Reasoning about Knowledge in Distributed Systems Using Datalog

Matteo Interlandi

University of Modena and Reggio Emilia
matteo.interlandi@unimore.it

Abstract. Logic programming has been considered a viable solution for distributed computing since the Fifth Generation Computer Systems project [8]. Nowadays, this line of thought is gaining new verve, pushed by the need for new programming paradigms for addressing new emerging issues in distributed computing. We argue that a missing piece in the current state-of-the-art is the capability to express statements about the knowledge state of distributed nodes. In fact, reasoning about the knowledge state of (group of) nodes has been demonstrated to be fundamental in order to design and analyze distributed protocols [7]. To reach this goal, we designed Knowlog: Datalog$^\neg$ augmented with a set of epistemic modal operators, allowing the programmer to directly express what a node "*knows*" instead of low level communication details.

1 Introduction

Since the Fifth Generation Computer Systems project [8], many authors have stated how logic programming [10,9] could be used to express distributed programs' specifications. New emerging trends that are arising nowadays provide new chances for proving how logic programming can be used to tackle distributed computing concerns. For instance, the tight coupling among distributed nodes and the related overhead introduced by traditional mechanisms implementing ACID properties starts to be considered unacceptable by cloud computing operators [5]. To address these issues, monotonic logic programming has been employed to formally specify eventually consistent distributed programs [3].

Motivated by all these facts, our goal is to open a new direction in the investigation on how Datalog could be adopted to implement distributed systems. We conjecture, in fact, that a missing piece in the litterature exists: this is the possibility to express in Datalog statements about the knowledge state of distributed nodes. The ability to reason about the knowledge state of (group of) nodes has been demonstrated to be a fundamental tool in multi-agent systems in order to specify global behaviors and properties of protocols [7]. Therefore, inspired by previous works in distributed logic programming [9,3] and knowledge-base programs for multi-agent systems [7], we develop Knowlog: Datalog$^\neg$ leveraged with a set of epistemic modal operators. In this way programmers are able to directly express nodes' state of knowledge instead of low level communication details. The advantage of this formalism is that it abstracts away all the mechanisms by which the knowledge is exchanged (message passing, shared memory, etc) and permits to explicitly reason about the nodes' state of knowledge. To support our assertions, we describe an implementation of the two phase commit protocol.

The remainder of the paper is organized as follows: Section 2 contains some preliminary notations about Datalog$^{\neg}$ and Datalog$^{\neg}$ augmented with a notion of time. Section 3 describes what we intend for a distributed system and introduces some concepts such as *global state* and *run* that will be used in Section 4 to specify the modal operators K, E and D. In addition, Section 4 contains the two phase commit protocol implementation in Knowlog. The paper finishes with Section 5 which specifies Knowlog semantics, and conclusions.

2 Preliminaries

As usual, a Datalog$^{\neg}$ rule is an expression in the form:

$$H(\bar{u}) \leftarrow B_1(\bar{u}_1), ..., B_n(\bar{u}_n), \neg C_1(\bar{v}_1), ..., \neg C_m(\bar{v}_m)$$

where $H(\bar{u})$, $B_i(\bar{u}_i)$ and $C_j(\bar{v}_j)$ are *atoms*, H, B_i, C_j are relation names in **relname** and $\bar{u}, \bar{u}_i, \bar{v}_j$ are tuples of appropriate arities. Tuples are composed by *terms* and each term can be a constant in the domain **dom** or a variable in the set **var**, with both **dom** and **var** disjoined from **relname**. In the followings we will interchangeably use the words *predicate*, *relation* and *table*. A *literal* is an atom (in this case we refer to it as *positive*) or the negation of an atom. We will usually refer to a ground atom as a *fact*. We allow *built-in* predicates to appear in the body of rules. Thus, we allow relation names such as $=$,$\neq$,$\leqslant$, $<$, $\geqslant$, and $>$. We also allow *aggregate operations* in rule heads in the form $R(\bar{u}, \Lambda < \bar{\mu} >)$ with Λ one of the usual aggregate functions and $< \bar{\mu} >$ defining the grouping of arguments $\bar{\mu}$. In this paper we assume each rule to be *safe*, i.e. every variable occurring in a rule head appears in at least one positive literal of the rule body. Then, a *Datalog$^{\neg}$ program Π* is a set of safe rules. As usual, we refer to *idb*(Π) as the *itensional* part of the database schema, while we refer to the *extensional* schema as *edb*(Π). Given a database schema $\mathbf{R}$, a *database instance* is a finite set $\mathbf{I}$ of facts.

As introductory example, we use the program depicted in Listing 1.1 where we employed an *edb* relation `link` to specify the existence of a link between two nodes. In addition, we employ an intensional relation `path` which is computed starting from the `link` relation (`r1`) and recursively adding a new path when a link exists from `A` to `B` and a path already exists from `B` to `C` (`r2`).

```
r1: path(X,Y):-link(X,Y).
r2: path(X,Z):-link(X,Y),path(Y,Z).
```

Listing 1.1. Simple Recursive Datalog Program

2.1 Time in Datalog$^{\neg}$

Distributed systems are not static, but evolving with time. Therefore it will be useful to enrich Datalog$^{\neg}$ with a notion of time. For this purpose we follow the road traced by Dedalus$_0$ [6]. Thus, starting with considering time isomorphic to the set of natural numbers $\mathbb{N}_0$, a new schema $\mathbf{R}^T$ is defined incrementing the arity of each relation $R \in \mathbf{R}$ by one. By convention, the new extra term, called *time suffix*, appears as the last attribute in every relation and has values in $\mathbb{N}_0$. We will sometimes adopt the term *timestamp* to refer to the time suffix value. In fact, each tuple can be viewed as timestamped with the evaluation step in which it is valid. For conciseness we will employ the term *time-step* to denote an evaluation step. By incorporating the time suffix term in the schema definition, we now have multiple instances for each relation, one for each timestamp. In this situation, with $\mathbf{I}[0]$ it is named the *initial database instance* comprising at least all ground atoms existing at the initial time 0, while with $\mathbf{I}[n]$ the instance at time-step n. In accordance with this approach, tuples by default are considered *ephemeral*, i.e., they are valid only for one single time-step. In order to make tuples *persistent* - i.e., once derived, for example at time-step s, they will eventually last for every time-step $t \geq s$ - a new built-in relation `succ` with arity two and ranging over the set of natural numbers is introduced. $\texttt{succ}(x, y)$ is interpreted true if $y = x + 1$. Program rules are then divided in two sets: *inductive* and *deductive*. The former set contains all the rules employed for transferring tuples through time-steps, while the latter encompasses rules that are instantaneous, i.e, local into a single time-step. Some syntactic sugar is adopted to better characterize rules: all time suffixes are omitted together with the `succ` relation, and a `next` suffix is introduced in head relations to characterize inductive rules. For a complete discussion on how to incorporate time in Datalog$^\neg$, we refer the reader to [6].

In Listing 1.2 the simple program of the previous section is rewritten in order to introduce the new formalism. Rule `r1` is a *persistency* rule which moves towards time-steps tuples that are not been explicitly deleted. To note that relation *del_P* is a not mandatory *idb* relation which contains all the facts of P that must be deleted (will not appear in P at state $t = s + 1$).

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y).
r2: del_link(X,Y):-link_down(X,Y).
r3: path(X,Y):-link(X,Y).
r4: path(X,Z):-link(X,Y),path(Y,Z).
```

Listing 1.2. Inductive and Deductive Rules

3 Distributed Logic Programming

Before starting the discussion on how we leverage the language with epistemic operators, we first introduce our distributed system model and how communication among nodes is performed. We define a distributed message-passing system to be a non empty finite set $\mathcal{N} = \{id_1, id_2, ..., id_n\}$ of share-nothing nodes joined by bidirectional communication links. Each node identifier has a value in the domain **dom** but, for simplicity, we assume that a node id_i is identified by its subscript i. Thus, in the followings we consider the set $N = \{1, ..., n\}$ of node identifiers, where n is the total number of nodes in the system.

With *adb* we denote a new set of *accessible* relations encompassing all the tables in which either facts are created remotely or they need to be delivered to another node. These relations can be viewed as tables that are horizontally partitioned among nodes and through which nodes are able to communicate. Each relation $R \in adb$ contains a *location specifier* term [12]. This term maintains the identifier of the node to which every new fact inserted into the relation R belongs. Hence, the nature of *adb* relations can be considered twofold: for one perspective they act as normal relations, but from another perspective they are local buffers associated to relations stored in remote nodes. As pointed out in [9,6], modeling communication using relations provides major advantages. For instance, the disordered nature of sets appears particularly appropriate to represent the basic communication channel behavior by which messages are delivered out of order.

Continuing with the same example of previous sections, we can now employ it to actually program a distributed routing protocol. In order to describe the example of Listing 1.3 we can imagine a real network configuration where each node has the program locally installed, and where each `link` relation reflects the actual state of the connection between nodes. For instance, we will have the fact `link(A,B)` in node A instance if a communication link between A and node B actually exists. The location specifier term is identified by the prefix `@`.

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y).
r2: del_link(X,Y):-link_down(X,Y).
r3: path(@X,Y):-link(X,Y).
r4: path(@X,Z):-link(X,Y),path(@Y,Z).
```

Listing 1.3. Inductive and Deductive Rules

The semantics of the program of Listing 1.3 is the same of the previous sections' ones, even though operationally it substantially differs. In fact, in this new version, computation is performed simultaneously on multiple distributed nodes. Communication is achieved through rule `r4` which, informally, specifies that a path from a generic node A to node C exists if there is a link from A to another node B and this last knows that a path exists from B to C.

3.1 Local State, Global State and Runs

In every point in time, each node is in some particular *local state* encapsulating all the information the node is in possess. The local state s_i of a node $i \in N$ can then be defined as a tuple $(\Pi_i, \mathcal{I}_i)$ where Π_i is the finite set of rules composing node i's program, and $\mathcal{I}_i \subseteq \mathbf{I}[n]_i$ is a set of facts belonging to node i. We define the *global state* of a distributed system as a tuple $(s_1, ..., s_n)$ where s_i is the node i's state. We define how global states may change over time through the notion of *run*, which binds (real) time values to global states. If we assume time values to be isomorphic to the set of natural numbers, we can define the function $r : \mathbb{N} \to \mathcal{G}$ where $\mathcal{G} = \{S_1 \times ... \times S_n\}$ with S_i be the set of possible local states for node $i \in N$. We refer to the a tuple (r, t) consisting of a run r and a time t as a *point*. If $r(t) = (s_1, ..., s_n)$ is the global state at point (r, t), we define $r_i(t) = s_i$

[7]. A system may have many possible runs, indicating all the possible ways the global state of the system can evolve. We define a *system* as a set of runs. Using this definition we are able to deal with a system not as a collection of interacting nodes but, instead, directly modeling its behavior, abstracting away many low level details. We think that this approach is particularly important in our scope of maintaing in our language an high level of declarativity.

4 Reasoning about Knowledge in Distributed Systems

In the model we have developed so far, all computations that a node can accomplish are consequences of its local state. If we consider two runs of a system, with global states respectively $g = (s_1, ..., s_n)$ and $g' = (s'_1, ..., s'_n)$, g and g' are *indistinguishable* for node i, and we will write $g \sim_i g'$ if i has the same local state both in g and g', i.e. $s_i = s'_i$. It has been shown in [7] that a system $\mathcal{S}$ can be viewed as a *Kripke frame.* A Kripke frame is a tuple $\mathcal{F} = (W, \mathcal{K}_1, ..., \mathcal{K}_n)$ where W is a non empty set of *possible worlds* (in our case a set of possible global states) and $\mathcal{K}_i$ with $i \in N$ is a binary relation in $W \times W$ which is intended to capture the *accessibility relation* according to node i: this is, $(w, u) \in \mathcal{K}_i$ if node i consider world u possible given its information in world w. Or, in other words, we want $\mathcal{K}$ to be equivalent to the $\sim$ relation, therefore maintaining the intuition that a node i considers u possible in global state w if they are indistinguishable, i.e., in both global states, node i has the same local state. In order to model this situation, $\mathcal{K}$ must be an *equivalence relation* on $W \times W$.

To map each rule and fact to the global states in which they are true, we define an *interpreted system Γ* as the tuple $(\mathcal{S}, \pi)$ with $\mathcal{S}$ a system over a set of global states $\mathcal{G}$ and π an interpretation function which maps first-order clauses to global states [7]. More formally, we build a structure over the Kripke frame $\mathcal{F}$ in order to map each program Π_i and each ground atom in $\mathcal{I}_i$ to the possible worlds in which they are true. To reach this goal, we define a *Kripke structure* $\mathcal{M} = (\mathcal{F}, U, \pi)$ where $\mathcal{F}$ is a Kripke frame, U is the *Herbrand Universe*, π is a function which maps every possible world to a *Herbrant interpretation* over first-order clauses $\Sigma_{\Pi,\mathbf{I}}$ associated with the rules of the program Π and the input instance $\mathbf{I}$, and $\Pi = \bigcup_{i=1}^{n} \Pi_i$, $\mathbf{I} = \bigcup_{i=1}^{n} \mathbf{I}_i[0]$. To be precise, $\Sigma_{\Pi,\mathbf{I}}$ can be constructed starting from the program Π and translating each rule $\rho \in \Pi$ in its first-order *Horn clause* form. This process creates the set of sentences Σ_{Π}. To get the logical theory $\Sigma_{\Pi,\mathbf{I}}$, starting from Σ_{Π} we add one sentence $R(\bar{u})$ for each fact $R(\bar{u})$ in the instance [2,11]. A valuation v on $\mathcal{M}$ is now a function that assign to each variable a value in U. In our settings both the interpretation and the variables valuation are fixed. This means that $v(x)$ is independent of the state, and a constant c has the same meaning in every state in which exists. Thus, constants and relation symbols in our settings are *rigid designators* [7,14]. Given a Kripke structure $\mathcal{M}$, a world $w \in W$ and a valuation v on $\mathcal{M}$, the *satisfaction relation* $(\mathcal{M}, w, v) \models \psi$ for a formula $\psi \in \Sigma_{\Pi,\mathbf{I}}$ is:

- $(M, w, v) \models R(t_1, ..., t_n)$ with $n = \mathit{arity}(R)$, iff $(v(t_1), ..., v(t_n)) \in \pi(w)(R)$
- $(M, w, v) \models \neg\psi$ iif $(M, w, v) \models \psi$

- $(M, w, v) \models \psi \wedge \phi$ iff $(M, w, v) \models \psi$ and $(M, w, v) \models \phi$
- $(M, w, v) \models \forall \psi$ iif $(M, w, v[x/a]) \models \psi$ for every $a \in U$ with $v[x/a]$ be a substitution of x with a constant a

We use $(M, w) \models \psi$ to denote that $(M, w, v) \models \psi$ for every valuation v. It could be also interesting to know not only whether certain formula ψ is true in a certain world, but also the formulas that are true in all the worlds of W. In particular, a formula ψ is *valid* in a structure M, and we write $M \models \psi$, if $(M, w) \models \psi$ for every world w in W. We say that ψ is valid, and write $\models \psi$, if ψ is valid in all structures. We now introduce the modal operator K_i in order to express what a node i "*knows*", namely which of the sentences in $\Sigma_{\Pi,\mathbf{I}}$ are known by the node i. Given $\psi \in \Sigma_{\Pi,\mathbf{I}}$, a world w in the Kripke structure $\mathcal{M}$, the node i knows ψ - we will write $K_i\psi$ - in world w if ψ is true in all the worlds that i considers possible in w [7]. Formally:

$$(\mathcal{M}, w, v) \models K_i\psi \text{ iff for all } w \text{ such that } (w, u) \in \mathcal{K}_i$$

This definition of knowledge has the following valid properties that are called *S5*:

1. *Distributed Axiom*: $\models (K_i\psi \wedge K_i(\psi \rightarrow \phi)) \rightarrow K_i\phi$
2. *Knowledge Generalization Rule*: For all structures M, if $M \models \psi$ then $M \models K_i\psi$
3. *Truth Axiom*: $\models K_i\psi \rightarrow \psi$
4. *Positive Introspection Axiom*: $\models K_i\psi \rightarrow K_iK_i\psi$
5. *Negative Introspection Axiom*: $\models \neg K_i\psi \rightarrow K_i\neg K_i\psi$

Informally, the first axiom allows us to distribute the epistemic operator K_i over implication; the knowledge generalization rule instead says that if ψ is valid, then so is $K_i\psi$. This rule differ from the formula $\psi \rightarrow K_i\psi$, in the sense that the latter tells that if ψ is true, then node i knows it, but a node does not necessarily know all things that are true. Even though a process may not know all facts that are true, axiom 3 says that if it knows a fact, then it is true. The last two properties say that nodes can do introspection regarding their knowledge: they know what they know and what they do not know [7].

4.1 Incorporating Knowledge: Knowlog$^{\boldsymbol{K}}$

In the previous section we described how knowledge assumptions can be expressed using first-order Horn clauses representing our program. We can now move back to the rule form. We use symbols $\square$ and $\boxdot$ to denote a (possibly empty) sequence of modal operators K_i, with i specifying a node identifier. Given a sentence in the modal Horn clause form, we use the following statement to express it in a rule form:

$$\square(H \leftarrow B_1, ..., B_n, \neg C_1, ..., \neg C_m) \tag{1}$$

with $n, m \geq 0$ and each positive literal in the form $\boxdot R$, while negative literals are in the form $\square \boxdot R$.

Definition 1. *The modal context $\Box$ is the sequence - with the maximum length of one - of modal operators K appearing in front of a rule.*

We put some restriction on the sequence of operators permitted in $\boxdot$.

Definition 2. *Given a (possibly empty) sequence of operators $\boxdot$, we say that $\boxdot$ is in restricted form if it does not contain K_iK_i subsequences.*

Definition 3. *A Knowlog^K program is a set of rules in the form (1), containing only (possibly empty) sequences of modal operators in the restricted form and where the subscript i of each modal operator K_i can be a constant or a variable.*

Informally speaking, given a Knowlog^K program, with the modal context we are able to assign to each node the rules the node is responsible for, while atoms and facts residing in the node i are in the form $K_i \boxdot R$. In order to specify how communication is achieved we define communication rules as follows:

Definition 4. *A communication rule in Knowlog^K is a rule where no modal context is set and body atoms have the form $K_i \boxdot R$ - they are all prefixed with a modal operators pointing to the same node - while the head atom has the form $K_j \boxdot R'$, with $i \neq j$.*

In this way, we are able to abstract away all the low level details about how information is exchanged, leaving to the programmer just the task to specify *what* a node should know, and not *how*.

The Two-Phase-Commit Protocol. Inspired by [4], we implemented the two-phase-commit protocol (2PC) using the epistemic operator K. 2PC is used to execute distributed transaction and it is divided in two phases: in the first phase, called the *voting phase*, a coordinator node submits to all the transaction's participants the willingness to perform a distributed commit. Consequently, each participant sends a vote to the coordinator, expressing its intention (a *yes vote* in case it is ready, a *no vote* otherwise). In the second phase - namely the *decision phase* - the coordinator collects all votes and decides if performing global *commit* or *abort*. The decision is then issued to the participants which act accordingly. In the 2PC implementation of Listing 1.4, we assume that our system is composed by three nodes: one coordinator and two participants. We considerably simplify the 2PC protocol by disregarding failures and timeouts actions, since our goal is not an exhaustive exposition of the 2PC. In addition, we employ some syntactic sugar to have a more clean code: we omit the modal context in each rule, and instead we group rules in *programs* identified by a name and the identifier of the node where the program should be installed. If the program must be installed on multiple nodes, we permit to specify, as location, a relation ranging over node identifiers.

```
    \\Initialization at coordinator
    #Program Initialization @C
r1:   transaction(Tx_id,State)@next:-transaction(Tx_id,State),
        ¬K_Cdel_transaction(Tx_id,State).
```

```
r2:   log(Tx_id,State)@next:-log(Tx_id,State).
r3:   part_cnt(count<N>):-participants(N).
r4:   transaction(Tx_id,State):-log(Tx_id,State).
r5:   participants(P1).
r6:   participants(P2).

    \\Initialization at participants
    #Program Initialization @participants
r7:   transaction(Tx_id,State)@next:-transaction(Tx_id,State),
        ¬K_Cdel_transaction(Tx_id,State).
r8:   log(Tx_id,State)@next:-log(Tx_id,State).

    \\Decision Phase at coordinator
    #Program DecisionPhase @C
r9:   yes_cnt(Tx_id,count<Part>):-vote(Vote,Tx_id,Part),Vote == "yes").
r10:  log(Tx_id,"commit")@next:-part_cnt(C),yes_cnt(Tx_id,C1),C==C1,
         State=="vote-req",transaction(Tx_id,State).
r11:  log(Tx_id,"abort"):-vote(Vote,Tx_id,Part),Vote == "no",
         transaction(Tx_id,State),State =="vote-req".

    \\Voting Phase at participants
    #Program VotingPhase @participants
r12:  log(Tx_id,"prepare"):-State=="vote-req",K_ctransaction(Tx_id,State).
r13:  log("abort",Tx_id):-log(Tx_id,State),State=="prepare",db_status(Vote),
         Vote=="no".

    \\Decision Phase at participants
    #Program DecisionPhase @participants
r14:  log(Tx_id,"commit"):-log(Tx_id,State_l),State_l=="prepare",
         State_t=="commit",K_ctransaction(Tx_id,State_t).
r15:  log(Tx_id,"abort"):-log(Tx_id,State_l),State_l=="prepare",
         State_t=="abort",K_ctransaction(Tx_id,State_t).

    \\Communication
r16:K_Xtransaction(Tx_id, State):-K_Cparticipants(@X),K_Ctransaction(Tx_id,State).
r17: K_Cvote(Vote,Tx_id,"sub1"):-K_P1log(Tx_id,State),State=="prepare",
       K_P1db_status(Vote).
r18: K_Cvote(Vote,Tx_id,"sub2"):-K_P2log(Tx_id,State),State=="prepare",
       K_P2db_status(Vote).
```

Listing 1.4. Two Phase Commit Protocol

4.2 Incorporating Higher Levels of Knowledge: Knowlog

Rules `r10`, `r11` of Listing 1.4 indicates that each participant, once written `"prepare"` in the log, sends to the coordinator its status together with its identifier. Then, the votes are aggregated at coordinator side and the final decision is issued. This process can also be seen in another way: the coordinator node will deliver `"commit"` for transaction `Tx_id` if it knows that every participant knows the fact `vote("yes",Tx_id)`; `"abort"` otherwise. Consequently, the decision phase at the coordinator will become as in Listing 1.5.

```
r10_a: log(Tx_id,"commit")@next:-K_s1vote("yes",Tx_id),K_s2vote("yes",Tx_id),
       transaction(Tx_id,State),State=="vote-req".
r11_a: log(Tx_id,"abort")@next:-K_xvote(Vote,Tx_id),Vote=="no",
       participants(X),transaction(Tx_id,State),State=="vote-req".
```

Listing 1.5. 2PC coordinator's program revisited

We chose this new revisited form described in Listing 1.5 for a purpose, in fact we want to show that other types of knowledge could be appealing to be incorporated in our language. For example, a node could be interested not only in knowing some fact, but also in knowing if *every* node in the system know something (rule `r1`) or a new information is derived by *combining* the knowledge belonging to different nodes (rule `r2`). The discussion about higher levels of knowledge can be started pointing out that both rules in Listing 1.5 have a common denominator: i.e., the notion of knowledge inside a *group of nodes*. In fact, both the above mentioned rules are used to declare which is the state of knowledge inside the group of participant nodes. Thus, given a non empty set of nodes G, we can hence augment Knowlog^K with modal operators E_G and D_G, which respectively are informally stating that "every node in the group G knows" and "it is distributed knowledge among the nodes in G". From a more operational point of view, $E_G\psi$ states that the sentence ψ is replicated in all the nodes belonging to G, while $D_G\psi$ states that ψ is fragmented among the nodes in G. We can easily extend the definition of satisfiability to handle the two new types of knowledge just introduced. $E_G\psi$ is true exactly if everyone in the group G know ψ:

$$(M, w, v) \models E_G\psi \text{ iff } (M, w, v) \models K_i\psi \text{ for all } i \in G$$

On the other side, a group G has distributed knowledge of ψ if the coalesced knowledge of the members of G implies ψ. This is accomplished by eliminating all worlds that some agent in G considers impossible:

$$(M, w, v) \models D_G\psi \text{ iff } (M, u, v) \models \psi \text{ for all } t \text{ that are } (w, u) \in \bigcap_{i \in G} R_i$$

Not surprisingly, for both E_G and D_G axioms analogous to the Knowledge Axiom, Distribution Axiom, Positive Introspection Axiom, and Negative Introspection Axiom all hold. In addition, distributed knowledge of a group of size one is the same as knowledge, so if G contains only one node i [7]:

$$\models D_{\{i\}}\psi \leftrightarrow K_i\psi$$

and the larger the subgroup, the greater the distributed knowledge of that subgroup is :

$$\models D_G\psi \rightarrow D_{G'}\psi \text{ if } G \subseteq G'$$

Before reformulating the definition of Knowlog and rewriting Listing 1.5 using the new operators, we first update the definition 2 in order to incorporate the new operators D_G and E_G.

Definition 5. *Given a (possibly empty) sequence of operators $\boxdot$, we say that $\boxdot$ is in restricted form if it does not contain either K_iK_i, D_GD_G or $E_{G'}E_{G'}$ subsequences, with i specifying a node identifier, G a group of nodes $G \subseteq N$ and G' is singleton.*

Definition 6. *A Knowlog program is a Knowlog^K program augmented with operators E_G and D_G, with $G \subseteq N$ and where the sequence of operators $\boxdot$ is in the restrict form of definition 5.*

Listing 1.6 shows Knowlog version of Listing 1.5.

```
r10_b: log(Tx_id,"commit")@next:-Exvote("yes",Tx_id),participants(X),
        transaction(Tx_id,State),State=="vote-req".
r11_b: log(Tx_id,"abort")@next:-Dxvote(Vote,Tx_id),Vote=="no",
        participants(X),transaction(Tx_id,State),State=="vote-req".
```

Listing 1.6. Knowlog 2PC coordinator's program

Operationally, E_G is used when we want that a fact, to be considered true, is correctly replicated in every node $i \in G$. On the other side, D_G is employed when facts that are fragmented inside multiple relations distributed in the node enclosed in G must be assembled in one place for computation. Employing the E_G operator in the head of communication rules we are able to express the sending of a message to multiple destinations, therefore emulating the multicast primitive behavior.

Definition 7. *A communication rule in Knowlog is a Knowlog^K communication rule where the body may contains atoms both in the form $D_G \boxdot R$ and $E_G \boxdot R$, while head atoms may also have the form $E_G \boxdot R$, with R a relation and $G \subseteq N$.*

As a future work we will investigate how the operator D can be used in front of communication rule to implement data dissemination [15].

5 Knowlog Semantics

The first step towards the definition of the Knowlog semantics will be the specification of the *reified* version of Knowlog. For this purpose, we augment **dom** with a new set of constants $\triangle$ which will encompass the modal operators symbols. We also assume a new set of variables O that will range over the just defined set of modal operator elements. We then construct $\mathbf{R}^{TK}$ adding to each relation $R \in \mathbf{R}^T$ a new term called *knowledge accumulator* and a new set of build-in relations `K`, `D`, `E` and $\oplus$. A tuple over the $\mathbf{R}^{TK}$ schema will have the form $(k, t_1, ..., t_n, s)$ where $k \in O \cup \triangle$ identify the knowledge accumulator term, $s \in S \cup \mathbb{N}$ and $t_1, ..., t_n \in \mathbf{var} \cup \mathbf{dom}$. Conversely, a tuple over *adb* relations, i.e., relations in the head of at least one communication rule, will have the form $(k, l, t_1, ..., t_n, s)$, with l the location specifier term. If the knowledge operator used in front of a non-*adb* relation is a constant, i.e. `KsKRinput("value")`, the reified version will be `input(Y,"value",n),Y = K_S ⊕ K_R` for example at time-step n. The operator $\oplus$ is hence employed to concatenate epistemic operators.

Instead, in case the operator employes a variable to identify a particular node (as in rule `r10_a` of Listing 1.5) or a set of nodes (as shown in rule `r10_b` of Listing 1.6), we need to introduce in $\mathbf{R}^{TK}$ relations `K(X,Y)`, `E(<X>,Y)`, and `D(<X>,Y)` in order to help us in the effort of building the knowledge accumulator term. The first term of the `K` relation is a node identifier $i \in N$ (respectively a set of node identifiers for relations `E`, and `D`) and the `Y` term is a value in $\triangle$ determined by the relation name and the node identifier(s). So for example, the reified version of `E_X vote("yes",Tx_id),participants(X)` will be `E(<X>,Y),vote(Y,"yes",Tx_id,n),participants(X)`.

For what concern communication rules, the process is the same as above, but this time we have to fill also the location specifier field of the head-relation. To accomplish this, if the head relation $R \in adb$ is in the form $K_i\Box R(t_1,...,t_n)$, the reified version will be $R(K_i\Box, i, t_1, ..., t_n, s)$. On the other side, if the head relation is in the form $E_G\Box R(t_1,...,t_n)$ the rule that includes it, is rewritten in m rules, one for each node identifier in G, and each of them having the head relation in the form $K_i\Box R(t_1,...,t_n)$ with $i \in G$. The reified version is then computed as described above. Using this semantics, nodes are able to communicate using the mechanism described in Section 3.

5.1 Operational Semantics

Given as input a Knowlog program Π in the reified version, first Π is separated in two subset: Π^l containing local rules (informally these are the rules that the local nodes i knows) and Π^r containing rules that must be installed in remote nodes. These last are rules having as a modal context K_j with $j, i \in N$, i the identifier of the local node and $i \neq j$. Following the delegation approach illustrated in [1] given a program Π_i local to node $i \in N$, we denote with Π^r_{ij} the remote rules in i related to node j and with $\Pi_j \leftarrow \Pi^r_{ij}$ the action of installing Π^r_{ij} in j's program. For what concern the evaluation of local rules, we partition the local program Π^l in inductive and deductive rule sets, respectively Π^i and Π^d. Then a pre-processing step orders the deductive rules following the dependency graph stratification. After this pre-processing step, the model $\mathcal{M}_\Pi$ is computed. In order to evaluate the stratified program Π^d we use the semi-naive algorithm depicted in [16]. To correctly evaluate rules and facts with modal operators, the *saturation (Sat)* and the *normalization* (*Norm*) operators are used to assist the immediate consequence operators T_{Π^d} [14]. This because Knowlog facts and rules are labeled by modal operators and therefore the immediate consequence operator must be enhanced in order to be employed in our context. More precisely, given a Knowlog instance $\mathbf{I}$ as input, $Sat(\mathbf{I})$ saturate facts in the instances following operators' properties. Lastly, the *Norm* operator converts to restricted form the modal operators in $T_{\Pi^d}(Sat(\mathbf{I}))$.

6 Conclusion and Future Work

We have presented Knowlog, a programming language for distributed systems based on Datalog$^\neg$ leveraged with a notion of time and modal operators. We described the communication and knowledge model behind Knowlog and we introduce as example, an implementation of the two phase commit protocol. As a future work, we will incorporate in Knowlog the *common knowledge* operator that has been proven to be linked to concepts such as coordination, agreement and consistency [7]. The successive step will be the definition in Knowlog of weaken forms of common knowledge such as *eventual* common knowledge.

References

1. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: PODS 2011, pp. 293–304. ACM, New York (2011)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.R.: Consistency analysis in bloom: a calm and collected approach. In: CIDR, pp. 249–260 (2011)
4. Alvaro, P., Condie, T., Conway, N., Hellerstein, J.M., Sears, R.: I do declare: consensus in a logic language. Operating Systems Review 43(4), 25–30 (2009)
5. Birman, K., Chockler, G., van Renesse, R.: Toward a cloud computing research agenda. SIGACT News 40(2), 68–80 (2009)
6. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer, Heidelberg (2011)
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (2003)
8. Furukawa, K.: Logic programming as the integrator of the fifth generation computer systems project. Commun. ACM 35(3), 82–92 (1992)
9. Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec. 39, 5–19 (2010)
10. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16, 872–923 (1994)
11. Lloyd, J.: Foundations of logic programming. Symbolic Computation: Artificial Intelligence. Springer (1987)
12. Loo, B.T., Condie, T., Garofalakis, M., Gay, et al.: Declarative networking: language, execution and optimization. In: SIGMOD 2006, pp. 97–108. ACM, New York (2006)
13. Ludäscher, B.: Integration of Active and Deductive Database Rules. DISDBIS, vol. 45. Infix Verlag, St. Augustin (1998)
14. Nguyen, L.A.: Foundations of modal deductive databases. Fundam. Inf. 79, 85–135 (2007)
15. Suri, N., Benincasa, G., Choy, S., Formaggi, S., Gilioli, M., Interlandi, M., Rota, S.: Disservice: a peer-to-peer disruption tolerant dissemination service. In: Proceedings of the 28th IEEE Conference on Military Communications, MILCOM 2009, pp. 2514–2521. IEEE Press, Piscataway (2009)
16. Zaniolo, C.: Advanced database systems. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers (1997)