Controlled Re-Execution of Distributed Programs with Disjunctive Predicates

Karim Serhan
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas
kfserhan@gmail.com

Elie Antoun
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas
elieantoun@utexas.edu

***Abstract*—Debugging a distributed application is a difficult task. The non-deterministic nature of a distributed computation rids the developers of the seemingly straightforward ability of reproducing a bug. A mechanism is needed to allow for the re-execution of a computation under the same conditions and ordering of events as a faulty run, in order to allow the developer to reproduce a bug. This mechanism falls under the general problem of predicate control.**

In this paper, we develop a tool that runs an offline predicate control algorithm for a specific type of predicates, disjunctive predicates, by adding additional synchronization to the computation trace. We complement our tool with a random trace generator and an interface that shows a graphical representation of the tool.

Keywords—distributed computing; distributed debugging; disjunctive predicates; controlled re-execution

I. INTRODUCTION

With the proliferation of distributed computing, and their ever-increasing complexity, new tools are needed to complement the design and implementation of distributed systems. The observation of a distributed system is one of the essential tools that help in the process of developing distributed applications, especially in terms of debugging and testing these applications. Debugging and testing distributed systems is known to be a difficult problem [3], as it is inherently dependent on the non-deterministic factors of a distributed program. These factors include uncertainties that characterize the nature of the channels, such as unreliable message delivery, variability in the communication latency, and out-of-order delivery of messages. Other factors include the possibility of process failures, and execution time differences among the different machines due to the performance difference of each node. All of these factors imply that the events of a distributed computation cannot be totally ordered, and only a partial order dictated by the messages exchanged between the processes can be achieved. Because of this non-deterministic nature of distributed systems, bug reproducibility is not always possible in such systems. This is why the observation of the computation alone is not always enough to debug and test the computation, and one may need to control a distributed computation to be able to efficiently carry out these debugging and testing tasks. By controlling the computation, one is able to re-execute a faulty execution with the same event ordering, thereby reproducing the exact conditions that led to a bug.

Controlling a distributed computation can be done in several ways. The algorithm can work in either an online or an offline fashion. Moreover, the control can either be done by inserting delays between events, or by changing the order of events [1]. In [4], it was shown that the generic predicate control problem is NP-complete. However, there are predicate structures that can be controlled more efficiently, specifically, disjunctive predicates and mutex predicates [1]. In this paper, we restrict our attention to offline control of disjunctive predicates.

The problem can be stated as follows. Given the trace of a distributed computation, and a disjunctive predicate, we attempt to add synchronization to the computation so that the predicate is always maintained, without violating the liveness property.

Disjunctive predicates can be useful in many applications of distributed debugging, as they can state that at least one local condition must be satisfied at all times. An example of one such predicate is that “at least one process must be alive at any given time” [2]. This predicate can be represented as the disjunction of the local predicates “process i is alive”.

The rest of the paper is organized as follows. In Section II we review the background material and various notations needed, as well as the theorems on which the algorithm relies on. In Section III, we present the algorithm for disjunctive predicate control, and analyze its time and space complexity. The implementation of our tool as well as some experimental results are discussed in Section IV. We conclude the paper in section V.

II. BACKGROUND

In this section, we introduce the concepts behind our Java tool that solves the problem of offline controlled re-execution for disjunctive predicates. First, we give a brief overview of the notations and model of a distributed computation, and then we introduce the notions of admissible sequences and true event graph that are at the heart of algorithm that solves the offline controlled re-execution problem for disjunctive predicated. Finally, we go over that algorithm and analyze its complexity in terms of the number of processes n and the number of events m per process.

A. Model of a distributed computation

A distributed computation consists of a number of *n* processes $P = \{p_0, p_1, \dots p_{n-1}\}$ that each execute a certain set of *m* events or instructions. The different processes do not share a common physical clock. Rather, they send messages to each other over a set of channels to synchronize their tasks. In this paper, we assume that messages are never lost and are guaranteed to eventually arrive to their respective destination, that is message delivery is reliable. However, we do not assume that the channels on which processes send messages are FIFO, meaning that messages can arrive out of order.

We introduce the notations used throughout the paper and their corresponding denotations in Table 1.

Notation	**Denotation**
Lowercase letters *e* , *f*	Local events on a process
Greek letters α, ß	Sequence of events
e.proc	The process on which event *e* occurs
e.happened	Returns whether the event *e* has happened
e.pred	The first event locally preceding *e*
e.succ	The first event locally succeeding *e*
$E.\top$	Set of finals events on all processes $P_0 \dots P_{n-1}$
$E.\bot$	Set of initial events on all processes $P_0 \dots P_{n-1}$
$\prec$	Locally precedes
$\rightsquigarrow$	Remotely precedes

Table 1: Notations used and their corresponding denotations

The set of events happening on different proceses in a computation E for a distributed system forms a reflexive partial order. We use the notation $(E, \rightarrow)$ to denote a computation in a distributed system, where E is the set of events happening on all processes and → the partial order relation known as Lamport's happened-before relation. The happened-before relation can be defined formally as follows. We say that $e \rightarrow f$ if any of the two following conditions hold:

1. $\forall e, f \in E: e \rightarrow f \stackrel{\text{def}}{=} (e \prec f) \vee (e \rightsquigarrow f)$
2. $\exists\, g: (e \rightarrow g) \wedge (g \rightarrow f)$

A cut is a with respect to a computation is formally defined as follows.

$$cut(C, E) \stackrel{\text{def}}{=} \forall e \in E: e \in C \Rightarrow e.pred \in C$$

We also need the notion of frontier cut, which was introduced in [2]. A frontier cut C is the set of events in the cut C, whose successor are not in C. Formally, a frontier cut can be defined as follows where C is the frontier cut.

$$frontier(C, E) \stackrel{\text{def}}{=} \forall e \in E: e \in C \Rightarrow e.succ \notin C$$

Having formally defined a cut and a frontier cut, we can define a consistent cut. A cut is said to be consistent if for all events e belonging to the cut C, all events that happened-before the event e must also belong to the cut. Formally, a consistent cut is defined as follows.

$$consCut(C, E) \stackrel{\text{def}}{=} cut(C, E) \wedge (\forall e, f \in E : (f \in C) \wedge (e \rightarrow f) \Rightarrow (e \in C)$$

A consistent cut C is said to be legal with respect to an sequence α if for all events α_i belonging to the sequence α that are in the cut C, all events that happened before αi are also part of a the cut C. Formally, a legal cut C with respect the a sequence of events α can be defined as follows.

$$legal\ (C, E, \alpha) \stackrel{\text{def}}{=} consCut(C, E) \wedge (\forall \alpha_i, \alpha_j \in \alpha : i \leq j : \alpha_j \in C \Rightarrow \alpha_i \in C$$

B. Admissible Sequence

We know introduce the notion of admissible sequence that is at the heart of the solving the controlled re-execution problem for disjunctive predicates. The following theorem was proved by Garg and Mittal in [2].

THEOREM 1: *A predicate is controllable in a computation* $(E, \rightarrow)$ *iff there exists an admissible sequence of events with respect to the predicate B and the computation* $(E, \rightarrow)$

As seen from the above theorem, finding an admissible sequence of events with respect to the predicate and the computation $(E, \rightarrow)$ is the key to solving the controlled re-execution problem for disjunctive predicates.

We now define and elaborate upon the notion of an admissible sequence:

For a given computation $(E, \rightarrow)$ and a predicate B, an admissible sequence of events α captures some safe execution of events S, where $S \subseteq E$. An admissible sequence of events must obey to certain properties and conditions for any given execution of a computation. These properties and conditions are detailed below.

- Agreement: an admissible sequence α should be consistent with the happened-before relation →. Formally, $\forall\, i, j : i < j \;\Rightarrow\; \alpha_j \nrightarrow \alpha_i$
- Boundary Condition: an admissible sequence α must start with an initial state in the computation and end in a final state in the computation. Formally, the boundary condition is defined as follows. $(\alpha_0 \in E.\perp) \wedge (\alpha_{n-1} \in E.T)$ where α_0 is the first event in the sequence, α_{n-1} is the last event in the sequence and n is the number of events in the sequence α.
- Continuity: an admissible sequence α must be a continuous execution. We can define this property more formally as follows: $\alpha_i, \alpha_j \notin E.T : (i < j) \wedge (\alpha_i.succ \rightarrow \alpha_j) \;\Rightarrow\; \alpha_{i+1} = \alpha_i.succ$
- Safety: the admissible sequence α must be a safe execution. Meaning, that for any cut that is legal with respect to the sequence α and a predicate B, and where some event in α belongs to the frontier of C, the predicate in question is true. That is,

$$legal(C, E, \alpha) \wedge (\exists\, \alpha_i \in \alpha \wedge$$
$$\alpha_i \in \text{frontier}\,(\mathrm{C}, \mathrm{E}) \;\Rightarrow C(B) = True)$$

C. True Event Graph

Another notion that is at the heart of solving controlled re-execution problem for disjunctive predicate is the true event graph (TEG), which was also defined in [2]. Given the trace of a computation $(E, \rightarrow)$ and a disjunctive predicate B, a true event graph is constructed based on the following conditions and considerations.

- An event *e* belongs to the true event graph if the local predicate for the process *e.procc* is true at the event *e*.
- There exists an edge between two events *e* and *f* in the true event graph if $e.succ \nrightarrow f$
- An event *e* in the true event graph is labeled "initial" if $e \in E.\perp$
- An event e in the true event graph is labeled "final" if $e \in E.T$

We now present the following two theorems that are crucial to solving the controlled re-execution problem for disjunctive predicates. The two theorems presented are proven in [2]. It is clear from the two theorems presented that the key to solving the problem is to construct the true event graph *G* and to find the shortest permissible path in *G*.

THEOREM 2 *Let G = (v, ε) be the true event graph corresponding to a disjunctive predicate B and a computation (E, →). The shortest permissible path in G, if it exists, corresponds to an admissible sequence of events.*

THEOREM 3: *Let G = (v, ε) be the true event graph corresponding to a disjunctive predicate B and a computation (E, →). If B is controllable in (E, →) then there exists a permissible path in G.*

Finally, in [2], the specific synchronization messages that should be added to the computation in order to control it is shown to be $A \cup B$, where A and B are the sets defined by:

$$A = \{(\alpha_i, \alpha_j) | 0 \le i < j < |\alpha|\}$$

$$B = \{(\alpha_{i+1}, \alpha_i.next) | 0 \le i < j < |\alpha| - 1, \alpha_i \notin E.T \; and \; \alpha_i.process \neq \alpha_{i+1}.\text{process}\}$$

III. ALGORITHM

We present an algorithm based on the theorems and results in [2], which were presented in the previous section. We first need to compute the admissible sequence, before adding the necessary synchronization messages. We do this by finding the shortest path in the graph from any initial state to any final state. Thus, we are computing the shortest admissible sequence, in order to minimize the number of additional synchronization messages that should be added to the trace. The algorithm's pseudocode is given below:

```
findAdmissibleSequence(trace):
  graph = constructTEG(trace)
  admissible_seq = NULL
  min_length = INFINITY
  for e in graph.initial_nodes
    for f in graph.final_nodes
      path = graph.shortest_path(e, f)
      if (length(path) < min_length)
        admissible_seq = path;
        min_length = length(admissible_seq)
      end if
    end for
  end for
  return admissible_seq
```

Figure 1: Pseudo-code for the algorithm to find an admissible sequence

Dijkstra's algorithm complexity is $O(|E| + |V|\ log\ |V|)$, where E is the set of edges, and V is the set of vertices in the graph. Given a computation with N processes, and up to m external events per process, the true event graph can contain up to Nm nodes, and in the order of $(Nm)^2$ edges. The sets of initial nodes and final nodes both have cardinality N, so the shortest path computation is executed N^2 times. This leaves the algorithm with a complexity of $O(N^2(Nm)^2) = O(N^4m^2)$.

A better alternative, which we did implement, is to compute the shortest paths between all pairs of nodes in the graph in one go, before finding the shortest path between an initial and a final node. We use the Floyd-Warshall all-pairs shortest path algorithm for this purpose, which is relies on dynamic programming to compute the shortest path for any pair of nodes in the graph. The complexity of the Floyd-Warshall algorithm is $O(|V|^3)$, leaving our algorithm with a complexity of $O(N^3m^3)$

Note that even though we need to save the shortest paths between all pairs of vertices in the case of the Floy-Warshall approach, the space complexity is still in the same order as the approach that uses Dijkstra's algorithm, as there is an efficient matrix representation from which the paths can be easily retrieved. This matrix representation has the same dimension as the adjacency matrix used for representing the true event graph.

IV. FRAMEWORK

We implement a tool in java that takes in the trace of a computation, and attempts to control the predicate of interest by adding additional synchronization messages to the trace.

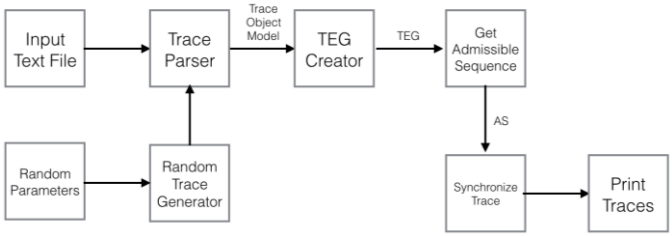

Figure 2: High-level block diagram of the framework

The image above shows a high-level block diagram of our tool. We represent a computation in a trace file that contains information about the events on each process, as well as the predicate value for each event, and the messages exchanged. Our Trace Parser module builds an object model of the trace for easier manipulation in subsequent modules. The TEG Creator constructs the true event graph corresponding to that computation. This will be fed into the main algorithm block, which computes all pairs shortest path in the graph between initial and final nodes, and then outputs the shortest admissible sequence. We then feed this admissible sequence into the Trace Synchronizer module, which will compute all the synchronization messages that should be added to the trace, as described in a subsequent paragraph. Our results are fed into a GUI block which draws the diagram corresponding to the computation, together with the additional synchronization messages. We now briefly describe the tasks of each of our modules.

Trace parser: This module creates an in-memory object model representation of the trace.

TEG Creator: The true event graph corresponding to the computation will be created according to the definition in section II.C.

Admissible Sequence Computation: This module will take the TEG graph as input, with the labeled initial and final nodes, and compute the shortest admissible sequence according to the algorithm presented in section III.

Trace Synchronizer: The trace synchronizer will compute all synchronization messages that should be added, according to the description in section III. The pseudo-code for this module is given below.

```
addSyncrhonizationMessages(admissibleS_seq):
  // add messages from set A in section II
  for i in [0, |α|)
    for j in (i, |α|)
      if (αi.process ≠ αj.process)
        addSyncMessage(αi, αj)
      end if
    end for
  end for

  // add messages from set B in section II
  for i in [0, |α|-1)
    if (αi.process ≠ αi+1.process)
      addSyncMessage(αi+1,αi.next)
    end if
  end for
```

Figure 3: Pseudo-code for finding the synchronization messages

GUI: Our results are sent to the graphical module, which will generate the trace diagram corresponding to the computation. The following diagram, outputted from our tool, shows how to control the given computation.

Events colored in green represents events on which the value of the predicate was true right before executing the event. The red arrows represent the additional synchronization messages needed to control the predicate.

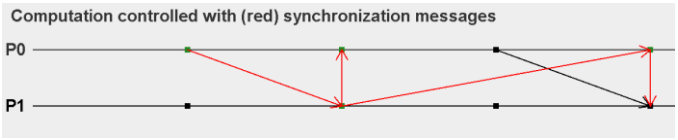

Figure 4: Example of synchronizing a computation

Below, we summarize some performance results of our algorithm. The algorithm was run on a computer with Intel Core i3 with 2.1 GHz and 8GB of memory running Mac OS 10. We simulated traces with varying number of processes, N, total number of events E, and total number of messages M.

N	E	M	t (sec)
3	50	10	0.10
5	100	20	2.10
10	150	30	10.24
15	200	40	23.12

Table 2: execution time of the offline algorithm

V. Conclusion

In this paper, we presented an implementation tool of the offline predicate control algorithm for disjunctive predicates, and evaluated this algorithm analytically and experimentally. Our tool is based on the algorithm presented by Garg and Tarafdar for disjunctive predicate control. We used the notion of a true event graph and admissible sequence to generate the synchronization messages that should be added to the trace. We use the Floyd-Warshall all-pair shortest path algorithm to efficiently find the shortest path between the nodes of interest in the graph, and achieve a worst case time complexity of $O(N^3m^3)$, N being the number of processes and m being the maximum number of external events per process.

Future work includes improving on the time complexity of the algorithm. Since we only need to compute the shortest path between initial and final nodes in the graph, we could investigate other graph algorithms that do not compute the shortest path between all pairs of nodes. Another direction for future work is integrating our tool in an end-to-end distributed debugging framework, and implementing the predicate control algorithm for mutex predicates as well.

References

[1] Garg, Vijay K. Elements of distributed computing. John Wiley & Sons, 2002.

[2] Mittal, Neeraj, and Vijay K. Garg. "Debugging distributed programs using controlled re-execution." Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. ACM, 2000.

[3] Tarafdar, Ashis, and Vijay K. Garg. "Predicate control for active debugging of distributed programs." Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998. IEEE, 1998.

[4] Tarafdar, Ashis, and Vijay K. Garg. "Software fault tolerance of concurrent programs using controlled re-execution." Distributed Computing. Springer Berlin Heidelberg, 1999. 212-225.