Efficient Decentralized Monitoring of Safety in Distributed Systems

Authors: Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu

Animesh Jain
Poorvasha Dhananjay Chauhan

IIT Dharwad

April 27, 2025

Introduction

Goal: Ensure safety in distributed systems through monitoring
Challenges:

1. No global clock or global state
2. Asynchronous message passing
3. Need for local, efficient monitoring

Approach: Use Past-Time Distributed Temporal Logic (PT-DTL)

Goal and Key Concepts

We describe an efficient decentralized monitoring algorithm that monitors the execution of a distributed program to check for violations of the safety properties.
For this we will be using following concepts:

1. **PT-DTL**, a variant of past-time linear temporal logic
2. **KNOWLEDGE VECTOR**(vector clock)
3. Describing the implementation of the algorithm in a tool called **DIANA**

Motivation

1. Challenge with LTL: Monitoring with LTL requires collecting consistent global snapshots and exploring all possible interleavings of events, which is computationally expensive and impractical in large-scale distributed systems—even with techniques like partial order reduction.
2. PT-DTL Advantage: PT-DTL overcomes this by enabling each node to locally monitor properties using only its causal knowledge, eliminating the need for global snapshots or exhaustive interleaving checks.

Distributed Systems Model

Consider a distributed system composed of n processes:
$p_1, p_2, ..., p_n$
Each process p_i has:

1. Its own local state s_i, which evolves as events occur
2. The ability to communicate asynchronously with other processes by sending and receiving messages

The computation of each process is abstracted out in terms of events. There can be three types of events:

1. Internal events – actions that affect only the local state of a process
2. Send events – when a process sends a message to another
3. Receive events – when a process receives a message sent by another

Since there's no global clock i.e. processes do not share a common notion of time, and events are only partially ordered

Events and Partial Orders

Let's denote: E = Union of all event sets from each process:
$E = \bigcup_{i=1}^{n} E_i$ where E_i is the set of events on process p_i.
Also let $\lessdot \subseteq E \times E$ be defined as follows:

1. $e \lessdot e$' if e and e' are events of the same process and e happens immediately before e'.
2. $e \lessdot e$' if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

Partial Order Cont.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. This partial order captures the causality relation among the events in different processes. The structure described by $\mathcal{C} = (E, \prec)$ is called a distributed computation. In what follows, we assume an arbitrary but fixed distributed computation $\mathcal{C}$. Let us define $\preceq$ as the reflexive and transitive closure of $\lessdot$.

In Fig. 2, $e_{11} \prec e_{23}$, $e_{12} \prec e_{23}$, and $e_{11} \lessdot e_{23}$. However, $e_{12} \not\lessdot e_{23}$.

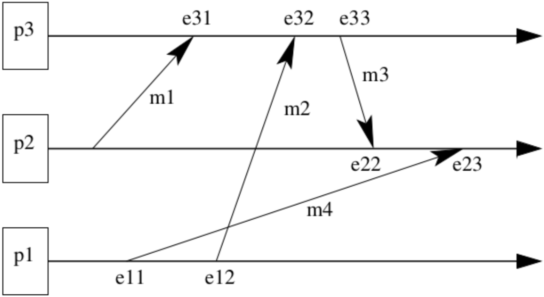

Figure: Sample Distributed Computation

Local State

The local state of a process is abstracted out in terms of a set of events. For $e \in E$, we define

$$\downarrow e = \{e` \mid e` \preceq e\},$$

that is, $\downarrow e$ is the set of events that causally precede e. For $e \in E_i$, we can think of $\downarrow e$ as the local state of p_i when the event e has just occurred.
This state contains the history of events of all processes that causally precede e.

Extended Patial Order

We extend the definitions of $\lessdot$, $\prec$, and $\preceq$ to local states such that:

$$\downarrow e \lessdot \downarrow e` \iff e \lessdot e`,$$
$$\downarrow e \prec \downarrow e` \iff e \prec e`,$$
$$\downarrow e \preceq \downarrow e` \iff e \preceq e`.$$

We denote the set of local states of a process p_i by

$$LS_i = \{\downarrow e \mid e \in E_i\}$$

and let $LS = \bigcup_i LS_i$. We use the symbols $s_i, s`_i, s``_i$, and so on to represent the local states of process p_i. We also assume that each local state s_i of each process p_i associates values to some local variables V_i, and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process p_i.

Causal

We use the notation $\text{causal}_j(s_i)$ to refer to the latest state of process p_j of which process p_i knows while in state s_i. Formally, if $\text{causal}_j(s_i) = s_j$, then $s_j \in LS_j$ and $s_j \preceq s_i$, and for all $s_j' \in LS_j$, if $s_j' \preceq s_i$ then $s_j' \preceq s_j$.
For example, in Fig 2, $\text{causal}_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$, then $\text{causal}_j(s_i) = s_i$.

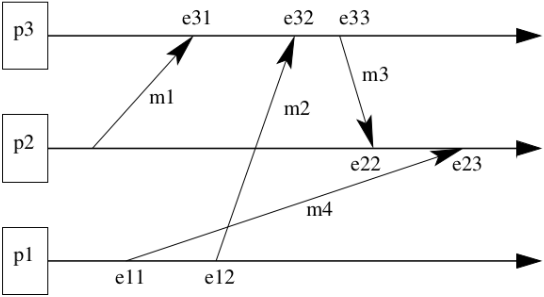

Figure: Sample Distributed Computation

Past-time Linear Temporal Logic (PT-LTL)

Past-time Linear Temporal Logic (PT-LTL) has been successfully used in the past to express, monitor, and predict violations of safety properties in software systems.

$$
\begin{aligned}
F ::= &\ true | a \in A | \neg F | F op F && \textit{propositional} \\
&| \bigcirc^{-1} F | \Diamond^{-1} F | \Box^{-1} F | F \mathcal{S} F && \textit{temporal}
\end{aligned}
$$

Where *op* are the standard binary operators, namely $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and

$\bigcirc^{-1} F$ should be read as "previously",
$\Diamond^{-1} F$ as "eventually in the past",
$\Box^{-1} F$ as "always in the past", and
$F_1 \mathcal{S} F_2$ as "F1 since F2".

PT-LTL Cont.

The logic is interpreted on a finite sequence of states or a run. If $\rho = s_1 s_2 \ldots s_n$ is a run, then we let ρ_i denote the prefix run $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Figure 3.
For ex, the formula $\Box^{-1}((\text{action} \wedge \bigcirc^{-1} \neg \text{action}) \rightarrow (\neg \text{stop} \, \mathcal{S} \, \text{start}))$ states that "whenever action starts to be true, it is the case that start was true at some moment in the past and since then stop was never true", or in other words that the action is taken only when the system is active.

PT-LTL Cont.

$\rho \models \text{true}$	for all ρ,
$\rho \not\models \text{false}$	for all ρ,
$\rho \models a$	iff a holds in the state s_n,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \ op\ F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \diamondsuit F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \boxdot F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \ \mathcal{S}\ F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $j < i \leq n$,

Figure: Semantics of PT-LTL

PT-DTL - Overview

PT-DTL allows local monitoring with knowledge of remote states
Extension of PT-LTL with epistemic operators (@j)
Can refer to remote variables and formulae
Enables processes to monitor global properties locally

PT-DTL Syntax

i-formulae (F_i): Local to process pi
i-expressions (ξ_i): Include local/remote expressions
Epistemic operators:

1. $@_j F_j$: Formula about pj known to pi
2. $@_j \xi_j$: Value of expression ξ_j on process p_j as known to process p_i

PT-DTL Semantics

PT-DTL semantics are defined over causal states rather than linear traces

For a configuration $\mathcal{C}$ and local state s_i of process p_i:

$(\mathcal{C}, s_i)[[@_j \xi_j]] = \xi_j$ at $(casual_j(s_i))$

ξ_j:an expression from process p_j

$@_j \xi_j$: the value of ξ_j at process p_j, as known to process p_i

$casual_j(s_i)$: the most recent state of p_j known to p_i at its local state s_i

Why it matters:

1. Enables process p_i to make decisions based on what it knows about other processes
2. Avoids global synchronization and linearization

This captures remote state awareness efficiently and forms the basis for local monitoring of global safety properties.

PT-DTL Semantics

$$
\begin{array}{lll}
\mathcal{C}, s_i \models \text{true} & \text{for all } s_i \\
\mathcal{C}, s_i \not\models \text{false} & \text{for all } s_i \\
\mathcal{C}, s_i \models P(\xi_i, \ldots, \xi_i') & \text{iff } P((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!]) = \text{true} \\
\mathcal{C}, s_i \models \neg F_i & \text{iff } \mathcal{C}, s_i \not\models F_i \\
\mathcal{C}, s_i \models F_i \; op \; F_i' & \text{iff } \mathcal{C}, s_i \models F_i \text{ op } \mathcal{C}, s_i \models F_i' \\
\mathcal{C}, s_i \models \odot F_i & \text{iff if } \exists s_i' \,.\, s_i' \lessdot s_i \text{ then } \mathcal{C}, s_i' \models F_i \text{ else } \mathcal{C}, s_i \models F_i \\
\mathcal{C}, s_i \models \Diamond F_i & \text{iff } \exists s_i' \,.\, s_i' \preccurlyeq s_i \text{ and } \mathcal{C}, s_i' \models F_i \\
\mathcal{C}, s_i \models \boxdot F_i & \text{iff } \mathcal{C}, s_i \models F_i \text{ for all } s_i' \preccurlyeq s_i \\
\mathcal{C}, s_i \models F_i \; \mathcal{S} \; F_i' & \text{if } \exists s_i' \,.\, s_i' \preccurlyeq s_i \text{ and } \mathcal{C}, s_i' \models F_i' \text{ and } \forall s_i'' \,.\, s_i' \prec s_i'' \preccurlyeq s_i \text{ implies } \mathcal{C}, s_i'' \models F_i \\
\mathcal{C}, s_i \models @_j F_j & \text{iff } \mathcal{C}, s_j \models F_j \text{ where } s_j = causal(s_i)
\end{array}
$$

$$
\begin{array}{ll}
(\mathcal{C}, s_i)[\![v_i]\!] & = s_i(v_i), \text{ that is, the value of } v_i \text{ in } s_i \\
(\mathcal{C}, s_i)[\![c_i]\!] & = c_i \\
(\mathcal{C}, s_i)[\![f(\xi_i, \ldots, \xi_i')]\!] & = f((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!]) \\
(\mathcal{C}, s_i)[\![@_j \xi_j]\!] & = (\mathcal{C}, s_j)[\![\xi_j]\!] \text{ where } s_j = causal_j(s_i)
\end{array}
$$

Figure: Semantics of PT-DTL

$$
\begin{array}{lll}
\mathcal{C}, s_i \models \Diamond F_i & = & \mathcal{C}, s_i \models F_i \text{ or } (\exists s_i' \,.\, s_i' \lessdot s_i \text{ and } \mathcal{C}, s_i' \models \Diamond F_i) \\
\mathcal{C}, s_i \models \boxdot F_i & = & \mathcal{C}, s_i \models F_i \text{ and } (\exists s_i' \,.\, s_i' \lessdot s_i \text{ implies } \mathcal{C}, s_i' \models \boxdot F_i) \\
\mathcal{C}, s_i \models F_i \mathcal{S} F_i' & = & \mathcal{C}, s_i \models F_i' \text{ or} \\
 & & (\mathcal{C}, s_i \models F_i \text{ and } \exists s_i' \,.\, s_i' \lessdot s_i \text{ and } \mathcal{C}, s_i' \models F_i \mathcal{S} F_i')
\end{array}
$$

Figure: Recursive Semantics of PT-DTL

PT-DTL Examples

1. Leader Election: "if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader"
   Property: Only one leader
   Formula:
   $leaderElected \rightarrow (state = leader \rightarrow \bigwedge_{j \neq i}(@_j(state \neq leader))$
2. Voting: "if the resolution is accepted then more than half of the processes say yes"
   Property: Majority votes yes
   Formula:
   $accepted \rightarrow (@_1(vote) + \ldots + @_n(vote)) > n/2$

PT-DTL Examples Continue

3. Server Reboot: "server accepts to reboot only after knowing that each client is inactive and aware of the warning to reboot"
   Property: Reboot only if all clients are inactive and know about warning
   Formula:
   $rebootAccepted \rightarrow$
   $\bigwedge_{client}(@_{client}(inactive \wedge @_{server} rebootWarning))$

PT-DTL Examples

1. Request and Reply:
   Property: If a has received a value then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past

$$\text{receivedValue} \rightarrow @_b(\Diamond^{-1}(\text{computedValue} \wedge @_a(\Diamond^{-1}\text{requestedValue})))$$

2. Temperature Alarm:
   Property: If my alarm has been set then it must be the case that the difference between my temperature and the temperature at process b exceeded the allowed value

$$\text{alarm} \rightarrow \Diamond^{-1}((\text{myTemp} - @_b\text{otherTemp}) > \text{allowed})$$

3. Airplane Landing:
   Property: If my airplane is landing, then the runway assigned by the airport matches the one that I plan to use

$$\text{landing} \rightarrow (\text{runway} = (@_{airport}\text{allocRunway}))$$

Monitoring Algorithm for PT-DTL

In this section, we describe an automated technique to synthesize efficient distributed monitors for safety properties in distributed systems expressed in PT-DTL

The synthesized monitor is distributed, in the sense that it consists of separate, local monitors running on each process

A local monitor can attach additional information to any outgoing message from the corresponding process. This information can subsequently be extracted by the monitor on the receiving side without changing the underlying semantics of the distributed program

Monitoring Algorithm for PT-DTL

The key guiding principles in the design of this technique are:

1. The local monitors should be fast, so that monitoring can be done online
2. The local monitors should have little memory overhead, in particular, it should not need to store the entire history of events on a process since this can be quite large
3. Additional messages needed to be sent between processes solely for the purpose of monitoring should be minimized and ideally, should be zero. Further, additional information piggybacked on regular messages(those generated as part of the distributed computation) should be small

Knowledge Vectors

The key guiding principles in the design of this technique are:

1. The local monitors should be fast, so that monitoring can be done online
2. The local monitors should have little memory overhead, in particular, it should not need to store the entire history of events on a process since this can be quite large
3. Additional messages needed to be sent between processes solely for the purpose of monitoring should be minimized and ideally, should be zero. Further, additional information piggybacked on regular messages(those generated as part of the distributed computation) should be small

Knowledge Vectors, Motivation

Consider evaluating a remote j-expression $@_j\ \xi_j$ at process p_i .

The naive solution is that process p_j simply piggybacks the value of ξ_j evaluated at p_j , with every message that it sends out.

The recipient process p_i can extract this value and use it as the value of $@_j \xi_j$

So the problem with this approach is that:

1. It add more overhead to each messages sent and
2. More importantly that the messages from p_j could reach p_i in arbitrary order

Knowledge Vectors

To keep track of the causal history, or in other words the most recent knowledge, we also need the event number at p_j at which these expressions were sent out in messages so that stale information in a reordered message sequence can be discarded.

Causal ordering can be effectively accomplished by using an array called KNOWLEDGE VECTOR with an entry for any process p_j for which there is an occurrence of $@_j$ in any PT-DTL formula at any process.

Knowledge vectors are motivated and inspired by vector clocks

Knowledge Vectors

The size of KNOWLEDGE VECTOR is not dependent on the number of processes but on the number of remote expressions and formulae.

Let $KV[j]$ denote the entry for process p_j on a vector KV. $KV[j]$ contains the following fields:

1. The sequence number of the last event seen at p_j , denoted by $KV[j]$.seq;
2. A set of values $KV[j]$.values storing the values j-expressions and j-formulae.

Each process pi keeps a local KNOWLEDGE VECTOR denoted by KV_i. The monitor of process pi attaches a copy of KV_i with every outgoing message m. We denote the copy by KV_m

Update Algorithm

The algorithm for the update of KNOWLEDGE VECTOR KVi at process p_i is given below:

1. **[internal]:** Update $KV_i[i]$. For this, we evaluate $\text{eval}(\xi_i, s_i)$ and $\text{eval}(F_i, s_i)$ for each $@_i\xi_i$ and $@_iF_i$, respectively, and store them in the set $KV_i[i]$.values;
2. **[send m]:** $KV_i[i].\text{seq} \leftarrow KV_i[i].\text{seq} + 1$. Send KV_i with m as KV_m;
3. **[receive m]:** For all j, if $KVm[j].\text{seq} > KVi[j].\text{seq}$ then $KVi[j] \leftarrow KVm[j]$, that is, $KVi[j].\text{seq} \leftarrow KV_m[j].\text{seq}$, and $KV_i[j].\text{values} \leftarrow KV_m[j].\text{values}$.

Proposition

Proposition 1: For any process p_i and any j, the entry for ξ_j or F_j in $KV_i[j]$.values contains the value of $@_j\xi_j$ or $@_jF_j$, respectively.

$KV_i[j]$.value contains the latest values that p_i has for j-expressions or j-formulae. Therefore, for the value of a remote expression or formula of the form $@_j\xi_j$ or $@jF_j$, process p_i can just use the entry corresponding to ξ_j or F_j in the set $KV_i[j]$.values.

Notes

Note:

1. the sequence number needs to be incremented only when sending messages.
2. The algorithm above tries to minimize the local work when sending a message. However, notice that the values calculated at step 1 are needed only when an outgoing message is generated at step 2, so one could have just evaluated all the expressions ξ_i and F_i at step 2, right before the message is sent out. This would reduce the runtime overhead at step 1 but would increase it at step 2. Depending on the specific application under consideration, one may prefer one way or the other.

Notes

Note:

1. The initial values for all the variables in the distributed program can be found either by a static analysis of the program or by a distributed broadcast at the beginning of the computation. Thus, it is assumed that each process p_i has complete knowledge of the initial values of remote expressions for all processes. These values are then used to initialize the entries $KV_i[j]$.values in the KNOWLEDGE VECTOR of p_i for all j.

Algorithms

```
array now; array pre; int index;

boolean eval(Formula F_i, State s_i){
  if binary(op(F_i)) then{
    lval ← eval(left(F_i), s_i);
    rval ← eval(right(F_i), s_i);}
  else if unary(op(F_i)) then
    val ← eval(subformula(F_i), s_i);
  index ← 0;
  case(op(F_i)) of{
    true : return true;   false : return false;
    P(ξ⃗_i) : return P(eval(ξ_i, s_i), ..., eval(ξ'_i, s_i)));
    op :  return rval op lval;  ¬ :  return not val;
    S : now[index] ← (pre[index] and lval) or rval;
      return now[index++];
    ⊡ : now[index] ← pre[index] and val;
      return now[index++];
    ◊ : now[index] ← pre[index] or val;
      return now[index++];
    ⊙ : now[index] ← val; return pre[index++];
    @_j F_j : return value of F_j from KV_i[j].values;
  }
}
```

Figure: Algorithm 1 eval Formula

```
value eval(Expression ξ_i, State s_i){
  case(ξ_i) of{
    v_i: return s_i(v_i);  c_i: return c_i;
    f(ξ_i^1, ..., ξ_i^k): return f(eval(ξ_i^1, s_i), ..., eval(ξ_i^k, s_i));
    @_j ξ'_j: return value of ξ'_j from KV_i[j].values;
  }
}
```

Figure: Algorithm 2 eval Expression

```
boolean init(Formula F_i, State s_i){
  if binary(op(F_i)) then{
    lval ← init(left(F_i, s_i));
    rval ← init(right(F_i, s_i));}
  else if unary(op(F_i)) then
     val ← init(subformula(F_i, s_i));
  index ← 0;
  case(op(F_i)) of{
    true : return true;   false : return false;
    P(ξ⃗_i) : return P(eval(ξ_i, s_i), ..., eval(ξ'_i, s_i)));
    op :  return rval op lval;  ¬ :  return not val;
    S :  now[index] ← rval; return now[index++];
    ⊡, ◊, ⊙ :  now[index] ← val; return now[index++];
  }
}
```

Figure: Algorithm 3 init

Frame Title

We use the function `eval` after every internal event or before sending any message to update the set $KVi[i].values$. We assign `now` to `pre` and, if a monitord PT-DTL formula F_i is specified for a process p_i, we designate p_i as the formula's owner.

At the owner process, we evaluate F_i using `eval` after each internal and receive event, following the update of the KNOWLEDGEVECTOR. If F_i evaluates to false, we report a violation warning.

The time and space complexity of this algorithm at every event is $\Theta(m)$, where m is the size of the original local formula.

Example

Let us consider three processes, p_1, p_2, and p_3. Process p_1 has a local variable x with an initial value of 5. Process p_2 has a local variable y with an initial value of 7.

Process p_2 monitors the formula $\mathcal{F}(y \geq @_1 x)$. An example computation is shown in Figure 9.

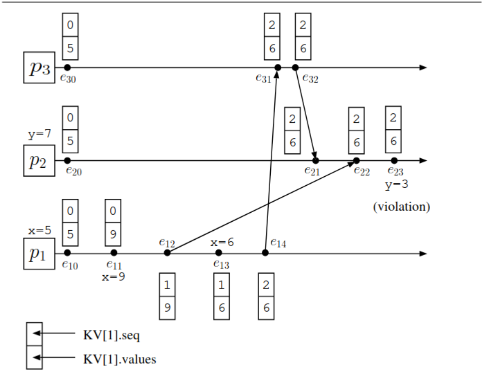

Figure: Monitoring of $\mathcal{F}(y \geq @1x)$ at p_2

Thank You