

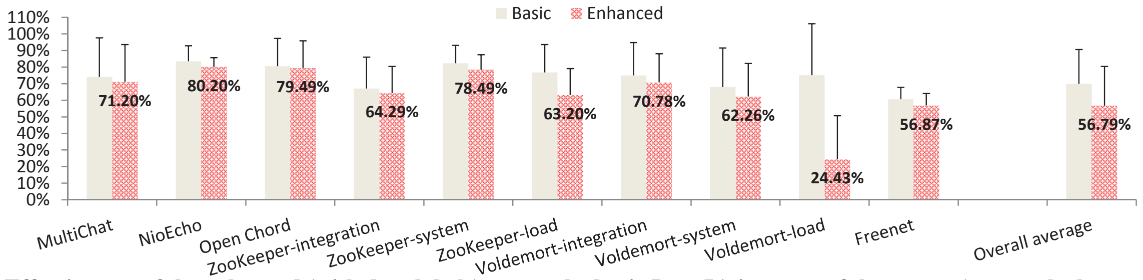

Figure 5: Effectiveness of the enhanced (with data labels) versus the basic DISTIA in terms of the means (y axes, the lower the better) and their standard deviations (caps atop the bars) of per-query impact-set size ratios to the baseline, per subject/input and overall.

can propagate almost as extensively to remote processes as within local ones in (the executions of) distributed programs, confirming the necessity of analyzing impacts beyond process boundaries.

Another finding is that, for almost all queries, there were same methods potentially impacted in both local and remote processes. This implies that, in distributed systems, components often share common functionalities. The sizes of common impact sets could be a metric of coupling and code reuse among distributed components. In the majority of these ten cases, there were as many potentially impacted methods shared by both local and remote processes as those only appeared in either, suggesting that the component-level code reuse is not only widely existent but also quite significant.

The rightmost two plots show the composition with the enhanced version in contrast to the results with the basic version (left to the vertical line) for the two best cases (ZooKeeper-load and Voldemort-load) to highlight the effectiveness improvement. These two cases are consistent with the two best seen in Figure 5. The *enhanced* results for other eight cases are omitted due to space limit, and *basic* results are shown for all cases to partly explain/justify the enhancement gained via reducing the sizes of remote impact sets.

5.3.3 *RQ3: Efficiency*

Table 3 (the rightmost six columns) lists all relevant costs of the basic DISTIA for this study, including the time cost of (bytecode) instrumentation, run-time overhead measured as ratios of the execution time of the original program (*Normal run*) over the instrumented one (*Instr. run*), and impact querying time.

The instrumenter took longer for larger subjects as expected, yet remained within 3 minutes even on the largest system Freenet. Note that this is a one-time cost (for the single program version analyzed by DISTIA) as the instrumented code can be executed on any inputs and used for computing any queries afterwards. Run-time and querying costs are consistently correlated to subject sizes as well as the type of inputs and the size of execution traces (last column), with the worst case seen by ZooKeeper and Voldemort, respectively. Nevertheless, the run-time overhead is at worst 21% and the longest querying time is less than one second.

Storage costs are also tightly connected to the type of inputs in addition to subject sizes, of which the largest is less than 1MB for the load test of Voldemort. In other cases, this cost is at most 0.5MB, with an overall average of less than 0.3MB.

In all, the results suggest that the basic DISTIA is highly efficient in both time and space dimensions, and that it appears to be readily scalable to large systems: It costs on average one minute for instrumentation and 56ms for computing the impact set per query, with mean run-time overhead of about 7%.

Tracing the first message-receiving events and utilizing them in addition, the enhanced version is expected to incur higher time and space costs. Yet, our results (not shown here) reveal that the increases are quite marginal and thus negligible: on overall average, the run-time and space overhead is less than 1% higher, and the querying time is 9.8ms longer. Since both versions share the same instrumenter, the instrumentation cost remains the same. Thus, in all, the enhanced version tends to be much more cost-effective.

5.4 Case Studies

To further investigate the effectiveness and practical utility of DISTIA, we conducted two exploratory case studies. For space reasons, we leave the details to a technical report on DISTIA [15].

In the first study, we randomly picked two queries of MultiChat and three of Voldemort with an input for each query also randomly chosen from the tests used above. We then manually determine the ground-truth impact sets of these queries to gauge the accuracy of DISTIA results. Overall, DISTIA had an average precision of 56.9% with the basic version and of 71.2% with the enhanced version, and both versions always had 100% recall for any query.

In the second study, we used the DISTIA impact sets to help understand the inter-component interaction in NioEcho and ZooKeeper: none of the authors had knowledge about internal workings of these programs. We picked two queries from the client and three

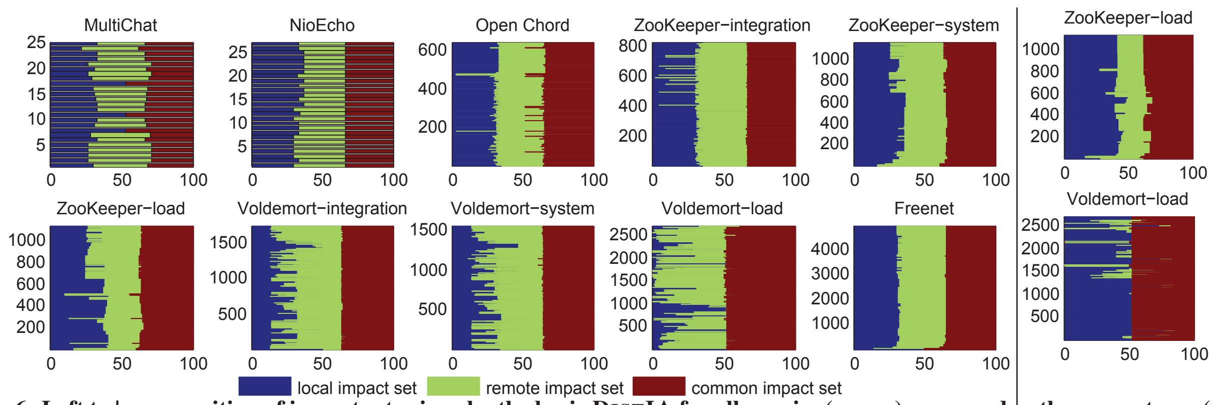

Figure 6: Left to |: composition of impact sets given by the basic DistIA for all queries (y axes) expressed as the percentages (x axes) of local, remote, and common impact sets in the whole impact set per query, for each subject and input type (atop each plot as the title). Right to |: composition of impact sets given by the enhanced DistIA for two subjects/inputs with the best effectiveness gains.

Table 3: Mean effectiveness, time-cost breakdown, and storage costs of the basic DistIA

Subject & input	Mean impact-set sizes			Mean impact size ratios			Time costs in millisecond (ms)					Storage costs in KB
	Local	Remote	All	Local	Remote	All	Instrumentation	Normal run	Instr. run	Run-time overhead	Querying (stdev)	
MultiChat	14.0	4.5	18.5	71.70%	85.03%	74.08%	12,817	5,461	5,735	5.02%	4 (2)	7
NioEcho	11.4	11.3	21.7	84.21%	83.60%	83.47%	13,365	3,213	3,619	12.64%	4 (2)	5
Open Chord	248.0	245.9	276.0	77.75%	77.09%	80.46%	14,533	4,856	4,931	1.54%	8 (5)	73
ZooKeeper-integration	284.1	265.5	492.1	66.90%	67.72%	67.04%		37,239	38,396	3.11%	11 (2)	94
ZooKeeper-system	672.3	672.8	948.3	81.91%	74.89%	82.25%	39,124	15,385	18,565	20.67%	24 (6)	132
ZooKeeper-load	577.1	560.7	837.8	73.12%	62.86%	76.80%		94,187	98,891	4.99%	22 (5)	142
Voldemort-integration	654.6	578.1	1052.7	74.85%	65.87%	74.98%		17,755	18,662	5.11%	22 (7)	315
Voldemort-system	522.1	478.6	842.9	67.41%	62.54%	67.97%	132,536	11,136	12,232	9.84%	19 (4)	197
Voldemort-load	286.9	709.9	967.6	49.37%	55.93%	75.06%		21,066	21,198	0.63%	29 (5)	782
Freenet	2787.8	2780.0	2956.6	57.56%	57.42%	60.64%	165,174	54,794	61,876	12.92%	114 (17)	527
Overall average	**820.4**	**850.6**	**1037.8**	**63.74%**	**62.01%**	**69.99%**	**62,924.8**	**26,509.2**	**28,410.5**	**7.17%**	**56.2 (45.1)**	**227.4**

from the server of NioEcho that looked relevant to messaging. The *remote* impact sets clearly revealed how the client initiates a request handled by the server which later sent a response back to the client. The case of ZooKeeper was much more complicated, yet examining both the local and remote impacts of one fundamental operation `getData` [28] actually enabled us to identify the major data transaction protocol between two nodes. In both cases, the event *ordering* was essential for the interaction understanding. The common impact sets, which reveal functionalities shared by components, also helped with the system structure comprehension.

5.5 Threats to Validity

The main threat to *internal* validity lies in possible implementation errors in DistIA and experiment scripts. To reduce this threat, we did a careful code review for our tool and used the two smallest subjects to manually validate their functionalities and analysis results. An additional such threat concerns possible missing (remote) impacts due to network I/Os that were not monitored at runtime. However, we checked the code of all subjects and confirmed that they only used the most common message-passing APIs monitored by our tool when executing the program inputs we utilized. In general, for arbitrary distributed systems, the soundness of DistIA relies on the identification of all such API calls used in the system.

The main threat to *external* validity is that our study results may not generalize to all other distributed programs and input sets. In this study, we considered only limited number of subjects, which may not represent all real-world distributed systems, and only subsets of inputs, which do not necessarily represent all behaviours of the studied systems. To reduce this threat, we have chosen subject programs of various sizes and application domains, including the four industry-scale systems in different areas. In addition, we considered different types of inputs, including integration, system, and load tests. Most of these tests came as part of the subjects except for the integration tests, which we created according to the official online documentation (quick-start guide) of these programs.

The main threat to *construct* validity is the metrics used for the evaluation. Without directly comparable peer techniques in the literature, we assumed that developers would use coverage-based approach like *MCov*, as a representative alternative to DistIA, to narrow down the search space of potential impacts in the context of distributed executions. To mitigate this threat, we examined the composition of each impact set and analyzed the effectiveness with respect to its local and remote subsets in addition to that of the holistic impact set to help demonstrate the usefulness of DistIA.

Finally, a *conclusion* threat concerns the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). Also, the present study only considered potential changes in single methods for each query, while in practice developers may plan for changes in multiple methods at a time, which may lead to different results. To minimize this threat, we adopted the strategy for all experiments and calculated the metrics for every possible query.

6. RELATED WORK

Dynamic Impact Analysis. The EAS approach [4] which partially inspired DistIA is a performance optimization of its predecessor PathImpact [39]. Many other dynamic impact analysis techniques also exist [40], aiming at improving precision [11], recall [42], efficiency [10], and cost-effectiveness [13,14] over PathImpact and EAS. However, these techniques did not address distributed or multiprocess programs that we focus on in this work.

Two recent advances in dynamic impact analysis, DIVER [11] and the unified framework in [14], utilize hybrid program analysis to achieve higher precision and more flexible cost-effectiveness options over EAS-based approaches, but still target centralized programs only. As a first step, DISTIA sacrifices precision for high efficiency. However, it would be interesting to adopt hybrid approaches for distributed systems too. For instance, one may immediately gain better analysis precision by first using static dependencies to prune false-positive impacts *within* each process, as DIVER did, and then propagating impacts across process boundaries by means of the analysis algorithms used in DISTIA. More aggressive pruning may further be obtained by leveraging more and/or finer-grained interprocess dependencies, such as communication dependencies and synchronization dependencies [31, 44], as already exploited by distributed-program slicing techniques [7].

Dependence Analysis of Parallel and Distributed Programs. Using fine-grained dependency analysis, a large body of work attempted to extend traditional slicing algorithms to concurrent programs [25, 35, 47, 65, 66] yet mostly focusing on centralized, and primarily multithreaded, ones. For those programs, traditional dependence analysis was extended to handle additional dependencies due to shared variable accesses, synchronization, and communication between threads and/or processes (e.g., [35, 47]). While DISTIA also handles multithreaded programs, it targets multiprocess ones running on distributed machines, and aims at lightweight predictive impact analysis instead of fine-grained slicing.

For systems running in multiple processes where interprocess communication is realized via socket-based message passing, an approximation for static slicing was discussed in [35]. Various dynamic slicing algorithms have been proposed too, earlier for procedural programs only [16, 18, 27, 31, 34] and recently for object-oriented software also [7, 44, 52]. And a more complete and detailed summary of slicing techniques for distributed programs can be found in [66] and [7]. Although these slicing algorithms were rarely evaluated against large real-world distributed systems, it can be anticipated that they would face scalability issues with large systems based on the limited empirical results they reported and the heavyweight nature of their technical design.

In contrast to these fine-grained (statement-level) analysis, DISTIA aims at a highly efficient method-level dynamic impact analysis that can readily scale to large distributed systems. A few more static analysis algorithms for distributed systems exist as well but focus on other (special) types of systems, such as RMI-based Java programs [59], different from the common type of distributed systems [17] DISTIA addresses. At coarser levels, other researchers resolve dependencies in distributed systems too but for different purposes such as enhancing parallelization [54], system configuration [32], and high-level system modeling [1, 8], or limited to static analysis [24, 46, 53]. In contrast, DISTIA performs code-based analysis while providing more focused impacts relative to concrete program executions than static-analysis approaches.

An impact analysis dedicated to distributed systems, Helios [53] can predict impacts of potential changes to support evolution tasks for DEBS. However, it relies on particular message-type filtering and manual annotations in addition to a few other constraints. Although these limitations are largely lifted by its successor Eos [24], both approaches are *static* and limited to DEBS only, as is the latest technique [64] which identifies impacts based on change-type classification yet ignores intra-component dependencies hence provides merely incomplete results. While sharing similar goals, DISTIA targets a broader range of distributed systems than DEBS using *dynamic* analysis and without relying on special source-code information (e.g., interface patterns) as those techniques do.

Unlike our dependence-based approach, a traceability-based solution [9] is presented in [60] which relies on a well-curated repository of various software models. The dynamic impact analysis for component-based software in [20] works at architecture level, different from ours working for distributed programs at code level.

Logging for Distributed Systems. Targeting high-level understanding of distributed systems, techniques like logging and mining run-time logs [8, 41] infer inter-component interactions using textual analysis of system logs, relying on the availability of particular data such as informative logs and/or patterns in them. DISTIA also utilizes similar information (i.e., the Lamport timestamps) but infers the happens-before relation between method-execution events mainly for code-level impact analysis. Also, DISTIA automatically generates such information it requires rather than relying on existing information in the original programs (e.g., logging statements).

The Lamport timestamp used is related to vector clocks [21, 43] used by other tools, such as ShiVector [1] for ordering distributed logs and Poet [36] for visualizing distributed systems executions. While we could utilize vector clocks also, we chose the Lamport timestamp as it is lighter-weight yet suffices for our current design of DISTIA. In addition, unlike ShiVector, which requires accesses to source code and recompilation using the AspectJ compiler, DISTIA does not have such constraints as it works on bytecode.

Tracing message-passing systems was explored before but for different purposes, such as overcoming non-determinacy/race detection [48] and reproducing buggy executions [33]. Tools like RoadRunner [22] and ThreadSanitizer [58] targeted multithreaded, centralized programs. None of these solutions work for large, diverse distributed systems without perturbations as DISTIA did.

Dynamic Partial Order Reduction (DPOR). DPOR has been used to improve the performance of model checking concurrent software by avoiding the examination of independent transitions [23]. Sharing the spirit of DPOR, especially distributed DPOR [67], DISTIA synchronizes only the first and last method-execution events for more efficient computation of partial ordering of methods *within* processes, and exploits message-passing semantics to avoid partial-ordering independent methods *across* processes.

However, unlike existing DPOR techniques which target multi-threaded programs, DISTIA focuses on multi-process, distributed programs. On the other hand, DPOR may be adopted for dynamic impact analysis of distributed programs with a few extensions and/or accommodations. First, the execution conditions of methods can be modeled as method-level states (versus statement-level states in the original DPOR). Also, to represent state transitions across processes, the identifies of parent processes need to be incorporated in thread identifiers. Finally, partial ordering algorithms like LTS may be employed to determine transition dependence at process level.

7. CONCLUSION

We presented DISTIA, the first dynamic impact analysis for common distributed programs. By partially ordering method-execution events and exploiting message-passing semantics, DISTIA can safely predict for any query the potential impacts both within and across all processes. Through an evaluation on large real-world distributed programs, we have shown the high efficiency of DISTIA and its superior effectiveness over existing options, and initially demonstrated its benefits for distributed system understanding.

8. ACKNOWLEDGMENT

This work was partially supported by ONR Grant N000141410037 to the University of Notre Dame and the faculty start-up funds given by Washington State University to the first author.

9. REFERENCES

[1] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 598–599, 2014.

[2] Apache. Voldemort. https://github.com/voldemort, 2015.

[3] Apache. ZooKeeper. https://zookeeper.apache.org/, 2015.

[4] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 432–441, 2005.

[5] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 169–179, 2013.

[6] Bamberg University. Open Chord. http://sourceforge.net/projects/open-chord/, 2015.

[7] S. S. Barpanda and D. P. Mohapatra. Dynamic slicing of distributed object-oriented programs. *IET software*, 5(5):425–433, 2011.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 468–479, 2014.

[9] S. A. Bohner and R. S. Arnold. *An introduction to software change impact analysis*. Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, June 1996.

[10] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 143–152, 2005.

[11] H. Cai and R. Santelices. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of International Conference on Automated Software Engineering*, pages 343–348, 2014.

[12] H. Cai and R. Santelices. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software*, 103:248–265, 2015.

[13] H. Cai and R. Santelices. A framework for cost-effective dependence-based dynamic impact analysis. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 231–240, 2015.

[14] H. Cai, R. Santelices, and D. Thain. Diapro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Transactions on Software Engineering and Methodology*, 25(2):18, 2016.

[15] H. Cai and D. Thain. Distea: Efficient dynamic impact analysis for distributed systems. *arXiv preprint arXiv:1604.04638*, 2016.

[16] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Proceedings of Advances in Parallel and Distributed Computing*, pages 370–377, 1997.

[17] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition, 2011.

[18] E. Duesterwald, R. Gupta, and M. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511. Springer, 1993.

[19] P. Eugster and K. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of European Conference on Object-Oriented Programming*, pages 570–594. Springer, 2009.

[20] T. Feng and J. I. Maletic. Applying dynamic change impact analysis in component-based architecture design. In *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 43–48, 2006.

[21] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[22] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2010.

[23] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[24] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 367–377, 2013.

[25] D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engineering*, 16(2):197–234, 2009.

[26] GoogleCode. MultiChat. https://code.google.com/p/multithread-chat-server/, 2015.

[27] D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In *IEEE International Conference on High Performance Computing*, pages 15–26. 2000.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[29] D. Jackson and M. Rinard. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–145, 2000.

[30] K. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *Proceedings of International Conference on Distributed Event-Based System*, pages 113–124, 2011.

[31] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 222–229, 1995.

[32] F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE concurrency*, 8(1):26–36, 2000.

[33] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *International Parallel and Distributed Processing Symposium*, pages 219–227, 2000.

[34] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2(2):199–215, 1992.

[35] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, volume 28, pages 178–187, 2003.

[36] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.

[37] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. Soot - a Java bytecode optimization framework. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[39] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 308–318, 2003.

[40] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23:613–646, 2013.

[41] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, 44(1):91–96, 2010.

[42] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero. The hybrid technique for object-oriented software change impact analysis. In *Proceedings of European Conference on Software Maintainance and Reengineering*, pages 252–255, 2010.

[43] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[44] D. P. Mohapatra, R. Kumar, R. Mall, D. Kumar, and M. Bhasin. Distributed dynamic slicing of Java programs. *Journal of Systems and Software*, 79(12):1661–1678, 2006.

[45] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed event-based systems*, volume 1. Springer, 2006.

[46] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[47] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Transactions on Programming Languages and Systems*, 28(6):1088–1144, 2006.

[48] R. H. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[49] Oracle. Java NIO. http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html, 2015.

[50] Oracle. Java Socket I/O. http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html, 2015.

[51] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 128–137, 2003.

[52] S. Pani, S. M. Satapathy, and G. Mund. Slicing of programs dynamically under distributed environment. In *Proceedings of International Conference on Advances in Computing*, pages 601–609. Springer, 2012.

[53] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic. Impact analysis for distributed event-based systems. In *Proceedings of International Conference on Distributed Event-Based Systems*, pages 241–251, 2012.

[54] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3):196–213, 2004.

[55] V. Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, 2006.

[56] V. Rajlich. Software evolution and maintenance. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–144, 2014.

[57] P. Rovegard, L. Angelis, and C. Wohlin. An empirical study on views of importance of change impact analysis issues. *IEEE Transactions on Software Engineering*, 34(4):516–530, 2008.

[58] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[59] M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, 2006.

[60] H. M. Sneed. Impact analysis of maintenance tasks for a distributed object-oriented system. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 180, 2001.

[61] SourceForge. NioEcho. http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode, 2015.

[62] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, pages 51:1–51:11, 2012.

[63] The Freenet team. The Free Network. https://freenetproject.org/, 2015.

[64] S. Tragatschnig, H. Tran, and U. Zdun. Impact analysis for event-based systems using change patterns. In *Proceedings of ACM Symposium on Applied Computing*, pages 763–768, 2014.

[65] J. Xiao, D. Zhang, H. Chen, and H. Dong. Improved program dependence graph and algorithm for static slicing concurrent programs. In *Advanced Parallel Processing Technologies*, pages 121–130. Springer, 2005.

[66] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[67] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Model Checking Software*, pages 58–75. Springer, 2007.