Cheaper matrix clocks

authors : **Frédéric Ruget**

project :

state : Draft

classification : Public

distribution :

keywords : PRO, REP

abstract : Matrix clocks have nice properties that can be used in the context of distributed database protocols and fault tolerant protocols. Unfortunately, they are costly to implement, requiring storage and communication overhead of size $\mathcal{O}(n^2)$ for a system of n sites. They are often considered a non feasible approach when the number of sites is large.

In this paper, we firstly describe an efficient incremental algorithm to compute the matrix clock, which achieves storage and communication overhead of size $\mathcal{O}(n)$ when the sites of the computation are "well synchronized". Secondly, we introduce the *k-matrix clock*: an approximation to the genuine matrix clock that can be computed with a storage and communication overhead of size $\mathcal{O}(kn)$. k-matrix clocks can be useful to implement fault-tolerant protocols for systems with crash failure semantics such that the maximum number of simultaneous faults is bounded by $k-1$.

A shorter version of this paper was presented at the 8th. international workshop on distributed algorithms, Terschelling, the Netherlands, October 1994.

Cheaper matrix clocks

Frédéric Ruget
Chorus Systèmes, 6 avenue Gustave Eiffel
78182 Montigny le Bx, France
ruget@chorus.fr

Abstract

Matrix clocks have nice properties that can be used in the context of distributed database protocols and fault tolerant protocols. Unfortunately, they are costly to implement, requiring storage and communication overhead of size $\mathcal{O}(n^2)$ for a system of n sites. They are often considered a non feasible approach when the number of sites is large.

*In this paper, we firstly describe an efficient incremental algorithm to compute the matrix clock, which achieves storage and communication overhead of size $\mathcal{O}(n)$ when the sites of the computation are "well synchronized". Secondly, we introduce the k-*matrix clock*: an approximation to the genuine matrix clock that can be computed with a storage and communication overhead of size $\mathcal{O}(kn)$. k-matrix clocks can be useful to implement fault-tolerant protocols for systems with crash failure semantics such that the maximum number of simultaneous faults is bounded by $k-1$.*

Key words: distributed systems, causality, logical time, matrix time, fault tolerance.

1 Introduction

Matrix clocks have been introduced in the context of asynchronous distributed systems. They have nice properties that can be used to design distributed database protocols and fault tolerant protocols [WB84, KB91]. Unfortunately, they are costly to implement: in a distributed system that consists of n sites, the naive algorithm to compute the matrix clock on the fly requires that an $n \times n$ matrix of integers be stored at each site and tagged onto each message: if the number of sites is large, it is necessary to use an optimized algorithm.

[WB84] describes several such optimized algorithms. Usually, optimized algorithms do not actually compute the genuine matrix clock, but an approximation thereto. The approximation is less expensive to compute but provides less information than the genuine clock. However, many applications will be happy with an approximated matrix clock, as long as it meets their needs (for example, the approximated matrix clock of [WB84] meet the needs of a distributed dictionary and a distributed log protocols).

This paper brings (as far as we know) two contributions to the theory of matrix clocks. Firstly, we derive an efficient incremental algorithm to compute the genuine matrix clock from work done in the context of the MANETHO fault tolerance project [EZ92, EZ93]. The algorithm achieves storage and communication overhead of size $\mathcal{O}(n)$ when the sites of the computations are "well synchronized".

Secondly, we introduce a new approximation to the matrix clock: the k-matrix clock. The k-matrix clock can be useful to implement optimistic fault tolerance mechanisms for a system with crash failure semantics [CASD85] such that the maximum number of simultaneous site failures is bounded by $k-1$. We propose an algorithm to compute the k-matrix clock on the fly, with storage and communication overhead of size $\mathcal{O}(kn)$. This bound does not depend on the relative synchronization between the sites of the computation.

The remainder of the paper is organized as follows. In Sect. 2 we recall the well known matrix clock. We mention some of its properties and describe how it can be implemented. In Sect. 3, we give a small taxonomy of known cheaper approximations to the matrix clock (this part is based on [WB84]). Section 4 presents our efficient incremental algorithm to compute the matrix clock. Section 5 presents our "k-matrix clock" approximation to the matrix clock, describes its properties and gives possible examples of its use. We conclude in Sect. 6.

2 Logical clocks

Logical clocks have been introduced in the framework of asynchronous, distributed systems [Lam78]. Let us quote the definition from [PT92]. The term *distributed* means that the system is composed of a set of sites that can communicate only by sending messages along a fixed set of channels. The term *asynchronous* means that there is no global clock in the system, no assumptions about the relative speed of sites, no assumptions about the delivery time of messages, and the sending and the receiving of a message are two distinct actions.

It is not possible to totally order the events that occur in a computation of an asynchronous distributed system. It is however possible to causally order [Lam78] the events of the computation. We say that an event e_1 causally precedes event an e_2 (denoted $e_1 \leq e_2$) if either **(1)** e_1 and e_2 occur on the same site S_i and e_1 occurs before e_2 (denoted $e_1 \leq_i e_2$, or **(2)** e_1 is the emission of a message and e_2 is its receipt, or **(3)** there exists an event $e3$ such that $e_1 \leq e_3$ and $e_3 \leq e_2$. The causal order is a partial order of the events of the computation.

A logical clock δ associates a date $\delta(x) \in D$ to each event $x \in E$ of the computation.

$$\begin{aligned} \delta : E &\rightarrow D \\ x &\mapsto \delta(x) \end{aligned}$$

The set D of clock values is partially ordered, and all logical clocks satisfy the following

(CLK) condition:

$$e_1 < e_2 \implies \delta(e_1) < \delta(e_2) \tag{CLK}$$

that is, the clock never goes backwards. Most logical clocks actually satisfy the stronger (S-CLK) condition:

$$e_1 \leq e_2 \iff \delta(e_1) \leq \delta(e_2) \tag{S-CLK}$$

that is, the clock exactly represents the causal structure of the partial order of events.

For example, the linear clock introduced by Lamport [Lam78] only satisfies (CLK). Vector clock (introduced independently by Fischer and Michael [FM82] and Liskov and Ladin [LL86], then formalized by Fidge [Fid91] and Mattern [Mat89]) satisfies (S-CLK).

2.1 Matrix clock

Matrix clock (denoted δ_{mat}) associates a square matrix of $n \times n$ integers to each event of the computation. The definition of matrix clock is:

$$\begin{aligned} \delta_{\mathrm{mat}} : E &\rightarrow \mathbb{N}^{n \times n} \\ x &\mapsto \left(\mathrm{card}(\downarrow_{E_j} \downarrow_{E_i} \{x\})\right)_{i,j \in \{1,\ldots,n\}} \end{aligned}$$

where $\downarrow_{E_i} \{x\}$ is the predecessor set of $\{x\}$ in E_i, that is the set of elements of E_i that are lower than x [1] and $\mathrm{card}(X)$ denotes the number of elements in the set X.

The definition of matrix clock is illustrated by Fig. 1.

Intuitively, if x is an event occurring on site S_i, then component $[j, k]$ of $\delta_{\mathrm{mat}}(x)$ represents S_i's view of S_j's view of the progress of S_k's local time. The matrix clock satisfies the (S-CLK) condition.

Matrix clocks have been introduced by Wuu and Bernstein [WB84] and Sarin and Lynch [SL87] as a means to discard obsolete information: since matrix clocks gives information about the other sites views, it makes it possible for a site to stop diffusing an information as soon as it knows that it is in all other sites' views.

Let us recall the algorithm for computing the matrix clock on the fly. Each site S_i maintains its own view $M_i \in \mathbb{N}^{n \times n}$ of the matrix clock and tags it onto all outgoing messages. M_i is initially set to zero. Rules (INT-M) and (MSG-M) are applied:

INT-M: Before S_i performs an event:

$$M_i[i, i] \leftarrow M_i[i, i] + 1$$

[1] Formally, $x \in \downarrow_B (A) \iff x \in B \wedge \exists y \in A, x \leq y$

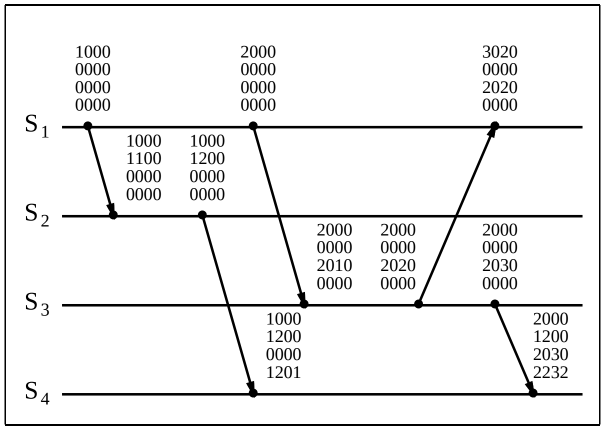

Figure 1: The matrix clock

MSG-M: Before S_i receives message (m, M) from S_j:

$$\begin{aligned} \forall l, M_i[i,l] &\leftarrow \max(M_i[i,l], M[j,l]) \\ \forall k,l, M_i[k,l] &\leftarrow \max(M_i[k,l], M[k,l]) \end{aligned}$$

3 Approximations to the matrix clock

Section 2.1 describes a "naive" algorithm to compute the matrix clock on the fly that generates a storage overhead of size $\mathcal{O}(n^2)$ per site and a communication overhead of size $\mathcal{O}(n^2)$ per message. This is very expensive. If the number of sites is large, it is necessary to use an optimized algorithm.

In [WB84], Wuu and Bernstein describe several such optimized algorithms. These optimized algorithms do not actually compute the genuine matrix clock, but an approximation thereto. The approximation is less expensive to compute but provides less information than the genuine clock. However, many applications (e.g. the distributed dictionary and the distributed log protocols of [WB84]) will be happy with an approximated matrix clock, as long as it meets their needs.

In this section, we briefly recall the various optimization strategies proposed by Wuu and Bernstein. For that purpose we have considered the sample computation of Fig. 2 and the message represented by a dotted arrow in the figure. For each strategy, we have indicated the part of the matrix clock that is stored at the sending site and the part that is tagged onto the message.

NAIVE strategy

The strategy described in Sect. 2.1. Storage and communication overheads are $\mathcal{O}(n^2)$.

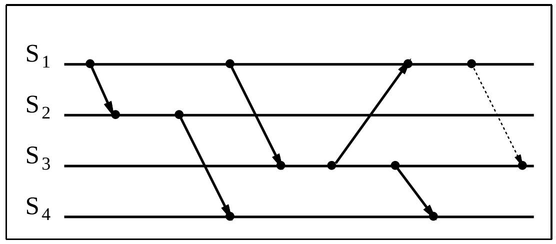

Figure 2: Sample computation

VECTOR strategy

Each site maintains a complete matrix, but outgoing messages are tagged only with the row that corresponds to the sending site. Communication overhead is thus only $\mathcal{O}(n)$. Unfortunately, VECTOR strategy does not guarantee progress. This is illustrated in Fig. 3: the drift between the genuine matrix clock and the clock obtained with VECTOR strategy grows without bound [2].

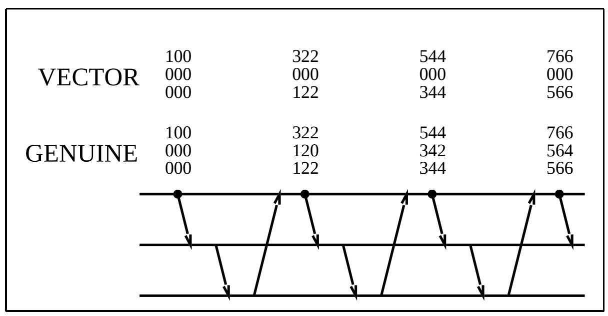

Figure 3: VECTOR does not guarantee progress

NEIGBH1 strategy

NEIGHB1 strategy assumes a fixed topology of the communication network. Each site stores only its row and a row for each of its neighbors [3]. It sends only those rows which correspond to neighbors of the target site. Storage and communication overhead is thus $\mathcal{O}(kn)$, where k is a bound of the number of neighbors a site may have.

NEIGBH2 strategy

Each site stores only the components of the matrix clock which correspond to a channel that belongs to the same network area [4] as the site (i.e., site S_i stores component $[j, k]$ of the

[2] To compensate for this drift, VECTOR strategy needs to be joined to a gossip mechanism [HHW89], so that all sites can regularly update their views of the matrix clock.

[3] Two sites are called *neighbors* if they are directly connected by a communication channel.

[4] An *area* of the communication network is a maximum sub-network such that each site in the area is

matrix clock if and only if sites S_i, S_j and S_k belong to the same area). It sends only those components of the matrix clock which correspond to channels in the same area as the target site. Communication overhead is $\mathcal{O}(k^2)$ where k is a bound of the number of neighbors a site may have. Storage overhead $\mathcal{O}(k^2)$ for each area to which the site belongs.

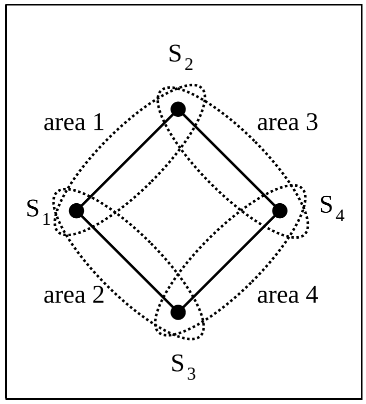

Figure 4: Sample network topology

Figure 6 illustrates these various strategies (for NEIGHB1 and NEIGHB2, we have assumed the network topology described in Fig. 4).

4 Incremental matrix clock

In this section, we describe an efficient incremental algorithm to compute the matrix clock on the fly, based on Elnozahy and Zwaenepoel's *antecedence graph* algorithm [EZ92, EZ93]. Let us first recall Elnozahy and Zwaepenoel's work, which itself builds on [SY85], [JZ87] and [SBY88].

The framework is optimistic recovery from failures. Elnozahy and Zwaenepoel consider system computations that consist of a number of recovery units (RU's) [SY85] which communicate only by messages over an asynchronous network. The execution of an RU consists of a sequence of piecewise deterministic state intervals, each started by a non deterministic event (such as the receipt of a message). They denote by σ_i^p the ith state interval of RU p. They define the *antecedence graph* (AG) of a state interval σ_i^p as the set of all state intervals that "happened before" σ_i^p [Lam78].

Elnozahy and Zwaenepoel propose an algorithm to compute the AG of the current state interval of each RU on the fly. This algorithm is based on (conceptually) piggy-backing the AG on outgoing messages: when an RU sends a message, it (conceptually) piggy-backs the AG of its current state interval on the message. The receipt of the message starts a new sate

connected to every other site in the area. This is illustrated in Fig. 4.

interval in the receiving RU, and the AG of that state interval is constructed from the AG of the previous state interval and the AG piggy-backed on the message.

The algorithm is incremental because the whole AG is not actually piggy-backed on the outgoing message: instead, the sending RU piggy-backs only those parts of the AG for which it does not know if they have already been stored at the receiving RU.

We transform Elnozahy and Zwaenepoel's algorithm in the following manner. Firstly, we translate RU's into a sites. Secondly, we translate states intervals into events [5]. With these translations, we obtain an incremental algorithm to compute the AG of an event, i.e. the set of all events that have "happened before" this event.

It is clear that the genuine matrix clock can be deduced from the AG, because the AG contains all the events that could possibly be involved in the computation of the matrix clock.

The paradox is that it might be cheaper to maintain the AG than to directly maintain the $n \times n$-dimensional matrix clock. This is possible because **(1)** the AG can be maintained incrementally as described in [EZ92, EZ93] and because **(2)** it is not necessary to store the whole AG: a garbage collecting algorithm along the lines of [WB84] or [SL87] can be used to discard "obsolete" events from the AG.

4.1 The algorithm

We now describe the incremental algorithm for computing matrix clock on the fly. We assume that every event is tagged with **(1)** the identifier of the site to which it belongs, and **(2)** its sequence number on that site. We let e_i^l denote the l-th. event produced on site S_i. We also let ϵ_i denote the current event of site S_i (i.e. the last event that was produced so far on S_i).

Each site S_i maintains a graph AG_i. AG_i is initially empty.

Throughout the execution of the algorithm, AG_i will be a subgraph of the antecedence graph $\mathrm{AG}(\epsilon_i)$ of the current event of S_i. The algorithm enforces an additional constraint on AG_i. For all j and k, consider the last event e of S_k that causally precedes the last event of S_j that causally precedes ϵ_i. Then, either e does not exist, or e belongs to AG_i. Formally, the algorithm enforces the (AG) constraint:

$$\forall j, k, \max(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\}) \in \mathrm{AG}_i \tag{AG}$$

The (AG) constraint is necessary to guarantee that the matrix clock can be computed from the events in AG_i. Indeed, recall the definition of the matrix clock given in Sect. 2.1:

$$\delta_{\mathrm{mat}}(\epsilon_i)[j,k] = \mathrm{card}(\downarrow_{E_k}\downarrow_{E_j} \{\epsilon_i\})$$

[5] We identify a state interval with the event that has lead to that state interval

assuming the (AG) constraint, this definition is equivalent to [6]:

$$\delta_{\mathrm{mat}}(\epsilon_i)[j,k] = \text{seq-num}\Big(\max\downarrow_{E_k\cap \mathrm{AG}_i}\Big\{\max\downarrow_{E_j\cap \mathrm{AG}_i}\{\epsilon_i\}\Big\}\Big)$$

Thus, the matrix clock can indeed be computed from AG_i.

The algorithm follows. Each site S_i piggybacks its AG_i on outgoing messages. Rules (INT-I), (MSG-I) and (GC-I) are applied:

INT-I: Before S_i performs an event e_l^i:

$$\mathrm{AG}_i \leftarrow \mathrm{AG}_i \cup \{e_l^i\}$$

MSG-I: Before S_i receives message (m, AG) from S_j:

$$\mathrm{AG}_i \leftarrow \mathrm{AG}_i \cup \mathrm{AG}$$

GC-I: At any time, S_i may remove "obsolete" events from its AG_i:

$$\begin{aligned} \forall j,k,\ M_i[j,k] &\leftarrow \text{seq-num}\Big(\max\downarrow_{E_k\cap \mathrm{AG}_i}\Big\{\max\downarrow_{E_j\cap \mathrm{AG}_i}\{\epsilon_i\}\Big\}\Big) \\ \mathrm{AG}_i &\leftarrow \mathrm{AG}_i - \Big\{e_l^j \in \mathrm{AG}_i \,|\, \forall k, M_i[k,j] > l\Big\} \end{aligned}$$

Rule (MSG-I) merges the local AG with the AG piggybacked on the received message, as explained in [EZ92, EZ93]. Rule (GC-I) does the garbage collection along the lines of [WB84, SL87]. It is possible to prove that these rules actually preserve the (AG) constraint (Cf. appendix A). The algorithm is illustrated in Fig. 6, as strategy "INCR".

4.2 Performance of the incremental algorithm

The algorithm achieves a small overhead if the sites of the computation are well synchronized. Indeed, in that case, thanks to the garbage collection mechanism, the size of the stored and piggybacked AG_i's will stay small: typically $\mathcal{O}(n)$ (see the example below). Thus both the storage and communication overheads will be of size $\mathcal{O}(n)$. This compares favorably with the overhead of size $\mathcal{O}(n^2)$ required by the naive matrix clock algorithm. However, there is a price to pay:

[6] In the formula, seq-num returns the sequence number of the considered maximum event, or 0 if this event does not exist.

- Firstly, to achieve a storage overhead of size $\mathcal{O}(n)$, the incremental algorithm does not maintain the matrix clock directly, instead it maintains the AG_i. For that reason, some computation is necessary to produce the actual components of the matrix clock. This result in additional time overhead.
- Secondly, if the computation is not well synchronized, the size of AG_i may increase without bound. To overcome this problem, a practical solution is to synchronize the computation by having gossip messages [HHW89] regularly visit all sites of the computation, in a ring or spanning tree fashion. This will keep the size of AG_i small, but at the expense of additional control messages.

We now give an example of a "well synchronized computation". The computation consists of n sites that communicate via a token ring network, as illustrated in Fig. 5. In the (somewhat difficult to read) figure, we have indicated the AG_i's corresponding to each receipt event. We can see that the AG_i's are of size $3n + 3$ ($2n + 2$ nodes and $n + 1$ edges [7]). In other words, in this example, the storage and communication overheads of the incremental algorithm are indeed of size $\mathcal{O}(n)$.

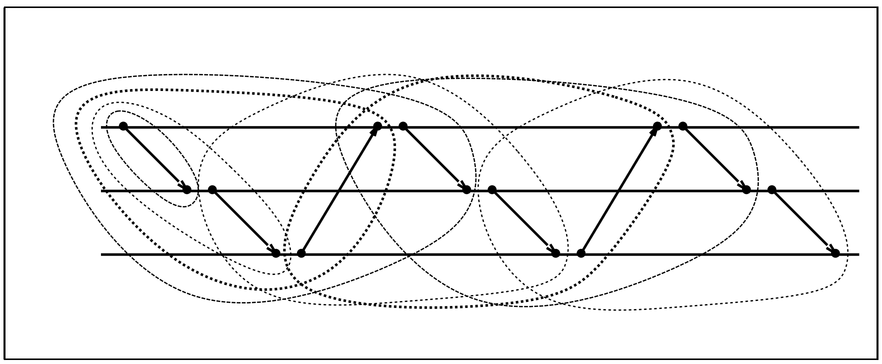

Figure 5: Size of AG_i

5 Another approximation to the matrix clock

In this section, we introduce our *k-matrix clock* approximation to the genuine matrix clock. We propose an algorithm to compute the k-matrix clock on the fly with storage and communication overhead of size $\mathcal{O}(kn)$. This bound does not depend on the relative synchronization between the sites of the computation. We describe possible applications of the k-matrix clocks in Sect. 5.4.

We first introduce the concept of k-approximation to a vector or a matrix.

[7] It is necessary to take the number of edges into account, because in general, a graph with n nodes may have up to n^2 edges.

5.1 k-approximation to a vector

Definition 1 *Let $\mathbb{N}$ denote the set of non negative integers and consider two n-dimensional vectors of integers $a, b \in \mathbb{N}^n$. Consider an integer $k \le n$. We say that b is a k-approximation of a and we denote $b \preceq_k a$, the fact that b contains the k greatest components in a. Formally:*

$$b \preceq_k a \stackrel{\text{def}}{=} \exists I \subseteq \{1, .., n\}, \begin{cases} \text{card}(I) = k & \\ \forall i \notin I, b_i \le a_i & (\text{I}\le) \\ \forall i \in I, b_i = a_i & (\text{I}=) \\ \forall i \notin I, \forall j \in I, a_i \le a_j & (\text{I}) \end{cases}$$

Intuitively, (I) says that I contains the indexes of the k greatest components of a. (I=) says that for each index in I, the corresponding components in a and b are equal. (I$\le$) says that for each index not in I, the corresponding component in a is greater than the component in b.

For example:

$$\begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \preceq_2 \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}; \begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \preceq_1 \begin{bmatrix} 0 \\ 6 \\ 6 \end{bmatrix}; \begin{bmatrix} 0 \\ 4 \\ 5 \end{bmatrix} \not\preceq_1 \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}$$

Whereas an n-dimensional vector is of size $\mathcal{O}(n)$, it is always possible to find a k-approximation of the vector of size only $\mathcal{O}(k)$ (by keeping only the k greatest components and setting the other components to 0).

Proposition 2 *The set of n-dimensional vectors of integers is partially ordered by $\preceq_k$ ($k \le n$).*

The proof of the proposition is given in appendix B.

Proposition 3 *Let* max *be the operator that takes the component-wise maximum of two vectors.* max *and $\preceq_k$ "commute".*

More precisely, let A and B be two n-dimensional vectors of integers. Let M be the component-wise maximum of A and B. Consider a and b, two k-approximations of A and B, and m, the component-wise maximum of a and b. Then m is a k-approximation of M. Formally:

$$
\forall A, B, M, a, b, m \in \mathbb{N}^n,
$$
$$
\left\{\begin{array}{l} M = \max(A, B) \\ a \preceq_k A \\ b \preceq_k B \\ m = \max(a, b) \end{array}\right. \implies m \preceq_k M
$$

The proposition is demonstrated in appendix B. It is illustrated by the following example.

$$
\begin{array}{cccccccccc}
\begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix}, & \begin{bmatrix} 2 \\ 7 \\ 3 \end{bmatrix} & \stackrel{\max}{\rightarrow} & \begin{bmatrix} 2 \\ 7 \\ 6 \end{bmatrix} & \qquad & \begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} & \begin{bmatrix} 4 \\ 7 \\ 2 \end{bmatrix} & \stackrel{\max}{\rightarrow} & \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} \\
\downarrow\preceq_2 & \downarrow\preceq_2 & & \downarrow\preceq_2 & & \downarrow\preceq_2 & \downarrow\preceq_2 & & \downarrow\preceq_2 \\
\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, & \begin{bmatrix} 4 \\ 7 \\ 3 \end{bmatrix} & \stackrel{\max}{\rightarrow} & \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} & \qquad & \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} & \begin{bmatrix} 4 \\ 7 \\ 3 \end{bmatrix} & \stackrel{\max}{\rightarrow} & \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix}
\end{array}
$$

5.2 k-approximation to a matrix

Definition 4 *Let A and B be two $n \times n$ matrices of integers. We say that B is a k-approximation of A (denoted $B \preceq_k A$) when the columns of B are k-approximations of the columns of A. Formally* [8]*:*

$$
\forall A, B \in \mathbb{N}^{n \times n},
$$
$$
B \preceq_k A \stackrel{\text{def}}{=} \forall 1 \le j \le n, B[\star, j] \preceq_k A[\star, j]
$$

For example:

$$
\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 2 & 0 & 3 \end{bmatrix} \preceq_2 \begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 0 & 3 \end{bmatrix}
$$
$$
\begin{bmatrix} 5 & 3 & 3 \\ 0 & 5 & 0 \\ 5 & 0 & 6 \end{bmatrix} \preceq_2 \begin{bmatrix} 5 & 3 & 3 \\ 4 & 5 & 3 \\ 5 & 3 & 6 \end{bmatrix}
$$

Whereas an $n \times n$-dimensional matrix is of size $\mathcal{O}(n^2)$, it is always possible to find a k-approximation of the matrix of size only $\mathcal{O}(kn)$.

[8] In the formula, $A[\star, j]$ denotes the jth. column of matrix A.

5.3 k-matrix clock

Definition 5 *A* [9] *k-matrix clock (denoted δ_k) is such that the values it returns are k-approximations of the genuine matrix times. Formally:*

$$\delta_k : E \rightarrow \mathbb{N}^n$$
$$\forall e \in E,\ \delta_k(e) \preceq_k \delta_{\mathrm{mat}}(e)$$

We now give an algorithm to efficiently compute a k-matrix clock on the fly.

Each site S_i maintains its view $A_i \in \mathbb{N}^{n \times n}$ of the k-matrix clock. A_i is initially set to zero. A_i is tagged onto all outgoing messages. Rules (INT-A) and (MSG-A) are applied:

INT-A: Before S_i performs an event:

$$A_i[i,i] \ \leftarrow\ A_i[i,i] + 1$$

MSG-A: Before S_i receives message (m, A) [10]:

$$\forall l, A_i[i,l] \ \leftarrow\ \max(A_i[i,l], A[j,l])$$
$$\forall k, l, A_i[k,l] \ \leftarrow\ \max(A_i[k,l], A[k,l])$$
$$A_i \ \leftarrow\ \mathrm{apr}(A_i), \text{ such that } \mathrm{apr}(A_i) \preceq_k A_i$$

Figure 6 gives an example of 2-matrix clock (strategy "2-MATRIX" in the figure).

Proposition 6 *The k-matrix clock A produced by the algorithm above is indeed a k-approximation to the matrix clock M. Formally, let $M(e)$ (resp. $A(e)$) denote the value produced just after occurrence of event e by the genuine matrix clock algorithm (resp. by the k-matrix clock algorithm above), then:*

$$\forall e \in E, A(e) \preceq_k M(e)$$

The proof is based on propositions 2 and 3. It is given in appendix B.

[9] We write "A matrix clock" intentionally: there are many possible k-matrix clocks.

[10] (m, A) is the received message, tagged with A, the sending site's view of the approximated matrix clock at sending time.

5.4 Applications of the k-matrix clock

Recall that intuitively, the genuine matrix clock M_i is such that component $M_i[j,k]$ represents site S_i's view of S_j's view of S_k's progress. The k-matrix clock A_i contains the k greatest components of M_i. In other words, A_i gives site S_i's view of the k most up-to-date views of S_k's progress.

A possible application of the k-matrix clock A_i is in the field of optimistic fault-tolerance. A typical optimistic fault-tolerant mechanism will for example guarantee that each recovery unit [SY85] piggy-backs on outgoing messages the part of its antecedence graph [EZ92, EZ93] for which it does not know whether it has already been stored by a quorum of other recovery units. Let us assume that the system obeys a crash failure semantics [CASD85] such that the maximum number of simultaneous site failures is bounded by $k-1$. Then it is sufficient for each recovery unit to piggy-back on outgoing messages the part of its antecedence graphs for which it does not know whether it has already been stored by k processes (including itself). For this purpose, the whole matrix clock is not necessary, the k-approximation suffices.

Another (similar) application is the implementation of a stable event log facility for a system with crash failure semantics such that at most $k-1$ faults may occur simultaneously. For example, we plan to use k-matrix clocks to implement a crash resilient event logging facility for the CDB distributed debugger [Rug94].

There is a similarity between k-matrix clocks and k-bounded ignorance [KB91]. However, whereas k-bounded ignorance is a distributed database technique to guarantee that a given transaction cannot be ignorant of more than k (causally) preceding transactions, k-matrix clocks guarantee that no more than k computation sites can be ignorant of a (causally) preceding event.

5.5 Clock condition for the approximated matrix clock

It is possible to define an order on $\mathbb{N}^{n\times n}$ such that the k-matrix clock satisfies the strong (S-CLK) condition. This is explained in appendix C.

6 Conclusion

In this paper, we have given an efficient incremental algorithm to compute the matrix clock of an asynchronous distributed system on the fly. This algorithm is derived from MANETHO's antecedence graph algorithm [EZ92, EZ93]. In the optimal case where the sites are "well synchronized", our algorithm achieves storage and communication overhead of size $\mathcal{O}(n)$, where n is the number of sites of the system. However, in the general case, the storage and communication overheads are not bounded. It is thus necessary to join a gossip mechanism to the algorithm, to make sure that the sites are kept relatively well synchronized.

A second contribution of this paper is the definition of the k-matrix clock: an approximation to the genuine matrix clock. The k-matrix clock is conceptually obtained by keeping only the k greatest entries in each column of the genuine matrix clock. Like the genuine matrix clock, it gives an exact representation of the causality order ((S-CLK) condition).

"Intuitively", the k-matrix clock gives the local site's view of the k most up-to-date site views of the progress of every site. It may thus be used to discard obsolete information in a system where information known by at least k sites is obsolete. A possible application is in the field of optimistic fault tolerance, to implement a stable event log for a system with crash semantics such that the maximum number of simultaneous faults is bounded by $k - 1$.

We have proposed an algorithm to compute the k-matrix clock on the fly with storage and communication overheads of size $\mathcal{O}(kn)$. This bound does not depend on the relative synchronization between the sites of the computation.

We plan to use either of the above algorithms to implement a crash resilient event logging facility for CHORUS's CDB distributed debugger.

References

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-tolerant Computing*, June 1985.

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast ouput commit. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[EZ93] E. N. Elnozahy and W. Zwaenepoel. Fault tolerance for a workstation cluster. In *Proc. of the Workshop on Hardware and Software Architectures for Fault Tolerance*, May 1993.

[Fid91] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, Aug. 1991.

[FM82] M. J. Fischer and A. Michael. Sacrifying serializability to attain high availability of data in an unreliable network. In *Proc. ACM SIGACT-SIGMOD Symp. on principles of database systems*, pages 70–75, Los Angeles, March 1982.

[HHW89] A. Heddaya, M. Hsu, and W. E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1):35–57, 1989.

[JZ87] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *The 17th Int. Symp. on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, June 1987.

[KB91] Krishnakumar and Bernstein. Bounded ignorance in replicated systems. In *Proc. ACM symp. on Principles of Database Systems*, 1991.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LL86] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proc. 5th ACM Symp. on PODC*, pages 29–39, 1986.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of Int. Workshop on Parallel and Distributed Algorithms, Bonas (France)*, pages 215–226. Cosnard, Quinton, Raynal and Robert editors, 1989.

[PT92] P. Panangaden and K. Taylor. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–94, September 1992.

[Rug94] F. Ruget. A distributed execution replay facility for CHORUS. In *Proc. of the 7th Int. Conf. on Parallel and Distributed Systems (PDCS'94)*, Las Vegas, Nevada, October 1994.

[SBY88] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49, June 1988.

[SL87] S. K. Sarin and L. Lynch. Discarding obsolete information in a replicated database system. *IEEE Trans. on Soft. Eng.*, SE 13(1):39–46, Jan. 1987.

[SY85] R. E. Strom and S. A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on computer Systems*, 3(3):204–226, August 1985.

[WB84] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proc. 3rd ACM Symp. on PODC*, pages 232–242, 1984.

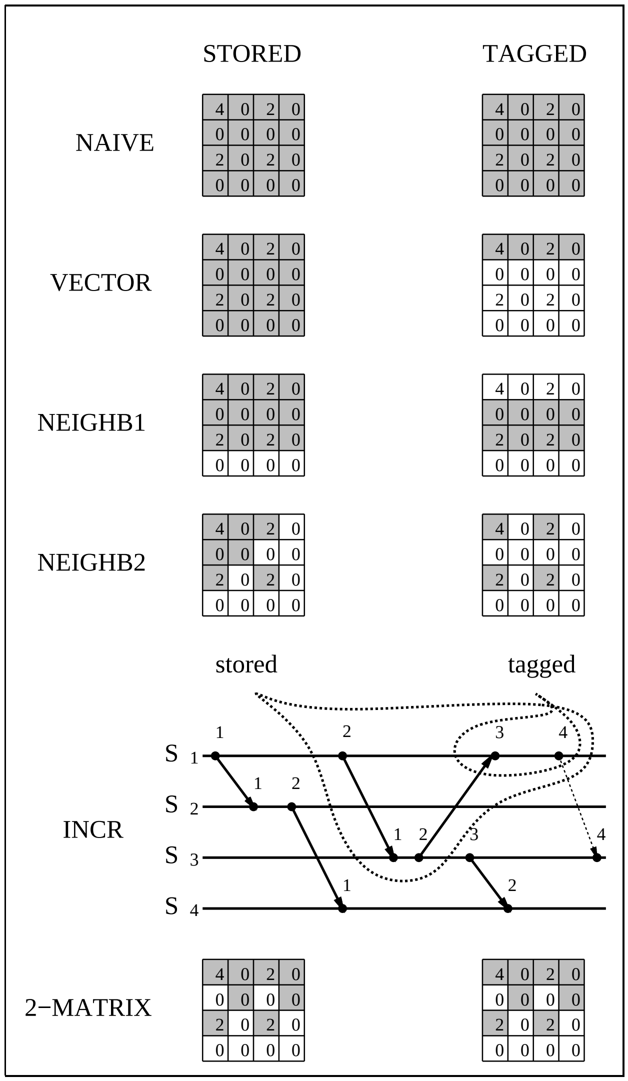

Figure 6: Approximation strategies

Appendix

The appendix gives the mathematical proofs promised in the body of the paper.

A Incremental matrix clock algorithm

Proposition *Rules (INT-I), (MSG-I) and (GC-I) preserve the (AG) constraint.*

Proof: First recall the (AG) constraint on the subgraph AG_i maintained by site S_i:

$$\forall j, k, \max(\downarrow_{E_k}\downarrow_{E_j}\{\epsilon_i\}) \in \mathrm{AG}_i \tag{AG}$$

where ϵ_i is the last event that occurred on S_i and $\mathrm{AG}(\epsilon_i)$ is the antecedence graph of ϵ_i.

It is easy to see that the (AG) constraint is preserved by rules (INT-I) and (MSG-I) defined in Sect. 4.1. Let us show that it is also preserved by the (GC-I) rule. Let us denote by AG_i and AG'_i the values of the AG_i before and after the garbage collection by rule (GC-I). We have:

$$\mathrm{AG}'_i = \mathrm{AG}_i - \left\{ e^j_l \in \mathrm{AG}_i \middle| \, \forall k, M_i[k,j] > l \right\}$$

We assume that AG_i satisfies constraint (AG). We want to prove that AG'_i also does. Consider any j and k, and event $e^k_l = \max(\downarrow_{E_k}\downarrow_{E_j}\{\epsilon_i\}) \in \mathrm{AG}_i$. e^k_l is the lth event of site S_k. By definition of the matrix clock, $M_i[j,k] = \mathrm{card}(\downarrow_{E_k}\downarrow_{E_j}\{\epsilon_i\}) = l$. Thus, $M_i[j,k] \not> l$ and as a consequence, the garbage collection mechanism does not remove e^k_l from AG_i. In other words: $e^k_l = \max(\downarrow_{E_k}\downarrow_{E_j}\{\epsilon_i\}) \in \mathrm{AG}'_i$. □

B k-approximations

Recall the formal definition of the k-approximation operator $\preceq_k$:

$$b \preceq_k a \stackrel{\mathrm{def}}{=} \exists I \subseteq \{1,..,n\}, \left\{ \begin{array}{cr} \mathrm{card}(I) = k & \\ \forall i \notin I, b_i \leq a_i & (\mathrm{I}\leq) \\ \forall i \in I, b_i = a_i & (\mathrm{I}=) \\ \forall i \notin I, \forall j \in I, a_i \leq a_j & (\mathrm{I}) \end{array} \right.$$

Proposition 2 *The set of n-dimensional vectors of integers is partially ordered by $\preceq_k$ ($k \leq n$).*

Proof: $\preceq_k$ is obviously reflexive and antisymmetric. We prove that it is also transitive (this is illustrated by Fig. 7). Consider $c \preceq_k b \preceq_k a$. The definition of $\preceq_k$ tells us that there exist two subsets $S, T \subseteq \{1, .., n\}$ with cardinal k such that:

$$\forall i \notin S, c_i \leq b_i \qquad (S\leq)$$
$$\forall i \in S, c_i = b_i \qquad (S=)$$
$$\forall i \notin S, \forall j \in S, b_i \leq b_j \qquad (S)$$
$$\forall i \notin T, b_i \leq a_i \qquad (T\leq)$$
$$\forall i \in T, b_i = a_i \qquad (T=)$$
$$\forall i \notin T, \forall j \in T, a_i \leq a_j \qquad (T)$$

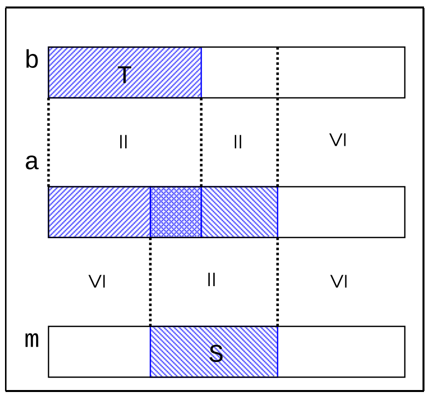

Figure 7: $\preceq_k$ is transitive.

From $(S\leq)$, $(T\leq)$ and $(T=)$ it follows that

$$\forall i \notin S, c_i \leq a_i \qquad (U\leq)$$

Now consider $i \in S$. We prove by contradiction that $c_i = a_i$. So assume that $c_i \neq a_i$. From $(S=)$, $(T\leq)$ and $(T=)$ it follows that $i \notin T$ and $b_i < a_i$. Together with $(T=)$ and (T), this yields $\forall j \in T, b_i < b_j$. This contradicts (S). Hence

$$\forall i \in S, c_i = b_i = a_i \qquad (U=)$$

Consider $j \in S$. One of the following is true. Either $S = T$ which implies $\exists i \in T, b_i \leq b_j$, or $S \neq T$ which also implies $\exists i \in T, b_i \leq b_j$ because of (S). With $(U=)$ and $(T=)$ this yields:

$$\forall j \in S, \exists i \in T, a_i \leq a_j \qquad (1)$$

We now prove:

$$\forall i \notin S, \forall j \in S, a_i \leq a_j \qquad (U)$$

Consider $i \notin S$ and $j \in S$. Either **(a)** $i \in T$, then $a_i = b_i$ (from (T=)), $b_i \leq b_j$ (from (S)), $b_j = a_j$ (from (U=)). Or **(b)** $i \notin T$, then (1) gives us a $j' \in T$ such that $a_{j'} \leq a_j$, and (from (T)) $a_i \leq a_{j'}$. In both cases (U) is verified.

Finally (U$\leq$),(U=) and (U) prove that $c \preceq_k a$. $\square$

Proposition 3 *Let* max *be the operator that takes the component-wise maximum of two vectors.* max *and* $\preceq_k$ *"commute".*

More precisely, let A and B be two n-dimensional vectors of integers. Let M be the component-wise maximum of A and B. Consider a and b, two k-approximations of A and B. Let m be the component-wise maximum of a and b. Then m is a k-approximation of M. Formally:

$$\forall A, B, M, a, b, m \in \mathbb{N}^n, \quad \left\{ \begin{array}{l} M = \max(A, B) \\ a \preceq_k A \\ b \preceq_k B \\ m = \max(a, b) \end{array} \right. \Longrightarrow m \preceq_k M$$

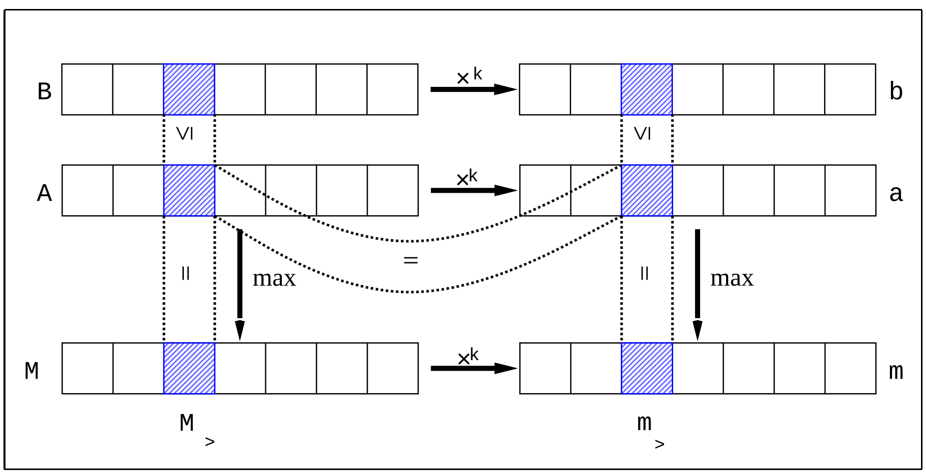

Figure 8: max and $\preceq_k$ "commute".

Proof: The proposition follows by induction on n (this is illustrated by Fig. 8 [11]). The proposition is true for $n = 1$. Suppose the proposition is true for $(n-1)$. Let us prove it is true for n. Consider $k \leq n$ and A, B, M, a, b, m as defined in the proposition. The case $k = 0$ is easy. If $k \neq 0$, let $M_\top = m_\top$ be a greatest component of M that is also in m. Remove component $\top$ from vectors A, B, M, a, b and m to obtain $(n-1)$-dimensional vectors A',

[11] In Fig. 8, we have assumed that $A_\top = M_\top = a_\top = m_\top$

B', M', a', b' and m'. These vectors satisfy the conditions of the proposition for $(n-1)$ and $(k-1)$, thus $m' \preceq_{(k-1)} M'$. It follows that $m \preceq_k M$. □

Proposition 6 *The approximated matrix clock A is indeed a k-approximation to the matrix clock M:*

$$\forall e \in E, A(e) \preceq_k M(e)$$

Proof: We do not consider the case $k = 0$ which is obvious. So assume that $k > 0$. The proof is by induction on the cardinal of $\downarrow_E \{e\}$ [12]. If card$(\downarrow_E \{e\}) = 0$, then both $A(e)$ and $M(e)$ are filled with zeroes and the proposition is true. Assume that the proposition is true for all e such that card$(\downarrow_E \{e\}) < x$. Consider an event e on site S_i, such that card$(\downarrow_E \{e\}) = x$.

(a) If e is not a message receipt, let p denote e's immediate predecessor event (on site S_i). By induction hypothesis, $A(p) \preceq_k M(p)$. Note that $M(p)[i,i]$ is strictly greater than any $M(p)[h,i]$. Since $A(p) \preceq_k M(p)$ and $k > 0$, we must then have $A(p)[i,i] = M(p)[i,i]$. Hence $A(e) \preceq_k M(e)$ (because $A(e)$ and $M(e)$ are computed by adding one to component $[i,i]$ of $A(p)$ and $M(p)$ respectively).

(b) If e is the receipt of message m, let p denote e's immediate predecessor event on site S_i and let event s be the sending of m, say by site S_j. By induction hypothesis, $A(p) \preceq_k M(p)$ and $A(s) \preceq_k M(s)$. Let us denote A_1 the matrix obtained by adding one to component $[i,i]$ of A(p), and A_2 the matrix obtained by taking the component wise maximum of A_1 and $A(s)$. Let us define M_1 and M_2 in the same way. Again, $A(p)[i,i] = M(p)[i,i]$ is strictly greater than any $M(p)[h,i]$, and consequently, $A_1 \preceq_k M_1$. Successive applications of proposition 3 ($\preceq_k$ and max "commute") guarantee that $A_2 \preceq_k M_2$. Finally, $A(e) \preceq_k M(e)$ because $A(e)$ and $M(e)$ are obtained by replacing component i of each column of A_2 and M_2 by the maximum of components i and j of the column.

To show this last point, consider two n-dimensional vectors a and b such that $b \preceq_k a$. Consider A and B obtained by replacing component i of each column of a and b by the maximum of components i and j of the column. We can assume without loss of generality that $i = 1$ and $j = 2$. There are four possible cases, depending on the position of the set S of indexes of the k greatest components of a, relatively to i and j. The four cases are illustrated in Fig. 9 (in the figure, the set S is indicated with a hashed area). The reader may check that in all cases: $B \preceq_k A$. □

C k-approximations and strong clock condition

In this section, we assume that $k > 0$. Let us first note that the component-wise order on $n \times n$-dimensional matrices is not adequate to compare approximated matrix clocks. Indeed, the clock conditions (CLK) or (S-CLK) (see Sect. 2) are not satisfied with this order. To show

[12] $\downarrow_E \{e\}$ is the predecessor set of e in the set E of all events of the computation.

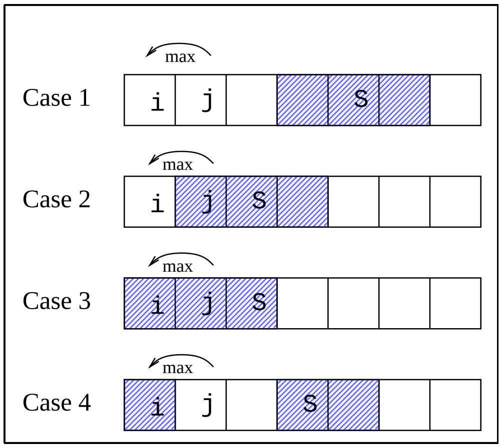

Figure 9: Relative position of i, j, and S.

this, consider events e_2 and e_3 and their 1-approximated matrix clocks $M(e_2)$ and $M(e_3)$ given in Fig. 10. We can check that $M(e_2) \not\leq M(e_3)$, in contradiction with (CLK) and (S-CLK). Thus the question is to know whether there exists an order on $n \times n$-dimensional matrices such that (CLK) or preferably (S-CLK) is satisfied. This is the purpose of the remainder of this section.

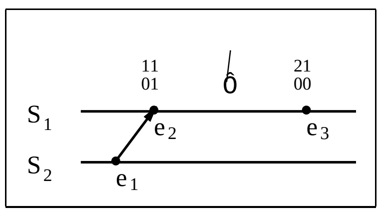

Figure 10: Component-wise order is not adequate

Definition 7 *Consider $a, b \in \mathbb{N}^n$, two n-dimensional vectors of integers. We define the $\leq_k$ relationship as follows.*

$$b \leq_k a \stackrel{\text{def}}{=} \exists i_1, j_1, i_2, j_2, \ldots, i_n, j_n, \begin{cases} a[i_1] \geq a[i_2] \geq \ldots \geq a[i_n] \\ b[j_1] \geq b[j_2] \geq \ldots \geq b[j_n] \\ \forall l \leq k, b[i_l] \leq a[i_l] \end{cases}$$

Informally, $b \leq_k a$ if and only if the k greatest components of a are greater than the k greatest components of b. We say that a is k-greater than b, or equivalently b is k-lower than a.

For example:

$$\begin{bmatrix} 0 \\ 5 \\ 6 \end{bmatrix} \leq_2 \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} ; \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix} \leq_2 \begin{bmatrix} 6 \\ 6 \\ 0 \end{bmatrix} ; \begin{bmatrix} 0 \\ 4 \\ 5 \end{bmatrix} \not\leq_2 \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix}$$

Of course, if $a \preceq_k b$ then $a \leq_k b$

Definition 8 *Consider $A, B \in \mathbb{N}^{n\times n}$, two $n\times n$-dimensional matrices of integers. We denote $B \leq_k A$ the fact that each column in B is k-lower than the corresponding column in A. Formally* [13]*:*

$$\forall A, B \in \mathbb{N}^{n\times n},$$
$$B \leq_k A \stackrel{\text{def}}{=} \forall 1 \leq j \leq n, B[\star, j] \leq_k A[\star, j]$$

For example:

$$\begin{bmatrix} 5 & 3 & 3 \\ 2 & 5 & 0 \\ 4 & 0 & 6 \end{bmatrix} \leq_2 \begin{bmatrix} 5 & 3 & 3 \\ 1 & 5 & 3 \\ 5 & 3 & 6 \end{bmatrix}$$

Proposition 9 *If the set of $n \times n$-dimensional matrices is endowed with the $\leq_k$ relationship, then the k-matrix clock A satisfies the strong (S-CLK) condition:*

$$\forall e_1, e_2 \in E, e_1 \leq e_2 \iff A(e_1) \leq_k A(e_2)$$

Proof: $\Longrightarrow$ is easy, we now prove $\Longleftarrow$.

Consider two events of a computation of the system: $e^i_l \in E_i$ and $e^j_m \in E_j$ such that $A(e^i_l) \leq_k A(e^j_m)$. We show that $e^i_l \leq e^j_m$.

The case $i = j$ is easy. So assume $i \neq j$. First note that for any event $e \in E$, the greatest component of the cth column of the k-approximated matrix clock $A(e)$ is exactly $M(e)[c, c]$ (this is because we assume that $k > 0$). Now only one of the following three cases is possible: **(a)** $e^i_l > e^j_l$: in that case, $M(e^i_l)[j, j] > M(e^j_m)[j, j]$. This contradicts $A(e^i_l) \leq_k A(e^j_m)$ (consider column j). **(b)** e^i_l and e^j_l are concurrent: in that case, $M(e^i_l)[j, j] < M(e^j_m)[j, j]$ and $M(e^i_l)[i, i] > M(e^j_m)[i, i]$. This contradicts $A(e^i_l) \leq_k A(e^j_m)$ twice (consider columns i and j). So it only remains case **(c)** $e^i_l < e^j_l$. $\square$

The reader can check that the computation of Fig. 10 actually satisfies (S-CLK) when $\mathbb{N}^{n\times n}$ is endowed with $\leq_1$.

[13] In the formula, $A[\star, j]$ denotes the jth. column of matrix A.

Of course, $\leq_k$ is not a partial order on $\mathbb{N}^{n \times n}$. For example:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \leq_1 \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \leq_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

In fact, $\leq_k$ only satisfies reflexivity and transitivity, which makes it a pre-order. However, since condition (S-CLK) is satisfied, $\leq_k$ is actually a partial order on the subset of $\mathbb{N}^{n \times n}$ that consists of the clock values obtained during an actual computation of the system.

To summary: if the set of clock values obtained during an actual computation of the system is partially ordered by the $\leq_k$ relationship, then the k-approximated matrix clock satisfies the strong (S-CLK) condition.