OWL API for iOS: Early Implementation and Results

Michele Ruta(✉), Floriano Scioscia, Eugenio Di Sciascio, and Ivano Bilenchi

Politecnico di Bari, via E. Orabona 4, 70125 Bari, Italy
{michele.ruta,floriano.scioscia,eugenio.disciascio}@poliba.it,
ivanobilenchi@gmail.com

Abstract. Semantic Web and Internet of Things are progressively converging, but the lack of reasoning tools for mobile devices on the iOS platform may hinder the progress of this vision. The paper presents an early redesign of OWL API for iOS. A partial port has been developed, effective enough to support mobile reasoning engines in a moderately expressive fragment of OWL 2. Both architecture and mobile-oriented optimization are sketched and preliminary performance results are discussed.

1 Introduction and Motivation

Semantic Web technologies are a key enabler of interoperability and intelligent information processing not only in the WWW, but also in the so-called Internet of Things (IoT). Application scenarios include supply chain management [5], (mobile) sensor networks [14], building automation [15] and more. The Semantic Web and the IoT paradigms are progressively overlapping in the *Semantic Web of Things* (SWoT) vision [14,17]. SWoT enables semantic-enhanced pervasive computing by associating informative fragments to multiple heterogeneous micro-devices in a given environment, each acting as a knowledge micro-repository. Rather than the batch processing of large ontologies and complex inferences prevalent in traditional Semantic Web scenarios, SWoT requires quick reasoning and query answering on sets of relatively elementary resources, in order to provide mobile agents with on-the-fly autonomous decision capabilities. The ever-increasing computing potentialities of mobile devices allow processing of rich and formally structured information without resorting to centralized nodes and support infrastructures. For a full accomplishment of this vision, reasoning engines and library interfaces are needed on the most relevant mobile device platforms.

iOS is the second largest mobile Operating System (OS) worldwide, with over 1 billion iPhone units sold (as of July 2016 [1]) as well as iPad and iPod Touch devices. While Android has a larger active device count, iOS has been more eagerly adopted in business [22]. Higher hardware and OS uniformity, a stricter security model [12], enterprise IT (Information Technology) department support tools and a stronger focus on usability are among the reasons. Business sectors

M. Dragoni et al. (Eds.): OWLED-ORE 2016, LNCS 10161, pp. 141–152, 2017.
DOI: 10.1007/978-3-319-54627-8_11

ranging from healthcare to sales management and research exhibit a thriving market of iOS software solutions. Nevertheless, a full adoption of Semantic Web technologies has not been possible on iOS so far. A recent survey [11] found no Web Ontology Language (OWL) [21] reasoners implemented in Objective-C or Swift, the only two languages natively supported on iOS. In fact Java is by far the most popular implementation language for that. Several reasoners originally developed for Java Standard Edition have been ported to the Java-based Android platform so as to run on mobile devices [3]; likewise Java-based reasoning engines expressly designed for mobile devices also exist, including *mTableau* [19] and *Mini-ME* [18], which work on Java Micro Edition and Android, respectively. Similarly, all main OWL Knowledge Base (KB) management libraries are Java-oriented. Among them the *OWL API* [7] is the most adopted one. Java code requires a rewriting effort toward Objective-C or Swift in order to be adopted on iOS (whereas C/C++ list can be reused in Objective-C projects by writing proper wrappers).

The lack of iOS Semantic Web tools hampers the development of multi-platform semantic-enabled mobile applications to follow the rapid pace of the IoT (r)evolution, which may stifle the SWoT vision as a whole [6]. Although toolkits (such as *Oracle Mobile Application Framework*[1] and *Codename One*[2]) allow cross-platform mobile development in Java language and deployment to iOS devices, they are affected by various cost, efficiency and inconvenience issues. Automatic source transpilers from Java to Objective-C (such as *J2ObjC*[3]) also exist, but they are primarily intended to allow multi-platform projects to share as much business logic code as possible: transpiling existing software is significantly harder from a development point of view, especially considering that library dependencies must be recursively translated, or suitable alternatives need to be found or developed. Automatic translation is also not very flexible, as the core architecture of the source project cannot be altered without a considerable amount of work: this is an issue in this specific case, since significant architectural changes to the OWL API internals are desirable in order to ensure high performance (both in terms of time and memory) in SWoT scenarios.

In order to allow developing mobile reasoners for iOS, we present here the first results of porting the OWL API to iOS. This approach was preferred over writing a new application programming interface because the OWL API is a *de facto* standard for manipulating DL KBs and has a large user community. A functional subset of the OWL API was implemented, able to load and process KBs in an OWL 2 fragment corresponding to the $\mathscr{ALEN}$ Description Logic (DL) –with the addition of role hierarchies– in RDF/XML syntax. The ported library was written in Objective-C, to be used by both Objective-C and Swift applications. It runs on iOS and macOS without modification, as it does not use iOS-specific APIs. Experimental tests verified the correctness of the implementation and exhibit satisfactory results also in comparison with the original Java OWL API

[1] http://www.oracle.com/technetwork/developer-tools/maf/overview/index.html.
[2] https://www.codenameone.com/.
[3] http://j2objc.org.

on macOS. The library is released[4] as open source under the *Eclipse Public License* and can already support a future Mini-ME port for iOS.

The remainder of the paper is as follows: Sect. 2 provides background on the OWL API and porting strategies, while Sect. 3 describes the developed library; experimental results are in Sects. 4 and 5 closes the work.

2 Background

The OWL API [7] is the most commonly used front-end for OWL-based Knowledge Base Management Systems (KBMS) [3,11]. Other interfaces include Jena[5], Protégé-OWL API [8] and OWLlink [10]. The Jena library provides ontology manipulation APIs for Resource Description Framework (RDF) [16], RDF Schema (RDFS) [4] and OWL models, and an inference API to support reasoning and rule engines. The Protégé-OWL API [8] leverages Jena on OWL and is particularly effective for developing graphical applications. OWLlink [10] is a client/server protocol on top of HTTP for KB management and reasoning. The OWLlink API [13] implements OWLlink on top of the OWL API and therefore could be also ported to iOS.

The OWL API is a Java library defining a set of interfaces to manipulate OWL 2 KBs. It supports loading and saving in several syntaxes, including RDF/XML, Turtle, the Manchester Syntax and more. The implemented model gives an abstract representation of concept, property, individual and axiom types in OWL 2 through four interface hierarchies, all having `OWLObject` as a common ancestor. The model interfaces do not depend on any particular concrete syntax. The `OWLOntologyManager` interface allows creating, loading, changing and saving KBs, alleviating the burden of choosing the appropriate parsers and renderers. Finally, `OWLReasoner` is the main interface for interacting with OWL reasoners. It provides methods to check satisfiability of classes or ontologies, to compute class and property hierarchies and to check whether axioms are entailed by a KB.

The benefits of porting traditional Semantic Web reasoners like *FaCT++* [20] to mobile platforms should be questioned, as they were designed primarily to run inference services such as classification and consistency check on large ontologies and/or expressive DLs. In ubiquitous contexts, ABox reasoning and non-standard inference services are often more useful, because mobile agents must provide on-the-fly answers to usually smaller problems in moderately expressive KBs [18]. On the other hand, importing a C/C++ library for RDF parsing can be a sensible choice to build an OWL manipulation library or a reasoner. Among the many available tools, the *Redland* [2] suite stands out for functional completeness, standards compliance and code maturity. Other tools like *owlcpp* [9] are less suitable for working in an OWL API port, as they only parse individual RDF triples.

[4] GitHub repository: https://github.com/sisinflab-swot/OWL-API-for-iOS.
[5] Apache Jena project: https://jena.apache.org/.

3 Reasoning on iOS Devices: OWL API Porting

The proposed software is a port of the OWL API version 3.2.4. It was implemented in Objective-C –deemed as more mature and stable than Swift– as an *iOS Framework*, *i.e.*, a library easily used by applications through dynamic linking. The following subsections report on the general architecture and devised performance optimization, respectively.

3.1 Models and Architecture

The OWL API entry point is the `OWLManager` class implementing the `OWLOntologyManager` interface, which allows loading and manipulating a KB. As shown in Fig. 1, the library architecture includes two basic components, the *OWL Model* and the *OWL Parser*. Java *interfaces* were translated to the corresponding Objective-C *protocols*, therefore the Model is interface-wise as the one of the OWL API. The current version does not model the whole OWL 2 language, but a fragment of it exhaustive enough to manage KBs in the $\mathcal{ALEN}$ DL with role hierarchies. In more detail, classes; property restrictions; Boolean class expressions; object properties; declaration, subclass, disjointness, equivalence, domain, range, class assertion and object property assertion axioms are modeled.

The Parser module uses the *Raptor* RDF parser from Redland to deserialize RDF/XML documents (other syntaxes were not considered at this early stage)

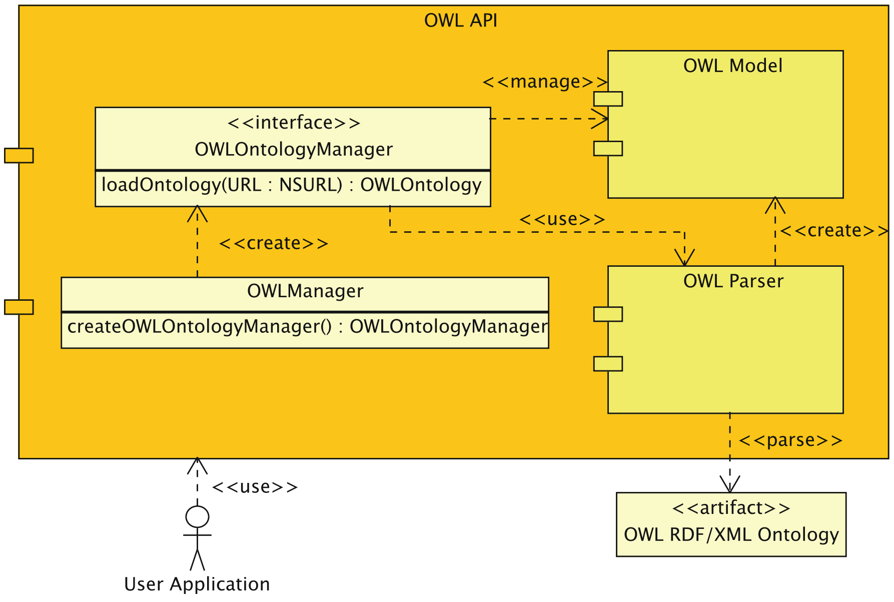

Fig. 1. Main components of the ported library

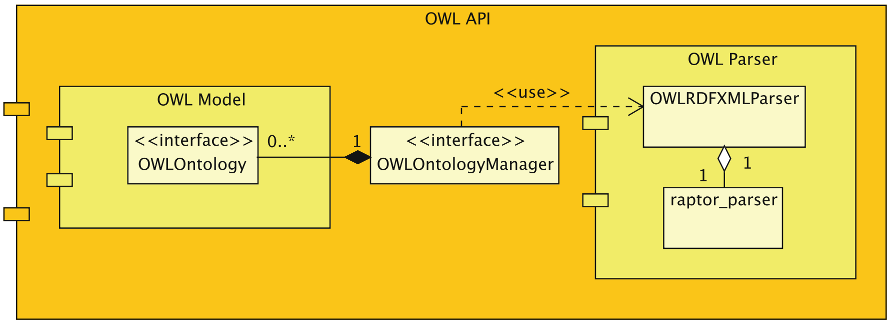

Fig. 2. Detail of the interaction between the Model and Parser modules

into streams of RDF statements. The OWLOntologyManager invokes Raptor through an `OWLRDFXMLParser` wrapper, which further processes the RDF statement stream in order to create an in-memory representation of the referenced OWL constructs and returns a fully populated `OWLOntology` object. The interaction between the Model and Parser modules is detailed in Fig. 2.

OWL ontology parsing from RDF triples does not follow the original OWL API approach. A simpler and leaner architecture was adopted, particularly fit for small and medium sized KBs. The implementation of OWLOntology interface is built through the `OWLOntologyInternals` class, which is populated incrementally during the parsing. It contains data structures such as maps and sets. As pictured in Fig. 3, `OWLStatementHandlerMap` associates each type of statement to a proper handler, as allowed by the Raptor library. Handlers are implemented as Objective-C *blocks*, which are similar to Java *lambdas* or C *function pointers*. Furthermore, the *builder* pattern was adopted to create instances within the Model component incrementally, because OWL axioms can derive from a variable number of RDF statements.

3.2 Optimization

Optimization effort basically focused on an efficient use of memory, which is the most constrained resource on mobile devices. Execution time was also profiled and optimized wherever possible. In what follows followed optimization directions are outlined.

Architectural optimization. The whole Model component is composed of *immutable* objects. This allows having just one copy of every instance in memory, saving space and time; moreover, it makes the whole component thread-safe. With immutable objects, object hashes can be cached to speed up the very frequent accesses to associative data structures. As a further optimization, if one guarantees that equal objects have the same memory address, the address itself is a perfect hash and equality check becomes just a pointer comparison. In order to make this property true, the library uses the *NSMapTable*

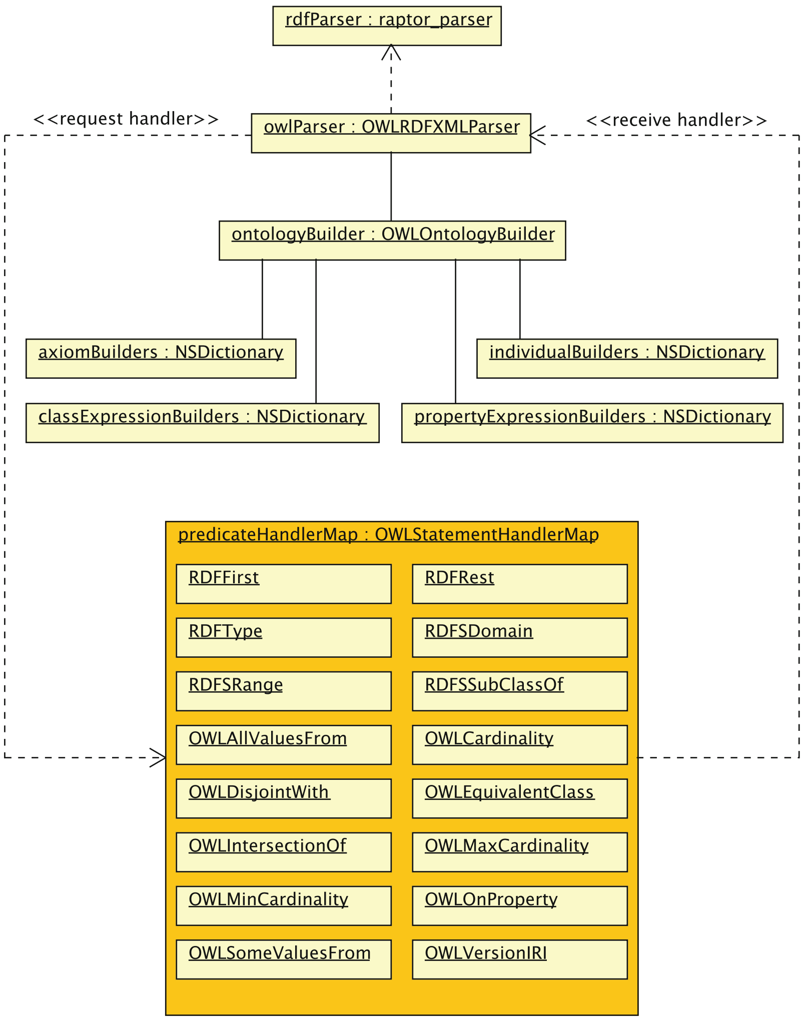
rdfParser : raptor_parser
<<request handler>>
owlParser : OWLRDFXMLParser
<<receive handler>>
ontologyBuilder : OWLOntologyBuilder
axiomBuilders : NSDictionary
individualBuilders : NSDictionary
classExpressionBuilders : NSDictionary
propertyExpressionBuilders : NSDictionary
predicateHandlerMap : OWLStatementHandlerMap
RDFFirst
RDFRest
RDFType
RDFSDomain
RDFSRange
RDFSSubClassOf
OWLAllValuesFrom
OWLCardinality
OWLDisjointWith
OWLEquivalentClass
OWLIntersectionOf
OWLMaxCardinality
OWLMinCardinality
OWLOnProperty
OWLSomeValuesFrom
OWLVersionIRI

Fig. 3. Main objects of the Parser module

class of the Objective-C Foundation framework as hash table, which supports pointer identity for equality and hashing. NSMapTable was set up to use *weak* references to allow de-allocation of unused objects. This approach, however, is beneficial only in hash tables with low collision rates: this was not found out to be true for all OWL API model classes. Therefore it was adopted just for entities (classes, object properties, individuals) and some axioms considered as performance-critical after profiling tests. These optimizations allowed to roughly halve the measured parsing turnaround times w.r.t. the initial implementation.
Parsing optimization. During the parsing process, each RDF triple is wrapped in a `RDFStatement` instance, which is discarded as soon as it is not used anymore.

Furthermore, builders cache the objects they populated, saving both time and memory (in case of similar but not identical instances). Finally, axiom builders are de-allocated in groups: this reduced the observed memory usage peak during parsing by about 30% in preliminary tests.

4 Experiments

The formal correctness and completeness of results provided by the iOS library was evaluated on a set of 34 KBs, obtained from the 2012 OWL Reasoner Evaluation Workshop reference dataset[6] considering all the KBs in the supported $\mathcal{AL}$, $\mathcal{AL}+$ and $\mathcal{ALE}$ DLs. The original Java OWL API 3.2.4 was leveraged as a test oracle. After parsing, the following tests were performed against each KB as significant examples: (i) retrieval of all axioms; (ii) retrieval of all axioms of a given kind; (iii) retrieval of all classes, individuals and properties; (iv) retrieval of all disjoint, equivalent and subclass axioms. The iOS library correctly parsed every KB in the test set, and the returned output of all retrieval tasks proved to be equivalent to the Java OWL API.

Performance evaluation was carried out on a subset of the KBs used for the correctness tests, reported in Table 1. They were selected because they are representative of both traditional and SWoT scenarios, while allowing to sample the performance of the iOS library when working with KBs of varying size. For each KB, three tests were performed: (i) parsing turnaround time; (ii) memory usage peak; (iii) query turnaround time. Each test was repeated five times: for turnaround time tests, the average of all runs was taken. For memory tests, the final result is the average of the last four runs, in order to consider a worst-case scenario due to potential memory leaks. Test devices are listed in Table 2.

Table 1. Knowledge bases used in the performance tests.

Knowledge base	DL	Category	Axioms	Size (kB)
spider_anatomy.owl	$\mathcal{ALE}$	Small	1392	187
brenda.owl	$\mathcal{ALE}$	Medium	14262	1515
mammalian_phenotype.owl	$\mathcal{AL}+$	Large	46081	4289
teleost_taxonomy.owl	$\mathcal{AL}$	Large	195351	21878

Figure 4 shows the results of parsing turnaround time tests: times grow linearly with the size of the parsed ontologies, and small-to-medium ontologies are parsed in about one second or less on devices more than two years old (iPhone 5s). This result is aligned with the performance goals of a mobile reasoner, especially considering that parsing only happens once per usage session, rather than each time a query is submitted to the reasoner.

[6] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/.

Table 2. Devices used for performance evaluation.

Device	OS	CPU	Arch.	RAM
Retina MacBook Pro 2014	OS X 10.11.5	Intel Core i7-4870HQ@2.5 GHz	64 bit	16 GB DDR3@ 1600 MHz
iPhone 6s	iOS 9.0.2	Apple A9@1.8 GHz	64 bit	2 GB LPDDR4
iPhone 5s	iOS 9.3.2	Apple A7@1.3 GHz	64 bit	1 GB LPDDR3
iPhone 5	iOS 9.3.2	Apple A6@1.3 GHz	32 bit	1 GB LPDDR2E

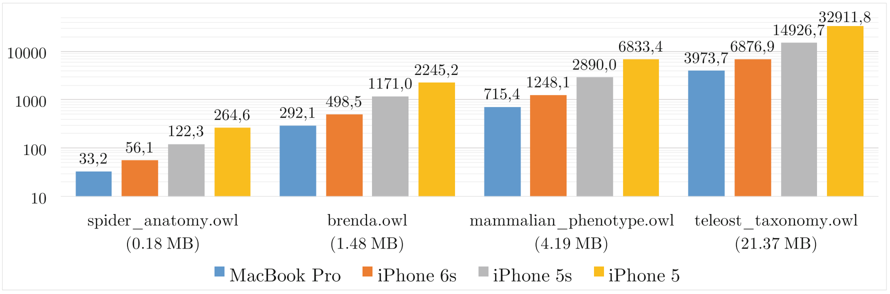

Fig. 4. iOS API parsing turnaround time (ms).

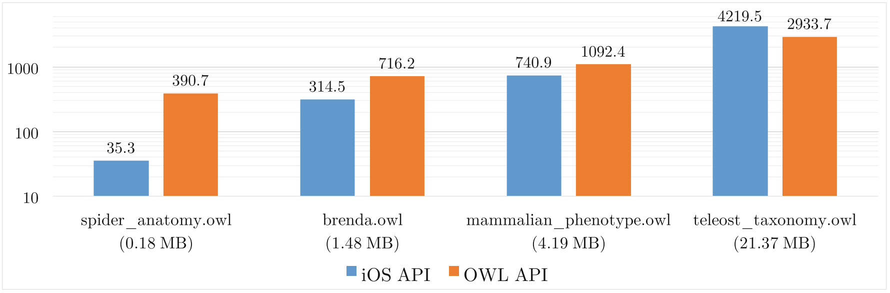

Fig. 5. Comparison of the parsing turnaround time between the iOS API and OWL API (ms).

Figure 5 compares parsing times provided by the iOS API with OWL API on the MacBook Pro testbed. First-run results were considered in this test only, in order to evaluate parsing performance in real usage, since a KB is usually loaded once and queried multiple times. Subsequent runs would provide less realistic results due to in-memory caching. The iOS API shows competitive performance on every test KB, outperforming the OWL API when parsing the small to medium-large ones.

Figure 6 reports on memory usage peak during parsing, which grows linearly with the size of the parsed ontology. Measured values are roughly similar on MacBook Pro, iPhone 6s and iPhone 5s, while they are about 40% lower on

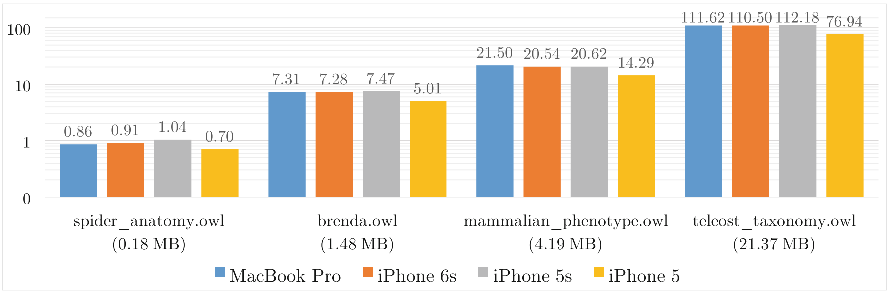

Fig. 6. Memory peak while parsing (MB).

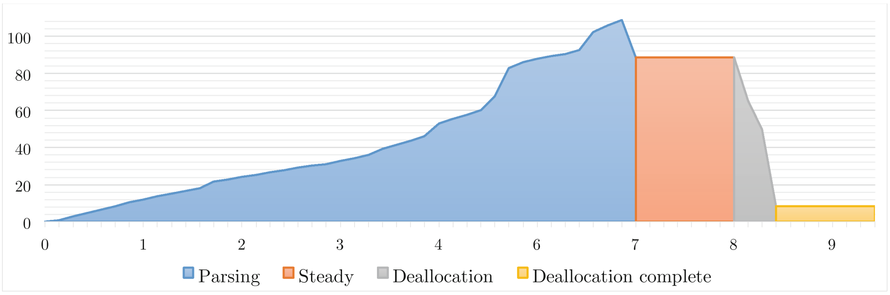

Fig. 7. Memory usage (MB) as a function of time (s).

iPhone 5: this is likely due to it being the only 32-bit device among the four. The results of this test were overall satisfactory, since the required memory is consistent with RAM availability of modern iOS devices.

Figure 7 shows the memory usage trend while parsing and querying the largest KB in the test set (*teleost_taxonomy.owl*) on iPhone 6s. Four phases can be pinpointed: memory usage raises and reaches its peak value during the ***parsing*** phase; during the ***steady*** phase the KB is fully loaded and can be queried; memory is released when the KB is ***de-allocated***.

Figure 8 shows the turnaround times for the retrieval of all classes in the ontology. This specific query is unrealistic, but it was chosen nonetheless as a stress test for the library. As also seen in the previous tests, times grow linearly with the size of the queried ontology. In order to contextualize the obtained results, query times were compared to OWL API on the MacBook Pro testbed: as reported in Fig. 9, the iOS API outperformed OWL API on every test ontology, confirming its suitability to be used in mobile and pervasive scenarios.

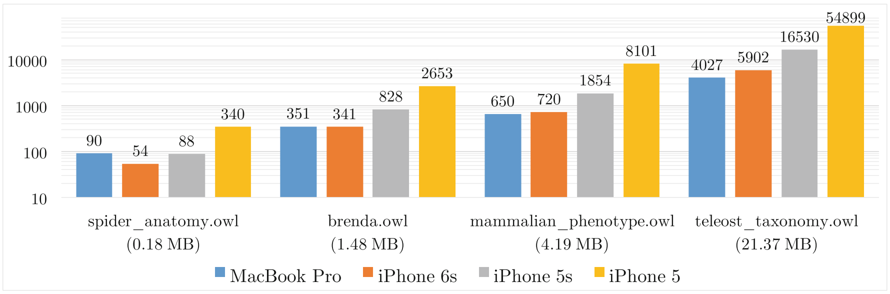

Fig. 8. All classes retrieval query turnaround time (µs).

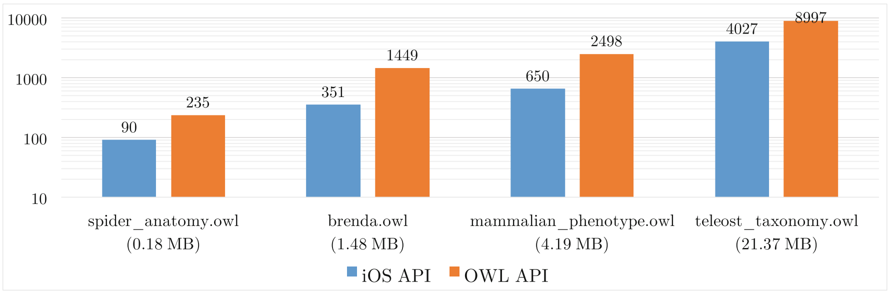

Fig. 9. Comparison of the turnaround time for all classes retrieval query between the iOS API and OWL API (µs).

5 Conclusion and Future Work

The paper presented early results of porting the OWL API to Objective-C, targeting mobile reasoning on the iOS platform. The developed library can run unmodified also on macOS. Early experiments on a small set of ontologies showed correctness of implementation and satisfactory performance in KB parsing and manipulation.

In its current form, the proposed library is ready to support the port of the Mini-ME mobile matchmaking and reasoning engine [18] to iOS, which was the main motivation for the endeavor and is the first planned future work. As a further hope, it will benefit the community as a whole and –possibly with the help of other developers– will grow toward a complete port, aligned with latest OWL API version.

Acknowledgment. The authors acknowledge partial support of Apulia region cluster project PERSON (PERvasive game for perSOnalized treatment of cognitive and functional deficits associated with chronic and Neurodegenerative diseases).

References

1. Apple Inc.: Apple celebrates one billion iPhones. http://www.apple.com/newsroom/2016/07/apple-celebrates-one-billion-iphones.html. Accessed 15 Sep 2016
2. Beckett, D.: The design and implementation of the Redland RDF application framework. Comput. Netw. **39**(5), 577–588 (2002)
3. Bobed, C., Yus, R., Bobillo, F., Mena, E.: Semantic reasoning on mobile devices: do Androids dream of efficient reasoners? Web Semant. Sci. Serv. Agents World Wide Web **35**, 167–183 (2015)
4. Brickley, D., Guha, R.V.: RDF schema 1.1. W3C Recommendation **25**, 2004–2014 (2014). https://www.w3.org/TR/rdf-schema/
5. Giannakis, M., Giannakis, M., Louis, M., Louis, M.: A multi-agent based system with big data processing for enhanced supply chain agility. J. Enterp. Inf. Manage. **29**(5), 706–727 (2016)
6. Hillerbrand, E.: Semantic web and business: reaching a tipping point? In: Workman, M. (ed.) Semantic Web: Implications for Technologies and Business Practices. Springer, Heidelberg (2016)
7. Horridge, M., Bechhofer, S.: The OWL API: a Java API for OWL ontologies. Semant. Web **2**(1), 11–21 (2011)
8. Knublauch, H., Fergerson, R.W., Noy, N.F., Musen, M.A.: The Protégé OWL plugin: an open development environment for semantic web applications. In: McIlraith, S.A., Plexousakis, D., Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 229–243. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30475-3_17
9. Levin, M.K., Cowell, L.G.: owlcpp: a C++ library for working with OWL ontologies. J. Biomed. Semant. **6**(1), 1 (2015)
10. Liebig, T., Luther, M., Noppens, O., Wessel, M.: Owllink. Semant. Web **2**(1), 23–32 (2011)
11. Matentzoglu, N., Leo, J., Hudhra, V., Sattler, U., Parsia, B.: A survey of current, stand-alone OWL reasoners. In: Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation, vol. 1387 (2015)
12. Mohamed, I., Patel, D.: Android vs iOS security: a comparative study. In: 2015 12th International Conference on Information Technology - New Generations (ITNG), pp. 725–730 (2015). doi:10.1109/ITNG.2015.123
13. Noppens, O., Luther, M., Liebig, T., Wagner, M., Paolucci, M.: Ontology-supported preference handling for mobile music selection. In: Proceedings of the Multidisciplinary Workshop on Advances in Preference Handling, Riva del Garda, Italy (2006)
14. Pfisterer, D., Römer, K., Bimschas, D., Kleine, O., Mietz, R., Truong, C., Hasemann, H., Kröller, A., Pagel, M., Hauswirth, M., et al.: SPITFIRE: toward a semantic web of things. IEEE Commun. Magaz. **49**(11), 40–48 (2011)
15. Ploennigs, J., Schumann, A., Lécué, F.: Adapting semantic sensor networks for smart building diagnosis. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8797, pp. 308–323. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11915-1_20
16. Schreiber, G., Raimond, Y.: RDF 1.1 Primer. W3C Working Group Note (2014). https://www.w3.org/TR/rdf11-primer/
17. Scioscia, F., Ruta, M.: Building a semantic web of things: issues and perspectives in information compression. In: Semantic Web Information Management (SWIM 2009), Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC 2009), pp. 589–594. IEEE Computer Society (2009)

18. Scioscia, F., Ruta, M., Loseto, G., Gramegna, F., Ieva, S., Pinto, A., Di Sciascio, E.: A mobile matchmaker for the ubiquitous semantic web. Int. J. Semant. Web Inf. Syst. (IJSWIS) **10**(4), 77–100 (2014)
19. Steller, L., Krishnaswamy, S.: Pervasive service discovery: mTableaux mobile reasoning. In: International Conference on Semantic Systems (I-Semantics), Graz, Austria (2008)
20. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). doi:10.1007/11814771_26
21. W3C OWL Working Group: OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation (2012). https://www.w3.org/TR/owl2-overview/
22. Weiß, F., Leimeister, J.M.: Why can't i use my iphone at work?: managing consumerization of IT at a multi-national organization. J. Inf. Technol. Teach. Cases **4**(1), 11–19 (2014). doi:10.1057/jittc.2013.3