The Formal Verification of Vector Clocks

Jeffrey Cheng

Advisor: Andrew Appel

April 28, 2024

Abstract

In distributed systems, events occur in parallel across numerous processes, and each process in the network must agree on the order in which these events occur. Vector clocks address this problem by assigning sortable timestamps to each event, establishing a universal ordering that preserves the causal relationship between events. This preservation of causality is the key invariant that vector clocks aim to maintain. This project uses Coq to provide a formal, machine-checked proof which shows that the vector clock algorithm upholds this invariant. It also demonstrates how the validity of this invariant provides correctness guarantees in a real-world system that employs vector clocks, namely distributed database operations.

1 Introduction

A distributed system is a network of computers that process events locally and communicate with each other to achieve a computational goal. With so many events occuring in parallel, different machines will order the events differently due to randomized latencies in communication. This can have detrimental consequences, such as separate copies of a database executing reads and writes in different orders.

One proposed solution to this situation is vector clocks. Vector clocks aim to order events in a manner that preserves causality, which is the idea that some event A "could have caused or affected" another event B. For example, if a process writes the value 30 and later reads that 30, the write event "could have affected" the read event, and the write event should be ordered before the read[1]. We say "could have" because if a variable already had the value 30 before the write 30, then technically the write did not affect the result of the read. Nevertheless, the write could have affected the value returned by the read if not for coincidence.

One of the original papers to propose the vector clock algorithm was a paper by Colin Fidge in 1988 [1]. In it, he provides a semi-formal, partial proof of the algorithm. To guarantee the validity of vector clocks, this project formally proves the correctness of the vector clock algorithm in Coq, an automated proof assistant and checker [1]. Coq only accepts proofs that abide by a set of universally accepted axioms, meaning all Coq-validated proofs, such as the one in this paper, must be completely correct. This paper also demonstrates how a correct vector clock algorithm provides real-world correctness guarantees.

2 Background

2.1 Vector Clocks

Distributed Systems Vector clocks are used in distributed systems, where events occur on parallel processes, as depicted in Figure 1.

[1]The proof code can be found in our GitHub repository

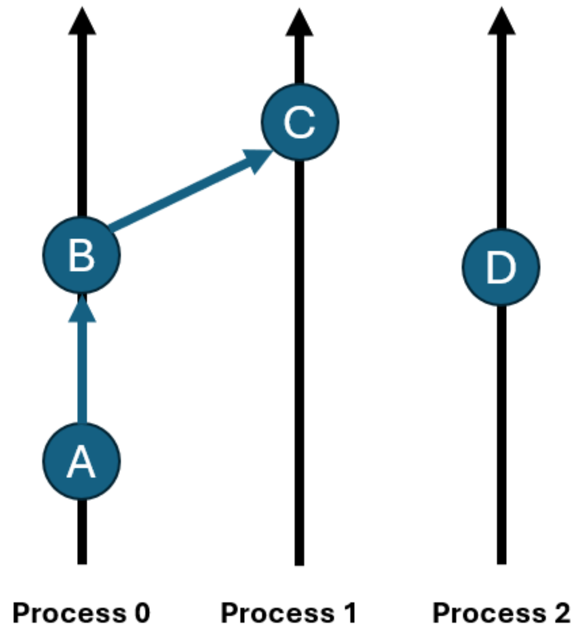

Figure 1: A example diagram of lettered events occuring on numbered processes.

In Figure 1, events A, B, C, and D happen accross processes 0, 1, and 2. These processes run independently from each other, but can transmit messages to communicate. In these scenarios, we consider three types of events:

1. Singleton Events (Event A): Events that occur locally on a process with no other effect

2. Send Message Events (Event B): Events that send a message to one or several processes

3. Receive Message Events (Event C): Events that receive a message from a different process

These three event types generalize quite well into most distributed system computations, and are the focus of vector clocks.

Causality Within these distributed computations, there is a notion of "causality." For example, in Figure 1, it would make sense to say that event A "could have caused" or "could have affected" event B. This is because event A occurs before event B on the same process, so anything done at

A is visible to B. Similarly, it would make sense to say that event B "could have caused" event C, because B sends a message that is directly received by C. Lastly, it would make sense to say that event A "could have caused" event C, because if event A could have caused event B and event B could have caused event C, then event A could have caused event C. These three relations are the three types of causality addressed by vector clocks, with causality usually being denoted with an arrow:

1. Sequential Causality (A $\rightarrow$ B): A occurs before B on the same process
2. Transmission Causality (B $\rightarrow$ C): B sends a message that C receives
3. Transitive Causality (A $\rightarrow$ C): A $\rightarrow$ B and B $\rightarrow$ C

On the other hand, it would not make sense to say that event C "could have caused" event D because these events occur on two different processes that do not communicate with each other in this situation. Thus, D would have no knowledge of C.

Vector Clocks The goal of vector clocks is to assign timestamps to each event in a way that preserves causality. Specifically, the vector clock algorithm aims to uphold the following invariant:

$$A \rightarrow B \iff \text{timestamp}(A) < \text{timestamp}(B)$$

This invariant means that for Figure 1, timestamp(A) < timestamp(B), timestamp(B) < timestamp(C), and timestamp(A) < timestamp(C). Conversely, this invariant also implies that timestamp$(C) \nless$ timestamp(D) and timestamp$(D) \nless$ timestamp(C).

Vector Clock Algorithm This paper uses the vector clock algorithm proposed by Colin Fidge in 1988 [1]. This section attempts to provide a brief, intuitive understanding of the algorithm, but consult the original paper for details. Note that the terms "timestamp," "vector time," and "clock time" are used interchangeably.

In accordance with its name, timestamps in the vector clock algorithm are represented by vectors of natural numbers with length n, where n is the number of processes in the distributed system of interest. Each process keeps track of its own vector timestamp, which can be thought of as that process's current clock time that gets updated as events occur. Every time an event occurs on some process p, p assigns its updated clock time to be the timestamp of that event.

The general algorithm rules are as follows, leaving out minor increments here and there for simplicity. All clock times begin as all zeroes, and when any event occurs on a process p, p updates index p of its clock time by 1. When a message transmission occurs from process p to process q, the send event of the transmission attaches its timestamp to the message, and process q updates each index i of its timestamp to $\max(\text{q\_clock}[i], \text{msg\_timestamp}[i])$. Consider the example below:

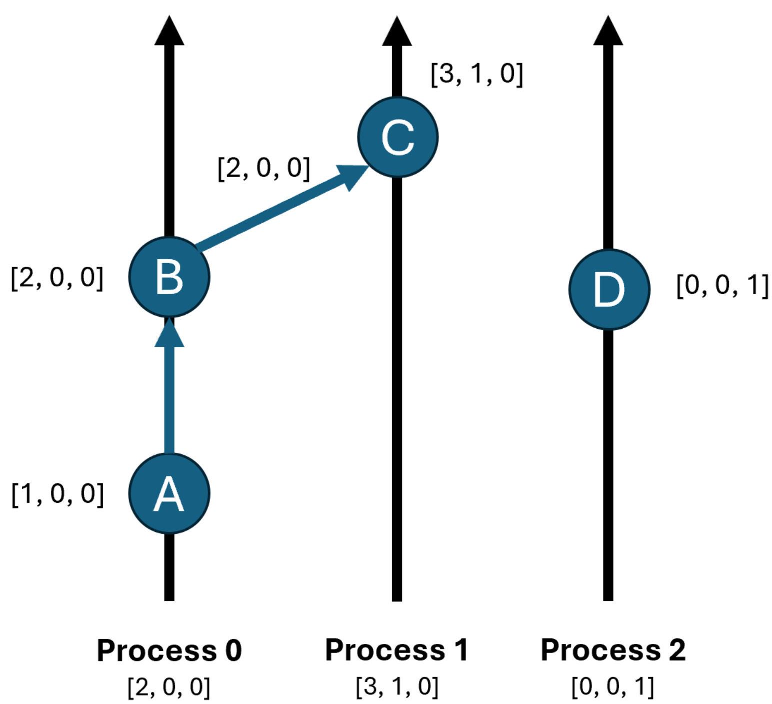

Figure 2: The distributed computation from Figure 1 with timestamps.

When event A occurs on process 0, clock_time(process 0)[index 0] increases by 1, and again for event B. Event B attaches its timestamp to the message, and process 1 updates each index of its clock to the message timestamp's respective value if it is larger, giving $[2, 0, 0]$. It then increments the indices of the send and receive processes to $[3, 1, 0]$. Process 2 has no idea that this is occuring because it has not received any messages, so it only has event D with a clock value of $[0, 0, 1]$. The vector times at the bottom represent the clock times of each process after this distributed computation has finished.

To compare the vector timestamps two events, the vector clock algorithm asserts the following invariant:

$$A \to B \iff \text{timestamp}(A)[p] < \text{timestamp}(B)[p] \quad \text{where A occurs on process p}$$

For example, $A \to B$ and timestamp$(A)[0]$ $<$ timestamp$(B)[0]$, $A \to C$ and timestamp$(A)[0]$ $<$ timestamp$(C)[0]$, and $D \nrightarrow C$ and timestamp$(D)[2]$ $\nless$ timestamp$(C)[2]$. From this point on, this invariant is referred to as the vector clock invariant, and is the main subject of interest for this paper.

Vector Clock Correctness Intuition Figure 3 below demonstrates the intuition behind why the vector clock algorithm works.

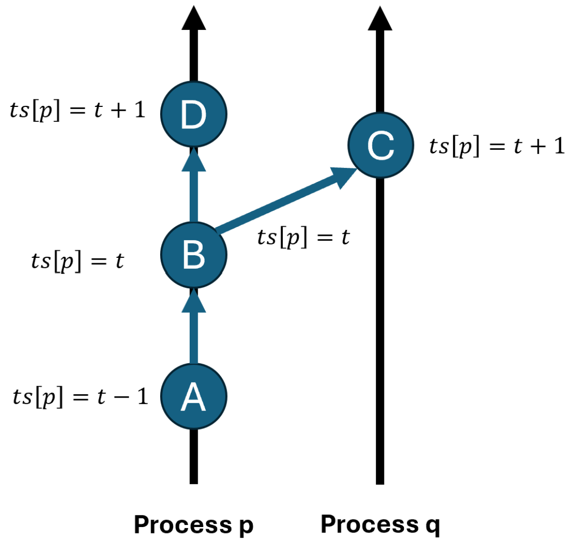

Figure 3: A distributed computation between processes p and q with timestamps abbreviated as ts.

Events A, B and D occur on process p, and their timestamps increase accordingly at index p. Event B sends a message to event C on process q, so event C "learns about" events A and B, but not D. The receive event timestamp calculation, referring to the max operations, is arranged so that event C's timestamp at index p is greater than that of every event from B and before, but no event afterward. In this case, this refers to event C's $ts[p]$ being $t+1$, which is greater than $ts_A[p] = t-1$ and $ts_B[p] = t$, but not $ts_D[p] = t + 1$.

A vectorized timestamp allows each process to track these relationships with every other process. The events that C learns about are the events that "could have caused" C, which is intuitively why the vector clock invariant is true. However, we would not like to believe in vector clocks based on intuition alone. Thus, this project's goal was to provide a formal, machine-checked proof that shows that the vector clock algorithm maintains the vector clock invariant. To accomplish this task, we use Coq.

Coq Coq is an automated proof checker and assistant for proving theorems about functional programs. In other words, Coq allows you to write functional programs, state theorems about those programs, and apply a series of proof steps to show that your theorems are correct. As you step through your proof, Coq will tell you exactly which steps are correct and incorrect using a set of universally accepted axioms, pre-proven Coq lemmas, and any lemma that is stated and proven by the user. See Figure 4 below for an example:

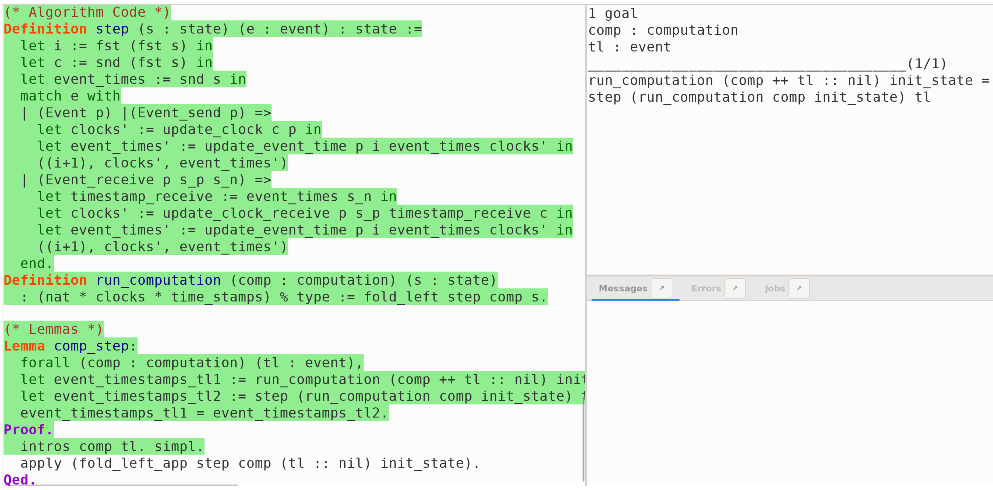

Figure 4: A screenshot of CoqIDE.

The top left quadrant of Figure 4 is part of the vector clock algorithm, implemented in Coq's functional programming language. The bottom left quadrant contains a lemma named "comp_step." It asserts:

Lemma (comp_step)**.** *For any variable* `comp` *of type computation and any variable* `tl` *of type event, we have:*

$$\texttt{run\_computation (comp ++ tl :: nil) init\_state} = \texttt{step (run\_computation comp init\_state) tl}$$

In other words, the two function calls, despite appearing different, always return the same value. In the top right quadrant, Coq will list defined variables, hypotheses, and the current manipulation of the theorem to prove.

In Fidge's original 1988 paper, he provides a semi-formal proof of

$$A \to B \Rightarrow \text{timestamp}(A)[p] < \text{timestamp}(B)[p]$$

where semi-formal means an argument in English and mathematical notation. This project uses Coq to provide a formal, machine-checked proof of the full vector clock invariant:

$$A \to B \iff \text{timestamp}(A)[p] < \text{timestamp}(B)[p]$$

where formal means listing the exact axiom or lemma used in each proof step.

3 Approach and Implementation

To complete this proof, there were five milestones to achieve in Coq:

1. Defining vector clocks
2. Implementing the vector clock algorithm
3. Stating the vector clock invariant
4. Proving the vector clock invariant
5. Proving a real-world vector clock guarantee

3.1 Defining Vector Clocks

The first step of the proof was to define all the elements of distributed systems and vector clocks in Coq's functional programming language.

3.1.1 Distributed Systems Elements

Processes and Events The code below shows how processes and events are represented in Coq.

`Definition` *process* := *nat*. (* nat is Coq's type for natural numbers *)

`Inductive` *event* : `Type` :=

| *Event* (*proc* : *process*)

| *Event_send* (*proc* : *process*)

| *Event_receive* (*proc* : *process*)

(*send_event_proc* : *process*) (*send_event_num* : *event_index*).

The `process` datatype was defined to be a natural number representing a process ID number. The event datatype was defined to have three constructors represnting each type of event:

1. <u>Event</u> (Singleton Event): A local event that occurs on a process `proc`

2. <u>Event_send</u> (Send Event): An event that sends a message from a process `proc`.

3. <u>Event_receive</u> (Receive Event): An event on process `proc` that receives a message sent by a send event `send_event_num` on the process `send_event_proc`

In Fidge's original paper, there is a one-to-one mapping between send and receive events [1]. The definition above only requires that every receive event has exactly one send event. The proof supports this broader definition without extra work, which allows for multicasts within the distributed system, meaning one event can send the same message to multiple other processes.

There is also a slight redundancy in this definition. Each receive event stores its respective send event and send process. The send event itself also stores its own process, so having the receive event store the send process as well is slightly redundant. This decision was made to simplify the proof process and requires a well-formedness guarantee that is defined in section 3.3.

Computations Vector clocks work on a distributed system with a set of events occuring over a set of processes. These situations are defined as a list of events, called a `computation`.

`Definition` *computation* := *list event*.

A computation is interpreted to be a set of events that occur on a distributed system in the order that they appear in the list. For example, [Event 0, Event 2, Event_send 0, Event_receive 1 0 2] would be the distributed computation in Figure 1. Note that since event D (Event 3) has no relationship to the other events, moving Event 3 around the list would represent an equivalent distributed computation.

It is important to see that (Event_receive 1 0 2) should store its respective send event, but its constructor only contains 3 natural numbers and no value of type event. This is due to the `event_index` datatype.

`Definition` *event_index* := *nat*.

The event constructors themselves do not provide a sufficient method of uniquely identifying events. For example, if there were two singleton events that occurred on the same process, their constructors would be exactly the same in a list of events. As a result, events are identified, stored, and compared using an index into a computation, defined using the datatype `event_index`. The last value in (Event_receive 1 0 2) is an `event_index` of 2, meaning this receive event's respective send event is the event at index 2 in the list of events that (Event_receive 1 0 2) is defined in, which is (Event_send 0) in [Event 0, Event 2, Event_send 0, Event_receive 1 0 2]. This has the important implication that the meaning of events depends on the list of events they appear in.

3.1.2 Causality

Having defined events in Coq, we must now define what it means for A $\rightarrow$ B given two events A and B. Recall the three types of causality: sequential causality, transmission causality, and transitive causality.

Sequential Causality Given a computation, an event A, and an event B, we define B to come sequentially after A in the computation if A and B occur on the same process and B comes later than A in the computation, embodying sequential causality.

To define this in Coq, we define a function `seq` that accepts a computation `comp`, an `event_index A`, and an `event_index B` that index two events A and B in `comp`. The function returns a proposition stating that events A and B occur on the same process and A happens before B in `comp`. Propositions in Coq are simply statements that can be true or false, and the proposition given by `seq comp A B` is true when B comes sequentially after A and false when B does not come sequentially after A.

```
Definition seq (comp : computation) (A B : event_index) : Prop :=
    (event_process comp A = event_process comp B) ∧ (A < B).
```

Transmission Causality Given a computation, an event A, and an event B, we define B to be the receive event of the send event A if B is a receive event, A is the send event stored in B's constructor, and the send message process number in B's constructor is A's process number, embodying transmission causality. We define a Coq function `msg` that again accepts a computation `comp`, an `event_index A`, and an `event_index B` that index events A and B in `comp`. It returns a proposition that is true when $A \rightarrow B$ by transmission causality and false otherwise.

```
Definition msg (comp : computation) (A B : event_index) : Prop :=
    match nth B comp (Event 0) with
    | Event_receive _ s_p s_n ⇒ A = s_n ∧ (event_process comp s_n = s_p)
    | _ ⇒ False
    end.
```

This function fetches the event B from `comp` using `event_index B` and pattern matches the constructor of B. If B is a singleton event or send event, it cannot be the case that $A \rightarrow B$ by transmission causality, so the proposition should always be false. If B is a receive event, it returns the proposition that `event_index A` matches the `event_index` of B's corresponding send event and event A occurs on the process that B received a message from according to B's constructor. Again, the returned proposition is true when $A \rightarrow B$ by the above definition of transmission causality and false otherwise.

General Causality Lastly, given a computation, an event A, and an event B, we define the general relation A $\rightarrow$ B using three constructors, one for each type of causality.

1. A $\rightarrow$ B if you can provide evidence that B comes sequentially after A
2. A $\rightarrow$ B if you can provide evidence that B is the receive event to the event send A
3. A $\rightarrow$ C if you can provide evidence that A $\rightarrow$ B and B $\rightarrow$ C for an event B in the computation

In Coq, we refer to A $\rightarrow$ B as the "after" relation, where saying B comes "after" A is the same as saying A "could have caused" B. We define the "after" relation below:

`Inductive after` : *computation* $\rightarrow$ *event_index* $\rightarrow$ *event_index* $\rightarrow$ `Prop` :=

| *after_sp* (*comp* : *computation*) (*A B* : *event_index*) (*H* : *seq comp A B*) : *after comp A B*

| *after_dp* (*comp* : *computation*) (*A B* : *event_index*) (*H* : *msg comp A B*) : *after comp A B*

| *after_trans* (*comp* : *computation*) (*A B C* : *event_index*) (*Htrans1* : *after comp A B*) (*Htrans2* : *after comp B C*) : *B* < *length comp* $\rightarrow$ *after comp A C*.

Similar to `seq` and `msg`, `after` accepts a computation `comp`, an `event_index A`, and an `event_index B`, as indicated to the right of the colon of each constructor. `after comp A B` is a proposition that is true when $A \rightarrow B$ and false otherwise. You can demonstrate that an `after` proposition is true using one of its three constructors. First, you could provide a proof that `seq comp A B` is true to show `after comp A B` by sequential causality using the `after_sp` constructor. Second, you could provide a proof that `msg comp A B` is true to show `after comp A B` by transmission causality using the `after_dp` constructor. Third, you could provide a proof that `after comp A B` is true, a proof that `after comp B C` is true, and a proof that `B` is a valid index into `comp` to show that `after comp A C` is true using the `after_trans` constructor.

Vector Timestamps The novelty of Fidge's paper comes in how events are timestamped. Each event receives a timestamp that is a vector of natural numbers, indexed by process numbers. Thus, a vector timestamp is defined in Coq as a function that maps process numbers to natural numbers. We did not use arrays because we wanted a strictly functional implementation to make the proof process easier. We did not use a list because a function allows for infinite processes, and updating the return value for one input of a function is easier than updating one element of a list.

`Definition` *time* := *nat*.

`Definition` *vclock_time* := *process* → *time*.

Each process maintains a vector timestamp as its clock value, so the set of all clock values is represented as a function mapping process numbers to their clock times. The vector clock algorithm aims to timestamp each event, so its output is represented as a mapping of events to their timestamp for a particular computation.

`Definition` *clocks* := *process* → *vclock_time*.

`Definition` *time_stamps* := *event_index* → *vclock_time*.

3.2 The Vector Clock Algorithm

The vector clock algorithm is a function named `run_computation` takes in a `computation` and returns a value of type `time_stamps`. It processes the list of events from head to tail, updating the process clocks and assigning a timestamp for each event. To make the proofs by induction easier, this was defined as a `fold_left` with a `step` function to process each event. The code below feeds a computation `comp` to the vector clock algorithm `run_computation` and tests if timestamp$(A)[p] <$ timestamp$(B)[p]$.

```
let p := event_process comp A in
let event_timestamps : time_stamps :=
    snd (run_computation comp init_state) in
event_timestamps A p < event_timestamps B p
```

3.3 Theorem Statement

$$A \to B \iff \text{timestamp}(A)[p] < \text{timestamp}(B)[p] \quad \text{where A occurs on process } p$$

This is the vector clock invariant that we want to prove. Below is the statement of the invariant in Coq.

```
Theorem order :
    ∀ (comp : computation) (A B : event_index) (p : process),
    A < length comp →
    B < length comp →
    p = event_process comp a →
    (∀ (i : event_index), well_formed comp i) →
    after comp A B ↔
    let event_timestamps : time_stamps :=
        snd (run_computation comp init_state) in
    event_timestamps A p < event_timestamps B p.
```

There are four assumptions required for the invariant. Events A and B must be in the given computation, p must be the process that A occurs on, and the computation must be well-formed. There are two well-formedness rules that are not encapsulated by the definitions of the vector clock elements and must be assumed as hypotheses. First, every receive event must come after its corresponding send event. Second, the send message process number stored in a receive event constructor must be equal to the process stored by the corresponding send event's constructor. Each of these well-formedness statements are written as Coq propositions, and the `well_formed` proposition that asserts that both the well-formedness propositions are true for some event in `comp`. The entire well-formedness hypothesis asserts that every event in `comp` is well-formed.

```
(* send event precedes receive event *)
Definition sender_prec (comp : computation) (a : nat) : Prop :=
    match nth a comp (Event 0) with
    | Event_receive _ _ s_n ⇒ s_n < a
    | _ ⇒ True
    end.
```

```
(* send event process matches send message process stored in receive event constructor *)
Definition sender_proc (comp : computation) (a : nat) : Prop :=
    match nth a comp (Event 0) with
    | Event_receive _ s_p s_n ⇒ event_process comp s_n = s_p
    | _ ⇒ True
    end.
```

3.4 Theorem Proof

To prove the theorem, both the forward and backward direction of the if-and-only-if had to be proven. The forward direction was broken into three subcases, one for each type of `after` constructor to show that the invariant holds for all three types of causality. From there, each subcase was proven using induction on the length of a list. This meant that the proof goal could be assumed as an induction hypothesis for any list of events of length n, and it had to be shown that appending an event to the end of the list would not break the invariant. Since the timestamps assigned to events do not change when an event is appended to the end of the list, our induction hypothesis for lists of length n is quite powerful when proving the invariant for lists of length $n + 1$. If A nor B are the tail element of the list, we can remove the tail element, and our induction hypothesis proves the invariant for A and B. This meant that each one of the subcase proofs only had to deal with the scenario where A or B is the tail element of the list.

3.4.1 The Forward Direction: Sequential Events

$$\textit{seq comp A B} \rightarrow \textit{event\_timestamps A p} < \textit{event\_timestamps B p}$$

The first subcase to prove is that if B occurs sequentially after A on the same process, then timestamp$(A)[p]$ < timestamp$(B)[p]$, where A occurs on process p. For each event that occurs on a process p, the vector clock algorithm increases p's clock time at index p by 1. Since A and B occur on the same process p and B occurs after A, it is quite straightforward to see how index p of B's timestamp would be larger than A's.

3.4.2 The Forward Direction: Message Transmission Events

$$\textit{msg comp A B} \rightarrow \textit{event\_timestamps A p} < \textit{event\_timestamps B p}$$

The second subcase to prove is that if B receives a message that A sends, timestamp$(A)[p]$ < timestamp$(B)[p]$, where A occurs on process p. According to the vector clock algorithm, timestamp$(B)[p] =$ max(local_clock$[p]$, timestamp$(A)[p]+1$). No matter which value is larger, timestamp$(B)[p]$ is assigned a value larger than timestamp$(A)[p]$.

3.5 The Forward Direction: Transitive Events

$$\texttt{after}\ \textit{comp A B} \rightarrow \texttt{after}\ \textit{comp B C} \rightarrow \textit{event\_timestamps A p} < \textit{event\_timestamps C p}$$

The last subcase of the forward direction is that if B comes after A and C comes after B, timestamp$(A)[p]$ < timestamp$(C)[p]$ where A occurs on process p. Given the first two subcases, this seems trivial, but there is a catch. When checking if $A \rightarrow B$, the index p is always the process that A, the first event, occurs on. For example, according to the invariant, $A \rightarrow B \Leftrightarrow$ timestamp$(A)[p] <$ timestamp$(A)[p]$ and $B \rightarrow A \Leftrightarrow$ timestamp$(B)[q] <$ timestamp$(A)[q]$ where A occurs on process p and B occurs on process q. By comparing at different indices based on the direction of the arrow, we can have $A \nrightarrow B$ and $B \nrightarrow A$ without being forced to assign timestamps of equal values.

What this means for the transitive property is that our invariant gives us $B \rightarrow C \Rightarrow$ timestamp$(B)[q] <$ timestamp$(C)[q]$, where B occurs on process q and q might not be equal to p. As a result, $B \rightarrow C$ does not guarantee timestamp$(B)[p] <$ timestamp$(C)[p]$, which we wanted for the transitive inequality. First, it must be proven that $A \rightarrow B \Rightarrow$ timestamp$(A)[i] \leq$ timestamp$(B)[i]$ for all i, which is proven in essentially the same manner as the forward direction of the original invariant. This lemma does have a trivial transitive subcase.

Once the less than or equal to lemma is proven, we can reason that $A \rightarrow B \Rightarrow$ timestamp$(A)[p] <$ timestamp$(B)[p]$ and $B \rightarrow C \Rightarrow$ timestamp$(B)[p] \leq$ timestamp$(C)[p]$, yielding timestamp$(A)[p] <$ timestamp$(C)[p]$ where A occurs on process p.

3.5.1 The Backward Direction

The backward direction of the vector clock invariant is the more interesting result. To prove it, we prove the contrapositive, meaning we show that

$$A \nrightarrow B \Rightarrow \text{timestamp}(A)[p] >= \text{timestamp}(B)[p]$$

which is logically equivalent to timestamp$(A)[p] <$ timestamp$(B)[p] \Rightarrow A \rightarrow B$. As explained earlier, we use induction on the length of a list, allowing us to assume $A \nrightarrow B \Rightarrow$ timestamp$(A)[p] >=$ timestamp$(B)[p]$ for any events A, B in a list of length n. We have to show that this invariant holds when an element is appended to the end of the list, and that event is either A or B.

If A is the last element on a process p, then timestamp$(A)[p]$ cannot be smaller than any other event's timestamp at index p. As a result, timestamp$(A)[p]$ $>=$ timestamp$(B)[p]$ for all other events B. The intuition for B as the tail element is more complicated.

Case 1 First, take the case where B is a receive event that inherited index p of its timestamp from a previous send event X during the max(local_clock$[p]$, msg_timestamp[p]) calculation. Since $X \rightarrow B$ from transmission causality and $A \nrightarrow B$ was assumed in the contrapositive, it must be that $A \nrightarrow X$, otherwise $A \rightarrow B$ by transitive causality. X precedes B, so it is not the tail element and we can use it with the inductive hypothesis, which is the contrapositive: $A \nrightarrow X \Rightarrow$ timestamp$(A)[p]$ $>=$ timestamp$(X)[p]$. To calculate the relation between timestamp$(X)[p]$ and timestamp$(B)[p]$, we do case analysis on all the placements of X as shown below:

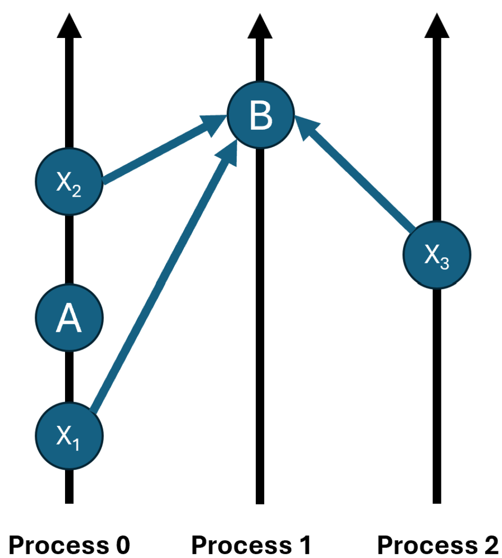

Figure 5: All placements of a send event X for a receive event B.

For X_1, timestamp$(X_1)[p] + 1 =$ timestamp$(B)[p]$ and timestamp$(X_1)[p] <$ timestamp$(A)[p]$, yielding timestamp$(A)[p]$ $>=$ timestamp$(B)[p]$. For X_2, X_2 comes sequentially after A, meaning $A \rightarrow X$, which is a contradiction. For X_3, timestamp$(X_3)[p]$ $=$ timestamp$(B)[p]$, giving timestamp$(A)[p]$ $>=$ timestamp$(B)[p]$. These three cases are exhaustive because X_1 and X_2 consider the cases where A and X are on the same process, and X_3 generalizes to any case where they are not.

Case 2 Now take any other case. This means that B is either a singleton event, a send event, or a receive event who inherited index p of its timestamp from its own process's clock during the max(local_clock$[p]$, msg_timestamp[p]) calculations. Let X be the last event on the same process as B excluding B. This means that B happens immediately after X on the same process, so $X \rightarrow B$. We now employ a similar argument as case 1.

Since $X \rightarrow B$ and $A \nrightarrow B$, $A \nrightarrow X$. By the inductive hypotheses, $A \nrightarrow X \Rightarrow$ timestamp$(A)[p]$ $>=$ timestamp$(X)[p]$. Since B comes immediately after X on some process q, q is the only timestamp index that would increase from X to B, so timestamp$(B)[p]$ $=$ timestamp$(X)[p]$. Note that q cannot be equal to p, otherwise B would come sequentially after A and $A \rightarrow B$, which is a contradiction. We can now say that timestamp$(A)[p]$ $>=$ timestamp$(B)[p]$. This concludes the final subcase for proving the main vector clock theorem.

3.6 Real-World Application

Having proved the main vector clock invariant, we can show how useful this guarantee is with a real-world algorithm that employs vector clocks. One such application of vector clocks is distributed database operations. For this case, we can think of each process being an independent user and each event being a database operation that a user executes, with each user beginning with a copy of the same database state. As users execute local operations, they modify their own copy of the database state, meaning each user could have a different database state after a few operations. On the other hand, users can also communicate with each other to merge their modified database states. In these scenarios, certain database operations follow directly from others. However, other sets of operations can be executed independently of each other because their users were not in communication. We can use vector clocks to know exactly which operations can affect other operations.

For simplicity, we define our database's state to be a single natural number. We consider every operation event, including send and receive events, to be a database write and database read, increasing the numerical state and assigning that state value to the event. Singleton and send events simply increment their process's state by one, but receive events set their process's state to the sum of the states of the send process and receive process, and then increment by 1 to represent a merging of states. That way, modifying events before either the receive event or its corresponding send event will modify the merged state value. As a result, modifying the list of database operations that occur may or may not modify certain operations' assigned database state.

We want to reason about how modifying one operation in a list of operations affects the database state assigned to other operations. One way of testing this is stifling the state update of an operation A, then seeing whether that affects the state value assigned to an operation B. The state calculation function takes in a list of operations and one index and prevents the operation at that index in the list from modifying its process's database state. If that operation is a send or receive event, the message transmission still works, just without a state increment or summation upon reception.

In the real world, this function would likely use vector clocks to tell when stifling the state update of operation A could affect an operation B. The developer would reason that $A \to B$ is synonymous with "stifling A affects B", and our vector clock invariant would allow them to say timestamp$(A)[p]$ $<$ timestamp$(B)[p]$ if and only if stifling A affects B. This allows them to use vector clocks to tell when certain operations affect others. Thus, our distributed database invariant is as follows:

`Theorem` *affects*:

$\forall$ (*comp* : *computation*) (*A B* : *event_index*) (*p* : *process*) (*db_state* : *database_states*),

A < *length comp* $\rightarrow$

B < *length comp* $\rightarrow$

p = *event_process comp A* $\rightarrow$

($\forall$ (*i* : *event_index*), *well_formed comp i*) $\rightarrow$

`let` *event_timestamps* : *time_stamps* :=

snd (*run_computation comp init_state*) `in`

event_timestamps A p < *event_timestamps B p* $\leftrightarrow$

`let` *op_states* := *apply_operations comp db_state* (*length comp*) `in`

`let` *op_states_rem_A* := *apply_operations comp db_state A* `in`

op_states B $\neq$ *op_states_rem_A B*.

For this proof and most other real-world application proofs, the intuitive approach would be to show that $A \rightarrow B \iff \texttt{op\_states B} \neq \texttt{op\_states\_rem\_A B}$, meaning $A \rightarrow B$ if and only if stifling A changes the state assigned to B. We could then use our vector clock theorem to relate event timestamps with the database state calculation algorithm. However, since the state calculation algorithm mirrors the vector clock algorithm quite closely, the proof was done by showing timestamp$(A)[p] <$ timestamp$(B)[p] \iff \texttt{op_states B} \neq \texttt{op_states_rem_A B}$ directly. Both these statements are equivalent, and the final result is

$$A \rightarrow B \iff \texttt{op\_states B} \neq \texttt{op\_states\_rem\_A B} \iff \text{timestamp}(A)[p] \geq \text{timestamp}(B)[p]$$

The proof of this theorem used similar techniques to the proof of the main vector clock theorem. Using induction on the length of the list, we only really need to worry about the case where B is the tail of the list. Find the event X that B inherits index p of its timestamp from. $\texttt{op\_states X} \neq \texttt{op\_states\_rem\_A X} \iff \text{timestamp}(A)[p] < \text{timestamp}(X)[p]$ by the induction hypothesis, and manually calculate the relationship between X and B for both their timestamps and database states, given B builds exactly one step off of X.

4 Results and Conclusion

From this project, we are now able to conclude the validity of vector clocks with 100% certainty. By providing Coq with incredibly detailed proof steps, we have shown that there is no possible corner case in the algorithm that was forgetten nor incorrectly reasoned about.

This provides a pleasant guarantee of correctness for those that employ vector clocks in their systems, such as distributed databases. By assigning vector timestamps to the events of a distributed system, developers can know exactly when one event can affect another. This allows them to remove, add, or change events to distributed computations and know exactly which states will be affected. For example, database users could delete database operations and know which database reads will come back differently to avoid unwanted consequences. GitHub users could delete commits and know which future commits will be affected. We have also formally proven that these systems express causality exactly as defined. In other words, $A \to B$ truly is synonymous with "stifling A's state update affects B's state," which is an intuitive but non-trivial idea. This shows that vector clocks are the correct tool of choice for these applications.

For most of the computer science world, semi-formal proofs are often enough to convince people of correctness, and vector clocks are commonly used without having a formal proof until now. However, this project does not just provide a proof to a widely accepted notion. There are many vector clocks optimizations whose correctness are not as obvious. To show that these optimizations are not to greedy and do not violate the vector clock invariant, this formal proof is a good foundation and model of how to show correctness in algorithms related to vector clocks. One straightforward corollary is the correctness of Lamport clocks, which only address the forward direction of the vector clock algorithm. Ultimately, this project provides a foundation and reference for many proofs to come, while allowing those who employ vector clocks in their own systems to sleep easier at night.

5 Acknowledgements

I would like to thank my advisor, Professor Andrew Appel, for his guidance and feedback throughout this project.

References

[1] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

I pledge my honor that this paper represents my own work in accordance with University regulations. - Jeffrey Cheng