PUC-Rio - Certificação Digital Nº 0510970/CA

5
Our Approach for Cooperative Reasoning

This chapter presents our proposal of an algorithm and protocol that implement the *cooperative reasoning process*, according with the design strategies discussed in the last chapter.

5.1
Strategy for Rule-based Context Reasoning

We propose a strategy in which two entities — a reasoner running on the user side, the *device reasoner*, and another one running on the ambient side, the *ambient reasoner* — interact to infer situations described by rules involving context variables depending on data collected from different sources and stored at both sides, performing what we defined as *cooperative reasoning*.

Each of these entities (called *cooperative reasoners*) aggregate context information obtained from local context providers available at each side and execute the reasoning of rules submitted by applications running on either side. As the interaction may start at the *ambient side* or at the *user side*, depending on which side the client application is running, we call *local reasoner* the one executing at the side where the interaction begins, and *remote reasoner* the other one.

Depending on the rule to be inferred, the reasoning operation may follow one of the patterns presented in Section 4.2: user side, ambient side or cooperative reasoning. Now we describe the cooperative reasoning process for a rule R, submitted to the reasoning service by a client application, highlighting each step of the process, and hence defining our proposed cooperative reasoning strategy. As discussed in Section 4.3, the client application may query the reasoner to get an immediate response about a submitted rule R or may subscribe to be notified whenever the situation described by R holds. As such, this cooperation can have two different general forms of interaction, the synchronous and the asynchronous interactions.

Figures 5.1 and 5.2 show box diagrams in which each of the thirteen steps of our strategy are represented. We divided these steps into two different groups, the synchronous interaction steps and the asynchronous interaction

steps, which we describe, in more detail, as follows.

5.1.1 Synchronous Interaction

The synchronous interaction starts when a client application submits a *synchronous query* to the *local reasoner*, as described in the following paragraphs.

S1 - Submission: First of all, an application submits an inference rule R to the local *cooperative reasoner*. In this step, a submission is identified either as being a *synchronous query*, if the application needs to check if the situation described by the rule holds at that current moment, or as a *subscription*, if the application is to be notified whenever the situation described by the rule holds.

S2 - Partitioning: After receiving a submission for rule R, the reasoner will parse it and split the antecedent of the original rule in two parts, a *local part* R_L, comprised by atoms that refer only to context information available at the *local reasoner*, and a *remote part* R_R, comprised by the atoms that refer to context information available at the *remote reasoner*. In our model we assume that (a) each reasoner knows whether an atom of the rule corresponds to context data available at its side, and (b) each atom corresponds to context data available either at the device side or at the ambient side, i.e., all predicates are valid. If all context information needed to evaluate the rule is available at the local knowledge base (local ABox), i.e, the remote part R_R is void, we call R a *local rule*. In this case, the *local reasoner* will perform the reasoning locally, and the next step is the *evaluation* step. Otherwise, the next step is the *pre-evaluation* step.

S3 - Pre-evaluation: The problem of reasoning about rules with variables is equivalent to finding a set of tuples of individuals that bind to that variables, satisfying the rule. As discussed in Section 4.4, when the rule R is split into R_L and R_R, the *local reasoner* has to partially evaluate R_L to determine a set of variables V, that are common to R_L and R_R, and a set of tuples $T = \{(c_{1,1}, c_{1,2}..c_{1,n}), (c_{2,1}, c_{2,2}..c_{2,n})..(c_{k,1}, c_{k,2}..c_{k,n})\}$, where each element $c_{i,j}$ of the tuple is a value of variable $v_j \in V$. This step is called pre-evaluation because it produces a partial result T that serves as input for the *remote reasoner* to calculate the final result S.

PUC-Rio - Certificação Digital Nº 0510970/CA

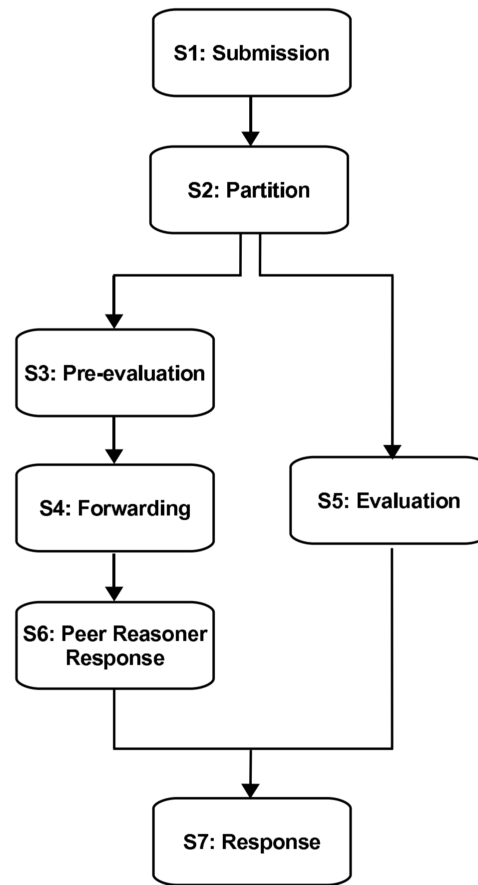

Figure 5.1: Box diagram representing the cooperative reasoning strategy for a synchronous query.

PUC-Rio - Certificação Digital Nº 0510970/CA

S4 - Forwarding: After pre-evaluating R_L, the *local reasoner* has to forward R_R to the *remote reasoner* (as a *synchronous query*), together with the list of variables V and the respective set of tuples T that were determined in the *pre-evaluation* step, if neither is empty. The *local reasoner* will wait for the response from the *remote reasoner* in the *peer reasoner response* step.

S5 - Evaluation: This step consists in performing the reasoning operation over the context data available locally to get the set of tuples S, corresponding to the tuples of individuals that satisfy the *local rule*. A rule that was forwarded by the other reasoner is also dealt in this step. In this case, for obtaining S the reasoner evaluates the rule bounded by the set of tuples T, containing the possible values for the set of variables in V. The next step is the *response* step.

S6 - Peer Reasoner Response: When the rule is not local, it will be partitioned and have the *remote part* R_R evaluated by the *remote reasoner*, according with the pre-condition imposed by the set of variables V and the set of tuples T associated, as described in Step S5. After forwarding R_R (Step

S3), the *local reasoner* will wait for the response from the *remote reasoner* containing the set of tuples S that corresponds to the result, going to the following step.

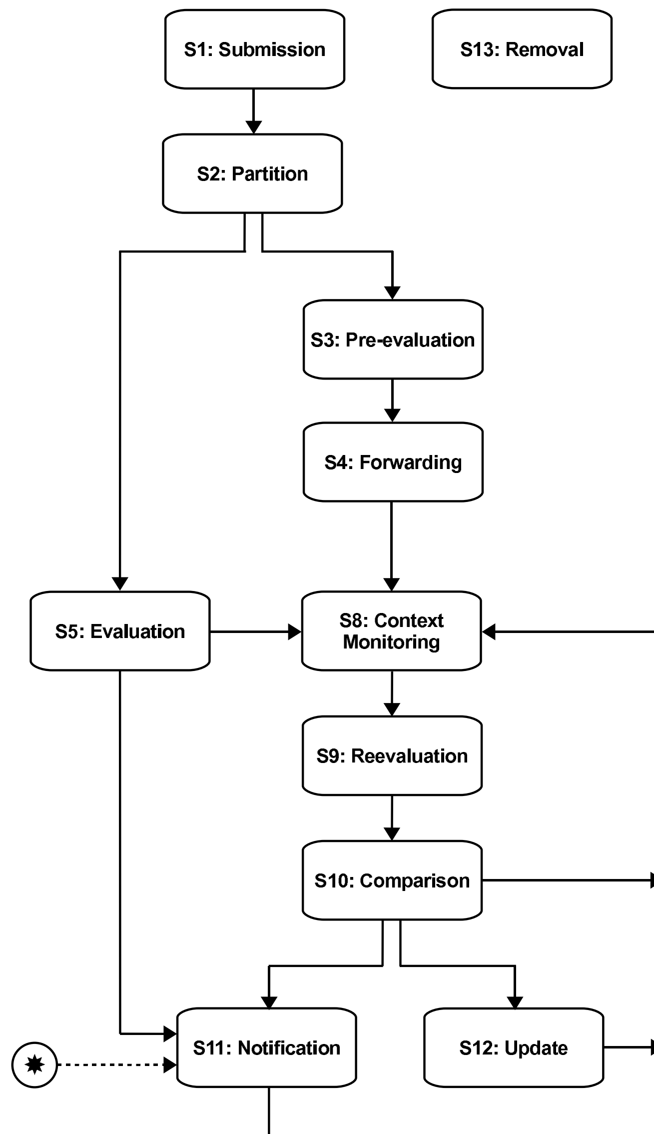

Figure 5.2: Box diagram representing the cooperative reasoning strategy for an asynchronous interaction.

S7 - Response: The result obtained for a *local rule* will be sent to the application client immediately after the *evaluation* (Step S5). For a rule that is not local, the *local reasoner* will go through Steps S3 and S4, and after that it will wait for the response from the *remote reasoner* (Step S6) and send to the application client the set of tuples S, received from the *remote reasoner*.

PUC-Rio - Certificação Digital Nº 0510970/CA

PUC-Rio - Certificação Digital Nº 0510970/CA

5.1.2 Asynchronous Interaction

The asynchronous interaction starts when a client application submits a rule R to the *local reasoner* as a *subscription*. The initial steps (S1 to S5) are very similar to what was described for the synchronous interaction in Subsection 5.1.1. After the rule R is submitted (Step S1), it is partitioned (Step S2), and if it is not a *local rule*, it goes through the *pre-evaluation* (Step S3) and the *forwarding* (Step S4), but as a subscription. After that, however, the reasoner goes to the *context monitoring* step (Step S8). On the other hand, if the submitted rule R is local, after the *partitioning* (Step S2) the next step will be the *evaluation* (Step S5) and after that the *notification* (Step S11) or *the context monitoring* (Step S8), as indicated in Figure 5.2. The Steps S8 to S13 in the asynchronous interaction are presented in the following paragraphs.

S8 - Context Monitoring: After the first evaluation of a *local rule* (Step S5), or after the *remote part* R_R of a rule is forwarded to the *remote reasoner*, each *subscription* is put into a list of subscriptions with the associated data. This comprises the information about the client application that submitted the subscription, the *local part* and *remote part* (if it exists) of the rule, the sets of free variables V and tuples T that were sent to the *remote reasoner* (or received from a *local reasoner*, in case the reasoning plays the role of the *remote reasoner* and R corresponds to a *forwarded rule*). Each time there is a change in context data that may affect one of the rules in the list, each of these rules is selected so that the reasoner can perform a *reevaluation* of the rule (Step S9).

S9 - Reevaluation: After a change in context data that may affect a rule associated with a *subscription*, if the rule is a *local rule*, the reasoner checks the rule performing a new reasoning operation to find a set of tuples of individuals S that satisfy the rule. If the rule is not local, similarly to what was described in Step S3 of the synchronous interaction, R_L will be evaluated to find an updated set of tuples T with values for each variable in V that are common in R_L and R_R. In Step 10, the results of this step are compared with the ones that were previously obtained.

S10 - Comparison: The sets of tuples S or T, which were determined in the *reevaluation* step, are compared with sets of tuples previously found. If there are no differences between the sets, no action will be taken and the process returns to (Step S8). Otherwise, the reasoner will store the new results for

future comparisons. If the rule is a *local rule*, the reasoner will proceed to the *notification* step (Step S11). Otherwise, it will proceed to the *update* step (Step S12), which will be explained ahead.

S11 - Notification: A notification for a client may be originated in several forms. (i) After the submission of a rule R (Step S1), if its is partitioned and identified as a *local rule* (Step S2), and in its evaluation (Step S5), a set of tuples of values S that satisfy the rule is found, the *local reasoner* sends this result to the client application as a notification. (ii) From then on, this rule will be monitored (Step S8), reevaluated when necessary (Step S9), and every time a new set S is found, i.e., different from the previous result (Step S10), the client will again be notified. (iii) For rules that had the *remote part* R_R forwarded to the *remote reasoner* (Step S4), upon being received there, R_R goes straight to the evaluation step (Step S5), and may generate a notification in the same way described in *i* and *ii*, but targeting the peer reasoner. On receiving this notification (as indicated by the dashed arrow in Figure 5.2), the *local reasoner* sends the received result S to the client application, but only if it meets the conditions that will be discussed in Subsection 5.1.3.

S12 - Update: A particular situation may occur if a rule R has a *local part* R_L that is being monitored by the *local reasoner*, and a *remote part* R_R, that was forwarded to the *remote reasoner*. While R_R is being monitored by the *remote reasoner*, changes in the context data may happen also at the local side, which may cause a change in the set of tuples T that were initially determined by the *local reasoner* in the pre-evaluation of R_L (Step S3) and previously forwarded to the *remote reasoner* (Step S4). Therefore, in Step S8 the *local reasoner* monitors the context variables present in the local part R_L of a rule, reevaluates the rule when necessary (Step S9), and updates this information at the *remote reasoner* whenever a new set of tuples T is found. This update is identified by an “update number”, that will be used to guarantee that a notification from the *remote reasoner* gives a valid result, as will discussed in Subsection 5.1.3.

S13 - Removal: At any time the client application can remove a subscription. If it corresponds to a *local rule*, the *local reasoner* simply removes it from the list of subscriptions. Otherwise, the *local reasoner* also requests for the *remote reasoner* to remove the part of the rule that was forwarded.

PUC-Rio - Certificação Digital Nº 0510970/CA

5.1.3 Stability of Context Data

A notification from the *remote reasoner* to the *local reasoner* about the result of a *forwarded rule*, is based on the context data available at the *remote reasoner* at the moment it was generated, and the latest set of tuples T received from the *local reasoner*. This set is first determined in Step S3 and may be subsequently reevaluated in Step S9. Before the notification from the *remote reasoner* arrives at the *local reasoner*, however, there may have been a context data change at the *local reasoner*, which caused a new update to be sent to the *remote reasoner*. In this case, the result S received from the *remote reasoner* can not be considered valid, because it is based on a set of tuples T that has changed.

To prevent the *local reasoner* from sending to the client application a result that is inaccurate, each time a new update is generated by the *local reasoner*, it receives an *update number*, which is sent to the *remote reasoner* together with the new set of tuples T. When the *remote reasoner* finds a set of values S that satisfy a rule, it notifies the *local reasoner*, sending the result S, together with the *update number* of the latest update it received, allowing the local reasoner to check if the result corresponds to the latest update. In Step S11, the *local reasoner*, after receiving this notification, will send the received result S to the client application only if the number of the last update received by the *remote reasoner* matches the number of the last update sent by the *local reasoner*. Otherwise, the result received from the *remote reasoner* will be ignored.

If there are frequent changes of the context data related with a rule R at the *local reasoner*, the reasoners might never converge to find a response and notify the client application. This means that this strategy is not adequate for reasoning with context data that are highly variable. The minimum time t_{reason} that the reasoners take to find a result comprises the periods of time needed for:

1. R_L to be evaluated (or reevaluated) at the *local reasoner*;
2. T to be updated at the *remote reasoner*;
3. R_R to be reevaluated by the *remote reasoner*, finding a result S;
4. the *local reasoner* to receive the notification from the *remote reasoner*;

Let us assume that these changes of the context data occur with mean periodicity of time t_{change}. The necessary condition for guaranteeing the

PUC-Rio - Certificação Digital Nº 0510970/CA

convergence of the inference process is that the context data is stable, i.e., that $t_{change} >> t_{reason}$. As the *performace* attribute identified in Section 4.3 indicates that t_{reason} should be adequate, it is directly related with stability of context data.

5.2 Algorithm

In Section 5.1 we described all the general steps that have to be executed to perform a *cooperative reasoning* process. From this description, we can identify that the events that trigger the actions in this process are:

- in a synchronous interaction:
  - the arrival of a new (or forwarded) query;
- in an asynchronous interaction:
  - the arrival of a new (or forwarded) subscription;
  - a change in context data that is being monitored.
  - the arrival of an update from the peer reasoner;
  - the arrival of a notification from the peer reasoner;
  - the removal of a subscription;

In this section we describe the distributed algorithm used to implement a service that performs the proposed process. In fact, the overall process may be divided in blocks of procedures, each triggered by one of the previously mentioned events.

PUC-Rio - Certificação Digital Nº 0510970/CA

Algorithm 5.1: ON RECEIVING A NEW QUERY

input: A rule R submitted to the reasoner by client C.

1 Partitions R to obtain R_L and R_R
2 **if** $R_R \neq \emptyset$ **then**
3 Pre-evaluates R_L to obtain V and T
4 Forwards R_R, V and T to the remote reasoner
5 Receives S from remote reasoner
6 **end**
7 **else**
8 Evaluates R to obtain S
9 **end**
10 Sends the result S to client C

Algorithm 5.1 shows the code that deals with the synchronous interaction, which is triggered by the arrival of a new or forwarded query. After

submission, the rule is partitioned (Line 1), and if it has a *remote part*, it is pre-evaluated to obtain the set of tuples T that is forwarded to the *remote reasoner* together with the list of common variables V (Lines 3 and 4). The *local reasoner* then waits for the reply from the *remote reasoner*, which contains the set of tuples S representing the result for the cooperative reasoning (Line 5). In contrast, if R is a *local rule*, it is immediately evaluated to obtain the set of tuples S that represent the result of the reasoning (Line 8). In either case, the result S is sent to the application client (Line 10). A *forwarded rule* is regarded by the *remote reasoner* in the same form as a *local rule*, and hence is also evaluated (Line 8), generating a response to the *local reasoner* (Line 10).

Algorithm 5.2: ON RECEIVING A NEW SUBSCRIPTION

input: A rule R submitted by client C.

```
1  Partitions R to obtain R_L and R_R
2  if R_R ≠ ∅ then
3      Pre-evaluates R_L to obtain V and T
4      Forwards R_R, V and T to the remote reasoner
5  end
6  else
7      Evaluates R to obtain S
8      if S ≠ ∅ then
9          Notifies the client C with the result S
10     end
11 end
12 Puts R_L in list L
```

Algorithm 5.2 implements the procedure triggered by the arrival of a new subscription from a client, which initiates an asynchronous interaction. As in the synchronous interaction, after the submission, the rule R is partitioned to obtain the *local part* R_L and the *remote part* R_R (Line 1). If R has a remote part R_L, it is pre-evaluated to obtain the set of tuples T (Line 3), which is forwarded to the *remote reasoner* as a subscription, together with the list of common variables V (Line 4). If R is a *local rule*, it is evaluated to obtain the set of tuples S that represents the result of the reasoning (Line 7). If this result is not empty, the client application is notified about the result S. As in the synchronous interaction, a *forwarded rule* is regarded by the *remote reasoner* as a *local rule* that is evaluated (Line 7), possibly generating a notification to the *local reasoner* (Line 9). In either case, the rule is put in a list of subscriptions so that it can be checked whenever an event that may change the result occurs, such as a change in context data, the arrival of an update message from the

PUC-Rio - Certificação Digital Nº 0510970/CA

local reasoner, or the arrival of a notification from the *remote reasoner*.

Algorithm 5.3: ON CONTEXT UPDATE

input: A change in context data D.

```
1  for each R ∈ L affected by D do
2      if R_R = ∅ then
3          Reevaluates R_L to obtain S
4          Compares S and LastResult
5          if S ≠ LastResult then
6              Stores S as LastResult
7              Notifies client C with S [and LastUpdateNumber ]
8          end
9      end
10     else
11         Reevaluates R_L to obtain T
12         Compares T and LastTuples
13         if T ≠ LastTuples then
14             Stores T as LastTuples
15             Increments LastUpdateNumber
16             Updates R_R, T and LastUpdateNumber at the remote
               reasoner
17         end
18     end
19 end
```

PUC-Rio - Certificação Digital Nº 0510970/CA

Each reasoner monitors its local context data (*context monitoring* step), and, as shown in Algorithm 5.3, whenever a change in context is perceived, the list of subscriptions L is checked to select any rule R that may have been affected by the change (Line 1), i.e., rules whose any atom is a predicate related with that context fact. If R is a *local rule*, it is reevaluated to obtain a new set of tuples S (Line 3). The new result S is compared with the previous one stored $LastResult$. If they differ from each other (Lines 4 and 5), the new result is stored as $LastResult$ (Line 6) and sent to the client C as a notification (Line 7). If the client C is the peer reasoner, i.e., the rule being monitored corresponds to a *remote part* received from that reasoner, then this notification must include as parameter the number of the last update received from it, $LastUpdateNumber$. If R has a *remote part*, it is reevaluated to obtain a new set of tuples T (Line 11). The new set of tuples T is compared with the last one stored $LastTuples$ and if they are different from each other (Lines 12 and 13), this set of tuples is stored as $LastTuples$ (Line 14) and sent to the *remote reasoner* as an update (Line 16). In this case, a variable $UpdateNumber$ is incremented to identify the number of this update (Line 15) and sent together with T and V.

Algorithm 5.4: ON RECEIVING UPDATE FROM PEER

input: An update for rule R containing a set of tuples T and an update number n.

1 Stores n as $LastUpdateNumber$
2 Reevaluates R to obtain S given T
3 Compares S $LastResult$
4 **if** $S \neq LastResult$ **then**
5   Stores S as $LastResult$
6   Notifies the local reasoner with S and $LastUpdateNumber$
7 **end**

As discussed in Step 12 of Subsection 5.1.2, changes in context data at the *local side* may cause a change of the variable values that were previously forwarded to the *remote reasoner*. In this case, the *local reasoner* sends an update to the *remote reasoner* as described in Algorithm 5.3, Line 16. Algorithm 5.4 implements the procedure triggered by the arrival of an update at the *remote reasoner*. Rule R is reevaluated to obtain a new set of tuples S (Line 2). The new result S is compared with the previous one stored $LastResult$ and if they are different from each other (Lines 3 and 4), this result is stored as $LastResult$ (Line 5) and sent to the *local reasoner* as a notification (Line 6).

Algorithm 5.5 implements the procedure triggered by the arrival of a notification from the *remote reasoner* at the *local reasoner*. In this case, the *local reasoner* compares the number of the update n, provided with the notification, with the last $UpdateNumber$ associated with that rule R (Line 1). If they are equal, the result S is sent to the client application (Line 2). Otherwise, the result is ignored.

Algorithm 5.5: ON RECEIVING NOTIFICATION FROM PEER

input: An notification message containing a set of values S and a version number n.

1 **if** $N = UpdateNumber$ **then**
2   Notifies the client C with result S
3 **end**

Finally, Algorithm 5.6 implements the procedure triggered by the arrival of a message from the client asking to remove the subscription associated with a rule R. If R has a *remote part* R_R, the *local reasoner* has to send a removal message to the *remote reasoner* asking for the removal of the subscription associated with R_R (Line 2). In any case, R is removed by the *local reasoner* (Line 4).

PUC-Rio - Certificação Digital Nº 0510970/CA

Algorithm 5.6: ON RECEIVING REMOVAL ASK

input: An message asking the removal of rule R.

1 **if** $R_R \neq \emptyset$ **then**
2   Removes R_R from the remote reasoner
3 **end**
4 Removes R from list L

5.3 Protocol

The reasoning process discussed in Section 5.1 is implemented as a service that receives messages from client applications — *synchronous queries* or *subscriptions* — containing rules that describe situations that are relevant for these applications. The reasoning process is performed by services, the *cooperative reasoners*, that exchange messages to execute the distributed algorithm described in Section 5.2. Nevertheless, the overall *cooperative reasoning process* is only completely specified after we define a communication protocol, describing all the messages exchanged by these services. As we assume that the communication channel is reliable, i.e., there is no loss of messages, confirmations messages were not included in our protocol.

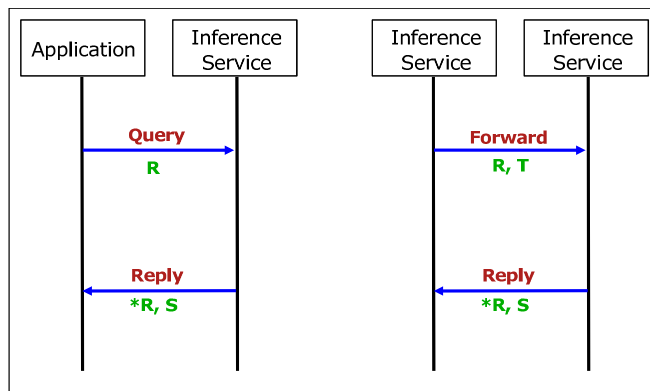

Figure 5.3: Synchronous interaction in the cooperative reasoning.

Figure 5.3 shows the protocol executed for performing the synchronous interaction of our cooperative reasoning process, as described in Section 5.1.1. The synchronous interaction starts when a `Query` message is sent from the client to the *local reasoner*, with rule R as parameter, triggering the procedure described in Algorithm 5.1 (Step S1). A `Forward` message is sent from the *local reasoner* to the *remote reasoner*, carrying the *remote part* of the rule R_R, if it exists, together with the set of variables V and the set of tuples T determined in the pre-evaluation of R (Steps S2, S3 e S4). The rule forwarded by the *local reasoner* is received by the *remote reasoner* as a query with some

PUC-Rio - Certificação Digital Nº 0510970/CA

extra parameters, and regarded as a *local rule*. It is evaluated by the *remote reasoner*, and the set of tuples S, found as result of the evaluation, is sent back to the *local reasoner* in a `Reply` message (Step S5). The *local reasoner* waits for this response (Step S6), and after receiving it, sends a `Reply` message to the client application containing this answer (Step S7). When the original rule is local, the only message sent is the `Reply` message containing the set of tuples S from the *local reasoner* to the client application (Step S7). Table 5.1 summarizes the synchronous protocol

Operation	**Description**
Query(R)	A client application sends a message to the local reasoner with a rule R, which describes the situation to be verified.
Forward(R_R, V, T)	The local reasoner sends to the remote reasoner this message containing R_R, the remote part of the original rule, the list of variables V that were evaluated locally and the set T of tuples with values for each variable.
Reply (S)	The local reasoner sends a message to a client — or the remote reasoner sends a message to the local reasoner — as a response to a synchronous query, containing a set of matches S that satisfy the originally proposed rule R.

Table 5.1: The protocol executed for performing the synchronous interaction of our cooperative reasoning process.

The messages exchanged in an asynchronous interaction are shown in Figure 5.4. This interaction starts when a `Subscribe` message is sent from the client to the *local reasoner*, with the rule R as parameter (Step S1). This message triggers the procedure described in Algorithm 5.2. A `Forward` message is sent from the *local reasoner* to the *remote reasoner* if R has a *remote part* R_R (Steps S2, S3 e S4), with R_R together with the set of variables V and the set of tuples T, determined in the pre-evaluation, as parameters.

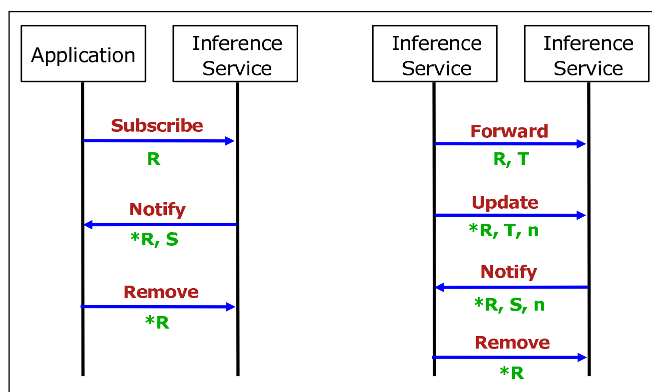

Figure 5.4: Asynchronous interaction in the cooperative reasoning.

PUC-Rio - Certificação Digital Nº 0510970/CA

An `Update` message is sent from the *local reasoner* to the *remote reasoner*, with a reference to rule R_R, the new set of tuples T and $UpdateNumber$ as parameters, whenever a change in context data in the *local reasoner* causes T to change (Steps S8, S9, S10 e S12). This message triggers the procedure described in Algorithm 5.4.

Operation	**Description**
Subscribe(R)	A client application sends a message to the local reasoner with a rule R describing a situation of interest.
Forward(R_R, V, T)	The local reasoner sends this message to the remote reasoner containing R_R, the remote part of the original rule, the list of variables V that were evaluated locally and the set T of tuples with values for each variable.
Update($*R_R$, T, n)	The local reasoner sends this message to the remote reasoner containing a reference to the rule R_R, previously forwarded, updated information to the corresponding set of tuples T and an update number n that identifies the update version.
Notify($*R$, S [, n])	The local reasoner sends this message to the client application — or the remote reasoner sends the message to the local reasoner — containing a reference to a rule R previously provided to the reasoner and a set of matches S that satisfy the rule R. If the notification is sent by the remote reasoner to the local reasoner, it contains also the number of the last update received by the sender n.
Remove($*R$)	A client application sends the message to the local reasoner — or the local reasoner sends the message to the remote reasoner — containing a reference to a rule R previously submitted, whose corresponding subscription must be removed.

Table 5.2: The protocol executed for performing the asynchronous interaction of our cooperative reasoning process.

A `Notify` message is sent from the *local reasoner* to the client application, with the result S as parameter, after the first evaluation of R (if S is not empty, Step S5) and whenever a change in context data in the *local reasoner* causes S to change (Steps S8, S9, S10 e S11). A `Notify` message may also be sent from the *remote reasoner* to the *local reasoner* whenever a change in context data in the *remote reasoner* causes the result S associated with a *forwarded rule* R_R to change, with the result S and the $LastUpdateNumber$ as parameters (Steps S8, S9, S10 e S11). This message triggers the procedure described in Algorithm 5.5. In this case, another `Notify` message is sent from the *local reasoner* to the client application, with the result S as parameter, if the $LastUpdateNumber$ received by the *local reasoner* has the same value of the local variable $UpdateNumber$ (Step S11).

PUC-Rio - Certificação Digital Nº 0510970/CA

The `Remove` message, sent from the client to *local reasoner*, triggers the procedure described in Algorithm 5.6. If the rule to be removed has a *remote part*, a `Remove` message is sent from the *local reasoner* to the *remote reasoner*, with a reference for rule R as parameter (Step S13). Table 5.2 summarizes the asynchronous approach.

5.4 Discussion

In this chapter, we proposed a strategy to execute *cooperative reasoning* and described the distributed algorithm and communication protocol to perform the complete process, according with the functional attributes of the design stratagies that we enumeratad in Section 4.3.

A fundamental part of our proposal for the split inference of facts is the partitioning of a rule in its local part, that is evaluated by the *local reasoner*, and its remote part, which is forwarded to the *remote reasoner*. When there are context variables that are common to the local and the remote part of the rules, however, the *remote reasoner* — to be able to evaluate the remote part of the rule — needs to know which are the possible values that the common variables may assume. As explained in Section 5.1, this is achieved by the Pre-evaluation/Forwarding and Reevaluation/Update steps of the reasoning strategy. These steps are associated with the *Update* and *Forward* messages, whose content comprises the tuples corresponding to the possible values that the set of variables may assume. In the reasoning process, these tuples represent a partial result for the *local reasoner* and a starting point for the *remote reasoner*. As the *local reasoner* forwards no complete RDF tuple for the *remote reasoner*, only tuples of individuals representing context variable values, no knowledge sharing happens between those reasoners. Particularly, when there is no variable in common between the local and remote parts of the rule, the variable values have to be solely determined in each side.

Providing asynchronous communication (publish/subscribe) is a particularly important attribute identified for this inference service. To achieve this goal, a *local reasoner* has to constantly update the information forwarded to the *remote reasoner*, in the *cooperative interaction*. For that reason, if there are frequent context changes at the *local reasoner*, not only the reasoning operation may never converge, bu also the great number of messages exchanged between the reasoners may cause a great communication overhead. This means that the proposed strategy is not adequate for reasoning with highly variable context data.

In our proposal of a *cooperative reasoning* protocol, however, we did

PUC-Rio - Certificação Digital Nº 0510970/CA

not consider aspects related with an important non-functional attribute, the robustness and resilience of the service. In our system model we assumed that the communication was reliable, i.e., there would be no loss of messages. As such, we did not include confirmation messages in our protocol, and hence, the loss of a message can cause an inference operation to be discontinued, with no warning being sent to the clients.

In the next, chapter we present a case study to show how — step-by-step — this strategy works, both for synchronous communication (queries) and asynchronous communication (publish/subscribe).

PUC-Rio - Certificação Digital Nº 0510970/CA