8
Evaluation

In this chapter we describe how our implementation of the Decentralized Reasoning Service (DRS) was tested in respect to its functional behavior and its performance and discuss the results that were found.

8.1
Testbed

In order to test the correct functioning and evaluate the performance of our implementation of DRS, we ran different batches of tests. To run the Ambient Decentralized Reasoning Service (DRS/A) — representing the ambient infrastructure — we used a desktop PC with a Core 2 Duo 2.40 GHz processor and Windows Vista operating system. To run the Device Decentralized Reasoning Service (DRS/D) — representing the mobile device — we used a notebook with an Atom 1.60 GHz processor Windows XP SP2 operating system. These computers were interconnected through an IEEE 802.11 wireless network with 54 Mbps data transfer rate. Depending on the purpose of each test, we used specific context data files — i.e., ontologies — created with specific features, as will be described later. In the next section we describe the functional tests performed to check if the service was operating as expected, i.e., meeting the previously identified design strategies. In Section 8.3, we discuss the tests we executed to measure the performance of the service.

8.2
Functional Test

A set of tests was performed to check if our implementation met the functional attributes of the design strategies identified in the specification of the decentralized reasoning service, as enumerated in Section 4.3.1. As this operation involved the simulation of the client applications subscribing the service and context changes happening in different sequences of events, we implemented a Test Management Application (TMA), with a graphical user interface to easily trigger the events and interpret the results. To be able to

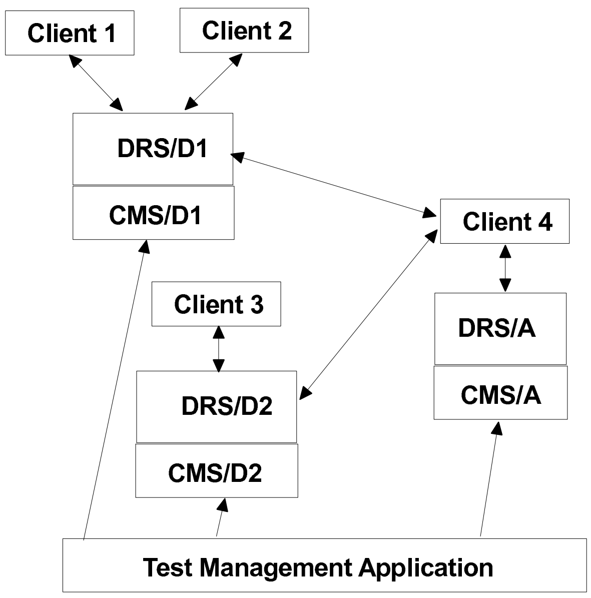
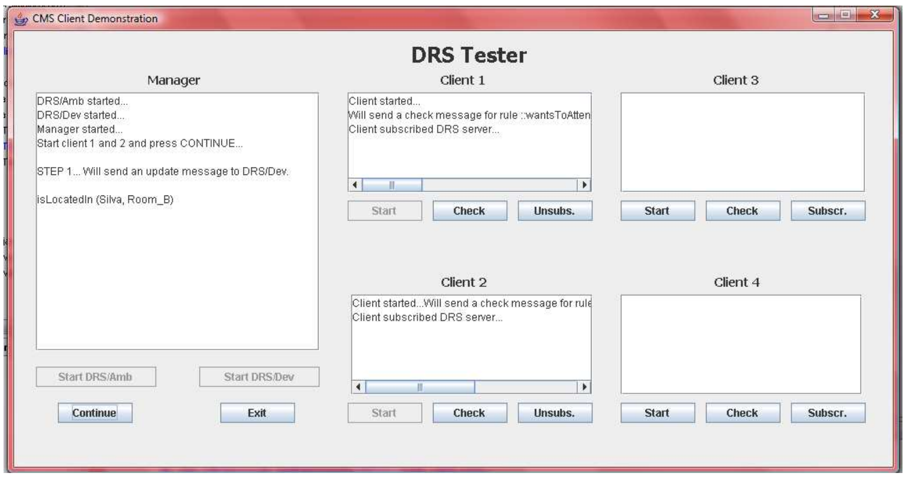


Figure 8.1: Architecture used for the functional test.

check any possible sequence of events, our simulation involved the execution of a reasoner service representing the ambient infrastructure (DRS/A), two reasoner services representing two users and their devices (DRS/D1 and DRS/D2), with the respective CMS servers, and four client applications, representing applications running on the users' devices (Client 1 and 2 running on device 1 and Client 3 running on device 2) and on the ambient infrastructure (Client 4). Figure 8.1 shows the architecture for this test.



```
DRS/Amb started...
DRS/Dev started...
Manager started...
Start client 1 and 2 and press CONTINUE...

STEP 1... Will send an update message to DRS/Dev.

isLocatedIn (Silva, Room_B)
```

```
Client started...
Will send a check message for rule ::wantsToAtten
Client subscribed DRS server...
```

```
Client started...Will send a check message for rule
Client subscribed DRS server...
```

Figure 8.2: GUI of the Test Management Application.

The interface, shown in Figure 8.2, contains buttons that allowed to control the actions of the TMA — responsible for simulating context changes for the reasoning services DRS/A, DRS/D1 and DRS/D2 — and the four client applications, and windows to show the output of each application. For each entity, there is a start button. At left, a window shows messages describing the actions taken by TMA, which is responsible for sending context update messages to each CMS, triggering the inference of some rules provided by the clients. Each time the button “Continue” is pressed, TMA sends a message to a specific CMS, in a pre-defined sequence of steps. At right, four windows show the behavior of each client, i.e., the queries, subscriptions and notifications. There are also the buttons “Check”, for sending a query, and “Subscribe/Unsubscribe”, for posting or removing a subscription. Using this interface we could verify if the queries were answered correctly and if the notifications were triggered at the right moment and with the correct result. We tested these responses varying the order of events (subscriptions and notifications), the number the simultaneous subscriptions and the form of the interaction, testing all possible patterns (see Section 4.2). In each case the response was correct.

8.3 Performance Test

A second set of tests was conducted in order to verify the response time, communication traffic and memory consumption of our implementation, three of the non-functional attributes of the design strategies identified in the specification of service (Section 4.3.2). Usually AmI systems are represented by ontology context models where the ABox is very much larger than the TBox. In such cases, the use of KAON2, which was developed to efficiently reason over large ABoxes rather than large TBoxes [103], is more appropriate. Accordingly, to represent our system in the evaluation process, we created ontologies with this same feature, but with some other specific characteristics, only for purpose of performance testing.

For our performance test we tried to create an ontology representing a realistic scenario — based on our conference example — to observe the behavior of the DRS server under heavy use conditions that we tried to simulate. As described in Figure 8.3, this ontology contains the four classes *Person*, *Activity*, *Subject* and *Environment* interrelated by the five binary properties *isLocatedIn*, *wantsToAttend*, *isInterestedIn*, *isRelatedWith* and *takesPlace*, and the unary property (subclass) *isAboutToStart*.

We generated a huge number of individuals and facts to create a large ABox, trying to mimic numbers that could possibly occur in a real scenario, e.g., a large conference. For example, in conference ACM SAC 2008, there were about 60 different sessions and the proceedings include approximately 1400 keywords. Based on these data, the ontology we generated has 500 individuals belonging to class *Person* ($Attendee_0$ to $Attendee_{499}$), 60 individuals

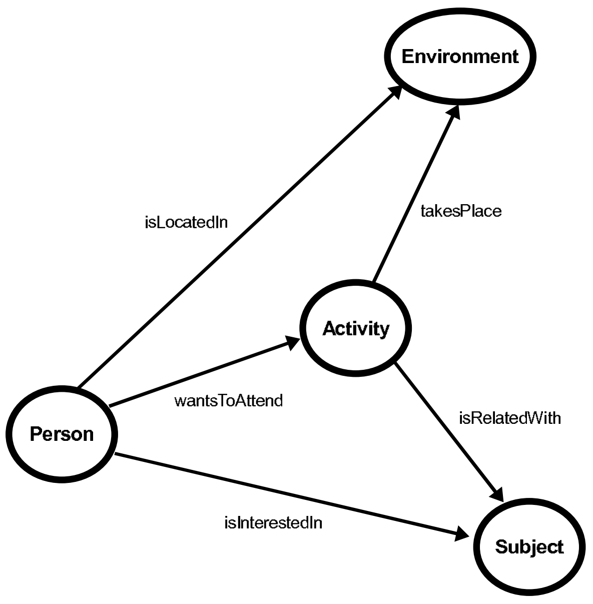


Figure 8.3: Classes and properties of the conference ontology.

belonging to class *Activity* ($Session_0$ to $Session_{59}$), 1400 individuals belonging to class *Subject* ($Keyword_0$ to $Keyword_{1399}$) and 10 individuals belonging to class *Environment* ($Room_0$ to $Room_9$). We then connected randomly each *Person* individual with one *Environment* individual (with *isLocatedIn* property), 10 *Activity* individuals (with *wantsToAttend* property) and 20 *Subject* individuals (with *isInterestedIn* property), each *Activity* individual with 30 *Subject* individual (with *isRelatedWith* property) and one *Environment* individual (with *takesPlace* property). No individual was defined as belonging to subclass *isAboutToStart*, for this was used to trigger the notification.

8.3.1 Response Time

In a first experiment, we measured the response time, i.e., the overall time since a context data change message is sent to the ambient side DRS, until an application client is notified by the user side DRS about the inference. We simulated the load of the server creating a variable number of subscriptions — 100, 200, 300, 400 or 500 — defined by a rule correlating the properties *wantsToAttend*, *takesPlace* and *isAboutToStart*, as in Rule 8.1.

Rule 8.1:

wantsToAttend(“Attendee5”,?s) ∧ takesPlace(?s,?r) ∧ isAboutToStart(?s) ⇒ shouldGoToRoom(“Attendee5”,?r)

The purpose of this simulation was to observe the overhead caused by the monitoring process. Since our server is optimized to check (i.e. evaluate the rules) only the subscriptions that involve properties that are changed, we tested the server under two different conditions: under no context change and with a context change message (changing the *takesPlace* property of a individual from class *Activity* selected randomly) being sent to the remote reasoner at each 50 milliseconds.

Centralized Reasoning

To be able to evaluate the decentralized reasoning strategy, we first measured the response time for a centralized configuration, i.e., with a central server collecting all context information and reasoning over a complete ontology, measuring the response time for the different number of subscriptions and load conditions. We created a test client that would subscribe the DRS with Rule 6.1 and then send a context change message that would trigger the context event. Hence, the total time observed goes from the sending of the context message until the receiving of the notification, encompassing the reasoning process.

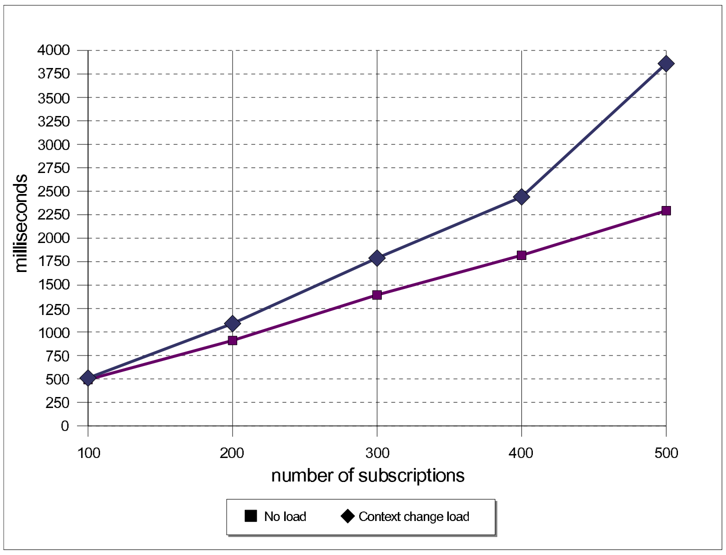


Figure 8.4: Response time measured for DRS working in a centralized configuration.

Figure 8.4 shows the different values measured. We can easily notice the influence of a greater number of subscribers — with the time measured going approximately from 0.5s to 2.25s under no load —, due to time spent on notifying each subscriber when a rule holds. We can conclude also that the influence of the message load becomes bigger when the number of subscribers gets greater, observing that for 500 subscriptions the difference between the two experiments goes to about 1.5s. That is due to the combination of the time spent monitoring and evaluating rules each time a message arrives with the time spent on notifying each subscriber.

Decentralized Reasoning Triggered at the Remote Reasoner

The decentralized reasoning was evaluated considering a local reasoner (DRS/D) that would have the part of the ontology concerning a user and its device, having information about the binary properties *isLocatedIn*, *wantsToAttend* and *isInterestedIn* while the remote reasoner (DRS/A) would have the part of the ontology having data about the properties *isRelatedWith* and *takesPlace*, and the unary property (subclass) *isAboutToStart*. As the reasoner running on the user's device is not expected to operate under heavy load conditions, all the load was simulated at the DRS/A.

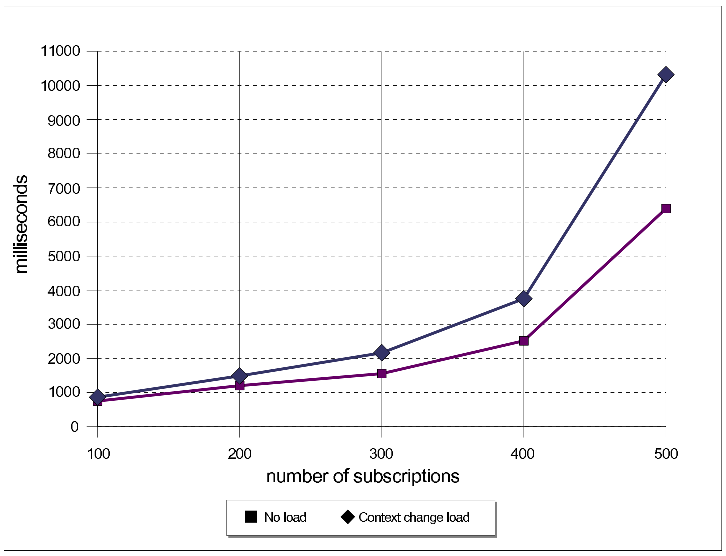


Figure 8.5: Response time measured for DRS working in a decentralized configuration and the reasoning being triggered at the remote reasoner.

We first measured the response time with the notification event being triggered by a message received by the remote reasoner. For that purpose, we used a test client that would subscribe the local DRS with Rule 6.1 — that would have part delegated to the remote reasoner — and then send a context change message to the remote reasoner that would trigger the context event. In this case, the total time observed goes from the sending of the context message to the remote reasoner until the receiving of the notification by the test client, encompassing also the reasoning process on the remote reasoner, the notification sent from the remote reasoner to the local reasoner and the validity verification of this notification by the local reasoner.

Figure 8.5 shows the two graphs. The influence of the number of subscribers is in even more perceptible in this configuration, making the measured time vary from approximately 0.8s to 6.4s under no load. This can be attributed to time spent by the remote reasoner to notify the simulated subscribers combined with the extra communication time. As observed in the centralized configuration, we notice that the influence of the message load becomes clearly bigger when the number of subscribers gets greater, observing that while for 100 subscriptions the difference between the two experiments is insignificant, for 500 subscriptions the value goes to approximately 3.7s. Again we believe that this is due to the combination of the time spent monitoring and evaluating rules each time a message arrives with the time spent on notifying each subscriber, added with the time spent in communication between the two reasoners.

Decentralized Reasoning Triggered at the Local Reasoner

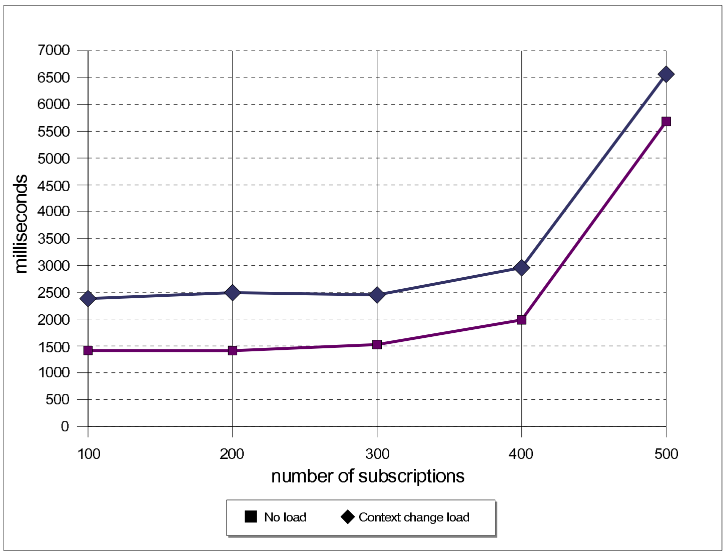


Figure 8.6: Response time measured for DRS working in a decentralized configuration and the reasoning being triggered at the local reasoner.

Finally, we measured the response time with the notification event being triggered by a message received by the local reasoner (DRS/D). As in the previous configuration, only the remote reasoner (DRS/A) was subject to the load simulation. The test client used in this case would subscribe to the local DRS submitting Rule 6.1, which would be split and have part delegated to the remote reasoner. Then the test client would send to the local reasoner a context change message that would cause the sending of an update message to the remote reasoner, what would finally trigger the notification event. The total time measured for this configuration comprises everything observed in the previous configuration, plus the time needed for the reasoning process executed on the local reasoner and the time for the update message going from the local to the remote reasoner.

In Figure 8.6, the two curves show the results for the experiments with and without load. In this graph we notice a peculiar behavior when comparing this configuration with the previously presented configuration. We can observe that the number of subscribers does have little impact until it becomes greater than 300. This is possibly due to the fact that most of the time of the reasoning operation is spent in communication among the reasoners.

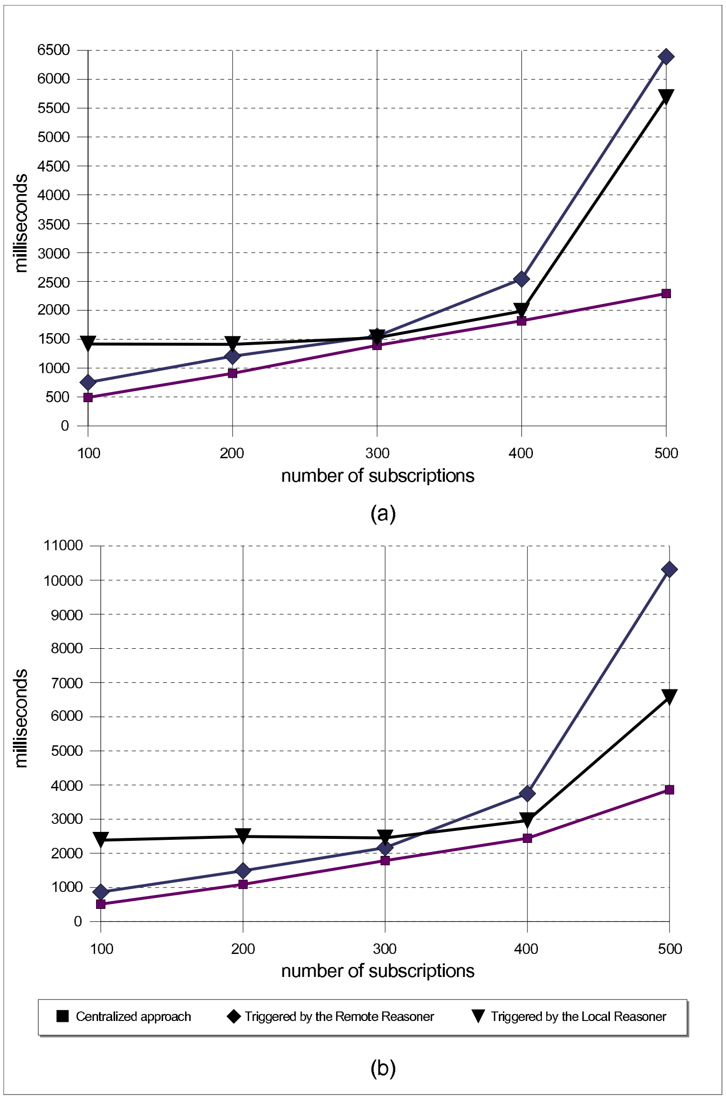


Figure 8.7: Comparison of the three configurations (a) with no load and (b) with context change load.

Result Analysis

Figure 8.7 shows the response time measured for DRS with different load conditions in two different graphs, allowing the comparison of the performance of the three different configurations in each case. In Figure 8.7 (a) we see results found under no load and, in Figure 8.7 (b) the results with the simulation of the arrival of context change messages.

A behavior that becomes clear from both graphs is the fact that for a small number of subscriptions the response time tends to be greater for the decentralized reasoning triggered at the local reasoner. This fact can be explained by the greater overhead imposed by the extra communication starting at the local reasoner, compared with the relatively small amount of time necessary to process less subscriptions. On the other hand, we can observe that increasing the number of subscriptions influences the response time of the system more drastically for the decentralized reasoning triggered at the remote reasoner under a simulated load condition. In this configuration we find the worst case, which is about 10.5s for 500 subscriptions. As, in this case, the remote reasoner is responsible for the *ambient side* reasoning, a context change will probably trigger several simultaneous notifications (to simulated subscribers), causing a high communication overhead.

8.3.2 Communication Traffic and Memory Footprint

In a second experiment, we compared DRS's decentralized reasoning approach both with a centralized and a simple peer-to-peer approach, all of them targeted at the evaluation of inference Rule 8.2, presented below, for the same ontology and test bed used in the first experiment.

Rule 8.2:

$$\textit{isLocatedIn}(\text{“}\textit{Silva}\text{”}, ?r) \wedge \textit{takesPlace}(?s, ?r) \wedge \textit{hasStarted}(?s) \longrightarrow \textit{isBusy}(\text{“}\textit{Silva}\text{”})$$

We measured the communication traffic between the mobile device and the network simulating different configurations. Distributed context providers were simulated by programs running in background and generating context change messages at each 5 seconds. To represent the mobile users, we implemented location providers, variating the *isLocatedIn* property. To represent the ambient context changes, we implemented activity status providers, indicating if a conference session *isAboutToStart*, *hasStarted* or *hasFinished*. While one location provider instance was executed on the netbook, all other context providers were deployed on the stationary machine.

For the decentralized reasoning approach, we set one location provider instance to send context change messages regarding the user's location to the user side DRS on the netbook, and the other location provider instances, together with the activity status provider instance, to send context change messages to the ambient side DRS. For simulating and evaluating the centralized reasoning approach, a DRS server executing on the *ambient side* was set to be the sole reasoner, receiving all context change messages described previously. Since we could not find performance results of any related system implementing peer-to-peer reasoning approach, we decided to simulate a simple P2P system, where each *user side* DRS is responsible for performing all inferences locally using context updates received from all other peers (as in P2P-DR, discussed in Section 3.5). Thus, for setting up the P2P configuration, we executed the location provider representing the user on the netbook, and the additional 499 instances of location providers and the activity status provider were executed on the stationary machine, all configured to periodically broadcast context change messages to the user side DRS.

	Centralized	**Decentralized**	**Peer-to-peer**
Communication traffic	790 Bytes/s	81.7 Bytes/s	303.5 KBytes/s
Memory footprint	—	20.2 KBytes	23.5 KBytes

Table 8.1: Communication overhead and memory footprint measured for different reasoning configurations simulated.

For these three approaches we measured the communication traffic at the netbook — considering both received and sent context change messages and the messages exchanged between the DRS services — during a simulation period of 5 minutes. We verified also the memory footprint of the reasoner executing on that device. From the measured values presented in Table 8.1, we observe that our decentralized approach minimizes the communication with the mobile device and, due to a smaller ontology, requires less memory at the user side.

8.4 Discussion

The response times observed in the first set of performance tests (Section 8.3.1) show that a some communication overhead is caused by the messages exchanged between the reasoners to perform the *cooperative reasoning*. In our simulation, the centralized reasoning showed a better performance in all cases, with a drastic difference for more than 300 subscribers. It is important to highlight, though, that in this configuration, the ambient's DRS and CMS would collect the context data of all sensors and mobile devices in the environment. In this experiment we did not take into account the huge communication overhead caused by periodic context data tranfer between the context providers and the ambient service in the centralized approach, as we were interested only in measuring the response time.

Thus, the results do not invalidate the use of the *cooperative reasoning process* in real scenarios. On the contrary, it proves its applicability, as in the worst case the response time was 10.5s, which is acceptable for our target scenario with up to 500 devices. It means that a user could be notified about the imminent start of a conference session 10.5s after it is signalled by the ambient infrastructure. Moreover, as discussed in Section 5.1.3, if the context change periodicity is not lesser than about 30s — which is an acceptable value for our scenario, as we do not expect a user to change his location more frequently than that — the inference process would converge to a stable result.

Nevertheless, the increase in the slope of the graphs for more than 300 subscribes indicates that the present implementation is not scalable, i.e., it is not ready for use in scenarios where a huge number of clients request the reasoning service. We observe that the response time was greatly affected by the communication overhead when the number of subscriptions grew. On the other hand, the results observed in the second experiment (Section 8.3.2) showed that the communication traffic was much higher in the centralized configuration than in the *cooperative reasoning*. Besides that, the reasoning service in the cooperative approach presented a smaller memory footprint.

In the next section, we discuss the contributions and limitations of the proposed approach, presenting also some topics of future work.