Local Mutual Exclusion for Dynamic, Anonymous, Bounded Memory Message Passing Systems

Joshua J. Daymude and **Andréa W. Richa** (Arizona State University)

Christian Scheideler (Universität Paderborn)

SAND 2022 (Virtual Event) — March 28–30, 2022

Motivation: Lifting Message Passing to Interactions

Traditionally, distributed computing models have lived in two communication paradigms: message passing and shared memory.

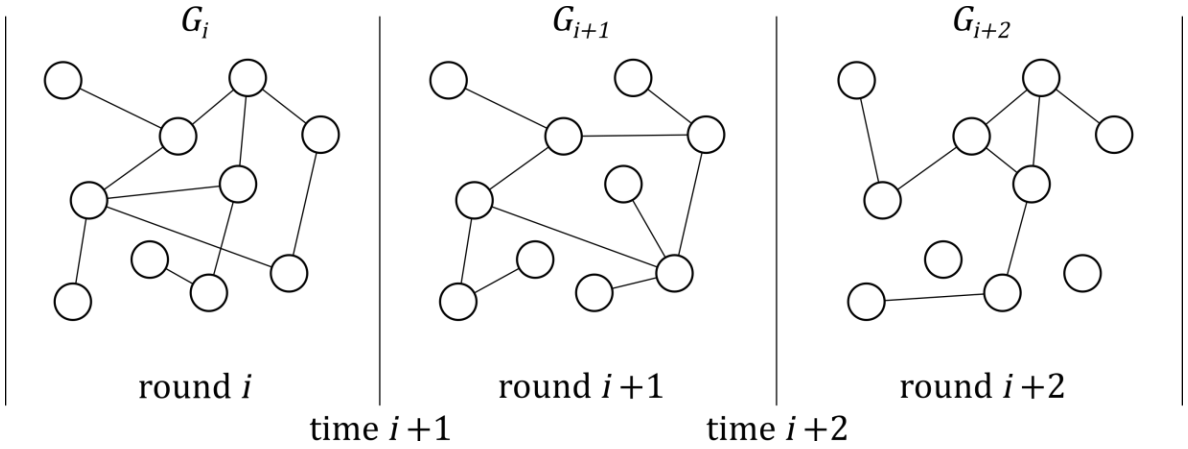
More recent application-focused models—especially for swarm robotics, programmable matter, mobile sensor networks, and so on—have flirted with more abstract **interactions**.

- Population protocols consider pairwise, stateful interactions.
- Chemical reaction networks consider "reactions" that convert entities based on type.
- Various programmable matter models, like the amoebot model, have traditionally assumed particles can access their neighbors' memory when performing actions.

How do we bridge these two paradigms, especially in concurrent, dynamic settings?

Time-Varying Graphs: Dynamics

We extend the established **time-varying graphs** model [Casteigts et al., 2012; Casteigts 2018] to interoperate with asynchronous message passing and either semi-synchronous or asynchronous node activation.



We assume an adversary controls which edges exist in each round, and that the underlying graph is complete (i.e., the adversary can introduce any edge).

Each node has the same fixed number of ports $\Delta > 0$; the adversary cannot assign a new edge to a node with no open ports. All nodes are anonymous, lacking unique IDs, but a node u can locally identify an incident edge $\{u, v\}$ with a port label $\ell_u(v) \in \{1, \ldots, \Delta\}$.

Time-Varying Graphs: Communication and Actions

Nodes communicate via (asynchronous) message passing. A message m sent from u to v remains in transit until either v receives it (at a time chosen by the adversary) or u and v are disconnected, at which point m is lost.

Distributed algorithms are encoded as actions: $\langle label \rangle : \langle guard \rangle \rightarrow \langle operations \rangle$.

A guard is a Boolean predicate determining whether the action is enabled for a node based on its state and any messages in transit that it might receive.

The operations specify what a node does when executing the action, following the form:

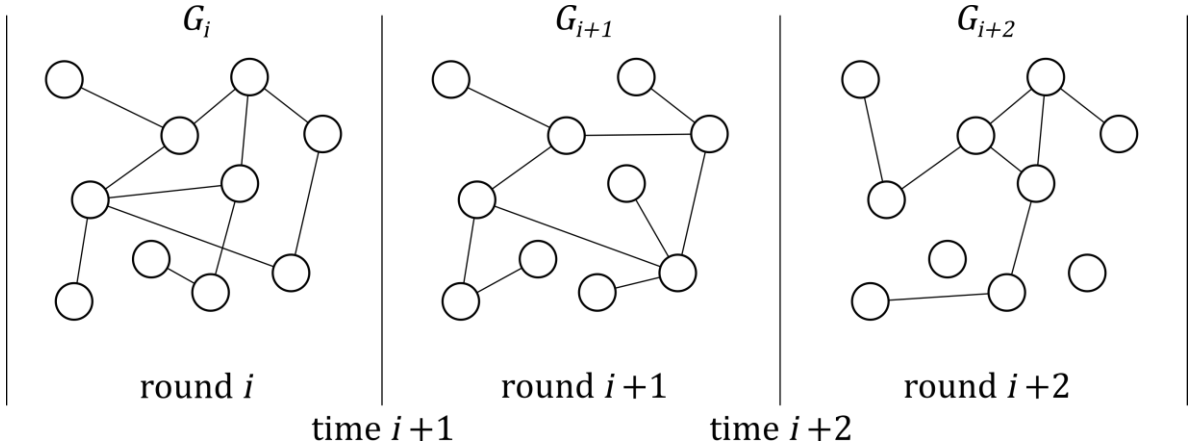
- Receive at most one message chosen by the adversary.
- Perform a finite amount of computation and/or state updates.
- Send at most one message per port label.

Time-Varying Graphs: Node Activation

An adversary controls when nodes are activated and which (enabled) actions they execute.

Semi-Synchrony. In each round, the adversary concurrently activates any (possibly empty) subset of nodes with enabled actions. Those action executions are guaranteed to complete within that round (i.e., before the adversary changes the topology again).

Asynchrony. Same setup semi-synchrony, but action executions may span multiple rounds (i.e., action executions may be concurrent with topology changes).



Weak Fairness. Any action that is continuously enabled for a node is eventually executed, and any message in transit on a continually existent edge is eventually processed.

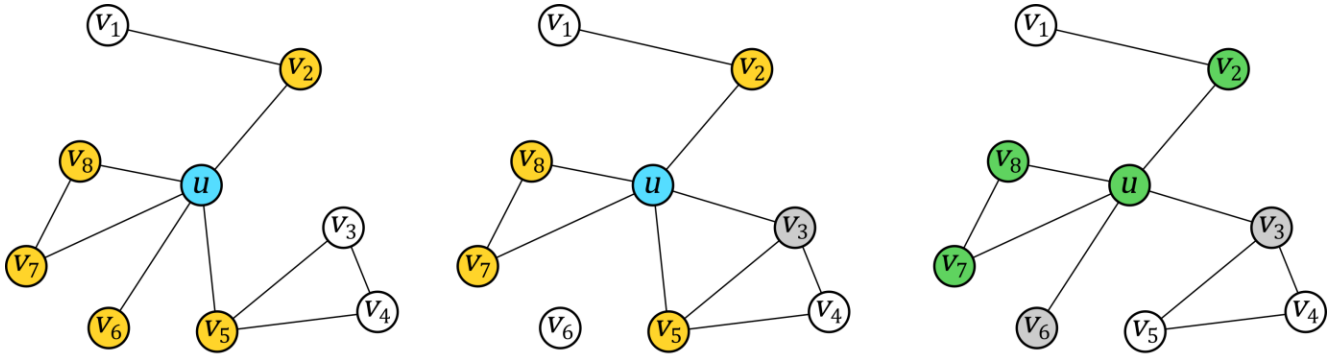
Local Mutual Exclusion

Informally, enable nodes exclusive access to themselves and their immediate neighbors, or to as many as possible given topological changes.

Each node u stores $\text{lock}(u) \in \{\perp, 0, \ldots, \Delta\}$ that is $\perp$ if u is unlocked, 0 if u has locked itself, and $\ell_u(v) \in \{1, \ldots, \Delta\}$ if u is locked by v.

The lock set of node u in round i is $\mathcal{L}_i(u) = \{\, v \in N_i(u) \mid \text{lock}(v) = \ell_v(u) \,\}$, which also includes u itself if $\text{lock}(u) = 0$.

Given rounds i and $j > i$, the persistent (closed) neighborhood of u are those neighbors that remain connected to u in rounds i through j: $\{u\} \cup \{\, v \in N_i(u) \mid \forall t \in [i, j], \{u, v\} \in G_t \,\}$.

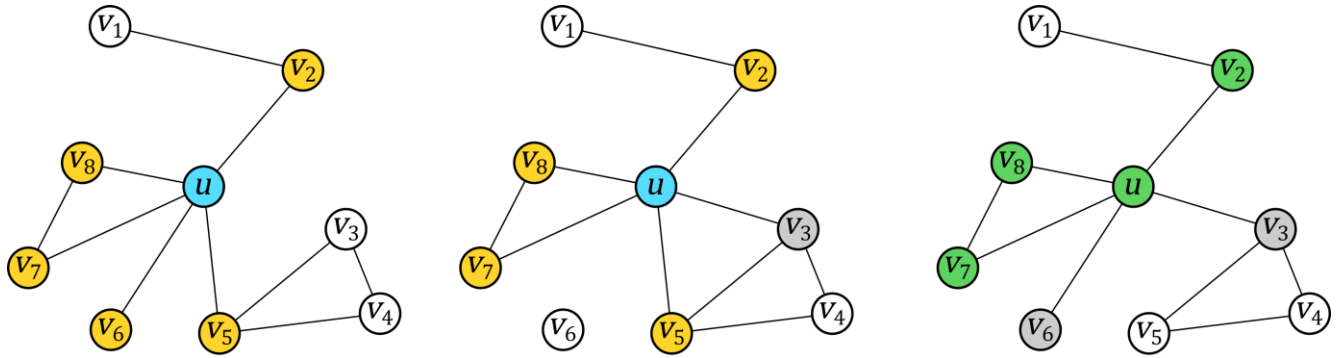


Local Mutual Exclusion

The local mutual exclusion problem requires Lock and Unlock operations satisfying:

Mutual Exclusion. For all rounds $i \in T$ and all pairs of nodes $u, v \in V$, $\mathcal{L}_i(u) \cap \mathcal{L}_i(v) = \emptyset$.

Lockout Freedom. Every issued lock request eventually succeeds—i.e., the issuing node's lock set equals its persistent neighborhood—with probability 1.



Note! Because each node's $\mathrm{lock}(u)$ variable can point to at most one node, mutual exclusion is trivially satisfied. Lockout freedom, however, remains challenging.

Related Work

Our focus is on dynamic, anonymous, bounded memory message passing systems.

- Classical mutual exclusion algorithms [Singhal, 1993 (survey)] often used some combination of unique identifiers, global coordination, and unbounded counters (Lamport-style clocks).
- The **arrow protocol** [Demmer and Herlihy, 1998] uses $\Theta(1)$ per node, but it's not obvious how to extend it to dynamic underlying topology, despite recent improvements [Ghodselahi and Kuhn, 2017; Khanchandani and Wattenhofer, 2019].
- Our mutual exclusion variant deals with exclusive access over neighborhoods. This is most closely related to **distance-3 independent sets** and (α, β)**-ruling sets** [Awerbuch et al., 1989; Schneider et al., 2013; Kuhn et al., 2018], whose algorithms typically assume static topologies, unique identifiers, and synchronous message delivery.
- Related problems appear in contention resolution in **mobile ad-hoc networks** [e.g., Attiya et al., 2010; Sharma et al., 2014], though these often rely on tokens or wall-clock timing.

A Randomized Algorithm for Local Mutual Exclusion

Initiators are nodes issuing lock requests.

- An initiator that calls **Lock** sends prepare() messages to its closed neighborhood.
- After receiving ready() messages from all (persistent) neighbors, it becomes competing and sends request_lock(p) messages to its participants, where p is a randomly chosen priority.
- If it receives at least one win(FALSE) message, it lost this competition and must compete again.
- Otherwise, if all responses are win(TRUE), it sends set_lock() messages, which then get acknowledged to conclude **Lock**.

Participants are those being locked/unlocked.

- On receiving a prepare() message, a participant puts the sending initiator into one of the following categories:
  - "On Hold", if a competition is already underway.
  - An "applicant" for the current competition, to which it replies ready().
- Applicants are promoted to candidates on receiving their request_lock(p) messages.
- The candidate with the unique highest priority wins win(TRUE), and all others lose win(FALSE), remaining to try again.
- Once there are no remaining candidates, initiators on hold are promoted to applicants.

Key Results

Theorem. If all nodes start with valid initial values, the local mutual exclusion algorithm satisfies the mutual exclusion and lockout freedom properties under semi-synchronous concurrency, requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$, and has at most two messages in transit along any edge at any time.

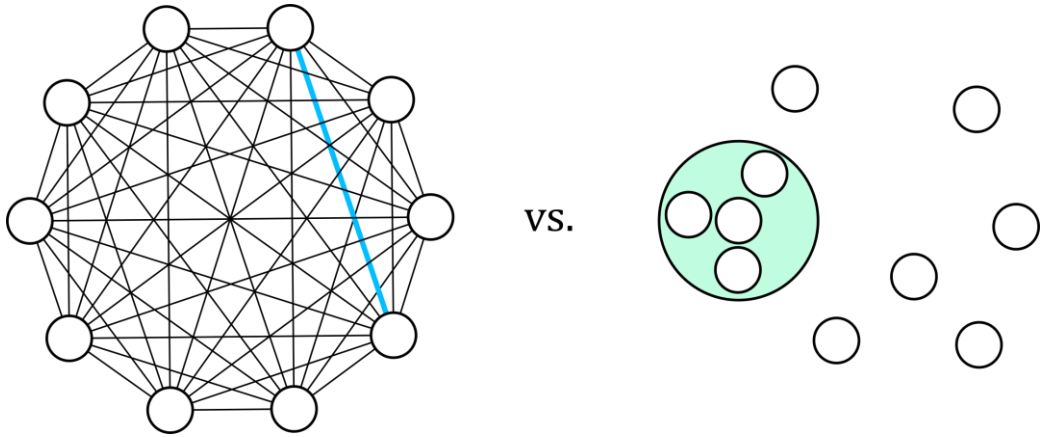
Lemma. For any asynchronous schedule $\mathcal{S}$, there exists a semi-synchronous schedule $\mathcal{S}'$ containing the same action executions as $\mathcal{S}$ that produces the same outcome for every action execution in $\mathcal{S}$.

Corollary. The local mutual exclusion algorithm also satisfies the mutual exclusion and lockout freedom properties under asynchronous concurrency.

Passively Dynamic Application: Population Protocols

Population protocols model passively mobile sensor networks [Angluin et al., 2006].

Typically, a sequential scheduler chooses one pair of agents to act per time. In reality, there may be many agents within interacting distance concurrently [Czumaj and Lingas, 2021].



Idea 1. Lock closed neighborhood, and interact with any locked node. This could, in concept, support multi-way interactions.

Idea 2. Choose one node to lock, and interact if successful. This may be more efficient in the setting where neighborhoods are typically large and only pairwise interactions are desired.

Actively Dynamic Application: Canonical Amoebot Model

The **canonical amoebot model** [DRS, 2021] is an updated formalism for the amoebot model of programmable matter dealing with concurrency.

The canonical amoebot model partitions amoebot functionality into:

- A higher-level application layer where algorithms are detailed in terms of operations.
- A lower-level system layer that executes an amoebot's operations via message passing.

Assumed the existence of **Lock** and **Unlock** operations (without giving implementation) for a concurrency control framework that converts sequential algorithms into concurrent ones by wrapping actions of the sequential algorithms in Lock/Unlock pairs.

Probing those results more carefully, we find that the asynchronous extension of our algorithm can directly implement these black box operations.

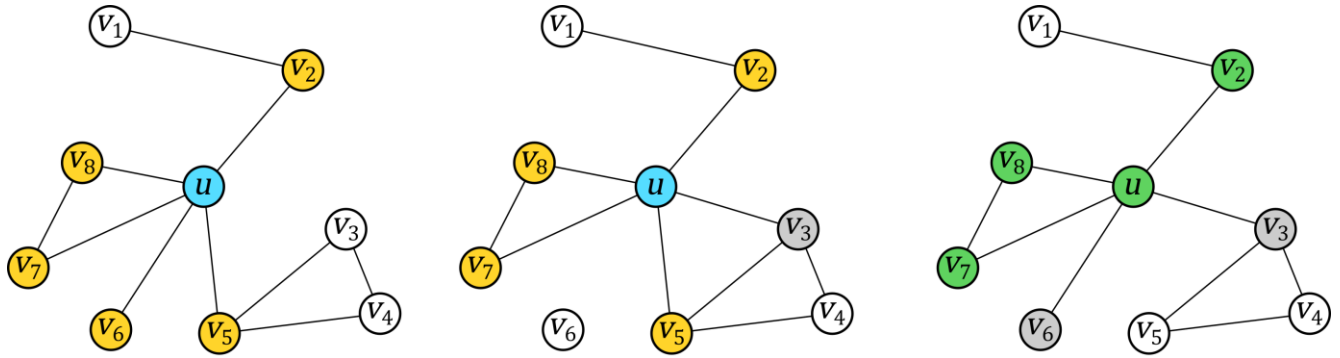
For the often-considered geometric space variant, where each amoebot has at most eight neighbors, our algorithm requires only $\mathcal{O}(\Delta) = \Theta(1)$ memory per node.

Conclusion

Local mutual exclusion is the bridge between traditional asynchronous message passing and atomic “interactions” that depend on the states of a node and its neighbors, even as those neighbors might connect and disconnect.

We gave a randomized algorithm that solves local mutual exclusion under both semi-synchronous and asynchronous concurrency using $\mathcal{O}(\Delta)$ memory and $\Theta(1)$-sized messages.

We illustrated applications of our algorithm in implementing some underlying assumptions of interaction in both passively and actively dynamic systems.



Thank you!

sops.engineering.asu.edu

jdaymude.github.io

@joshdaymude