Information-Provenance Clocks

Dan Fu and Ross Rheingans-Yoo
Harvard School of Engineering and Applied Sciences
dfu@college.harvard.edu || rry@eecs.harvard.edu

***Abstract*—The study of time, clocks, and causal orderings of events in distributed systems has been a subject of study within the field of computer science for almost forty years, in at least some reference frame. Nevertheless, relatively little work has been done to investigate the practical considerations of designing a distributed ordering system with the goal of enforcing a causally consistent ordering of events. We formalize the closely-linked concepts of *ordering schema* and *order-control policy*. We present an extension of Lamport's venerable *vector-clock* ordering schema, designed to reduce the frequency with which spurious information is integrated into clock vectors. We demonstrate that in certain systems that use such vector clocks as the basis for their order-control policy, this modification drastically reduces the incidence of false-positive identifications of mis-ordered events.**

I. Introduction

One fundamental capability often desired in distributed systems is *ordering*, the ability to specify that certain events 'occurred before' or 'occurred after' other events. However, Lamport demonstrated that it is impossible in general to found such specification in objective, physical terms [1], requiring that ordering schemata must in general be partial orderings specified logically, *within* a given distributed system. For this reason, we three possible relations between two events a, b:

- $a < b$: a **must precede** b; b **must succeed** a.
- $a \gtrless b$: a **may precede** b; a **may succeed** b; b **may preceded** a; b **may succeed** a.
- $a < b$: a **must succeed** b; b **must precede** a.

NB: When presented in boldface, these expressions refer to the relations of 'must' and 'may' with respect to a particular ordering schema. To instead refer to causal constraints on which events 'must' or 'may' come before others, we qualify the relation as 'causally may precede' and 'causally must precede', without boldface.

We define a *consistent ordering schema* to be one in which "a causally must precede b" $\implies$ "a **must precede** b. We may also desire that in most cases, the converse holds, though as we will see below, this is difficult to achieve in general in practice.

A. Logical Clocks

Lamport [1] introduced *logical clocks* as an ordering schema which assigns integer numbers to events, such that:

- any two events which occur within the same process are assigned distinct numbers, with the higher number assigned to the later-occurring event
- the sending and receipt of a message, respectively, are assigned distinct numbers, with the higher number assigned to the receipt event.

These conditions are achieved by specifying that each process maintains a *logical clock counter* which is incremented on every 'internal event' occurring local to the process. (Such an event is then assigned the new value as its *timestamp*.) Furthermore, on the receipt of any message sent with timestamp T_m (according to the sender's clock), this counter is advanced from C_j to $\max[C_j+1, T_m+1]$, assigning the new value as the timestamp of the receipt event. In this schema, we say that an event a **must precede** an event b iff the timestamp of a is less than b.

However, logical clocks have two significant disadvantages. First, integer timestamps present an *over-specified* model of the "happened before" partial ordering: it may assign a **must precede** relation to events which causally may have occurred in either order.

Second, logical clocks often exhibit gaps between the timestamps of subsequent events in a process. This makes it impossible in general to use timestamps to determine on a process-local level whether a remote event (which it may later receive) has

occurred which **must precede** an event it intends to process locally.

Together, these limitations mean that logical clocks provide no more than a rough ordering of *one possible sequence* of events, which gives no insight that can drive a consistent ordering policy from within the system.

B. Vector Clocks

One schema introduced to address the limitations of over-specification and spurious counter gaps is *vector clocks*, which assign to events a *vector* of logical-clock values, each component corresponding to a single process. Each process, then, maintains a vector, incrementing its own component on internal events, and advancing other components as necessary so that, for any message received with timestamp $\vec{T}$ by a process j with clock $\vec{C}$, the clock-vector becomes C' with:

$$\forall i, C'_i \geq \max[C_i, T_i] \quad (1)$$
$$C'_j = C_j + 1 \quad (2)$$

In this schema, an event a **must precede** an event b iff a 'dominates componentwise' b, *i.e.*:

- $\forall i, a_i \leq b_i$
- $\exists j : a_j < b_j$.

Note that, in any possible run, no process will advance the component corresponding to another process beyond the value of that process's component in its own clock-vector. It follows that any events with timestamps with identical values in a process i's component causally may have occurred without any intervening events from process i.

Additionally, any event b with a timestamp dominating another's a causally must have occurred on a machine that had received a message from (a machine that had received a message from...) the machine that produced event a. Under the assumption that any message a machine receives causally affects subsequent messages, a will thus precede b in any run of the system, and we specify that a **must precede** b. Correspondingly, any pair a, b which is *incomparable—i.e.*, neither dominates the other—may have occurred in either order (in at least some run of the system), and we specify that a **may precede** b.

II. VECTOR-BASED ORDER-CONTROL POLICY

We consider *ordering schemata* separately from *ordering policy*. The former, as discussed above, represents a posited (partial) order relation between events; the latter, which we have not yet discussed, refers to a systemwide policy for scheduling and deferring processing tasks to avoid processing messages in an undesired order. We further separate ordering policy into *order-control policy* (OCP) and *scheduling*. Order-control policy identifies when certain tasks *cannot* be processed; scheduling determines, subject to such constraints, which *will* be. Further discussion of scheduling in distributed systems is beyond the scope of this paper.

A. Order-control policy

The simplest order-control policy is the maximally permissive policy (or the *null policy*), which allows any task to be processed at any time. This policy makes no guarantees about the order in which tasks are processed, and may, for example, process a communication event b before another event a even if a **must precede** b (either according to the ordering schema, or causally). Ordering consistency, if required, must thus be guaranteed by some mechanism independent of the clock system.

By contrast, we define a *proper order-control policy* to be an OCP in which event b is never processed before event a if a **must precede** b. Under such a policy, if a_1 and a_2 are sent from A to B and C respectively and a_1 **must precede** a_2, and a_2 causes b_3 to be sent from C to B, then the OCP will guarantee that B handles a_1 before it handles b_1, regardless of the order in which messages arrive.

Intuitively, the practical effects of such a policy will depend on the dynamics of the underlying network. In the foregoing work, we consider the following network details:

- We assume that the network never drops messages.
- We assume that all messages are delivered correctly.
- We assume that the network never reorders messages between two participants.

We allow, however, for messages or machines to be *delayed* arbitrarily; our only liveness condition is that all sent messages get delivered *eventually*.[1] In

[1] Okay, we assume that all messages get delivered before any client-side message queue overflows and crashes the machine it lives on.

some cases, the policies under discussion will benefit from a liveness assumption that all pairs of nodes exchange communication with some frequency; we will note any such assumption explicitly when we invoke it.

B. Minimal proper OCPs for vector-based clocks

Given a vector-based ordering schema with the condition that messages sent from A to B **must succeed** each other in order, we implement the minimal proper OCP as follows:

- Never handle a message b with vector $\vec{b}$ if $\vec{b}$ dominates the vector $\vec{a}_i$ of any unhandled queued message a. (Recall that if $\vec{b} > \vec{a}_i$, then a **must precede** b.)
- Never handle a message b with vector $\vec{b}$ if $\vec{b}$ dominates any vector $\vec{a}_i^{\text{inf}}$ the minimal vector which could possibly be received next from node i. (This will be the vector last received from i, incremented in the ith component.)

This policy is *proper* because it prevents any instance in which a message a_i is received and its handling **must precede** the handling of b, which has already occurred. It is *minimal* because each rule only prevents the handling of messages which **must succeed** other messages either queued or potentially to-be-received (but which may still be received).

C. Examples

1) Minimal proper OCP for logical clocks:

If we take logical clocks as our ordering schema (as described above), then this policy may be implemented by tracking, for each conversation, the 'least next timestamp' (LNT), which is the timestamp of the first-queued message for any conversation with a queue, and $1+$ the last-received timestamp (of that conversation) otherwise. Then a message may only be processed if its conversation has a minimal LNT.

Assuming that all pairs of nodes converse with some frequency, no node will be blocked indefinitely, because eventually any conversation will receive enough messages to move it out of the minimal position where it blocks the others. However, if one node stops sending messages, all other nodes will eventually become blocked, as all other conversations' LNTs will advance until they pass that of the stopped conversation.

2) Minimal proper OCP for vector clocks:

If use vector clocks instead (as described above), then each node will instead retain a least next *vector*, and restrict processing only to conversations of with LNVs along the frontier of R-minimal LNVs, where R is the **must succeed** relation. As vector dominance has a wider allowance for incomparability than the almost-well-ordered integers, this will frequently result in multiple unblocked options to process next, unlike in the case of logical clocks.

Additionally, the OCP may be able to recover if one node stops sending messages indefinitely. Specifically, it will do so in the case of the disappearance of node i iff all nodes have the same value in the ith component of their logical clocks. This situation arises naturally before i has sent any messages—and so the system can begin processing messages before all nodes have come online—or can be arranged artificially, by allowing a node to send multiple messages with vectors with the identical value in its own component.

III. INFORMATION-PROVENANCE CLOCKS

A. (More) Granular Vector Clocks

Considering the motivation of OCP design, a natural extension of vector clocks is a schema which traces causal links more closely. A fundamental assumption of the vector clock schema as formulated above is that any piece of information that a process receives can causally affect any piece of information that the process subsequently sends. (This assumption informs our decision about when to roll vectors forward to 'cover' received timestamps, *i.e.*, on every receipt.) While this assumption may be valid in some cases, it might be quite far from the truth in others. If so, then vector clocks run the risk of incrementing clock values far too often. This can create gaps where they need not logically exist, over-specifying the history recorded, and makes it easy for the minimal proper OCP to become blocked.

Instead, we propose associating separate clock vectors (*information-provenance clocks*) to distinct pieces of information (or sets of information) that are unlikely to affect each other, even if located within a single process. If clock values are advanced only when one piece of information explicitly influences another, then 'most' clock instances within

a process will be unaffected by 'most' messages received. We may wish[2] that, if the separation of clocks is sufficiently granular, any clock advances will correspond to the reliance of one value on all influences indicated in the vector being advanced to. If so, then we can minimize the number of receipts which the minimal proper OCP blocks.

B. Disentangling separate 'conversations'

In particular, we hypothesize that information-provenance clocks should allow 'side conversations' to occur unhindered by one or more processes 'flooding' a system with unrelated messages. Consider a system under the minimal proper vector-based OCP with three processes, A, B, and C, and suppose that A is flooding the system with many messages, while B and C are attempting to send unrelated messages to each other at a slower rate. Further suppose that B has received n messages from A, but that C has not yet received (or not yet processed) any messages from A.

Now consider what happens when B tries to send a message to C. If A, B, and C are using the traditional vector clocks, then B's vector clock will have a value of at least n in A's component, where C's vector clock will have a much lower value. When B tries to send a message to C, C will infer a gap in its communication with A. Under the proper-OCP constraint, C will be disallowed from processing B's message until it has processed up to n of A's.

Now consider the same scenario with information-provenance clocks. If we know ahead of time that the messages B and C will be sending to each other are not affected by the messages sent between A and B or A and C, then we can keep one information-provenance clock each for the communication channels between A/B, A/C, and B/C. In this scenario, B's B/A clock will have a value for A of at least n, but its B/C will not have that A value. As a result, when B sends a message to C, C has no reason to believe that it has missed critical information from A and can process the message (within its messages-from-B logic) without worrying about receiving one from A that **must precede** the message from B.

Additional clocks, however, impose overhead on systems by requiring additional storage space and more-involved computations to update appropriately. We wish to understand the effect of clock granularity on side conversations by studying the character of clock gaps and blocking messages in a more-granular schema.

[2]We conjecture, but do not here prove, that *some* information-dependence structures cannot fully satisfy this condition under *any* granularization in this model. Nevertheless, we conjecture that finer granularizations will, in general, reduce the frequency of spurious clock gaps.

IV. Scale Model Simulation

A. Overview

To evaluate the effect of clock granularity on clock gaps, we constructed a scale model of a distributed system implementing information-provenance clocks. This model will simulate multiple processes running on separate machines, at different speeds (a certain number of steps per second, determined at initialization). Each process will manage multiple distinct 'threads', each handling its communications with one other node, protected by separate information-provenance clocks.

On each timestep, a process will randomly choose a message queue from those not blocked by OCP. If the queue is empty, the process will randomly choose to do an internal action (which increments its internal clock counter) or an external action (by sending a vector-stamped message to the node on the other end of the channel). If the queue is not empty, the process will simply handle the first message from the queue, applying the appropriate updates to the appropriate internal clocks. Additionally, processes occasionally engage in 'crosstalk' from one thread to another, which rolls the target's clock vector forward to cover the source's.

In this setting, we are able to vary a number of experimental variables to observe their effect on the traces that the model produces, specifically in terms of the frequency with which messages are blocked by OCP.

B. Simulating Granularity

To examine the effect of granularity in the system design, we used three different setups, each using 5 processes sending messages to each other:

- Each node uses a single vector for all of its conversations. (This is the traditional vector-clock model.)

- Each node uses two vectors: one vector for the conversations with the two nodes to its immediate left, and another for the two nodes to its right.
- Each node uses a separate vector for each conversation.

To model the 'actual' granularity in the structure of the data processed, we introduce the notion of random crosstalk. We assume that most messages that processes send over their channels only affect the information directly associated with that channel by the information-provenance clock, but that messages occasionally draw from more than one channel on the source machine. When a process sends a message, it will, with some probability, choose a message it has received from another process to 'influence' the outgoing message. When so, it rolls the vector of the outgoing channel forward to cover the source of the influence. In our tests, we use three different probabilities for crosstalk to occur on each send event: $0.9, 0.5, 0.1$, to simulate information that is highly-interconnected, information that is moderately connected, and information that is highly independent, respectively.

C. Speed Distribution

Another major experimental variable we manipulate is the relative speed of different machines. We are primarily interested in two cases - when all the machines are around the same speed, and when one machine is faster than the others. We test the case of when the speeds of the machine are relatively close to each other by randomly assigning each machine a speed of $4, 5$, or 6 steps per second, and test the case of one machine being faster than the rest and flooding the network by randomly assigning all $n-1$ 'slow' machines a speed of $1, 2$, or 3 steps a second, and assigning one 'fast' machine a speed of 6 steps a second. We also test a uniform case where all machines run at speed 6.

D. Policy Strictness

Lastly, test two different order-control policies. One policy, the *strict* policy, is the minimal proper OCP as described above. The second, which is *not* a proper OCP, only considers a message blocked if there is a *queued* message whose vector it dominates.

Our scale simulation implements the difference in these two policies by allowing processes to peek at the head of their message queues. The processes can see the vector clock associated with a message and, by peeking at the heads of all the other message queues, determine the **must succeed**-minimal sets of LNVs. The assumption underlying this model is that processing a message is costly relative to peeking and scheduling; the fundamental unit of time in a step represents the amount of time necessary to process a message and do the work associated with that message.[3]

E. Output Variables

We look primarily at one output variable: for each process, the fraction of the time that the process cannot handle the message it intends to because it is blocked by another message. We choose to use this indicator because it encapsulates how often a process chose to handle a message but couldn't. As such, it acts as a proxy measurement for how badly timing constraints hinder progress.

V. Results

The results of our experiments are shown in Figs. 1 and 2. We report the fraction of intended message-handles which are blocked by order-control policy, averaged across twelve 5-minute runs of each model, which each correspond to 1200-1800 timesteps[4] on each of the 5 machines.

The biggest factor affecting these ratios are the order-control policy. When processes use a strict policy, wherein they can only handle a message if they know that they cannot receive a message that **must precede** it, processes can spend up to 80% of their time in a blocking state. When they use a more lenient policy, they tend to spend no more than 10% of their time in a blocking state in the very worst case; when there is not a single fast process flooding the network, they tend to spend less than 10% of their time in a blocking state.

[3]It is worth noting that this assumption is not necessarily valid for all systems. The limiting factor in some systems might be the time it takes for a stream of messages to be read, or the time it takes a message to travel from machine to machine; in such a system, the assumption that message processing is costly relative to scheduling would not hold.

[4]The exception is the 'bimodal' case, where the fast machine has approximately 1800 timesteps and the slow machines have 300-900 each.

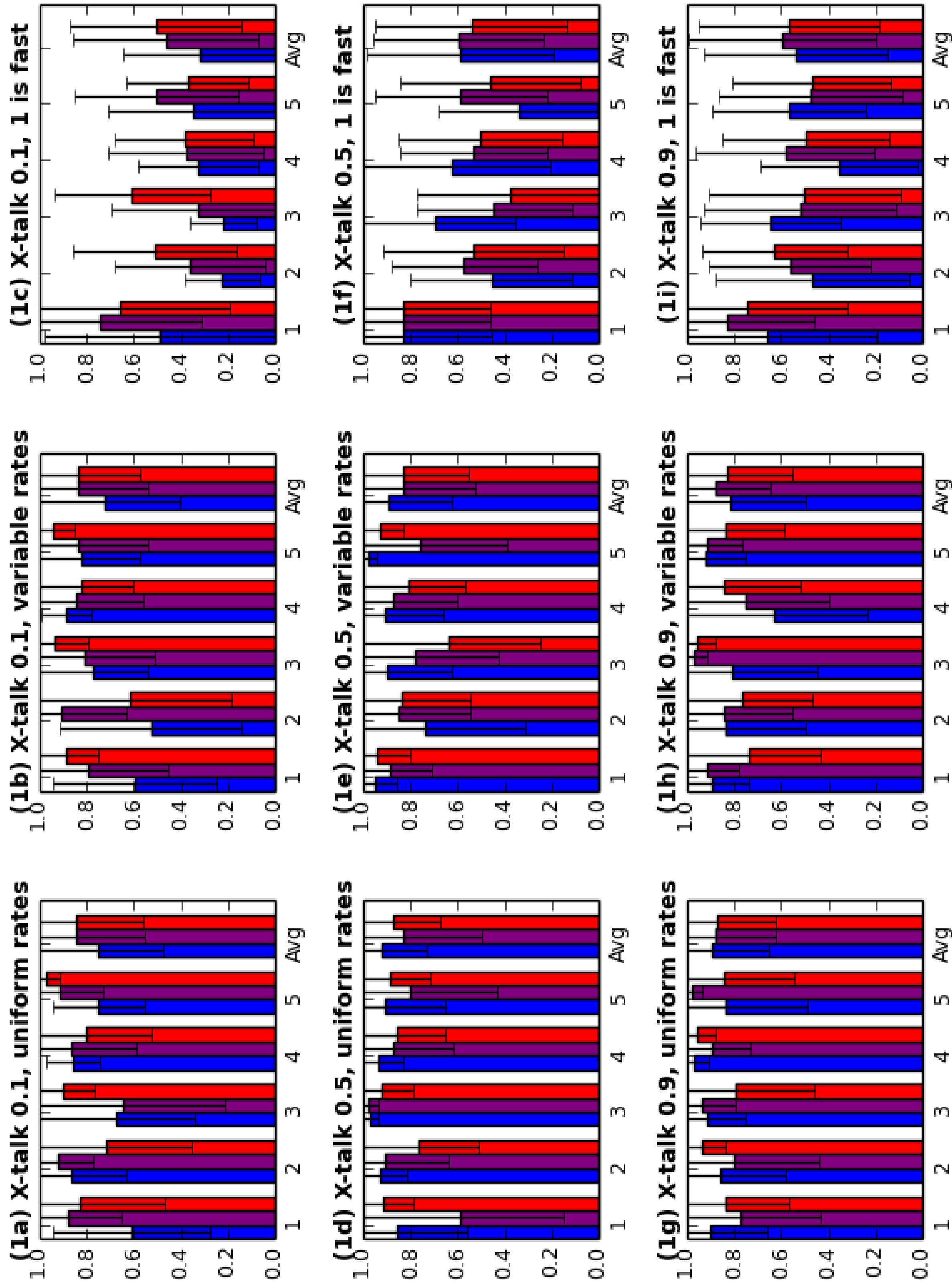

Fig. 1. Blocking ratios for models running with a strict OCP. **Left/center/right**: uniform rates, variable rates, bimodal rates. **Top/middle/bottom**: crosstalk $p = 0.1, 0.5, 0.9$. **Blue**: maximally granular (one vector per channel). **Purple**: one vector per two channels. **Red**: one vector per process / traditional vector clocks.

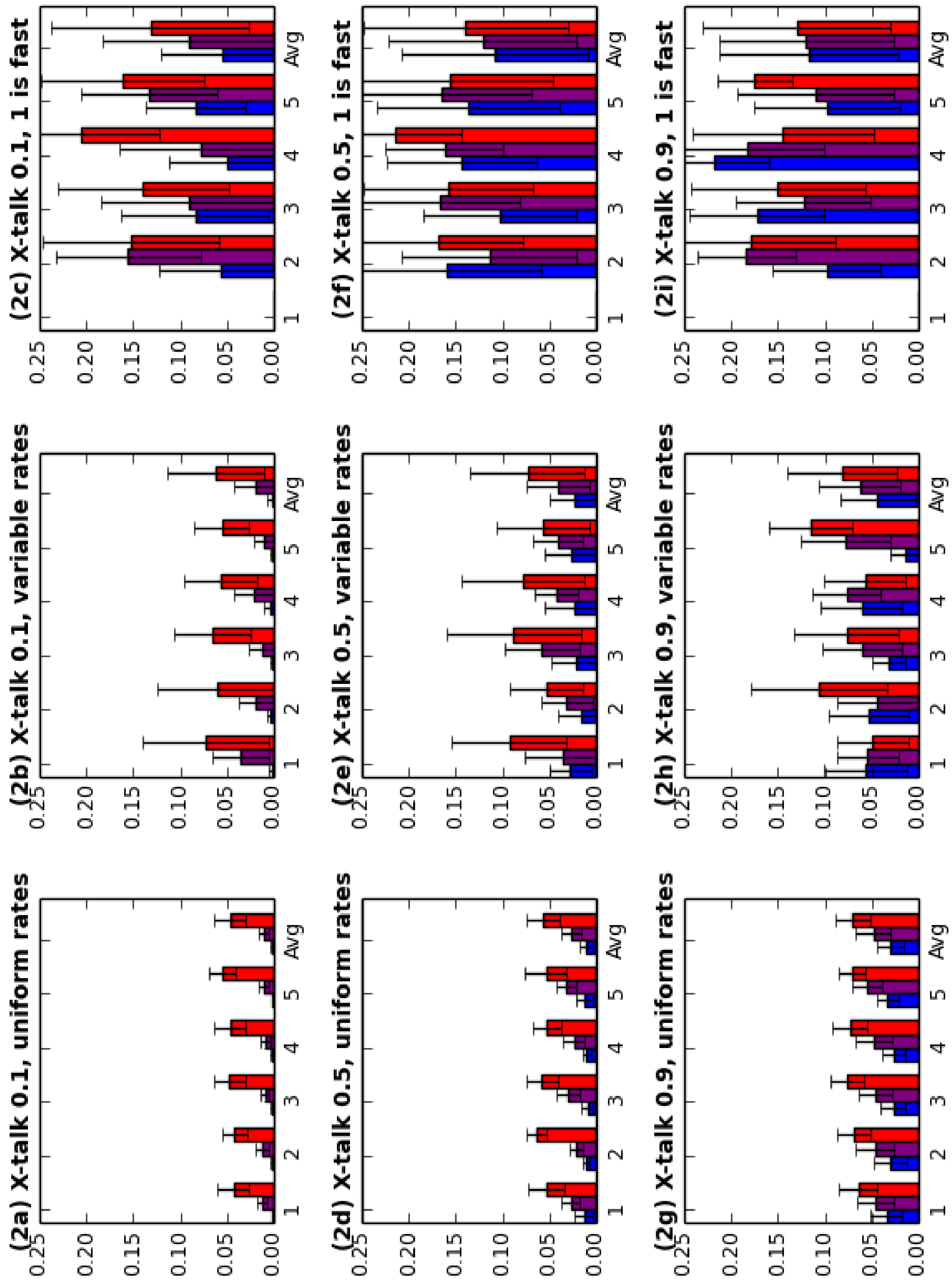

Fig. 2. Blocking ratios for models running with a non-strict OCP. **Left/center/right**: uniform rates, variable rates, bimodal rates. **Top/middle/bottom**: crosstalk $p = 0.1, 0.5, 0.9$. **Blue**: maximally granular (one vector per channel). **Purple**: one vector per two channels. **Red**: one vector per process / traditional vector clocks.

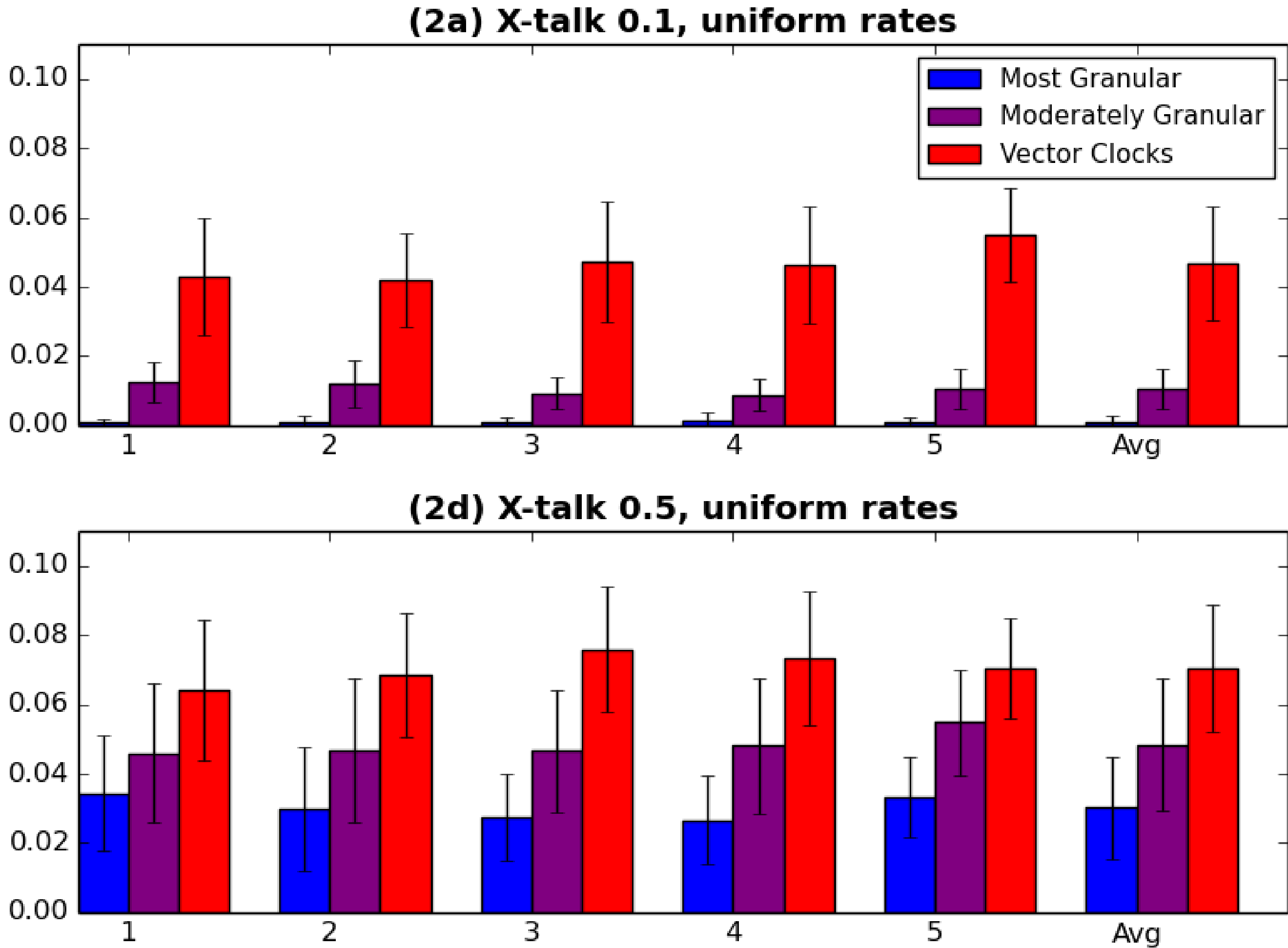

Fig. 3. Detail of subplots (2a) and (2d) from Figure 2 above. **Top**: low rate $p = 0.1$ of crosstalk. **Bottom**: moderate rate $p = 0.5$ of crosstalk. In both cases, all processes are running at the same speed. Each bar reports an average of twelve 5-minute runs, including approximately 1800 timesteps per process per run. **Blue**: maximally granular (one vector per channel). **Purple**: one vector per two channels. **Red**: one vector per process / traditional vector clocks.

When processes use a strict order-control policy, it is not clear whether information-provenance clocks have a discernable effect on the amount of time processes spend in a blocking state; because the fraction of time spent blocked is so high, there is relatively high variance in our results, and it is difficult to draw conclusions about the effect of using information-provenance clocks over vector clocks. In some cases, such as when there is little crosstalk and no variation in machine speed (Fig. 1a), the information-provenance clocks appear to offer a slight advantage over vector clocks. In most other cases, however, the information-provenance clocks perform similarly to vector clocks.

The more lenient ordering policy is much more promising. Under this regime, processes only consider a message blocked if they have received (but not yet processed) another message that **must precede** it. Here, the information-provenance clocks vastly outperform vector clocks. In the setups where the machines are proceeding at uniform rates (Figs. 2a, 2d, and 2g), the systems using information-provenance clocks spend orders of magnitude less time in blocking states than their counterparts using vector clocks. The magnitude of these advantages carry over to the case of low crosstalk and variable machine speeds and are reduced, but remain significant in the cases of moderate or near-constant crosstalk. Even when there is single process flooding the network with messages, information-provenance clocks still enjoy a significant advantage over vector clocks.

It is worth studying some of these graphs in more detail to more deeply understand the dynamics driving blocking states. Consider Figs. 2a and 2d (expanded in Fig. 3). There is virtually no difference between blocking rates for the systems using vector clocks, even though crosstalk increases five-fold from 2a to 2d. However, blocking rates do increase for the systems using information-provenance clocks; in other words, the information-provenance clocks adapt to independence of their data.

Interestingly, the systems using the most granular formulation of the information-provenance clocks perform similarly under high levels of crosstalk to the systems using a moderately granular formulation of the information-provenance clocks under low levels of crosstalk. This suggests a close relationship between those two cases - in the former case, crosstalk forces almost every outgoing message to be affected by two communication channels; in the latter case, this is formalized by the setup of the information-provenance clocks. Note that either case is still two steps removed from systems using vector clocks, where each outgoing piece of information is implicitly affected by four communication channels.

VI. Conclusion

Inquiry is certainly warranted into the question of how ordering schemata interact with order-control policy in real systems, but our experimental results suggest that information-provenance clocks show promise in reducing blocking overhead in systems which desire weak-to-moderate order-control guarantees. (Our results are inconclusive in the regime of strong order-control guarantees.)

Further research might investigate whether the high blocking overhead required by a strictly proper order-control protocol might be reduced by enabling nodes to send periodic or demand-driven 'update requests' to other nodes, prompting them to roll their shared channel's vector forward far enough to unblock the requester's other threads. In such a system, we would expect overall blocking load to decrease, and for similar results to emerge as observed in the weak order-control regime, with respect to the benefits of clock granularity.

Another question of interest is whether the benefits of information-provenance clocks persist in systems where the cost of scheduling and message inspection is a first-order consideration, in contrast to the regime we simulated, in which message-handling costs dominate.

One application for which we believe granular information-provenance clocks to be particularly apt is in distributed structured databases, especially graph databases. In this setting, the model of nodes carrying on separate conversations with occasional crosstalk approaches reality quite cleanly, considering a graph sharded across a network of nodes, after being organized by some clustering algorithm. The fine details, of course, go beyond our simple experiments, though we expect our main result—that granularity reduces blocking load—to generalize.

Acknowledgements

We wish to thank Jim Waldo, both for teaching an excellent course in distributed systems that motivated this research, and for steering us away from a proposed project concerning distributed holomorphic cryptography which was, to be frank, unlikely to end in any way but tears. We also wish to thank Margo Seltzer, for introducing us to the polysyllabic word 'provenance', and James Mickens, whose course in Systems Security precipitated the beginning of a fruitful academic partnership.

References

[1] L. Lamport. Time, clocks, and the order of events in a distributed system. *C. ACM*, 21(7):558–565. Jul 1978.

[2] Ö. Babaoğlu, K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, 2nd ed. pp. 55–96. ACM Press 1993.