HyLAR: Hybrid Location-Agnostic Reasoning

Mehdi Terdjimi, Lionel Médini, Michael Mrissa

HyLAR: Hybrid Location-Agnostic Reasoning

Mehdi Terdjimi, Lionel Médini, and Michael Mrissa

Université de Lyon, LIRIS
Université Lyon 1 - CNRS UMR5205
F-69622, France
{mehdi.terdjimi,lionel.medini,michael.mrissa}@liris.cnrs.fr

Abstract. The question of client-side reasoning is crucial to semantic web application design as client performances drastically increase. It is an opportunity for ubiquitous devices to use semantic technologies. In this paper, we propose a lightweight, modular and adaptive architecture developed in JavaScript for hybrid client/server side reasoning. We evaluate the performance of the reasoning process with different browsers, devices and network conditions, and discuss the best strategy with respect to the envisioned reasoning tasks.

Keywords: mobile reasoning, ubiquitous semantic web, client-side reasoning

1 Introduction

To address scalability concerns that arise with high numbers of simultaneous requests, web application designers dispose of several tools, among which caching static data and deferring code execution from the server to the client side. But even if in average, client processing resources augment at a fast pace, they remain heterogeneous and in some cases, too limited to execute heavy calculation processes. Adaptivity and flexibility depending on the client resources is therefore necessary. This concern also arises with semantic web technologies: solving SPARQL queries for a large number of clients can require heavy reasoning processes and cause endpoints unavailability. Client-side reasoning is therefore to consider while designing a semantics-enabled web application. Moreover, mobile devices and smart appliances provide an opportunity for semantic technologies to exploit the paradigm of ubiquitous computing and provide knowledge sharing and reasoning facilities wrt. standards on different devices. But again, their diversity and heterogeneity require the ability to defer reasoning tasks on a client or to perform them on the server if the client is unable to handle them.

In this paper, we propose an approach and a corresponding architecture for locating the different steps of a reasoning process on either server or client side, and evaluate the execution times of each of these steps depending on their locations. We overview different approaches allowing mobile reasoning in Section 2. In Section 3, we propose our contribution: we distinguish between the reasoning steps that can be pre-processed and those that must be processed at request

time. We then present an architecture to deploy these steps on either server or client-side. We evaluate performances in three different deployment situations. We discuss our results in Section 4 and give work perspectives in Section 5.

2 Mobile reasoning state of the art

On the one hand, web servers are often facing breakdowns and unavailability issues when serving semantic data as SPARQL endpoints. On the other hand, client-side semantic processing must deal with resource limitations, especially on mobile devices. The main motivation that conducted to the following reasoning approaches was the need to optimize reasoning for resource-constrained devices.

Krishnaswamy and Li [3] discuss challenges in mobile OWL reasoning. They describe how to reduce load by configuring reasoners for precise tasks using limited description logics. Kollia and Glimm [2] propose to rewrite costly-to-evaluate axiom templates into smaller templates. A Triple Pattern Fragments [7] (TPF) interface is a Web API to RDF data where clients can ask for triples matching a certain triple pattern. This approach relies on intelligent clients that query TPF servers to address the problem of scalability and availability of SPARQL endpoints. However, the use of a server is necessary.

Current existing mobile reasoners are based on first-order logic (FOL), managing Tbox (schema), Rbox (roles) and Abox (assertions). Sinner and Kleemann's KRHyper [6] is a novel-tableaux based algorithm for FOL, but encounters memory exhausting problems when the reasoning task becomes too large for the device, as pointed out in [3]. Based on $\mathcal{ALCN}$, Mine-ME 2.0 from Ruta et al. [5] is used on Android devices. Embedded reasoners such as the $\mathcal{EL}+$ reasoner proposed by Grimm et al. [1] are capable of reasoning on large Tboxes due to the limitations of $\mathcal{EL}$ (no individuals nor concept disjointness). But neither [5] nor [1] provide web client access. An approach to embed a reasoner in mobile devices is to rely on web standards and run it in a web browser in Javascript. Other works are oriented towards web-based technologies. They rely on Javascript reasoners that can be embedded in mobile devices and ran on the device browser. EYE[1] is a NodeJS[2]-compatible reasoner capable of inferring on FOL rules, performing server-side reasoning while a client widget renders a graphical interface for SPARQL querying. As far as we know, the reasoner has not been ported onto the client side. Based on the JSW Toolkit, OWLReasoner[3] allows client-side processing of SPARQL queries on OWL 2 EL ontologies. After parsing an ontology, a "classification" step performs its deductive closure to return its Tbox and Abox and converts them into a relational database. SPARQL queries sent to the reasoner are rewritten into SQL queries, and processed on the database. To the best of our knowledge, OWLReasoner is the only full-JavaScript OWL 2 EL that can be used offline in a web client. However, its SPARQL engine is limited to basic rule assertions.

[1] http://reasoning.restdesc.org/
[2] https://nodejs.org/
[3] https://code.google.com/p/owlreasoner/

3 Contribution

Our contribution aims at designing reasoning processes that "bridge the gap between the web and the semantic web"[4]. The first envisioned means to tackle this problem is to make better use of standard web mechanisms, such as HTTP caching and proxying. The second one is to cope with recent advances in web applications and exploit client resources by deferring code execution on the client. We focus on JavaScript-enabled reasoners, so that the same parts of code can both be deployed on the client and server sides, to provide an adaptable reasoning task. We also plan to build an architecture with respect to W3C standards, using description logics over FOL. For these reasons, our implementation is based on OWLReasoner.

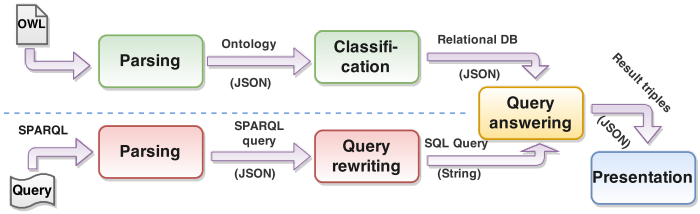

Fig. 1. Steps of classification and reasoning processes in OWLReasoner

We therefore aim at separating reasoning tasks executed once and preprocessed on server side (parsing and classification steps) and tasks executed when a query is sent to the reasoner (SPARQL query parsing, rewriting and reasoning). These steps are depicted in Figure 1. The following subsections characterize the most suitable architecture by evaluating the reasoning efficiency wrt. several parameters: client resource limitation, number of simultaneous clients requesting the SPARQL endpoint, size of the processed ontology and network latency.

3.1 Implementation

We here introduce the Hybrid Location-Agnostic Reasoner (HyLAR)[5] architecture, used to perform our experiments. HyLAR is based on the separation of OWLReasoner JSW modules that perform ontology classification (JSW Classifier), ontology and SPARQL query parsing (JSW Parser) and reasoning (JSW Reasoner). These steps are packaged as Node.js modules and AngularJS[6] services. This way, they can be executed on either the server or client. On the client side, the reasoner modules can be embedded either in a regular angular service,

[4] Phil Archer, W3C, Semweb.Pro Paris, Nov. 2014
[5] https://github.com/ucbl/HyLAR (GitHub)
http://dataconf.liris.cnrs.fr/owlReasoner/ (website)
[6] http://www.angularjs.org

or in a web worker. They are queried by an independent angular service using an asynchronous promise pattern, so that the main service is totally agnostic about the location of the reasoning modules.

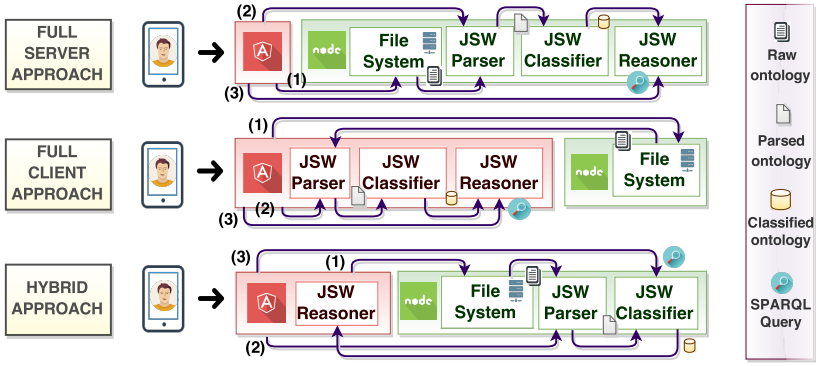

Fig. 2. Architectures used for our evaluation

3.2 Evaluation

We consider four scenarios, representing all possible steps of the reasoning process: the scenario (0) for loading client scripts; (1) for loading a raw ontology; (2) for performing ontology parsing, classification and loading the resulting JSON object; (3) for SPARQL query processing. We used the architecture presented above to evaluate the overall reasoning process times in three situations: full server-side, full client-side and hybrid (server-side parsing and classification, and client-side query processing). Figure 2 shows (1), (2) and (3) for each situation. Additionally, for the hybrid and full client-side variants, client-side parts are evaluated both with and without web worker. We assume that scripts and ontologies are available on the server. All scenarios conform to a query-processing-response pattern. In the result tables, we noted [Q] the time for the client's request to reach the server; [P] the processing time and [R] the time for the server response to reach the client. Depending on the scenario and location of the calculations, some parts of this steps/patterns are considered immediate (e.g. querying the local reasoner to process a query). They are noted in the result tables as not applicable. Each evaluation is tested on two ontologies[7]: A (1801 class assertions and 924 object property assertions) and B (12621 class and no object property assertions)[8].

[7] We chose ontologies of "reasonable" sizes, representing datasets that a web application can require. For instance, ontology B has actually been used to perform client-side recommendation in [4]

[8] Due to OWLReasoner query engine limitations that does not currently allow querying individuals nor data property assertions, our evaluations are limited to class and object property assertions. The reader will see in the discussion that even if ontology complexity changes calculation times, it leads to the same conclusions.

Ontologies A / B	**[R0]**	**[Q1]**	**[R1]**	**[Q2]**	**[R2]**	**[Q3]**	**[R3]**
Remote server	334	54	110 / 275	119 / 120	167 / 647	146 / 154	61 / 85

Table 1. Network delays (in ms)

Ontologies A / B	**[P2] (no worker)**	**[P2] (worker)**	**[P3] (no worker)**	**[P3] (worker)**
Inspiron (Chrome)	790 / 27612	764 / 26464	28 / 101	24 / 88
Lumia (IE)	1989 / 54702	1883 / 53801	156 / 198	144 / 185
Galaxy Note (Firefox)	2954 / 81255	2872 / 79752	465 / 2988	440 / 2872
Server (Node.js)	780 / 20972	n/a	35 / 37	n/a

Table 2. Classification [P2] and reasoning [P3] times (in ms)

A first evaluation shows network request and response delays for each scenario. It is realized by simulating a remote server with Clumsy 0.2[9]. [R0] is the time for the client to load scripts and following are the respective query/response times for [Q1]/[R1] retrieving the raw ontlogy, [Q2]/[R2] retrieving the classification result and [Q3]/[R3] sending the SPARQL query and retrieving results. A second evaluation compares processing times for [P2] classification and [P3] reasoning in three different configurations: a Dell Inspiron (with Chrome), a Nokia Lumia 1320 (Snapdragon S4 @ 1700 MHz, with Internet Explorer), a Samsung Galaxy Note (ARM cortex A9 Dual-Core @ 1,4 GHz, with Firefox) and a Node.js server set up in the Inspiron.

4 Discussion

As expected, we can see in Table 2 that the server has the best results for the classification processing time and can use caching. Even if the raw ontology is faster to load than the classification results, loading scripts and data on the client is much faster than performing the same classification step on each client. Therefore, it makes no sense to defer and duplicate heavy calculations onto clients, rather than pre-calculating them on the server and caching results. Table 2 shows an important difference between configurations: we keep reasonable processing time for the query answering task in good to average configurations (e.g. Inspiron and Lumia), but the older Galaxy Note is ten times slower than the server. For such limited resource devices, the server could therefore take over the answering process. More generally, for M clients and N queries/client, the three configuration calculation times can be calculated as follows[10]:

- server-side: $P2_{server} + M \times N \times (Q3 + P3_{server} + R3)$
- client-side: $M \times (R0 + Q1 + R1) + P2_{client} + N \times P3_{client}$
- hybrid: $P2_{server} + M \times (R0 + Q2 + R2) + N \times P3_{client}$

Globally, the evaluation shows that choosing a location for the query answering process is not as simple as for the classification step. For low client resources, ontology usage (number of queries per client) and server load (number of clients),

[9] http://jagt.github.io/clumsy/

[10] Server-side classification (performed once and then cached) and client-side calculations (performed in parallel) are only counted once.

it can be more efficient to perform this step on the server. But as these parameters grow, it appears that relocating query processing on the client can be a good strategy, since queries can be processed autonomously on each client. A more powerful server would shorten server-side response times, resulting in shifting the strategy switching point, but higher performance, and therefore scalability, can be achieved by deferring this step on clients.

5 Conclusion and future work

In this paper we propose HyLAR, an adaptable architecture for OWL reasoning, based on OWLReasoner. The main benefit of our architecture is the possibility to switch the different parts of the reasoning code on either client or server side. We evaluate three implementations (full server, full client or hybrid) on different devices, using two ontologies of different sizes. Experiments show that deductive closure should be performed on the server side. Besides that, client-side processing has an important initial cost and is, as always, dependent on the client resources. Therefore, as performing the whole process makes sense for a restricted number of queries, it is worth deploying a reasoner and loading ontologies on clients when scalability concerns come into play. Our next move is to define a context-aware approach to automatically adapt the reasoning process to ontology size, client and network conditions. Another perspective for our approach is to study the impact of INSERT and UPDATE queries, as well as other reasoning approaches. To do this, we need to improve or replace the limited reasoner embedded in HyLAR.

Acknowledgement

This work is supported by the French ANR (Agence Nationale de la Recherche) under the grant number <ANR-13-INFR-012>.

References

1. Grimm, S., Watzke, M., Hubauer, T., Cescolini, F.: Embedded $\mathcal{EL}$+ reasoning on programmable logic controllers. In: The Semantic Web–ISWC 2012, pp. 66–81. Springer (2012)
2. Kollia, I., Glimm, B.: Optimizing sparql query answering over owl ontologies. arXiv preprint arXiv:1402.0576 (2014)
3. Krishnaswamy, S., Li, Y.F.: The mobile semantic web. In: Proceedings of the companion publication of the 23rd international conference on World wide web companion. pp. 197–198. International World Wide Web Conferences Steering Committee (2014)
4. Médini, L., Bâcle, F., Nguyen, H.D.T.: DataConf: Enriching conference publications with a mobile mashup application (May 2013), http://liris.cnrs.fr/publis/?id=6032, lIME'2013 Workshop at WWW'2013 conference

5. Ruta, M., Scioscia, F., Loseto, G., Gramegna, F., Ieva, S., Di Sciascio, E.: Minime 2.0: powering the semantic web of things. In: 3rd OWL Reasoner Evaluation Workshop (ORE 2014)(jul 2014) (2014)
6. Sinner, A., Kleemann, T.: Krhyper–in your pocket. In: Automated Deduction–CADE-20, pp. 452–457. Springer (2005)
7. Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R.: Querying datasets on the web with high availability. In: The Semantic Web–ISWC 2014, pp. 180–196. Springer (2014)