View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Interval Tree Clocks

A Logical Clock for Dynamic Systems

Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte

DI/CCTC, Universidade do Minho
Largo do Paço, 4709 Braga Codex, Portugal
{psa,cbm,vff}@di.uminho.pt

Abstract. Causality tracking mechanisms, such as vector clocks and version vectors, rely on mappings from globally unique identifiers to integer counters. In a system with a well known set of entities these ids can be preconfigured and given distinct positions in a vector or distinct names in a mapping. Id management is more problematic in dynamic systems, with large and highly variable number of entities, being worsened when network partitions occur. Present solutions for causality tracking are not appropriate to these increasingly common scenarios. In this paper we introduce *Interval Tree Clocks*, a novel causality tracking mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralized creation of processes/replicas without need for global identifiers or global coordination. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. The representation is so compact that the mechanism can even be considered for scenarios with a fixed number of entities, which makes it a general substitute for vector clocks and version vectors.

Key words: Causality, logical clock, version vectors, vector clocks, dynamic systems.

1 Introduction

Ever since causality was introduced in distributed systems [12], it has played an important role in the modeling of distributed computations. In the absence of global clocks, causality remains as a means to reason about the order of distributed events. In order to be useful, causality is implemented by concrete mechanisms, such as Vector Clocks [7, 16] and Version Vectors [18], where a compressed representation of the sets of events observed by processes or replicas is kept.

These mechanisms are based on a mapping from a globally unique identifier to an integer counter, so that each entity (i.e. process or replica) keeps track of how many events it knows from each other entity. A special and common case is when the number of entities is known: here ids can be integers, and a vector of counters can be used.

Nowadays, distributed systems are much less static and predictable than those traditionally considered when the basic causality tracking mechanisms were created. In dynamic distributed systems [17], the number of active entities varies during the system execution and in some settings, such as in peer-to-peer deployments, the level of change, due to churn, can be extremely high.

Causality tracking in dynamic settings is not new [8] and several proposals analyzed the dynamic creation and retirement of entities [21, 9, 20, 13, 2]. However, in most cases localized retirement is not supported: all active entities must agree before an id can be removed [21, 9, 20] and a single unreachable entity will stall garbage collection. Localized retirement is only partially supported in [13], while [2] has full support but the mechanism itself exhibits an unreasonable structural growth that its practical use is compromised [3].

This paper addresses causality tracking in dynamic settings and introduces Interval Tree Clocks (ITC), a novel causality tracking mechanism that generalizes both Version Vectors and Vector Clocks. It does not require global ids but is able to create, retire and reuse them autonomously, with no need for global coordination; any entity can fork a new one and the number of entities can be reduced by joining arbitrary pairs of entities; stamps tend to grow or shrink, adapting to the dynamic nature of the system. Contrary to some previous approaches, ITC is suitable for practical uses, as the space requirement scales well with the number of entities and grows modestly over time.

In the next section we review the related work. Section 3 introduces a model based on fork, event and join operations that factors out a kernel for the description of causality systems. Section 4 builds on the identified core operations and introduces a general framework that expresses the properties that must be met by concrete causality tracking mechanisms. Section 5 introduces the ITC mechanism and correctness argument under the framework. Before conclusions, in Section 7, we present in Section 6 a simple simulation based assessment of the space requirements of the mechanism.

2 Related Work

After Lamport's description of causality in distributed system [12], subsequent work introduced the basic mechanisms and theory [18, 7, 16, 5]. We refer the interested reader to the survey in [22] and to the historical notes in [4]. After an initial focus on message passing systems, recent developments have improved causality tracking for replicated data: they addressed efficient coding for groups of related objects [14]; bounded representation of version vectors [1]; and the semantics of reconciliation [10].

Fidge introduces in [8] a model with a variable number of process ids. In this model process ids are assumed globally unique and are gradually introduced by process spawning events. No garbage collection of ids is performed when processes terminate.

Garbage collection of terminated ids requires additional meta-data in order to assess that all active entities already witnessed the termination; otherwise, ids cannot be safely removed from the vectors. This approach is used in [9, 21] together with the assumption of globally unique ids. In [20] the assumption of global ids is dropped and each entity is able to produce a globally unique id from local information. A typical weakness in these systems is twofold: terminated ids cannot be reused; and garbage collection is hampered by even a single unreachable entity. In addition, when garbage collection cannot terminate, the associated meta-data overhead cannot be freed. Since this overhead is substantial, when the likelihood of non termination is high, it can be more efficient not to garbage collect and keep the inactive ids.

The mechanism described in [13] provides local retirement of ids but only for restricted termination patterns (a process can only be retired by joining a direct ancestor); moreover, the use of global ids is required.

Our own work in [2] introduced localized creation and retirements of ids and presented Version Stamps, a dynamic substitute to version vectors. Although still of theoretical interest as it does not use counters, and although it inspired the id management technique used in ITC, the technique was later found out to exhibit very adverse growth in common scenarios [3]. The id management technique used in version stamps shares many properties with credit management techniques in termination detection algorithms [15, 11].

In order to control version vector growth, in Dynamo [6] old inactive entries are garbage collected. Although the authors tune it so that in production systems errors are unlikely to be introduced, in general this can lead to resurgence of old updates. Mechanisms like ITC may help in avoiding the need for these aggressive pruning solutions.

3 Fork-Event-Join Model

Causality tracking mechanisms can be modeled by a set of core operations: fork, event and join, that act on stamps (logical clocks) whose structure is a pair (i, e), formed by an id and an event component that encodes causally known events. Fidge used in [8] a model that bears some resemblance, although not making explicit the id component.

Causality is characterized by a partial order over the event components, $(E, \leq)$. In version vectors, this order is the pointwise order on the event component: $e \leq e'$ iff $\forall k.\ e[k] \leq e'[k]$. In causal histories [22], where event components are sets of event ids, the order is defined by set inclusion.

fork The fork operation allows the cloning of the causal past of a stamp, resulting in a pair of stamps that have identical copies of the event component and distinct ids; $\mathrm{fork}(i, e) = ((i_1, e), (i_2, e))$ such that $i_2 \neq i_1$. Typically, $i = i_1$ and i_2 is a new id. In some systems i_2 is obtained from an external source of unique ids, e.g. MAC addresses. In contrast, in Bayou [20] i_2 is a function of the original stamp $f((i, e))$; consecutive forks are assigned distinct ids since an event is issued to increment a counter after each fork.

peek A special case of fork when it is enough to obtain an *anonymous* stamp $(\mathbf{0}, e)$, with "null" identity, than can be used to transmit causal information but cannot register events, $\mathrm{peek}((i, e)) = ((i, e), (\mathbf{0}, e))$. Anonymous stamps are typically used to create messages or as inactive copies for later debugging of distributed executions.

event An event operation adds a new event to the event component, so that if (i, e') results from $\mathrm{event}((i, e))$ the causal ordering is such that $e < e'$. This action does a strict advance in the partial order such that e' is not dominated by any other entity and does not dominate more events than needed: for any other event component x in the system, $e' \not\leq x$ and when $x < e'$ then $x \leq e$. In version vectors the event operation increments a counter associated to the identity in the stamp: $\forall k \neq i.\ e'[k] = e[k]$ and $e'[i] = e[i] + 1$.

join This operation merges two stamps, producing a new one. If $\text{join}((i_1, e_1), (i_2, e_2)) = (i_3, e_3)$, the resulting event component e_3 should be such that $e_1 \leq e_3$ and $e_2 \leq e_3$. Also, e_3 should not dominate more that either e_1 and e_2 did. This is obtained by the order theoretical join, $e_3 = e_1 \sqcup e_2$, that must be defined for all pairs; i.e. the order must form a join semilattice. In causal histories the join is defined by set union, and in version vectors it is obtained by the pointwise maximum of the two vectors.

The identity should be based on the provided ones, $i_3 = f(i_1, i_2)$ and kept globally unique (with the exception of anonymous ids). In most systems this is obtained by keeping only one of the ids, but if ids are to be reused it should depend upon and incorporate both [2].

When one stamp is anonymous, join can also model message reception, where $\text{join}((i, e_1), (\mathbf{0}, e_2)) = (i, e_1 \sqcup e_2)$. When both ids are defined, the join can be used to terminate an entity and collect its causal past. Also notice that joins can be applied when both stamps are anonymous, modeling in-transit aggregation of messages.

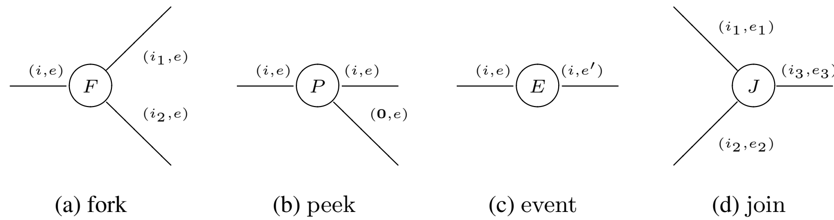


Fig. 1: Core operations.

Classic operations can be described as a composition of these core operations:

send This operation is the atomic composition of event followed by peek. E.g. in vector clock systems, message sending is modeled by incrementing the local counter and then creating a new message.

receive A receive is the atomic composition of join followed by event. E.g. in vector clocks taking the pointwise maximum is followed by an increment of the local counter.

sync A sync is the atomic composition of join followed by fork. E.g. In version vector systems and in bounded version vectors [1] it models the atomic synchronization of two replicas.

Figure 2 depicts graphical representations of these composite operations, but other composite operations could also be easily described using the same set of core operations. For instance, a message multicast could be modeled as the atomic composition of an event operation followed by a sequence of peek operations.

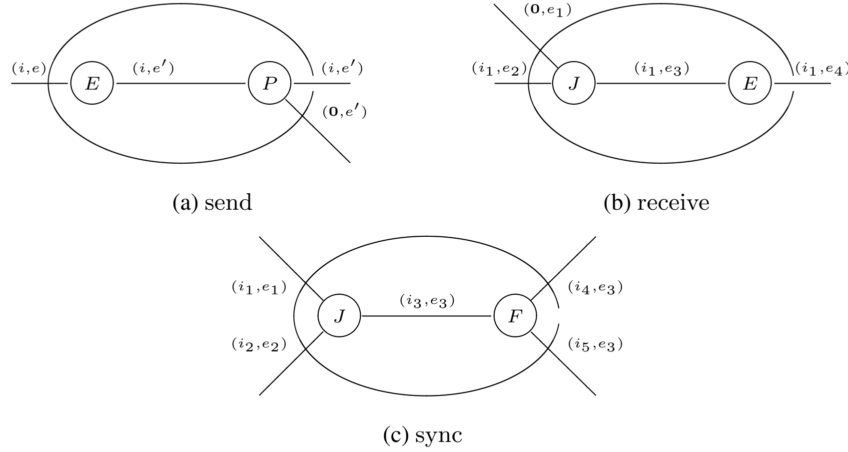


Fig. 2: Some composite operations.

Traditional descriptions assume a starting number of entities. This can be simulated by starting from an initial *seed* stamp and forking several times until the required number of entities is reached.

4 Function Space Based Clock Mechanisms

In this section we present a general framework which can be used to explain and instantiate concrete causality tracking mechanisms, such as our own ITC presented in the next section. Here stamps are described in terms of functions and some invariants are presented towards ensuring correctness. Actual mechanisms can be seen as finite encodings of such functions. Correctness of each mechanism will follow directly from the correctness of the encoding and from respecting the corresponding semantics and conditions to be met by each operation. In the following we will make use of the standard pointwise sum, product, scaling, partial ordering and join of functions:

$$(f + g)(x) \doteq f(x) + g(x),$$

$$(f \cdot g)(x) \doteq f(x) \cdot g(x),$$

$$(n \cdot g)(x) \doteq n \cdot g(x),$$

$$f \leq g \doteq \forall x.\ f(x) \leq g(x),$$

$$(f \sqcup g)(x) \doteq f(x) \sqcup g(x),$$

and of a function $\mathbf{0}$ that maps all elements to 0:

$$\mathbf{0} \doteq \lambda x. 0.$$

A stamp will consist of a pair (i, e): the identity and the event components, both functions from some arbitrary domain to natural numbers. The identity component is a

characteristic function (maps elements to $\{0, 1\}$) that defines the set of elements in the domain available to *inflate* ("increment") the event function when an event occurs. We chose to use the characteristic function instead of the set as it leads to better notation. The essential point towards ensuring a correct tracking of causality is to be able to inflate the mapping of some element which no other entity (process or replica) has access to[1]. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities. This is expressed by the following invariant over the identity components of all entities:

$$\forall i.\ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

We adopt a less general but more useful invariant, as it can be maintained by local operations without access to global knowledge. It consists of having disjointness of the parts of the domain that are mapped to 1 in each entity; i.e. non-overlapping graphs for any pair of id functions.

$$\forall i_1 \neq i_2.\ i_1 \cdot i_2 = \mathbf{0}.$$

Comparison of stamps is made through the event component:

$$(i_1, e_1) \leq (i_2, e_2) \doteq e_1 \leq e_2.$$

Join takes two stamps, and returns a stamp that causally dominates both (therefore, the event component is a join of the event components), and has the elements from both identities available for future event accounting:

$$\mathrm{join}((i_1, e_1), (i_2, e_2)) \doteq (i_1 + i_2, e_1 \sqcup e_2).$$

Fork can be any function that takes a stamp and returns two stamps which keep the same event component, but split between them the available elements in the identity; i.e. any function:

$$\mathrm{fork}((i, e)) \doteq ((i_1, e), (i_2, e)) \qquad \text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}.$$

Peek is a special case of fork, which results in one *anonymous* stamp with $\mathbf{0}$ identity and another which keeps all the elements in the identity to itself:

$$\mathrm{peek}((i, e)) \doteq ((i, e), (\mathbf{0}, e)).$$

Event can be any function that takes a stamp and returns another with the same identity and with an event component inflated on any arbitrary set of elements available in the identity:

$$\mathrm{event}((i, e)) = (i, e + f \cdot i) \qquad \text{for any } f \text{ such that } f \cdot i > \mathbf{0}.$$

An event cannot be applied to an anonymous stamp as no element in the domain is available to be inflated.

[1] If this property is not met it can still be possible to form an order that is compatible with causality, but where some concurrent events appear as ordered. This is the case in Lamport clocks [12] and in plausible clocks [23] where the stated invariant does not hold. A Lamport clock can be modeled by having the same identity in all entities.

5 Interval Tree Clocks

We now describe *Interval Tree Clocks*, a novel clock mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralized creation of processes/replicas without need for global identifiers. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. There are two essential differences between ITC and classic clock mechanisms, from the point of view of our function space framework:

- in classic mechanisms each entity uses a fixed, pre-defined function for id; in ITC the id component of entities is manipulated to adapt to the dynamic number of entities;
- classic mechanisms are based on functions over a discrete and typically finite domain; ITC is based on functions over a continuous infinite domain ($\mathbb{R}$) with emphasis on the interval $[0, 1)$; this domain can be split into an arbitrary number of subintervals as needed.

The idea is that each entity has available, in the id, a set of intervals that it can use to *inflate* the event component and to give to the successors when forking; a join operation joins the sets of intervals. Each interval results from successive partitions of $[0, 1)$ into equal subintervals; the set of intervals is described by a binary tree structure. Another binary tree structure is also used for the event component, but this time to describe a mapping of intervals to integers. To describe the mechanism in terms of functions, it is useful to define a *unit pulse* function[2]:

$$\mathbf{1} \doteq \lambda x. \begin{cases} 1 & x \geq 0 \wedge x < 1, \\ 0 & x < 0 \vee x \geq 1. \end{cases}$$

The id component is an *id tree* with the recursive form (where i, i_1, i_2 range over id trees):

$$i ::= 0 \mid 1 \mid (i_1, i_2).$$

We define a semantic function for the interpretation of id trees as functions:

$$\begin{aligned} [\![0]\!] &= \mathbf{0} \\ [\![1]\!] &= \mathbf{1} \\ [\![(i1, i2)]\!] &= \lambda x. [\![i_1]\!](2x) + [\![i_2]\!](2x - 1). \end{aligned}$$

These functions can be 1 for some subintervals of $[0, 1)$ and 0 otherwise. For an id (i_1, i_2), the functions corresponding to the two subtrees are transformed so as to be non-zero in two non-overlapping subintervals: i_1 in the interval $[0, 1/2)$ and i_2 in the interval $[1/2, 1)$. As an example, $(1, (0, 1))$ represents the function $\lambda x. \mathbf{1}(2x) + (\lambda x. \mathbf{1}(2x -$

[2] In this paper we use the *lambda calculus* notation for defining unary functions: a function is anonymously defined by a lambda expression which expresses its action on its argument. For instance, the "increment" function f such that $f(x) = x + 1$ would be expressed as $\lambda x. x + 1$

$1))(2x-1)$. We will also use a graphical notation, which is based on the graph of the function over $[0,1)$. Examples:

$$(1,(0,1)) \sim$$
$$((0,(1,0)),(1,0)) \sim$$

The event component is a binary *event tree* with non-negative integers in nodes; using e, e_1, e_2 to range over event trees and n over non-negative integers:

$$e ::= n \mid (n, e_1, e_2).$$

We define a semantic function for the interpretation of these trees as functions:

$$\begin{aligned} [\![n]\!] &= n \cdot \mathbf{1} \\ [\![(n, e1, e2)]\!] &= n \cdot \mathbf{1} + \lambda x.\, [\![e_1]\!](2x) + [\![e_2]\!](2x-1). \end{aligned}$$

This means that the value for an element in some subinterval is the sum of a *base* value, common for the whole interval, plus a *relative* value from the corresponding subtree. We will also use a graphical notation for the event component; again, it is based on the graph of the function, obtained by "stacking" the corresponding parts. An example:

$$(1,2,(0,(1,0,2),0)) \sim$$

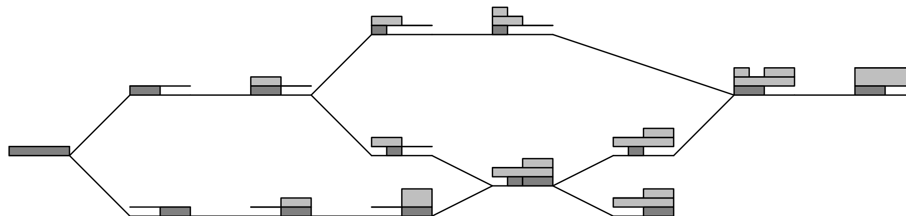
A stamp in ITC is a pair (i,e), where i is an id tree and e an event tree; we will also use a graphical notation based on stacking the two components:

$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \sim$$

ITC makes use what we call the *seed* stamp, $(1,0)$, from which we can fork as desired to obtain an initial configuration.

5.1 An Example

We now present an example to illustrate the intuition behind the mechanism, showing a run with a dynamic number of entities in the fork-event-join model. The run starts by a single entity, with the *seed* stamp, which forks into two; one of these suffers one event and forks; the other suffers two events. At this point there are three entities. Then, one entity suffers an event while the remaining two synchronize by doing a join followed by a fork.



The example shows how ITC adapts to the number of entities and allows simplifications to occur upon joins or events. While the first two forks had to split a node in the id tree, the third one makes use of the two available subtrees. The final join leads to a simplification in the id by merging two subtrees. It can be seen that each event always inflates the event tree in intervals available in the id. The event after the final join managed to perform an inflation in a way such that the resulting event function is represented by a single integer.

5.2 Normal Form

There can be several equivalent representations for a given function. ITC is conceived so as to keep stamps in a *normal form*, for the representations of both id and event functions. This is important not only for having compact representations but also to allow simple definitions of the operations on stamps (fork, event, join) as shown below. As an example, for the unit pulse, we have:

$$\mathbf{1} \sim 1 \equiv (1,1) \equiv (1,(1,1)) \equiv ((1,1),1) \equiv \ldots$$

This means that, if after a join the resulting id is $(1,(1,1))$, we can simplify it to 1. Normalization of the id component can be obtained by applying the following function when building the id tree recursively:

$$\begin{aligned} \mathrm{norm}((0,0)) &= 0, \\ \mathrm{norm}((1,1)) &= 1, \\ \mathrm{norm}(i) &= i. \end{aligned}$$

The event component can be also normalized, preserving its interpretation as a function. Two examples:

$$(2,1,1) \sim \quad \equiv \quad \sim 3,$$
$$(2,(2,1,0),3) \sim \quad \equiv \quad \sim (4,(0,1,0),1).$$

To normalize the event component we will make use of the following operators to "lift" or "sink" a tree:

$$\begin{aligned} n{\uparrow}^m &= n+m, \\ (n,e_1,e_2){\uparrow}^m &= (n+m,e_1,e_2), \\ n{\downarrow}_m &= n-m, \\ (n,e_1,e_2){\downarrow}_m &= (n-m,e_1,e_2). \end{aligned}$$

Normalization of the event component can be obtained by applying the following function when building a tree recursively (where m and n range over integers and e_1 and e_2 over normalized event trees) :

$$\begin{aligned} \mathrm{norm}(n) &= n, \\ \mathrm{norm}((n,m,m)) &= n+m, \\ \mathrm{norm}((n,e_1,e_2)) &= (n+m, e_1{\downarrow}_m, e_2{\downarrow}_m), \text{ where } m = \min(\min(e_1),\min(e_2)), \end{aligned}$$

where $\min$ applied to a tree returns the minimum value of the corresponding function in the range $[0, 1)$:

$$\min(e) = \min_{x \in [0,1)} [\![e]\!](x),$$

which can be obtained by the recursive function over event trees:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n + \min(\min(e_1), \min(e_2)), \end{aligned}$$

or more simply, assuming the event tree is normalized:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n, \end{aligned}$$

which explores the property that in a normalized event tree, one of the subtrees has minimum equal to 0. We will also make use of the analogous $\max$ function over event trees that returns the maximum value of the corresponding function in the range $[0, 1)$, and can be obtained by the recursive function:

$$\begin{aligned} \max(n) &= n, \\ \max((n, e_1, e_2)) &= n + \max(\max(e_1), \max(e_2)). \end{aligned}$$

5.3 Operations over ITC

We now present the operations on ITC for the fork-event-join model. They are defined so as to respect the operations and invariants from the function space based framework presented in the previous section. All the functions below take as input and give as result stamps in the normal form.

Comparison Comparison of ITC can be derived from the pointwise comparison of the corresponding functions:

$$(i_1, e_1) \leq (i_2, e_2) \doteq [\![e_1]\!] \leq [\![e_2]\!].$$

It is trivial to see that this can be computed through a recursive function over normalized event trees; i.e. $(i_1, e_1) \leq (i_2, e_2) \iff \mathrm{leq}(e_1, e_2)$, with leq defined as (where l and r range over the "left" and "right" subtrees):

$$\begin{aligned} \mathrm{leq}(n_1, n_2) &= n_1 \leq n_2, \\ \mathrm{leq}(n_1, (n_2, l_2, r_2)) &= n_1 \leq n_2, \\ \mathrm{leq}((n_1, l_1, r_1), n_2) &= n_1 \leq n_2 \wedge \mathrm{leq}(l_1{\uparrow}^{n_1}, n_2) \wedge \mathrm{leq}(r_1{\uparrow}^{n_1}, n_2), \\ \mathrm{leq}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= n_1 \leq n_2 \wedge \mathrm{leq}(l_1{\uparrow}^{n_1}, l_2{\uparrow}^{n_2}) \wedge \mathrm{leq}(r_1{\uparrow}^{n_1}, r_2{\uparrow}^{n_2}). \end{aligned}$$

Fork Forking preserves the event component, and must split the id in two parts whose corresponding functions do not overlap and give the original one when added.

$$\mathrm{fork}(i, e) \doteq ((i_1, e), (i_2, e)), \text{ where } (i_1, i_2) = \mathrm{split}(i),$$

for a function split such that:

$$(i_1, i_2) = \mathrm{split}(i) \implies [\![i_1]\!] \times [\![i_2]\!] = \mathbf{0} \wedge [\![i_1]\!] + [\![i_2]\!] = [\![i]\!].$$

This is satisfied naturally using the following recursive function over id trees, as the two subtrees of an id component always represent functions that do not overlap:

$$\begin{aligned}
\mathrm{split}(0) &= (0, 0),\\
\mathrm{split}(1) &= ((1, 0), (0, 1)),\\
\mathrm{split}((0, i)) &= ((0, i_1), (0, i_2)), \text{ where } (i_1, i_2) = \mathrm{split}(i),\\
\mathrm{split}((i, 0)) &= ((i_1, 0), (i_2, 0)), \text{ where } (i_1, i_2) = \mathrm{split}(i),\\
\mathrm{split}((i_1, i_2)) &= ((i_1, 0), (0, i_2))
\end{aligned}$$

Join Joining two entities is made by summing the corresponding id functions and making a join of the corresponding event functions:

$$\mathrm{join}((i_1, e_1), (i_2, e_2)) \doteq (\mathrm{sum}(i_1, i_2), \mathrm{join}(e_1, e_2)),$$

for a sum function over identities and a join function over event trees such that:

$$\begin{aligned}
[\![\mathrm{sum}(i_1, i_2)]\!] &= [\![i_1]\!] + [\![i_2]\!],\\
[\![\mathrm{join}(e_1, e_2)]\!] &= [\![e_1]\!] \sqcup [\![e_2]\!].
\end{aligned}$$

The sum function that respects the above condition and also produces a normalized id is:

$$\begin{aligned}
\mathrm{sum}(0, i) &= i,\\
\mathrm{sum}(i, 0) &= i,\\
\mathrm{sum}((l_1, r_1), (l_2, r_2)) &= \mathrm{norm}((\mathrm{sum}(l_1, l_2), \mathrm{sum}(r_1, r_2))).
\end{aligned}$$

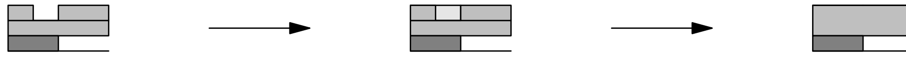
Likewise, the join function over event trees, producing a normalized event tree is:

$$\begin{aligned}
\mathrm{join}(n_1, n_2) &= \max(n_1, n_2),\\
\mathrm{join}(n_1, (n_2, l_2, r_2)) &= \mathrm{join}((n_1, 0, 0), (n_2, l_2, r_2)),\\
\mathrm{join}((n_1, l_1, r_1), n_2) &= \mathrm{join}((n_1, l_1, r_1), (n_2, 0, 0)),\\
\mathrm{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \mathrm{join}((n_2, l_2, r_2), (n_1, l_1, r_1)), \text{ if } n_1 > n_2,\\
\mathrm{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \mathrm{norm}((n_1, \mathrm{join}(l_1, l_2{\uparrow}^{n_2 - n_1}), \mathrm{join}(r_1, r_2{\uparrow}^{n_2 - n_1}))).
\end{aligned}$$

Event The event operation is substantially more complex than the others. While fork and join have a simple natural definition, event has a larger freedom of implementation while respecting the condition:

$$\text{event}((i,e)) = (i,e'),\ \text{subject to } [\![e']\!] = [\![e]\!] + f \cdot [\![i]\!] \text{ for any } f \text{ such that } f \cdot [\![i]\!] > \mathbf{0}.$$

Event cannot be applied to anonymous stamps; it has the precondition that the id is non-null; i.e. $i \neq 0$. We can use any subset of the available id to inflate the event function. The freedom of which part to inflate is explored in ITC so as to simplify the event tree. Considering the final event in our larger example:



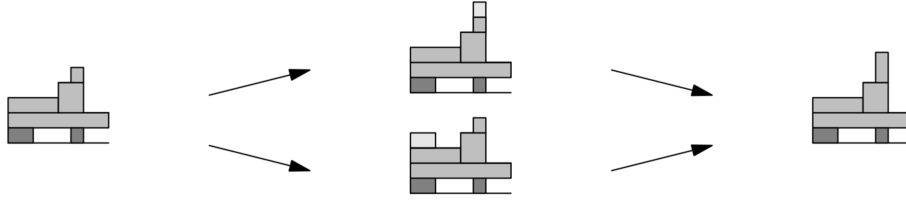
The event operation was able to fill the missing part in a tree so as to allow its simplification to a single integer. In general, the event operation can use several parts of the id, and may simplify several subtrees simultaneously. The operation performs all simplifications in the event tree that are possible given the id tree. If some simplification is possible (which means the corresponding function was inflated), the resulting tree is returned; otherwise another procedure is applied, that "grows" some subtree, preferably only incrementing an integer if possible. The event operation is defined resorting to these two functions (fill and grow) defined below:

$$\text{event}(i,e) = \begin{cases} (i, \text{fill}(i,e)) & \text{if } \text{fill}(i,e) \neq e, \\ (i,e') & \text{otherwise, where } (e',c) = \text{grow}(i,e). \end{cases}$$

Fill either succeeds in doing one or more simplifications, or returns an unmodified tree; it never increments an integer that would not lead to simplifying the tree:

$$\begin{aligned}
\text{fill}(0,e) &= e, \\
\text{fill}(1,e) &= \max(e), \\
\text{fill}(i,n) &= n, \\
\text{fill}((1,i_r),(n,e_l,e_r)) &= \text{norm}((n, \max(\max(e_l), \min(e'_r)), e'_r)), \\
&\quad \text{where } e'_r = \text{fill}(i_r, e_r), \\
\text{fill}((i_l,1),(n,e_l,e_r)) &= \text{norm}((n, e'_l, \max(\max(e_r), \min(e'_l)))), \\
&\quad \text{where } e'_l = \text{fill}(i_l, e_l), \\
\text{fill}((i_l,i_r),(n,e_l,e_r)) &= \text{norm}((n, \text{fill}(i_l,e_l), \text{fill}(i_r,e_r))).
\end{aligned}$$

In the following example, fill is unable to perform any simplification and grow is used. From the two candidate inflations shown in light grey, the one chosen requires a simple integer increment, while the other would require expanding a node:



Grow performs a dynamic programming based optimization to choose the inflation that can be performed, given the available id tree, so as to minimize the cost of the event tree growth. It is defined recursively, returning the new event tree and cost, so that:

- incrementing an integer is preferable over expanding an integer to a tuple;
- to disambiguate, an operation near the root is preferable to one farther away.

$$
\begin{aligned}
\mathrm{grow}(1, n) &= (n+1, 0),\\
\mathrm{grow}(i, n) &= (e', c+N), \text{ where } (e', c) = \mathrm{grow}(i, (n, 0, 0)),\\
&\quad \text{and } N \text{ is some large constant},\\
\mathrm{grow}((0, i_r), (n, e_l, e_r)) &= ((n, e_l, e'_r), c_r + 1), \text{ where } (e'_r, c_r) = \mathrm{grow}(i_r, e_r),\\
\mathrm{grow}((i_l, 0), (n, e_l, e_r)) &= ((n, e'_l, e_r), c_l + 1), \text{ where } (e'_l, c_l) = \mathrm{grow}(i_l, e_l),\\
\mathrm{grow}((i_l, i_r), (n, e_l, e_r)) &= \begin{cases} ((n, e'_l, e_r), c_l + 1) & \text{if } c_l < c_r,\\ ((n, e_l, e'_r), c_r + 1) & \text{if } c_l \geq c_r, \end{cases}\\
&\quad \text{where } (e'_l, c_l) = \mathrm{grow}(i_l, e_l) \text{ and } (e'_r, c_r) = \mathrm{grow}(i_r, e_r).
\end{aligned}
$$

The definition makes use of a constant N that should be greater than the maximum tree depth that arises. This is a practical choice, to have the cost as a simple integer. We could avoid it by having the cost as a pair under lexicographic order, but it would "pollute" the presentation and be a distracting element.

6 Exercising ITCs

In order to have a rough insight of ITC space consumption we exercised its usage for both dynamic and static scenarios, using a mix of data and process causality. For data causality in dynamic scenarios, each iteration consists of forking, recording an event and joining two replicas, each performed on random replicas, leading to constantly evolving ids. This pattern maintains the number of existing replicas while exercising id management under churn. For process causality in a static scenario, we operate on a fixed set of processes doing message exchanges (via peek and join) and recording internal events; here ids remain unchanged, since messages are anonymous.

The charts in Figure 3 depict the mean size (using the binary encoding shown in Appendix A) of an ITC across 100 runs of 25,000 iterations for process causality and 100,000 iterations for data causality and for different numbers of active entities (pre-created by forking a *seed* stamp before iterating). It shows that space consumption

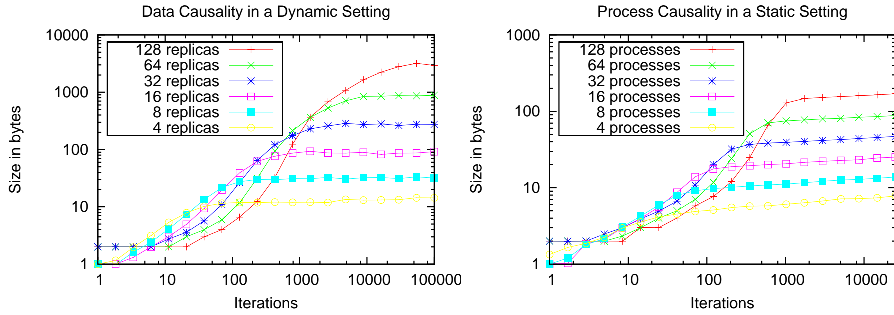


Fig. 3: Average space consumption of an ITC stamp, in dynamic and static settings.

basically stabilizes after a number of iterations. These results show that ITCs can in fact be used as a practical mechanism for data and process causality in dynamic systems, contrary to Version Stamps [2] that have storage cost growing unreasonably over time.

In order to put these numbers in perspective, the Microsoft Windows operating system [19] uses 128 bits Universally Unique Identifiers (UUIDs) and 32 bit counters. The storage cost of a version vector for 128 replicas would be 2560 bytes using a mapping from ids to counters and 512 bytes using a vector. The mean size of an ITC for this scenario (at the end of the iterations) would be less than 2900 bytes for dynamic scenarios and slightly above 170 bytes for static ones. While vectors can be represented in a more compact way (e.g. factoring out the smallest number), such optimizations would be irrelevant for dynamic scenarios, where most of the cost stems from the UUIDs.

7 Conclusions

We have introduced Interval Tree Clocks, a novel logical clock mechanism for dynamic systems, where processes/replicas can be created or retired in a decentralized fashion. The mechanism has been presented using a model (fork-event-join) that can serve as a kernel to describe all classic operations (like message sending, symmetric synchronization and process creation/retirement), being suitable for both process and data causality scenarios.

We have presented a general framework for clock mechanisms, where stamps can be seen as finite representations of a pair of functions over a continuous domain; the event component serves to perform comparison or join (performed pointwise); the identity component defines a set of intervals where the event component can be inflated (a generalization of the classic counter increment). ITC is a concrete mechanism that instantiates the framework, using trees to describe functions on sets of intervals. The framework opens the way for research on future alternative mechanisms that use different representations of functions.

Previous approaches to causality tracking for dynamic systems either require access to globally unique ids; do not reuse ids of retired entities; require global coordination for garbage collection of ids; or exhibit an intolerable growth in terms of space consumption

(our previous approach). ITC is the first mechanism for dynamic systems that avoids all these problems, can be used for both process and data causality, and requires a modest space consumption, making it a general purpose mechanism, even for static systems.

References

1. José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Rachid Guerraoui, editor, *DISC*, volume 3274 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2004.
2. Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps – decentralized version vectors. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 544–551. IEEE Computer Society, 2002.
3. Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Improving on version stamps. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part II*, volume 4806 of *Lecture Notes in Computer Science*, pages 1025–1031. Springer, 2007.
4. Roberto Baldoni and Michel Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.
5. Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, 1991.
6. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.
7. Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, 1989.
8. Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.
9. Richard G. Golden, III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing (JPDC)*, 55(1):109–120, 1998.
10. Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2006.
11. Shing-Tsaan Huang. Detecting termination of distributed computations by external agents. In *International Conference on Distributed Computing Systems*, pages 79–84, Newport Beach, California, 1989.
12. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
13. Tobias Landes. Tree clocks: An efficient and entirely dynamic logical time system. In Helmar Burkhart, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, as part of the 25th IASTED International Multi-Conference on Applied Informatics, February 13-15 2007, Innsbruck, Austria*, pages 349–354. IASTED/ACTA Press, 2007.

14. Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.
15. Mattern. Global quiescence detection based on credit distribution and recovery. *IPL: Information Processing Letters*, 30, 1989.
16. Friedemann Mattern. Virtual time and global clocks in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.
17. Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *Proceedings 24th IEEE Symposium on Reliable Distributed Systems (24th SRDS'05)*, pages 109–118, Orlando, FL, USA, October 2005. IEEE Computer Society.
18. D. Stott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *Transactions on Software Engineering*, 9(3):240–246, 1983.
19. Daniel Peek, Douglas B. Terry, Venugopalan Ramasubramanian, Meg Walraed-Sullivan, Thomas L. Rodeheffer, and Ted Wobber. Fast encounter-based synchronization for mobile devices. *Digital Information Management, 2007. ICDIM '07. 2nd International Conference on*, 2:750–755, 2007.
20. K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated database services for world-wide applications. In 7th *ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.
21. David Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997.
22. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 3(7):149–174, 1994.
23. F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.

A A Binary Encoding for ITC

Here we describe a compact encoding of ITC as strings of bits. It may be relevant when stamp size is an issue, e.g. when many entities are involved; it is appropriate to being transmitted or stored persistently as a single *blob*. We do not attempt to present an optimal (in some way) encoding, but a sensible one, which was used in the space consumption analysis.

As an event tree tends to have very few large numbers near the root and many very small numbers at the leaves; this prompts a variable length representation for integers, where small integers occupy just a few bits. Also, common cases like trees with only the left or right subtree, or with 0 for the base value are treated as special cases.

We use a notation (inspired by the bit syntax from the Erlang programming language) where: $\ll x, y, z \gg$ is a string of bits resulting from concatenating x, y and z; and $n{:}b$ represents number n encoded in b bits. An example: $\ll 2{:}3, 0{:}1, 1{:}2 \gg$ represents the string of 6 bits 010001.

$$enc((i,e)) = \ll enc_i(i), enc_e(e) \gg.$$

$$\begin{aligned}
enc_i(0) &= \ll 0{:}2, 0{:}1 \gg, \\
enc_i(1) &= \ll 0{:}2, 1{:}1 \gg, \\
enc_i((0,i)) &= \ll 1{:}2, enc_i(i) \gg, \\
enc_i((i,0)) &= \ll 2{:}2, enc_i(i) \gg, \\
enc_i((i_l,i_r)) &= \ll 3{:}2, enc_i(i_l), enc_i(i_r) \gg.
\end{aligned}$$

$$\begin{aligned}
enc_e((0,0,e_r)) &= \ll 0{:}1, 0{:}2, enc_e(e_r) \gg, \\
enc_e((0,e_l,0)) &= \ll 0{:}1, 1{:}2, enc_e(e_l) \gg, \\
enc_e((0,e_l,e_r)) &= \ll 0{:}1, 2{:}2, enc_e(e_l), enc_e(e_r) \gg, \\
enc_e((n,0,e_r)) &= \ll 0{:}1, 3{:}2, 0{:}1, 0{:}1, enc_e(n), enc_e(e_r) \gg, \\
enc_e((n,e_l,0)) &= \ll 0{:}1, 3{:}2, 0{:}1, 1{:}1, enc_e(n), enc_e(e_l) \gg, \\
enc_e((n,e_l,e_r)) &= \ll 0{:}1, 3{:}2, 1{:}1, enc_e(n), enc_e(e_l), enc_e(e_r) \gg, \\
enc_e(n) &= \ll 1{:}1, enc_n(n,2) \gg.
\end{aligned}$$

$$enc_n(n,B) = \begin{cases} \ll 0{:}1, n{:}B \gg & \text{if } n < 2^B, \\ \ll 1{:}1, enc_n(n-2^B, B+1) \gg & \text{otherwise.} \end{cases}$$