A Distributed Tabling Algorithm for Rule Based Policy Systems

Miguel Alves, Carlos Viegas Damásio
CENTRIA, Universidade Nova de Lisboa
Lisbon, Portugal
Email: cd@di.fct.unl.pt,mba@estg.ipvc.pt

Wolfgang Nejdl, Daniel Olmedilla
L3S Research Center and Hannover University
Hannover, Germany
Email: {nejdl,olmedilla}@l3s.de

***Abstract*— Distributed Peer-to-Peer and Grid infrastructure require distributed access control mechanisms. These mechanisms can be implemented in distributed trust management infrastructures and usually require reasoning on more than one peer, as soon as authority is delegated or requests involve several authorities. Building on previous work of the authors which formalized such a distributed trust management infrastructure based on distributed logic programs, we describe in this paper how reasoning can be implemented as distributed logic evaluation and how loops during this evaluation can be handled with. Our solution is based on a loop tolerant distributed tabling algorithm which includes in the process protection of sensitive policies and generation of proofs without increasing the complexity of the system.**

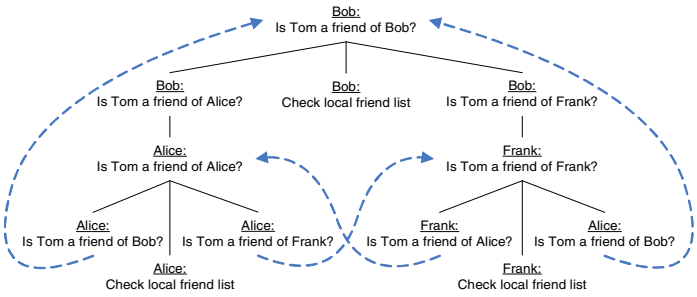

Fig. 1. SLD-like tree for Bob's example

I. Introduction

Access to sensitive information involves a process in which authentication and authorization play a crucial role. Authentication allows to identify which entity is requesting access and authorization is the process of checking whether that entity is allowed to get access to the requested resource. For example, a user authenticates into a system by means of the login mechanism (providing user and password) and the system consults its permission table before the user is granted access to a resource (authorization).

However, P2P and virtual organizations require more complex decisions than just matching identities (or groups) with a table of permissions. Due to the need to interact with entities with whom no previous transaction has been made, we require statements like "Grant access if requester provides a valid credit card and the resource is available at the moment". Logic based policy languages provide well-defined semantics [1] and have been chosen in the last years as an appealing solution to specify this kind of statements (e.g., [2], [3], [4], [5]) in order to allow more powerful authorization mechanisms. In addition, real-world policies [6] tend to be as complex as any piece of software when written down in detail; getting a policy right is as hard as getting a piece of software correct, and maintaining a large number of them is only harder. There exist some approaches that help local administrators in the specification of such policies (e.g., using ontologies [7]) and validating them. These tools typically guide the user on the specification of policies, ensuring that they are syntactically and semantically correct at writting time (static checking). In order to dynamically validate policies or to avoid infinite evaluation, techniques like model checking or tabling are applied. However, all these tools and techniques help in a local environment where the whole set of policies is known, and they cannot be used in a distributed environment. In particular, problems like loops among delegation rules cannot be detected.

From the point of view of declarative policy specification, loops may easily occur and should not be considered as errors. However, if not handled accordingly, they may end up in non-terminating evaluation. For example, suppose Bob wants to share his pictures with his friends. Bob protects his pictures with a policy that states "only my friends may access my pictures". However, he does not only have a list of his friends but also include that "all Alice's and Frank's friends are also my friends". Suppose Alice and Frank have a similar policy in which their friends list includes also all other's friends. Given that setup, imagine that Tom requests access to Bob's pictures. Bob's security agent (SA) checks that Tom is not his friend but, since he might be a friend of Alice or Frank, it asks their security agents. Now, Alice's SA checks locally if Tom is her friend, but some of her policies say that any friend of Bob or Frank is also her friend and therefore, it asks Bob's and Frank's SA. In parallel Frank's SA evaluates its policies and produces a similar situation asking back Bob's and Alice's SA, and so on. As the reader can see, if not detected and handled appropriately, the evaluation of this request would never terminate (see figure 1), even if answers exist. Real policies, including for instance business rules, are much more complex and higher in number than the ones in this example (and they are typically not under control of a single person) what increases the risk of loops and not termination during

dynamic policy evaluation. In addition, it is important to note that solutions like replication of information (e.g., sharing the lists of friends) or including in each communication all the previous requests made (in order to detect repetitions in the chain of requests) may not be possible due to privacy or even scalability issues (as it happens with search where in many cases federation is preferred over replication of resources).

In this paper we present a distributed tabling algorithm which can handle safely mutual recursive dependencies (loops) in distributed environments like P2P networks. Due to the security context, other aspects like private and public policies and proof generation must be taken into account. This algorithm allows a system to evaluate arbitrary authorizations with respect to arbitrary policies, without increasing the overall load of the network (the algorithm is polynomial), return the right answers and detect termination even when loops exist.

This paper is structured as follows: Section II briefly introduces the context of policy-driven negotiation for access control as well as the PEERTRUST language. An extended example is presented in Section III, and the problems of ordinary SLD-resolution appropriately illustrated. The distributed tabling algorithm and a detailed example of its application are described in section IV. Section V discusses related work and, finally, section VI presents the conclusions and further work.

II. POLICY-DRIVEN NEGOTIATION AND PEERTRUST

Open distributed environments like the World Wide Web or P2P networks offer easy sharing of information, but provide few options for the protection of sensitive information and other sensitive resources. Typically, this protection is based on the assumption that a requester is already known by the server (e.g., by means of previous registration and user/password authentication mechanisms). This way, the server is able to map the identity of the requester into a permissions table in order to grant or deny access to a resource.

Nowadays, due to the success of the WWW and P2P networks and therefore to the big amount of potential users a server might have, maintaining a table of authorizations based on identities is no longer desirable. Specifically, the Web provides an environment where parties may make connections and interact without being previously known to each other. In many cases, before any meaningful interaction starts, a certain level of trust must be established from scratch. Generally, trust is established through exchange of information between the two parties. Since neither party is known to the other, this trust establishment process should be bi-directional: both parties may have sensitive information that they are reluctant to disclose until the other party has proved to be trustworthy at a certain level.

To make controlled sharing of resources easy in such an environment, parties will need software that automates the process of iteratively establishing bilateral trust based on the parties' access control policies, i.e., *trust negotiation* [8] software. In this software, every party can define access control and release policies (*policies*, for short) to control outsiders' access to their sensitive resources. The policies describe what properties a party must demonstrate (e.g., ownership of a driver's license, membership of a specific project or holding of a citizen id of the European Union) in order to gain access to a resource.

There exist several requirements a policy language must satisfy [1]. In particular, an entity can delegate some of its authority to another entity. For example, the manager of a company can delegate its authority to buy new equipment to one of its employees or a system may delegate the validation of a student id to the university. However, as this process must be automated, *delegation of authority* [9] brings some new challenges that must be solved like the existence of loops in the policies which could bring a negotiation into an infinite evaluation.

PEERTRUST's language [3], [10] is based on first order Horn rules (definite Horn clauses), i.e., rules of the form

$$lit_0 \leftarrow lit_1, \ldots, lit_n$$

where each lit_i is a positive literal $P_j(t_1, \ldots, t_n)$, P_j is a predicate symbol, and the t_i are the arguments of this predicate. Each t_i is either a variable or a constant[1]. The head of a rule is lit_0, and its body is the set of lit_i. The body of a rule can be empty. The following policy can be used to specify that access to resources are granted to clients:

access(Resource) $ Requester ← client(Requester).

The ability to reason about statements made by other peers is central to trust negotiation. For example, in our initial example, Bob does not have information about Alice's friends and therefore must ask her to find out if Tom is her friend. One can think of this as a case of Bob *delegating evaluation* of the query "Is Tom Alice's friend?" to Alice. To express delegation of evaluation to another peer, we extend each literal lit_i with an additional *Authority* argument, " lit_i @ *Authority* " where *Authority* specifies the peer who is responsible for evaluating lit_i or has the authority to evaluate lit_i. The *Authority* argument can be a nested term containing a sequence of authorities, which are then evaluated starting at the outermost layer

As mentioned earlier, Bob may need a way of referring to the peer who asked a particular query (e.g., Tom). We accomplish this by including a *Requester* argument in literals, so that we now have literals of the form

lit_i @ *Issuer $ Requester.*

Using the *Issuer* and *Requester* arguments, we can delegate evaluation of literals to other parties and also express interactions and the corresponding negotiation process between parties. For instance, the policy for Bob is specified as follows

Bob Policy:

access(Picture) $ Requester ← friend(Requester).

[1]The original PEERTRUST language allows also complex terms, i.e. a function symbol and its arguments, which are themselves terms. These are not allowed here because termination of queries with complex terms is an undecidable problem.

friend(Name) $ Requester ← isMyFriend(Name).
friend(Name) $ Requester ← friend(Name) @ "Alice".

and Alice's policy only differs on the last policy rule where she delegates to Bob instead of herself.

III. Problem Statement

As shown in section I, the trust negotiation would get into an infinite evaluation due to a loop. It is desirable not only to detect the loop but also to allow the algorithm to find the right answers. In our example, if Tom was a friend of Alice, the evaluation would not terminate due to the existence of an infinite branch in the SLD-tree. Furthermore, special mechanisms may be applied in case loops are detected. For example, imagine two CIA agents with the policy "I show you my CIA badge if you show me yours first" [11]. None of them is going to show her CIA badge first and therefore, even though their negotiation could succeed it does not due to the existence of a deadlock. [11] describes an algorithm to solve this deadlock, which could be applied in case the loop is detected.

In addition, in trust negotiation, we must be able to distinguish between predicates that can be queried by external parties, and ones that cannot—analogous to the public and private procedures in object-oriented languages. These rules must also be kept hidden while generating proofs to be disclosed to other peers.

Additionally, the rules that the peer defines on its own are its *local* rules. A peer may also have copies of rules defined by other peers, and it may use these rules in its proofs in certain situations. In those cases, a signature is used to verify that the issuer really did issue the rule. Because of simplicity, the examples in this paper include only local and public rules.

In the rest of this section we introduce how the distributed evaluation of logic programs is used for access control, and describe an extended example which illustrates the PeerTrust language as well as the problems with the ordinary SLD-based proof-procedures of logic programming.

A. Distributed Evaluation for Access Control

Delegation of authority is widely used in access control. For example, a system may delegate the validation of a student id to the university or one company in a virtual organization might delegate on other partners whether a requester is a client or not. Therefore, the ability to reason about statements made by other parties is central to trust negotiation.

This scenario can be seen as the evaluation of distributed logic programs. The semantics of the PeerTrust language is an extension of that of SD3 [12] to allow the set of all PeerTrust programs to be viewed as a single global logic program. We refer the interested reader to [12], [3] for semantic details.

B. Running Scenario

A Library provides free access to its documents, and it makes available a public search mechanism based on keywords, returning URLs for further navigation. Moreover, it has an agreement with an international Publishing company (Pub) that allows redirection of requests from the Library to its own search service. This policy is specified as follows:

Library Policy:

```
getURL(Key,URL) ← libraryDoc(Key,URL).
getURL(Key,URL) ← getURL(Key,URL) @ "Pub".
libraryDoc(p2p,'http://library.org/url1').
```

The search engine of the Publisher is connected to other specific search engines. It also provides access to its own publications under a subscription. All topics are classified with respect to an access level: free, basic or full.

Publisher Policy:

```
getURL(Key,URL) ←
    searchEngine(Loc), getURL(Key,URL) @ Loc.
getURL(Key,URL) $ Req ←
    Req ≠ "Pub",
    topicProvided(Key,Level),
    accLevel(Req,Level),
    getURL(Key,URL).
getURL(Key,URL) $ "Pub" ←
    companyPublication(Key,URL).

searchEngine("Library").

topicProvided(p2p,free).
topicProvided(music,basic).

companyPublication(p2p,'http://my.com/url1' ).
companyPublication(p2p,'http://my.com/url2' ).
```

The first rule of `getURL/2` allows any user to use the search service of the associated search engines. The next rule, provides the access control policy to documents of Publisher, where it should be noticed in particular the last call in the body to `getURL/2`: since this rule is to be executed by the Publisher peer, then the requester is changed to "Pub", and no longer is Req. The last rule is to be executed solely by a call from "Pub" itself. The access level is controlled by the following rules:

Publisher Policy (cont):

```
accLevel(Req, basic) ← registeredUser(Req) @ "Music".
accLevel(Req, Level) ← hasSubscription(Req,Level).
accLevel(Req, Level) ←
    accLevel(Req,LevelH), accOrder(Level,LevelH).

accOrder(free,basic).
accOrder(basic,full).

hasSubscription( "Library", free ).
hasSubscription( "Music", full).
hasSubscription( "Alice", basic).
hasSubscription( "Bob", full).
```

Fig. 2. An infinite SLD branch

In particular, the publisher allows users of its partner company Music to have basic access to the documents of Publisher.

The Music company makes available to its users, information about the music topic obtained from Publisher. It also provides all services to users of Publisher company which have at least basic access.

Music Policy:

getMusic(URL) $ Req ←
registeredUser(Req),
getDocURL(music,URL) @ “Pub”.

registeredUser(User) $ “Pub” ← registered(User).
registeredUser(User) $ “Music” ← musicUser(User).
registeredUser(User) $ “Music” ←
accLevel(User, basic) @ “Pub”.

musicUser(”Frank”).

Notice that the query `accLevel("Bob",`Lev`)` does not terminate under ordinary SLD-resolution. Since in order to check that Bob has basic access to the Publisher services, then it should be first verified whether it is a registered user of “Music” company. But in order to check that Bob is registered at “Music”, it is also verified that Bob has basic access level to Publisher documents. An infinite branch of the SLD tree generated from the goal can be found in Figure 2, where each goal is attached with the peer and requester of the goal. Other SLD derivations exist, for instance by unifying the top-goal with the second and third rules of the policy rules for Publisher.

Independently of that, the last rule of access level is left recursive, and under ordinary SLD-resolution will not be able to answer any query. Furthermore, it is also clear that there are multiple dependencies between the queries getURL at Librarian and at Publisher, which introduce further problems for the SLD proof-procedure.

IV. Tabling Algorithm and Distributed Evaluation

Tabling is a technique for goal-oriented evaluation of logic programs by storing computed answers in tables, combining the characteristics of the traditional top-down strategy with those of the bottom-up evaluation [13], [14], [15], [16], [17]. Tabling is adequate for the evaluation of recursive queries with repeated calls in programs, and for cases where queries may not terminate under the ordinary top-down SLD-based inference mechanisms of the Prolog like logic programming languages, like the ones shown in the previous section. In the context of evaluation for distributed rule based policy systems this is particularly relevant since it is not known *a priori* if mutual dependencies between goals will exist among peers. In this section we extend a distributed tabling architecture [18] to P2P networks, which is based on the SLG proof procedures [15], [17].

A. Tabling

A tabling algorithm produces a forest of proof-trees, one for each goal executed in the system. Each tree will have an associated table, where the answers (lemmas) produced by that proof-tree are stored. The evaluation of a new goal is started by ordinary resolution with program clauses. As in SLD resolution, a sub-goal is selected for execution in the resolvent. However, instead of resolving again with program clauses, the sub-goal waits for the answers of the corresponding table (which might be created at that time). When a new answer is found, it is stored in the respective proof-tree table and it is propagated to the consumer sub-goals of this table. Since repeated answers are not propagated, looping is avoided and program termination is guaranteed for the Datalog case (no complex terms). The full forest constructed for the query `accLevel("Bob",`Lev`)` started at “Pub” is depicted in Figure 3, where goals are numbered by (a possible) order of creation.

The tabulation algorithm has four basic operations: creation of a new tree, selecting a node for execution, calling a goal, and answer resolution (or lemma resolution). The construction of the forest in Figure 3 starts by creating a proof-tree for the top goal, the node labeled by 1 in the forest. The goal labeling the root node is resolved with each clause that unifies with it; in this case obtaining nodes 2, 3, and 4. The computation proceeds by choosing a node for execution, and selecting a goal in the resolvent (which are underlined in the figure). Suppose node 3 is chosen; since there is only one goal in the node still to be executed, this is the selected one. Now, the proof-procedure starts execution of the goal `hasSubscription("Bob",L1) @ "Pub" $ "Bob"`, and a new tree is created for the goal (numbered 5), with variables renamed apart. The difference to SLD is now evident, since each new goal starts the creation (or reuses) a

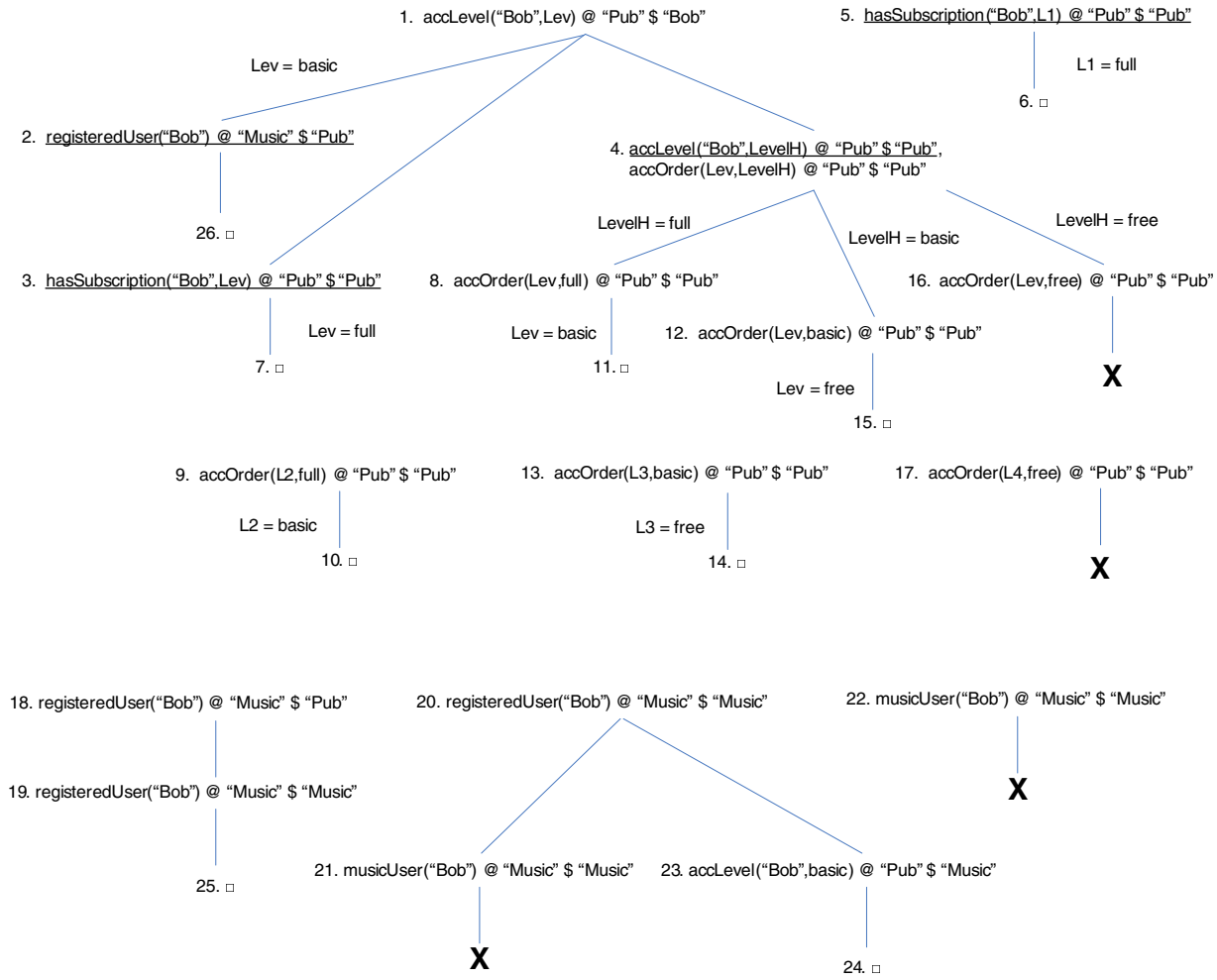

Fig. 3. Full forest constructed during the tabulated evaluation of the query `accLevel("Bob",`Lev`)` at “Pub”

proof-tree, and is not “inline” resolved. The goal at the root, node 5, is resolved with a fact in the program, obtaining the answer (empty) clause at node 6. This answer is resolved with the consuming goal(s), i.e. solely 3. By unifiying the answer `hasSubscription("Bob",full)` of goal 5 with the call at node 3, the first answer is obtained for the top goal, namely `Lev = full`, represented at node 7.

The goal at node 4 is then selected for execution, and the very interesting call `accLevel("Bob",`$LevelH$`)` occurs: this a variant (renaming) of the initial goal; therefore, the tree at node 1 is reused for providing answers to goal 4. This is a property of the tabling algorithm, avoiding the repetitive creation of proof-trees! Since there is already one answer for the top-goal ($full$), it is resolved with goal at 4, originating node 8. This node 8 starts the creation of a new tree (nodes 9 and 10), with a single answer ($basic$). This generates a new answer to the top goal, which is consumed again by node 4. This process further generates nodes 11–17.

Finally, a new proof-tree is created (root at node 18) by choosing node 2 for execution. The computation proceeds as previously, till node 23 is generated. Node 23 is labeled with a goal which is an instance of node 1, but the requester is distinct. By a simple syntactical analysis, it can be seen that the requester of `accLevel/2` is irrelevant, so the answers for tree 1 can be reused. Since it has been shown previously that Bob has basic access (node 11), the tree rooted at 20 succeeds with a single answer. This answer is then consumed by node 19, and propagated to node 2, originating node 26; so another proof granting basic access to Bob has been obtained. However, this answer was already generated and therefore it is not propagated to the consumers. The computation finishes without entering into a loop and obtaining the expected results: Bob has access level full, basic and free.

B. Distribution of the Tabling Algorithm

The challenge is now the adaptation of the previous tabling algorithm to the distributed setting, with the corresponding technological issues properly addressed. In first place, it is assumed the existence of an asynchronous message-passing system, with reliable FIFO channels and unbound capacity. Our basic distributed system requires three types of components:

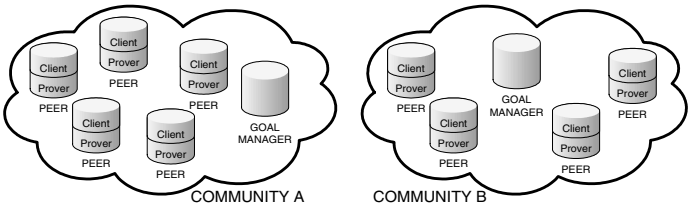

Fig. 4. Example of running architecture

- The goal manager which interfaces a community of peers with the outside world;
- The peer clients which keep the tables and corresponding answers for goal calls, avoiding redundant answers. Simultaneously, they manage the delivery of solutions to the appropriate invoking goals;
- The peer provers which perform the logical expansion operations on the set of active goals, i.e. construct and store the proof-trees;

Nothing prevents the coupling of a peer client with the correspondent peer prover in a single software component, if the protocol described in the rest of this section is followed. All peers in the system should be able to communicate among them (directly or via goal managers). The behavior of each component and the communication protocol is detailed below.

Goal Managers: In a running system, there exists a goal manager for each community of peers, several peer clients, one for each peer, and a prover client for each peer client (as in Figure 4). There are two types of queries that can be executed by the system: call for authorizations, obtaining answers, and requesting the explanation of a computed authorization answer, obtaining program clauses in the proof. The task of a goal manager is to start queries in peers of the corresponding community, and to provide the appropriate answers to the requesters. Its major function is to detect termination of the corresponding queries.

In Figure 5, we depict the several types of messages that the components of the architecture interchange. The dashed arrows between Peer Client and Peer Prover signify that the messages can be sent to the corresponding components of other peer, in the same community or not.

The authorization query is encapsulated in a message *call(Goal @ Peer, N)* where *Goal* is the goal to be executed at *Peer*, and *N* is an external identifier. Notice that an internal identifier is generated and a call message is sent to the appropriate peer, in public mode. The answers are received through *answer(Ans,i)* where *i* is the internally assigned identifier to goal *N*. Proofs are requested with

proof(Goal @ Peer, Ans @ Peer, M)

messages, where answer *Ans @ Peer* should have been received from a previously invoked query *Goal @ Peer*, and *M* is the external identifier for the proof; again an internal identifier is generated in order to avoid potential unintended mixing of external with internal identifiers. The clauses necessary to derive the answer are sent via *explanation(Rule,Id)* messages, one at a time. Done messages are sent to the involved components in order to announce termination of both types of query (c.f. Section IV-C).

Peer Clients: When a peer client receives a call message, it verifies whether the goal is already tabled or not. If the goal is not tabled, a new table for the goal is created and a *c*all message is sent to the prover client with the goal and table identifier, in order to be evaluated. If the goal is already tabled, all the answers in the table are sent to the caller. The peer client must keep the peer names from which it received *c*all message in order to send back every new answer generated (table consumers). There are two modes for goal execution: *public*, for goals that can be shown in proofs, and *internal*, for goals which must be hidden in proofs. For security reasons, goals in different modes are maintained in different tables. An initial query, requested by a goal manager, is always started in public mode.

When a peer client receives a proof message request, it simply forwards it to the prover client which is responsible for producing it. Again, for guaranteeing termination, a proof for a given answer to a goal is asked only once.

Peer Provers: The peer provers perform the logical operations on goals. The execution of a new goal at the prover client is initiated with a call message from its peer client. This goal can either be executed at the current peer, or forwarded to a different one. If a goal is to be executed explicitly at another peer, the prover client first checks whether it has delegated signed rules from that peer, which allow it to execute the goal locally. If not, the goal call is simply forwarded to that peer. Otherwise, in the cases that the goal is to be executed locally, the prover resolves the literal with the policy rules matching it (signed or not) in the program and then selects a literal in each resulting resolvent. Local rules of the peer can be marked public or internal. To avoid unintentional disclosure of private information, public calls must be ground in order to match with internal rules. Public rules can always be used, as well as internal rules called by goals in internal mode.

Upon reception of a matching answer for a given goal, the prover client unifies the answer with the selected literal. If the body is empty, a solution has been found and it is communicated to the peer client which requested the computation. Otherwise, a literal is selected in the resolvent and a new resolvent is created. The prover then sends a call message to its peer client, and waits for the answers. Notice that if the literal is to be executed at a different peer, then the prover later on will forward it, as we've explained before.

The peer prover keeps track of the logical operation which originated a resolvent in the proof-tree, either: clause resolution with an ordinary rule; clause resolution with a signed rule; forwarding a query to a different peer; and answer resolution, keeping a pointer to the parent resolvent. This information is then used to produce the explanation (proof) of an answer. Basically, a peer prover is capable of reconstructing the proofs

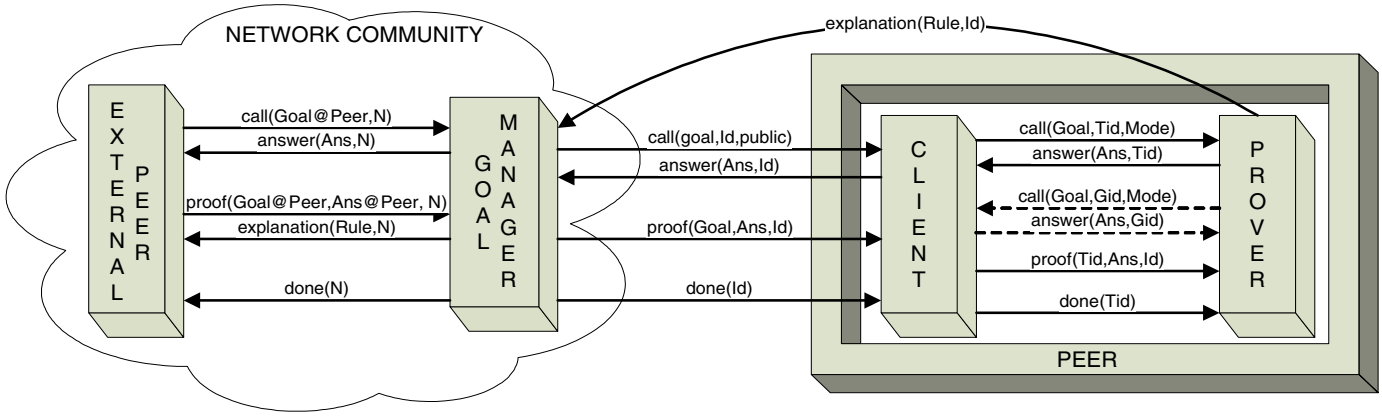

Fig. 5. Messages interaction between the components of the architecture

of answers to calls executed in that peer. If the answer was originated by an internal rule called in public mode, then it simply sends to the goal manager an explanation containing a rule with that answer in the head and body with the special literal *hidden*. Otherwise, it determines the policy rule which originated the answer and sends that rule to the corresponding goal manager. This rule is obtained by traversing the path from the answer to the root of the proof-tree. From this path, the peer prover obtains the sub-goals and corresponding answers that have been used to produce the answer to be explained. The proofs for used answers of sub-goals are then recursively asked to the corresponding peers. Some care must be taken in the implementation in order to avoid repeatedly sending rules and to guarantee termination in the proof generation.

C. Termination Detection

One important aspect of the proposed architecture is the capability to know that the authorization query or that the generation of the corresponding proof is finished. It is fundamental to be able to detect the termination of the distributed computation of each query executed in the system, in order to inform the entity who queried the system that there are no more answers. Also, termination detection allows freeing of allocated structures used in the evaluation of the query.

Several versions of termination detection have been implemented, by adapting the classical Dijkstra-Scholten's diffusing computation and Mattern's credit recover distributed termination detection algorithms (see for instance [19]). In the most general scenario, several peers in different communities can share computations. Whenever a new query is started at a goal manager, a new instance of the termination algorithm is initiated and all messages are tagged with the identifier of this instance. If during the distributed evaluation of query A, a goal calls a non-completed table started by other external query B then query A cannot complete before B is complete. In these situations, we run dynamically Tel's Phase Wave algorithm (see again [19]) among involved goal managers in order to synchronize their termination detection, obtaining an algorithm inspired in the Shavit-Francez decentralized computation termination algorithm, which can also be found in [19]. If two queries do not share incomplete tables, then their termination is detected independently. Notice again that already completed tabled goals can be shared without interference of computations. The advantage of this algorithm is that no propagation of graph dependencies are necessary to ensure termination, and thus no sensitive information is disclosed. However, if two queries share some goal then one of them might end waiting too long for the termination of the other. The number of control messages exchanged is in the worst-case linear in the number of messages of the basic algorithm, and the overhead is small.

We have also developed local termination algorithms that are capable to detect completion (termination) of individual tables. These are too complex to describe here, but the idea is that each call in the system starts a new instance of the termination algorithm. For guaranteeing correctness of the algorithm we use a sophisticated locking mechanism which requires the propagation of goal dependencies in order to avoid deadlock. For a single community of peers this is very appropriate, and empirical evidence suggests that the message complexity overhead is between one and two times the number of calls and answers of the algorithm described in the previous section. This is relatively small, and promises to have good application in P2P networks. However, sending the dependency among goals might not be acceptable for some peers, since it publicizes the peers involved in the computation. We are currently working in variations of the local algorithm with limited propagation of goal dependencies for the general setting.

D. Implementation

The architecture previously described was implemented[2] via Prolog meta-interpreters on top of PVM-Prolog [20], using the XSB Prolog system. The PVM-Prolog (PPVM) library is a Prolog interface for PVM system, offering to the logic programing application the capabilities of PVM such as process spawning and control, virtual machine management and

[2]Available from the PEERTRUST CVS repository at http://sourceforge.net/projects/peertrust/

failure handling. PVM-Prolog defines an intermediate layer supporting the parallel and distributed execution of Prolog programs. This way it is possible to build a real distributed environment where several peers interact with each other using the PVM message passing mechanisms.

We make use of the layer of Prolog code presented in [18], supporting basic services like name resolution and distributed termination algorithms. Furthermore, for development and debugging purposes, we use a random simulator of the distributed system. The system was tested in a 8 PC cluster.

E. Detailed Computation

Consider again the proof forest of Figure 3, and the query

```
accLevel("Bob",Lev) @ "Pub" $ "Bob"
```

Figure 6 depicts an UML 2.0 interaction diagram showing a particular execution of our distributed algorithm. We are using subsumptive tabling in this example, in order to reduce generation of tables and respective proof-trees. Notice that each basic operation in the non-distributed tabling algorithm corresponds to the exchange of several messages by the distributed procedure; the messages are numbered with the corresponding identifiers of nodes used in the forest of Figure 3.

The computation is started by a query of Bob to the Publisher peer, and for the sake of simplicity, all goals and rules are in public mode. Since Publisher is not in the same community of Bob, the call must go through the goal manager, where the identifier is renumbered (from 7 to 0), and is sent to the Publisher peer client. Since this is a new goal, a new table is created in the publisher peer client and a message is sent to the publisher peer prover in order to start execution of the goal. The often occurring pattern of a call message sent to the peer client followed by a message to the corresponding peer prover indicates the creation of a new tree in the distributed forest of the tabling algorithm. The second argument of call messages originated from peer clients or peer provers represent identifiers of tables or goals, respectively.

The proof-trees are kept inside the prover peers, and only call and answer messages are interchanged between the components. Notice that all goals selected for execution are always sent to the peer client, even in the cases that the goal is subsequently executed at the same prover (e.g., messages labeled with 3, 4, and 5). This clearly decouples storage of tables and execution of goals, at expense of some message overhead but with substantial modularity advantages; this overhead can be reduced by using threading.

A particularly interesting subsequence of messages is the one labeled with 2. This represents a call from a peer prover (Publisher) to the peer client of a different peer (Music). This is once more sent through the peer client of Publisher, before being forwarded to the Music peer client by the Publisher peer prover. This allows the Publisher peer prover to check if there are signed rules obtained from Music peer before sending the request, avoiding unnecessary delays as well as tabling of remote calls (it is assumed that a prover peer client have faster message channels to its peer clients than to other peers).

Notice that message 4 reuses the table created by message 1, avoiding non-termination of the computation. Answers are generated in peer provers, which are sent back to the peer clients. The peer clients check if the answer is not repeated, and in this case propagate the answer to the requesters which might generate more answers, see for instance messages labeled with 6 and 7. The first message 7 corresponds to an answer to the original top-goal. Since this table is being shared by two calls, it originates two answer messages, one for each consuming selected goal (for Bob and for publisher peer prover). Upon reception of message 7 by publisher peer prover, the computation continues with message 8.

Message 23 does not create a new table, reusing again the tree created at 1, because of the adoption of subsumptive tabling. The decision to reuse a table has significant impact in the size of the computation, and two versions are commonly found in the literature [17]. The simplest check consists in using variant tabling, where a new table is created unless there is already a tabled goal identical modulo renaming of variables. Another alternative is subsumptive tabling, which creates a new tree unless there is a more general tabled goal; this requires extra unification checks since a more general answer might not unify with a more particular goal. In general, the same goal from a different requester might have different solutions. However, if the requester is not mentioned at the head of all rules for a predicate, as in the `accLevel/2` rules, then it is safe to share the answers.

Also notice that a failed goal, like 17, has an associated table but no answers are generated. Finally, the computation is finished by sending the appropriate done messages to Bob and to the peer clients. The detection of termination requires extra control messages, which are not shown in the diagram.

V. Related work

The work on distributed tabled query evaluation of logic programs is rather limited [18], [21], [22], [1], due to the inherent difficulties of termination detection. The previous work by the authors [18], [22] probably contains the most complete discussion of the problems that can be found in distributed query evaluation of logic programs, as well as a thorough analysis of local and global completion algorithms. This paper generalizes previous work avoiding the assumption of a single goal manager and extends it to make it suitable for policy languages, including mechanisms for delegation, internal rules as well as proof explanation.

The work by Rui Hu [21] presents a distributed implementation of the SLG proof procedure [15], including support for negation. Unfortunately, some unrealistic assumptions are made in the description of the algorithms, namely the knowledge of some global data structures. Furthermore, it requires passing of the "call stack" in each invoking call message which is terribly inefficient and might disclose private information. No study of the complexity of the implemented system is given.

The authors of the Cassandra system contributed in [4] with a new language for trust management. They mention the use of

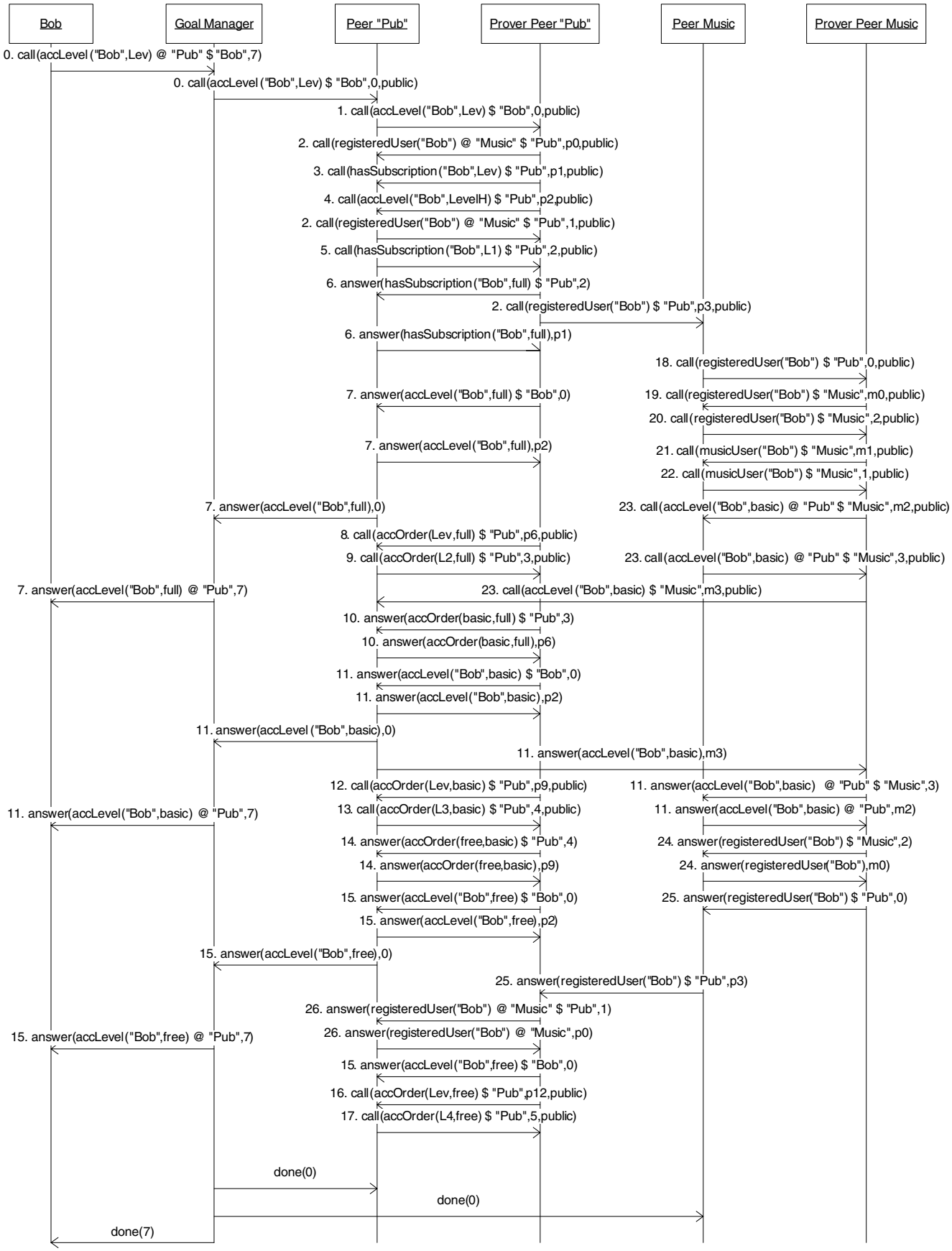

Fig. 6. Distributed tabulated evaluation of a query

a SLG tabled resolution algorithm extended with constraints, to deal with remote entities. In this sense is very similar to our initial ideas published in 2000. However, the contribution of that paper is a new language with tunable expressiveness and therefore they do not provide a description of the algorithm nor any details of it. Here we provide a full description of the distributed algorithm we have defined and how we deal with private and public rules as well as with generation of the proofs (which is not supported by the Cassandra system). In addition, we provide a fully distributed implementation of the algorithm and discuss the different solutions for the termination algorithm, which is probably the most important issue in the definition of a query evaluation algorithm in distributed environments. The authors in [4] report that they have only one sequential implementation.

We should mention also the OPTYap system, which is the unique parallel tabling system which we are aware of [23]. OPTYap is particularly efficient and scalable, however the techniques used do not generalize to the distributed setting of P2P networks or the WWW.

VI. Conclusions and further work

In this paper we have provided a solution in order to dynamically deal with loops during an authorization decision for access control. The algorithm takes care of internal and public rules, generates the proof of the evaluation and returns the answers (even if within a loop) in polynomial time. This work has been implemented and tested in a real distributed environment over a 8 PC cluster.

Further work includes an analysis of the current communication among goal managers in order to reduce the information interchanged. This reduction would allow a better management of the information that could lead to information leakage. In addition, we plan to integrate algorithms for local termination involving several peer communities. This is specially important in order to allow negation as failure in the PeerTrust language, under the well-founded semantics. We are also working in variations of the algorithms which are capable of handling communication faults.

Acknowledgements. This research has been partially supported by European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework project REWERSE number 506779.

References

[1] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu, "Requirements for policy languages for trust negotiation." in *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 5-7 June 2002, Monterey, CA, USA*. IEEE Computer Society, 2002, pp. 68–79.

[2] P. A. Bonatti, N. Shahmehri, C. Duma, D. Olmedilla, W. Nejdl, M. Baldoni, C. Baroglio, A. Martelli, V. Patti, P. Coraggio, G. Antoniou, J. Peer, and N. E. Fuchs, "Rule-based policy specification: State of the art and future work," Working Group I2, EU NoE REWERSE, Tech. Rep., Aug. 2004, http://rewerse.net/deliverables/i2-d1.pdf.

[3] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett, "No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web," in *1st European Semantic Web Symposium (ESWS 2004)*, ser. Lecture Notes in Computer Science, vol. 3053. Heraklion, Crete, Greece: Springer, May 2004, pp. 342–356.

[4] M. Y. Becker and P. Sewell, "Cassandra: Distributed access control policies with tunable expressiveness." in *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), 7-9 June 2004, Yorktown Heights, NY, USA*. IEEE Computer Society, 2004, pp. 159–168.

[5] P. A. Bonatti and D. Olmedilla, "Driving and monitoring provisional trust negotiation with metapolicies," in *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*. Stockholm, Sweden: IEEE Computer Society, June 2005, pp. 14–23.

[6] "Cassandra policy for national ehr in england. http://www.cl.cam.ac.uk/users/mywyb2/publications/ehrpolicy.pdf." [Online]. Available: http://www.cl.cam.ac.uk/users/mywyb2/publications/ehrpolicy.pdf

[7] W. Nejdl, D. Olmedilla, M. Winslett, and C. C. Zhang, "Ontology-based policy specification and management," in *2nd European Semantic Web Conference (ESWC)*, ser. Lecture Notes in Computer Science, vol. 3532. Heraklion, Crete, Greece: Springer, May 2005, pp. 290–302.

[8] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated trust negotiation," DARPA Information Survivability Conference and Exposition. IEEE Press, Jan 2000.

[9] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, 1992. [Online]. Available: citeseer.ist.psu.edu/lampson92authentication.html

[10] S. Staab, B. K. Bhargava, L. Lilien, A. Rosenthal, M. Winslett, M. Sloman, T. S. Dillon, E. Chang, F. K. Hussain, W. Nejdl, D. Olmedilla, and V. Kashyap, "The pudding of trust," *IEEE Intelligent Systems*, vol. 19, no. 5, pp. 74–88, 2004.

[11] N. Li, W. Du, and D. Boneh, "Oblivious signature-based envelope," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2003, pp. 182–189.

[12] J. Trevor and D. Suciu, "Dynamically distributed query evaluation," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Santa Barbara, CA, USA, May 2001.

[13] H. Tamaki and T. Sato, "OLD resolution with tabulation," in *Proc. of the Int. Conf. on Logic Programming'86*, ser. LNCS 225, Shapiro, Ed. Springer–Verlag, 1986, pp. 84–98.

[14] L. Vieille, "A database-complete proof procedure based on SLD-resolution," in *Proc. of the Int. Conf. on Logic Programming'87*, 1987, pp. 74–103.

[15] W. Chen and D. S. Warren, "Query evaluation under the well founded semantics," in *Proc. of the Twelfth Symposium on Principles of Database Systems (PODS'93)*, 1993.

[16] R. Bol and L. Degerstedt, "Tabulated resolution for well founded semantics," in *ILPS'93*. MIT Press, 1993.

[17] K. Sagonas, T. Swift, and D. S. Warren, "XSB as an efficent deductive database engine," in *Proc. of SIGMOD 1994 Conf. ACM*, 1994.

[18] C. V. Damásio, "A distributed tabling system," in *Proceedings of Tabulation in Parsing and Deduction 2000 (TAPD 2000)*, 2000.

[19] G. Tel, *Introduction to Distributed Algorithms*. Cambridge Univ. Press, 2000, 2nd Edition.

[20] J. C. Cunha and R. F. P. Marques, "Distributed algorithm development with pvm-prolog," in *5th EUROMICRO Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, 1997, pp. 211–2158.

[21] R. Hu, "Efficient tabled evaluation of normal logic programs in a distributed environment," Ph.D. dissertation, State University of New York at Stony Brook, 1997.

[22] M. B. Alves, "Distributed tabling architecture with termination detection (in portuguese)," Master's thesis, Universidade Nova de Lisboa, 2004.

[23] R. Rocha, "On applying or-parallelism and tabling to logic programs," Ph.D. dissertation, University of Porto, Portugal, 2001.