Detecting Causality in the Presence of Byzantine Processes: The Synchronous Systems Case

Anshuman Misra, Ajay Kshemkalyani

University of Illinois at Chicago

Contact email: amisra7@uic.edu, ajay@uic.edu

TIME 2023

Roadmap

1. What is causality and why it is important
2. Happens Before Relation
3. Problem Formulation (CD)
4. Replicated State Machine (RSM) Approach
5. RSM based causality testing algorithm

Introduction

- Causality is an important tool in understanding and reasoning about distributed systems
- Determining causality is the process of ordering events in a given execution trace
- Events that are not causally related are concurrent
- Applications of causality detection include deadlock detection, detecting race conditions, distributed debugging and monitoring

Introduction

- Sequential programs consist of totally ordered events
- Distributed programs consist of events that are not totally ordered
- The idea is to partially order events during execution
- Theoretically, the happens before relation defines causality
- In practice, logical clocks timestamp events which are used to determine causality

Introduction

In practice, the following mechanisms are used to track causality:

1. Causality Graphs
2. Scalar Clocks
3. Vector Clocks
4. Interval Tree Clocks
5. Bloom Clocks
6. Encoded Vector Clocks
7. Plausible Clocks
8. Incremental Clocks
9. Version Vectors

However, none of these mechanisms consider Byzantine failures.

Introduction

- Recently it has been proved that it is impossible to detect causality in the presence of Byzantine failures in an asynchronous system [1]
- In light of this result, this paper investigates the solvability of detecting causality in a synchronous system with Byzantine failures

[1] Misra, Anshuman, and Ajay D. Kshemkalyani. "Detecting Causality in the Presence of Byzantine Processes: There is No Holy Grail." In 2022 IEEE 21st International Symposium on Network Computing and Applications (NCA), vol. 21, pp. 73-80. IEEE, 2022.

Contributions

- This paper examines the solvability of causality detection in synchronous systems under Byzantine failures
- We establish a fundamental possibility result about causality detection in the presence of Byzantine processes
- We provide an algorithm that uses vector clock and replicated state machines to solve the causality detection problem
- We summarize the solvability of the family causality detection problems under a variety of settings in Table 2

Results

Mode of communication	Detecting "happens before" in asynchronous systems	Detecting "happens before" in synchronous systems
Multicasts	Impossible [2] FP, FN	Possible, Theorem 3 $\overline{FP}, \overline{FN}$
Unicasts	Impossible [2] FP, FN	Possible, Corollary 4 $\overline{FP}, \overline{FN}$
Broadcasts	Impossible [2] $\overline{FP}, FN$	Possible, Corollary 5 $\overline{FP}, \overline{FN}$

Table 1: Detecting causality between events under different communication modes in asynchronous and synchronous systems. FP is false positive, FN is false negative. $\overline{FP}/\overline{FN}$ means no false positive/no false negative is possible.

[2] Misra, Anshuman, and Ajay D. Kshemkalyani. "Detecting Causality in the Presence of Byzantine Processes: There is No Holy Grail." In 2022 IEEE 21st International Symposium on Network Computing and Applications (NCA), vol. 21, pp. 73-80. IEEE, 2022.

System Model

1. The distributed system is modelled as an undirected graph $G = (P, C)$. Here P is the set of processes communicating in the distributed system.
2. The channels are assumed to be FIFO and G is a complete graph.
3. The distributed system is assumed to be synchronous, i.e., there is a known fixed upper bound δ on the message latency, and a known fixed upper bound ψ on the relative speeds of processors.
4. The system assumes the presence of Byzantine processes. A correct process behaves exactly as specified by the algorithm whereas a Byzantine process may exhibit arbitrary behaviour including crashing at any point during the execution.

Definitions (1)

1. e_i^x, where $x \geq 1$, denotes the x-th event executed by process p_i
2. The sequence of events $\langle e_i^1, e_i^2, \ldots \rangle$ is called the execution history at p_i and denoted E_i
3. $E = \bigcup_i \{E_i\}$ and $T(E)$ denotes the set of all events in (the set of sequences) E
4. The *happens before* relation, denoted $\rightarrow$, is an irreflexive, asymmetric, and transitive partial order defined over $T(E)$

Definitions (2)

Definition 1

The happens before relation $\rightarrow$ on events $T(E)$ consists of the following rules:

1. **Program Order**: For the sequence of events $\langle e_i^1, e_i^2, \ldots \rangle$ executed by process p_i, $\forall\ x, y$ such that $x < y$ we have $e_i^x \rightarrow e_i^y$.
2. **Message Order**: If event e_i^x is a message send event executed at process p_i and e_j^y is the corresponding message receive event at process p_j, then $e_i^x \rightarrow e_j^y$.
3. **Transitive Order**: If $e \rightarrow e' \wedge e' \rightarrow e''$ then $e \rightarrow e''$.

Problem Formulation (1)

1. An algorithm to solve the causality detection problem collects the execution history of each process in the system
2. E_i is the actual execution history at p_i and F_i is the execution history at p_i as perceived and collected by the algorithm
3. Analogous to $T(E)$, $T(F)$ denotes the set of all events in F, therefore Definition 1 applies to $T(F)$ as well

Problem Formulation (2)

Definition 2

The causality detection problem $CD(E, F, e_i^*)$ for any event $e_i^* \in T(E)$ at a correct process p_i is to devise an algorithm to collect the execution history E as F at p_i such that $valid(F) = 1$, where

$$valid(F) = \begin{cases} 1 & \text{if } \forall e_h^x, e_h^x \to e_i^*|_E = e_h^x \to e_i^*|_F \\ 0 & \text{otherwise} \end{cases}$$

Problem Formulation (3)

When 1 is returned, the algorithm output matches the actual (God's) truth and solves CD correctly. Thus, returning 1 indicates that the problem has been solved correctly by the algorithm using F. 0 is returned if either

- $\exists e_h^x$ such that $e_h^x \rightarrow e_i^*|_E = 1 \wedge e_h^x \rightarrow e_i^*|_F = 0$ (denoting a false negative), or
- $\exists e_h^x$ such that $e_h^x \rightarrow e_i^*|_E = 0 \wedge e_h^x \rightarrow e_i^*|_F = 1$ (denoting a false positive).

Replicated State Machine

1. A replicated state machine (RSM) is a distributed service that ensures that every process in the system arrives at the same state after processing the same sequence of inputs
2. Under Byzantine failures, each process is modelled as an ensemble of $(2t + 1)$ replicas (of which at most t are Byzantine)
3. To ensure that all replicas actions and transitions are coordinated the following requirements must hold:
    1. Agreement
    2. Total Order

RSM based Algorithm (1)

1. Each process is modelled as a $(3t+1)$ replicated state machine, where at most t replicas can be Byzantine
2. In a system with n application processes there are $(3t+1)n$ replicas partitioned into n RSM ensembles
3. The Algorithm ensures that E matches F thereby preventing any false positives and false negatives

RSM based Algorithm (2)

1. When an ensemble receives an application message m, every correct replica processes m under the constraints of agreement and total order
2. Further, when an ensemble p sends a message m to ensemble j, correct replicas only consider m to be valid if $(t + 1)$ replicas from p_i have sent m
3. This essentially filters out any Byzantine behaviour in the system and ensures that only causality tracking metadata from correct sources are recorded at each application process

RSM based Algorithm (3)

Each RSM replica $p_{i,a}$ maintains the following data structures.

1. An integer $seq_{i,a}$, initialized to 0, that gives the sequence number of the latest local event at $p_{i,a}$.
2. A local F that is a set of sequences F_k. F contains $p_{i,a}$'s view of the recorded execution history F_k of each RSM p_k.
3. An integer matrix $LASKALSJ[n, n]$, where $LASKALSJ[j, k]$ gives the sequence number of the latest send event by p_k (as per/from the local F_k) at the point in time of the last send event to $p_{j,*}$. This data structure is for efficiently identifying to send to p_j only the *incremental updates* that have occurred to the local F_k at $p_{i,a}$ for each other process p_k, that need to be transmitted to the destinations p_j of a message send event since $p_{i,a}$'s last send to p_j.
4. $p_{i,a}$ also maintains an auxiliary integer matrix $V[|T(F_i)|, n]$, where $V[s, k]$ is $maxeventID(F_k)$ in $F(e^s_{i,a})$, i.e., the highest sequence number in $F_k (\in F)$ when the sth local event $e^s_{i,a}$ was executed at $p_{i,a}$.

Algorithm - Description

Data: Each process $p_{i,a}$ maintains (i) an integer $seq_{i,a}$, (ii) F which is the union of sequences F_k (history of events at p_k) for all k, (iii) integer matrix $LASKALSJ[n,n]$, (iv) integer matrix $V[|T(F_i)|, n]$.

Input: e_h^x, e_i^*

Output: $e_h^x \rightarrow e_i^*|_F \in \{true, false\}$

Algorithm - Unicast

when $p_{i,a}$ needs to send application message M to $p_{j,*}$: ▷ `Each other correct` $p_{i,a'}$ `state machine will execute likewise`

$seq_{i,a} = seq_{i,a} + 1$

append current send event to F_i; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

$(\forall k)$ include history from F_k after event $LASKALSJ[j,k]$ in inc_F

$(\forall k)$ $LASKALSJ[j,k] = maxeventID(F_k)$

send $(M, inc\_F, seq_{i,a}, j)$ to each $p_{j,*}$ via RSM layer (to satisfy RSM Total Order and Agreement for receiver ensemble p_j)

Algorithm - Multicast

when $p_{i,a}$ needs to send application message M to each $p_{j,*}$ for each $p_j \in G$: ▷ `Each other correct` $p_{i,a'}$ `state machine will execute likewise`

$seq_{i,a} = seq_{i,a} + 1$

append current send event to F_i; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

$(\forall k)$ include history from F_k after event $\min_{p_j \in G}(LASKALSJ[j,k])$ in inc_F

$(\forall p_j \in G)(\forall k)$ $LASKALSJ[j,k] = maxeventID(F_k)$

send $(M, inc\_F, seq_{i,a}, G)$ to each $p_{j,*}$ for each $p_j \in G$ via RSM layer (to satisfy RSM Total Order and Agreement$-M$ for each receiver ensemble p_j)

Algorithm - Internal Event

<u>**At internal event** at $p_{i,a}$:</u>

$seq_{i,a} = seq_{i,a} + 1$

append current internal event to F_i; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

Algorithm - Message Delivery

when $(M, inc\_F, seq_j, i/G)$ is *SR-delivered* to $p_{i,a}$ from p_j: ▷ `Happens when` $t+1$ `identical copies of` $(M, inc\_F, seq_j, i/G)$ `for` seq_j `(which equals` $seq_{j,*}$`) are` ***`TOA-delivered`*** `from` $p_{j,*}$

for *all* k **do**

  if $maxeventID(F_k) < maxeventID(inc\_F_k)$ **then**

    append history of events $\langle maxeventID(F_k)+1, \ldots, maxeventID(inc\_F_k)\rangle$ from inc_F_k to F_k

$seq_{i,a} = seq_{i,a} + 1$

append current receive event to F_i; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

Algorithm - Causality Testing

To determine $e_h^x \rightarrow e_i^*$ at correct state machine $p_{i,a}$ via call to **test**$(e_h^x \rightarrow e_i^*)$:

if e_h^x *is in* F_h *and* $* \leq maxeventID(F_i)$ **then**

  return$(e_h^x \rightarrow e_i^*|_F)$ ▷ `the test is whether` $V[*, h] \geq x$

else

  return$(false)$

Algorithm - Correctness (1)

Theorem 3

There are neither false negatives nor false positives in solving causality detection as per the RSM based causality testing Algorithm for the multicast mode of communication in synchronous systems.

Algorithm - Correctness (2)

Corollary 4

There are neither false negatives nor false positives in solving causality detection as per the RSM based causality testing Algorithm for the unicast mode of communication in synchronous systems.

Corollary 5

There are neither false negatives nor false positives in solving causality detection as per the RSM based causality testing Algorithm for the broadcast mode of communication in synchronous systems.

Results

Mode of communication	Detecting "happens before" in asynchronous systems	Detecting "happens before" in synchronous systems
Multicasts	Impossible [2] FP, FN	Possible, Theorem 3 $\overline{FP}, \overline{FN}$
Unicasts	Impossible [2] FP, FN	Possible, Corollary 4 $\overline{FP}, \overline{FN}$
Broadcasts	Impossible [2] $\overline{FP}, FN$	Possible, Corollary 5 $\overline{FP}, \overline{FN}$

Table 2: Detecting causality between events under different communication modes in asynchronous and synchronous systems. FP is false positive, FN is false negative. $\overline{FP}/\overline{FN}$ means no false positive/no false negative is possible.

[2] Misra, Anshuman, and Ajay D. Kshemkalyani. "Detecting Causality in the Presence of Byzantine Processes: There is No Holy Grail." In 2022 IEEE 21st International Symposium on Network Computing and Applications (NCA), vol. 21, pp. 73-80. IEEE, 2022.

Conclusion

1. The causality detection problem CD, is solvable under Byzantine failures
2. Having a system of $(3t + 1)n$ processes with at most t Byzantine processes partitioned by the RSM approach neutralizes Byzantine behaviour
3. However, the RSM approach is only applicable to synchronous systems
4. Future work is to investigate whether a more direct approach can be employed to solve CD