A Modular Lightweight Implementation of the Smart-M3 Semantic Information Broker

Fabio Viola, Alfredo D'Elia
ARCES - Advanced Research Center on Electronic Systems
University of Bologna
Bologna, Italy
{fabio.viola2, alfredo.delia4}@unibo.it

Luca Roffia, Tullio Salmon Cinotti
DISI - Department of Computer Science and Engineering
University of Bologna
Bologna, Italy
{luca.roffia, tullio.salmoncinotti}@unibo.it

Abstract—**Interoperability among heterogeneous devices is one of the main topics investigated nowadays to realize the Ubiquitous Computing vision. Smart-M3 is a software architecture born to provide interoperability through the Semantic Web technologies and reactivity thanks to the publish-subscribe paradigm. In this paper we present a new implementation in Python of the central component of the Smart-M3 architecture: the Semantic Information Broker (SIB). The new component, named pySIB, has been specifically designed for embedded or resource constrained devices. pySIB represents a new open source lightweight and portable SIB implementation, but also introduces new features and interesting performances. JSON has been introduced as the default information encoding notation as it offers the flexibility of XML with minor bandwidth requirements. Memory allocation on disk and at runtime is in the order of Kilobytes i.e. minimal, if compared with the other reference implementations. Performance tests on existing (SP^{2}B) and ad-hoc benchmarks point out possible improvements but also encouraging data such as the best insertion time among the existing SIB implementations.**

I. Introduction

The vision of Tim Berners Lee about a re-implementation of the Web [12] brought to a plethora of new languages for univocally identifying resources (URI), describing them (RDF [24]), providing meaning to information (RDFS [24] and OWL [27]) and retrieving it (SPARQL [30]). Most of these technologies, born under the wing of the Semantic Web, are currently used in other IT areas such as context-aware computing and the Internet of Things.

The application to IoT scenarios of semantic technologies for information representation provides a common vocabulary to achieve interoperability among the interacting entities but implies growth of bandwidth usage and verbosity. A proper software infrastructure is then needed to face the large amount of information shared in these contexts, to handle a high number of heterogeneous devices and to meet the strong requirements in terms of reliability and reactivity.

Among the proposed solutions there is Smart-M3 [17], a simple abstract software architecture based on a centralized software module for information management: the SIB (Semantic Information Broker). Different implementations of the SIB have been designed in the recent years to solve the new issues arisen by the incoming scenarios: RedSIB [25] (that introduced SPARQL Updates) and the OSGI SIB [22] (that brought portability and extensibility) are nowadays the reference implementations of this software module while CuteSIB [14] is emerging as a novel SIB implementation founded on RedSIB that also aims at suporting low-capacity devices.

The Smart-M3 platform is based on the Semantic Web technologies since the information is stored by the broker as an RDF graph where additional semantics can be specified using an OWL ontology. [21]. The entire framework M3 in which pySIB is included provides high interoperability and extensibility and it is easy to develop and operate with it, as it can be evinced by answering the fifteen technical questions proposed by Balandin and Waris in [10] where they highlight the main points to consider when developing smart spaces.

In this paper is presented pySIB[5]: a novel open source information broker bringing three main contributions to its field of research. First pySIB has been designed to be lightweight and so ideal for running on devices with poor computational resources or restrictive energy consumption requirements. Second a new serialization format has been defined for all the exchanged messages to substitute XML and contain the verbosity that Semantic Web formalisms unavoidably bring with them. Third, the performance evaluation carried out shows that, despite being lightweight, the pySIB shows comparable or even better performances than the state-of-the-art implementations.

This paper is organized as follows: in section II an overview of related research projects is reported. In section III the reference platform Smart-M3 is presented, then the architecture of the specific pySIB implementation is described. Section IV reports about the JSON implementation of the SSAP protocol, supported by the Python implementation of the semantic information broker. In section V the developed platform is evaluated against specific tests and compared with the other main information brokers. Eventually, in section VI, conclusions are drawn.

II. Related work

Many solutions have been proposed in literature to face the known issues of IoT scenarios among which the high number of heterogeneous devices, the big amount of exchanged data, the security, dependability and interpretability of information. One of the most common approaches consists in adopting middleware-based architectures [28], where a software layer is delegated to solve the mentioned issues [18].

Albano et al. in [9] provide a classification between middlewares for data exchange. Among the different data management approaches illustrated, the publish-subscribe paradigm is the one that best fits not only for the smart grid scenarios on which the article is focused on, but also for the ubiquitous computing scenarios and IoT in general. Publish Subscribe Message-Oriented Middlewares (PSMOM) satisfy the requirements of such applications in terms of scalability, dinamicity, latency, and jitter.

One of the most common ways of granting interoperability in middleware based solutions is to found the information representation on standards coming from the Semantic Web. This approach is very diffused, as demonstrated by the number of existing past and ongoing research project using it, like the one described by Gyrard et al. in [16], Task Computing Environment (TCE) [23], COntext BRoker Architecture [13] and SPITFIRE [29], just to name a few.

This article focuses on a component of one of these middleware based solutions: Smart-M3 [17]. Smart-M3 is a semantic interoperability platform based on the publish-subscribe paradigm. It was conceived in 2008 during the ARTEMIS joint undertaking European Project SOFIA (*Smart Object For Intelligent Applications*) and adopted in various EU past and ongoing projects (e.g. CHIRON, IoE, RECOCAPE, IMPRESS, ARROWHEAD [2] and communities (i.e. FRUCT [4]). The Python implementation described in this paper differs from the previous implementations since it is more oriented on portability, extensibility and optimization of resource usage.

III. Software architecture

A. The reference platform

Smart-M3 is the reference platform and pySIB is a novel implementation of the core of the platform, the Semantic Information Broker. It enables devices to interact through an information centric approach. Clients share semantic information (i.e. information represented with RDF and OWL). Kiljander et al. in [19] highlight how the main advantage of using RDF and OWL is that they provide a common way to describe information in generic machine-interpretable form and thus both provide means for semantic level interoperability and support information reusability.

Smart-M3 relies on the Explicit Context Model [28] where a central node is responsible for the context management and is completely application-independent and decoupled from the entities devoted to context acquisition, context pre-processing and context reasoning. A Smart-M3 application consists of two kind of agents that, in the Smart-M3 nomenclature, take the name of SIB (introduced in section I), and KPs (or *Knowledge Processors*) [26]. The KPs are information producers and consumers, constituting in their whole a smart application. KPs are software agents sharing data through the SIB by means of SPARQL UPDATE [15] and QUERY [30] languages.

KPs are developed to fit the application needs, using one of the Knowledge Processor Interfaces (KPIs) available for several programming languages (Java, Python, Ruby, C, C#, Javascript, PHP). KPs can be classified among producers (which only perform update of the KB), consumers (which only retrieve data) or aggregators (entities playing both the roles simultaneously).

The SIB-KPs interaction is mediated by the SSAP (*Smart Space Access Protocol*) that acts as the main integration point of the platform [17] and relies on XML (eXtensible Markup Language) as the default encoding language.

Fig. 1 summarizes the architecture of the Smart-M3 interoperability platform.

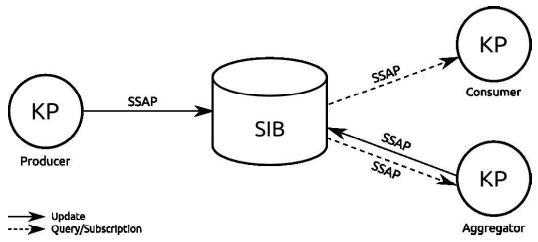

Fig. 1. Smart-M3 architecture

B. pySIB software architecture

The pySIB is a multi-threaded server whose software architecture is represented in Fig. 2.

pySIB is an information broker that holds information in the form of an RDF graph. Applications can store data into the RDF graph I in accord to a locally agreed ontology O. A smart space $S = (I, O)$ can then be easily set up as in a classic Smart-M3 application and KPs can exploit the KPIs or a higher-level ontology library for the interaction [21]. Future implementation of pySIB will provide support for handling several RDF graphs at a time.

The typical message flow starts with a request sent from a KP to the SIB that is received by the network module and forwarded to the proper parser. A synthetic representation of the message in the form of a python dictionary is then generated for the processor module with access to the information store. The processor module generates a dictionary, representing an abstract description of the response to be forwarded to the KP. The Message Builder processes the abstract response description and serializes it using the encoding which best fits the KP needs.

Message parser and *Message builder* are respectively the decoder and the encoder for the chosen SIB-KP communication protocol. The Message parser decodes the received message in order to build a python dictionary to be processed (independently from the communication protocol adopted). The builder is instead the encoder that takes the output of the elaboration of the SIB in order to build a message to forward to the KPs. The choice of the builder and of the parser module to be used is demanded to the system administrator who decides which protocol should be used by the SIB. The modular structure of the SIB, in fact, allows to easily replace the Parser and Builder modules providing support for new formats and encoding. pySIB is provided with a parser and a builder for the JSSAP, detailed in section IV. Thanks to the configurability of pySIB, we can affirm that this novel broker is multi-protocol. Support for the original SSAP can be developed as a pluggable module and then enabled through a simple configuration file.

Other two features potentially supported but not yet implemented are the support for quads and a security mechanism based on those. It is important to notice that, despite the important and generally invasive nature of the security mechanisms, thanks to the modular organization of pySIB it will be simple to add them in the information processing flow, as shown with the dashed lines in Fig. 2.

The security module will be designed following the approach presented in [32] that provides confidentiality and authenticity through its central component named RIBS (RDF Information Base Solution), a server used by KPs to store, retrieve, manipulate and subscribe information. This components determines security levels for each communication session leveraging on the data collected about methods and algorithms used by the clients.

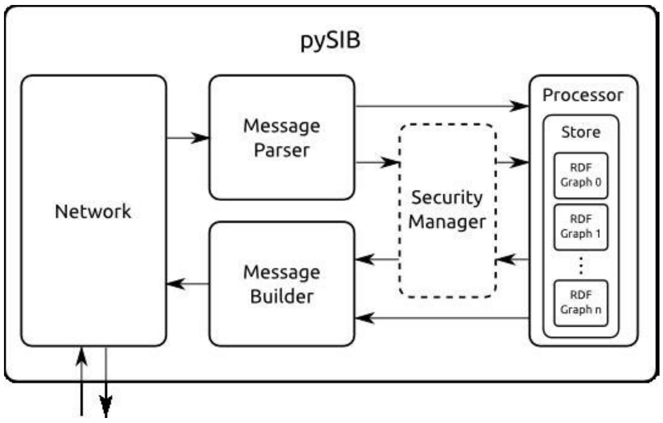

Fig. 2. Software architecture

IV. The JSSAP

The default protocol used in the communication between KPs and pySIB is the JSSAP. This protocol defines the standard format for every request and confirm message using JSON as the data serialization format for all the primitives in the standard SSAP specification. Remapping the SSAP protocol to a JSON encoded version allowed to reduce the length of the exchanged messages from 10% (for long messages) up to 40% (for short messages).

The main fields inherited from the standard SSAP protocols are:

- `node_id`: identifies the KP that performs a request (and receives the reply);
- `space_id`: an identifier for the smart space to which the KP belongs to;
- `transaction_id`: an univoque identifier for the request (and its reply);
- `transaction_type`: the transaction type identifies the kind of the request performed by the KP. The possible requests are detailed in subsection IV-A;
- `message_type`: the message type is used to mark a message as a request, a reply or an indication. Since the kind of message can be easily evinced by the software agent, this field is now not mandatory.

Each transaction type is also characterized by specific fields detailing the given request or the associated reply.

An example of a simple request performed by the KP to join a specific smart space is here reported:

```
{
    "transaction_id":"0",
    "transaction_type":"JOIN",
    "message_type":"REQUEST",
    "node_id":"ab12cd34",
    "space_id":"ARCES"
}
```

A. Primitives supported by pySIB

As documented in [17] and [20], the SSAP includes eight primitives, all supported by the SIB implementation presented in this paper.

Through the `join` and `leave` primitives it is possible for a KP to respectively access or exit from an M3 smart space.

The `insert` and `remove` operations allow to modify the information graph producing or deleting information. The `update` operation performs both removals and insertions. pySIB supports RDF-M3 as a serialization method for RDF triples. In addition, the SPARQL UPDATE language [15] is supported, so that more complex update operations can be performed.

The `query` primitive surfs the semantic graph allowing to explore it in detail. Even for the query two encoding are supported: RDF-M3 and SPARQL QUERY language [30]. The same choice is available for the `subscribe` primitive that allows a KP to declare its interest on a sub-graph and to be notified about changes on it. The `unsubscribe` primitive is used to cancel an existing subscription.

B. The JSSAP encoder and decoder

In order to provide an efficient support to JSSAP, the most commonly used Python modules for dealing with JSON have been compared. Four Python modules have been taken into consideration and tested using three ad-hoc software agents.

The four modules analyzed are:

- `json` [1]: the standard JSON encoder/decoder provided with the Python interpreter;
- `cjson` [6]: a C implementation of a JSON encoder/decoder for the Python programming language;
- `ujson` [8]: another C implementation compatible with both Python 3 and Python 2.5 or higher;
- `simplejson` [7]: a Python module with optional C extensions.

The machine where such tests have been performed is a Lenovo Thinkpad X220 provided with an Intel(R) Core(TM) i5-2520M CPU 2.50GHz 4-core processor and 4 GB of RAM running Linux Mint 17 Qiana. Every point on the charts is the mean value of a set of twenty samples where the variance can be neglected.

1) Encoding: The first test performed has the objective to compare the JSON libraries in the encoding phase. The comparison metric is the time needed by the Semantic Information Broker to build a reply to a SPARQL query in relation to the number of results. The query just takes all the pySIB content:

```
SELECT ?s ?p ?o
WHERE { ?s ?p ?o }
```

This kind of test is pretty much effective and relevant to characterize the encoding time, since the reply to a SPARQL query can be very large, depending on the number of bindings to be returned.

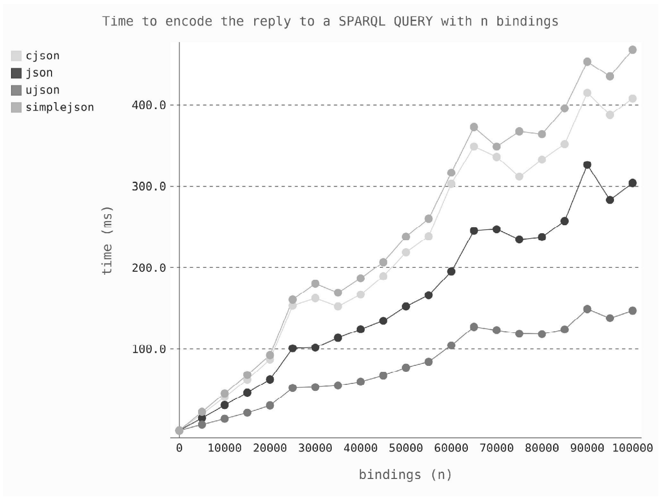

Fig. 3. Time to encode the result of a SPARQL QUERY request with a variable number of bindings

Fig. 3 summarizes the results on the chosen libraries. `Simplejson` module is the slowest one with every number of bindings. The time required by `ujson` to encode the results of a SPARQL query is always around the 30% of the one required by standard `json` resulting, in the worst case, in a difference of more than 340 ms.

2) Decoding: To characterize the available libraries in decoding phase, the time needed to decode a JSSAP request coming from a KP has been reported. As the decoding time depends from the JSON document tree structure, two main relevant scenarios have been taken into consideration: RDF-M3 insert, and SPARQL UPDATE. Even if the number of triples to insert is the same, with an RDF-M3 insert the resulting JSON message is composed by a multitude of small fields, while with SPARQL UPDATE the triples to be inserted are all in the same large field.

A block of triples to be inserted with RDF-M3 is represented as:

```
{
  "triples":{
    "triple":{
      "subject":"http://ns#subject0",
      "predicate":"http://ns#predicate0",
      "object":"http://ns#object0"
    },
    "triple":{
      "subject":"http://ns#subject1",
      "predicate":"http://ns#predicate1",
      "object":"http://ns#object1"
    },
    ...
   "triple":{
      "subject":"http://ns#subjectN",
      "predicate":"http://ns#predicateN",
      "object":"http://ns#objectN"
    },
  }
}
```

while the SPARQL UPDATE operation to insert $N + 1$ triples is:

```
PREFIX ns:<http://ns#>
INSERT {
  ns:subject0 ns:predicate0 ns:object0 .
  ns:subject1 ns:predicate1 ns:object1 .
  ...
  ns:subjectN ns:predicateN ns:objectN
}
```

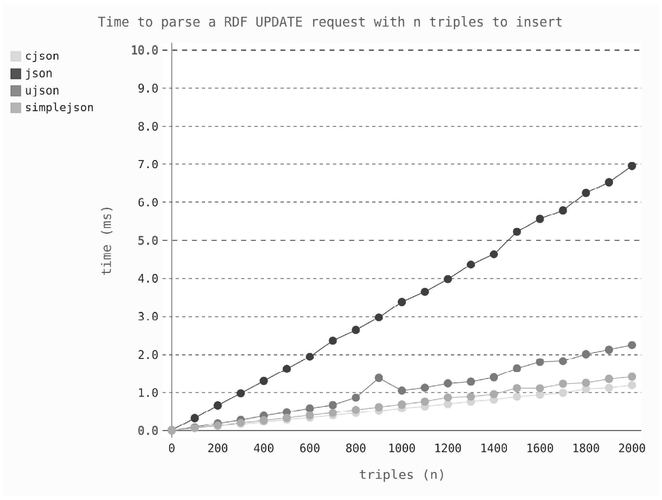

Fig. 4. Time to parse an RDF UPDATE request with a variable number of triples to insert

Fig. 4 and Fig. 5 show the results of the two tests. As expected, whatever library is used, the time needed to decode an RDF-M3 update is alwasys higher than the time elapsed to parse a SPARQL update with the same number of triples since there is a very high number of fields to be analyzed. Then, for the sake of clarity, the two bar diagrams utilize different scales in order to better appreciate the trend of each line. Both the charts, however, show a linear behaviour that highlights a

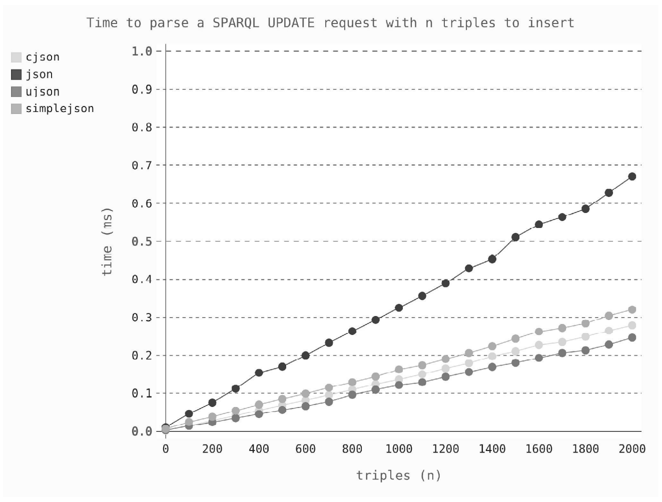

Fig. 5. Time to parse a SPARQL UPDATE request with a variable number of triples to insert

direct dependency of the elapsed time on the number of triples. The slope of each line strictly depends on the efficiency of the related library.

It can be observed how the default `json` module underpeforms with respect to the alternative implementations. Differently from what happens with the encoding test, here the differences between the performance of `cjson`, `ujson` and `simplejson` are negligible, since in the worst case the difference between the fastest and the slowest encoder is respectively less than 0.1 ms for a SPARQL update and less than 1.2 ms for an RDF update.

Thus, the library chosen to provide support for the JSSAP with pySIB is `ujson`, mainly due to the results of the test of the encoding.

V. Evaluation

The SIB presented in this paper has been evaluated on a personal computer Lenovo Thinkpad X220 with a quad-core Intel i5-2520M (2.50GHz) and 4 GB of RAM running Linux Mint 17 Qiana. Further tests will be performed in the following months on single-board computers with real-life use cases. Each test has been performed twenty times obtaining in every case neglibile variances.

Before starting the description of the evaluation, it is worth paying attention to the time required to perform an operation that can be formalized as:

$$\begin{aligned} t_{OP} &= t_{ENC\_REQ} + t_{SEND\_REQ} + t_{DEC\_REQ} \\ &+ t_{ELAB} + t_{ENC\_REP} + t_{SEND\_REP} + t_{DEC\_REP} \end{aligned} \tag{1}$$

where the first is the time elapsed by the KP to encode the request, the second is the time needed to transfer the TCP packet to the SIB; t_{ELAB} is the time needed by the SIB to elaborate the request (decoded in t_{DEC_REQ}) and perform the desired operation. Then a reply must be sent, so it is encoded (t_{ENC_REP}), sent (t_{SEND_REP}) and decoded by the KP (t_{DEC_REP}).

As mentioned in section IV-B, the encoding and decoding phases have been optimized with the choice of the best performing Python module for JSON in order to reduce the impact of t_{ENC_REQ}, t_{DEC_REQ}, t_{ENC_REP} and t_{DEC_REP} on the total elapsed time.

In each of the following benchmarks, the Python implementation of the Semantic Information Broker has been compared with:

- RedSIB, the C implementation, running in non-persistent way with support of hash tables;
- OSGi SIB, the SIB written exploiting the OSGi Java framework. Also this one has been used in non-persistent mode. Since the OSGi SIB also provides support for the JSSAP, this bundle has been preferred over the standard SSAP.

A. *Time to insert* n *triples*

The time required to insert a block of n triples, varying n has been measured on the above mentioned SIBs. Results are shown in Fig. 6. pySIB results to be the fastest among the analyzed SIBs in the majority of the tests. The simple regression analysis allows to predict future values with the equation $y = 7.79 + 0.06 \cdot x$ where x is the number of triples to insert and y the dependent variable representing the time required to perform the insertion.

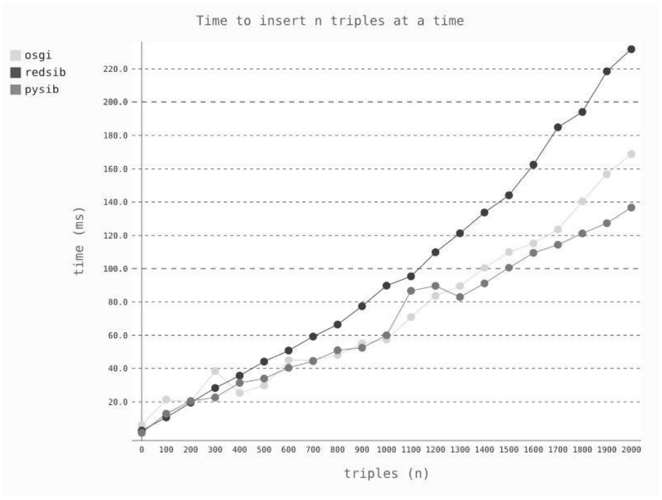

Fig. 6. Time to insert *n* triples

B. *Time to retrive* n *triples*

Fig. 7 compares the time took by pySIB and its alternatives in order to retrieve a variable number of triples with an RDF query (the number of triples retrieved corresponds to the full content of the SIB). The graph highlights a similar trend for pySIB and for RedSIB that results the fastest one for a large number of results. The effort spent in optimizing the encoding and the decoding phases brought to encouraging results, since the performance are comparable to the most known and diffused implementations of the SIB. The trend of the graph of pySIB is linear: the amount of time spent depends on the amount of triples and then the amount of data to be transferred over the TCP, but also on the efficiency of the parsing module. The linear regression analysis on the pySIB allow to build the following relation between the number of triples inserted (x) and the time required to perform the action (y):

$$y = -11.35 + 0.14 \cdot x$$

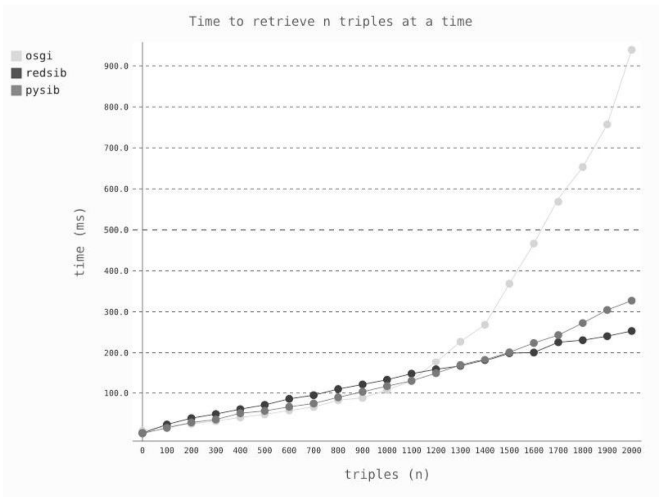

Fig. 7. Time to retrieve n triples

C. SP^{2}B

The SP^{2}B [31] is a benchmark designed to test the performance of SPARQL endpoints. This suite provides a data generator that creates an N3 file [11] with up to 25M triples according to the DBLP [3] ontology. The benchmark is composed by seventeen SPARQL queries properly designed to test the behaviour of the semantic data stores.

Given a knowledge base composed by 10k triples, and subsequently 50k triples, the response time to the seventeen queries has been measured. The time elapsed by pySIB is compared with the time needed by RedSIB and the OSGi SIB to perform the same tasks. Table I and table II report in the first column the query identifier followed by the number of expected results (if the query is a SELECT) or the expected Boolean (if the query is an ASK). For every triple *SIB/size of the knowledge base/query* is also reported the obtained result if different from the expected one. Time values are expressed in seconds.

The results show that pySIB behaves correctly with 12 of the 17 queries (for the small dataset) and with 11 queries on the large dataset. In these cases the response to the queries is correct and obtained within the time limit (set to five minutes). These results suggest the directions for a further investigation in order to improve the response to complex SPARQL queries obtaining a correct reply for every of each (as with the OSGi SIB) and in a shorter time.

TABLE I. SP^{2}B benchmark results with 10k triples

Query (Results)	pySIB	RedSIB	OSGi SIB
Q1 (1)	0.01	0.49 (0)	0.01
Q2 (147)	0.28	2.95	0.18
Q3a (846)	1.08	17.27	0.13
Q3b (9)	1.00	17.95	0.01
Q3c (0)	1.00	17.59	0.01
Q4 (23226)	timeout	timeout	75.44
Q5a (155)	50.90 (1)	1.16 (1)	0.61
Q5b (155)	24.25 (1)	timeout	0.30
Q6 (229)	21.13 (11)	0.24 (13)	0.73
Q7 (0)	0.27	21.86	0.19
Q8 (184)	0.05 (0)	17.92 (0)	0.05
Q9 (4)	1.00	35.00	0.02
Q10 (166)	0.28	0.07	0.08
Q11 (10)	< 0.01	0.03	0.04
Q12a (True)	< 0.01	< 0.01	< 0.01
Q12b (True)	< 0.01	17.90	< 0.01
Q12c (True)	< 0.01	0.07	< 0.01

TABLE II. SP^{2}B benchmark results with 50k triples

Query (Results)	pySIB	RedSIB	OSGi SIB
Q1 (1)	0.15	10.78 (0)	0.03
Q2 (965)	< 0.01	timeout	3.45
Q3a (3647)	4.86	timeout	1.23
Q3b (25)	4.39	649.38	0.01
Q3c (0)	4.13	timeout	0.01
Q4 (104746)	timeout	timeout	timeout
Q5a (1085)	timeout	timeout	21.78
Q5b (1085)	timeout	timeout	10.47
Q6 (1769)	345.53 (10)	4.28 (17)	21.78
Q7 (2)	1.19 (0)	timeout	5.63
Q8 (264)	0.05 (0)	timeout	0.30
Q9 (4)	1.37	timeout	0.07
Q10 (307)	0.06	0.18	0.06
Q11 (10)	0.93	0.12	0.02
Q12a (True)	1.94	0.11	< 0.01
Q12b (True)	0.13	timeout	< 0.01
Q12c (True)	0.11	< 0.01	< 0.01

D. Disk usage

In the current subsection, the disk space required by pySIB is compared with the one required by RedSIB and by the OSGi SIB. The analysis has been carried out on the operating system GNU/Linux, distribution Mint 17 Qiana for the architecture amd64. The disk space is reported in kilobytes and has been calculated using the binary packages in `.deb` format for the target Linux distribution. For each SIB the required disk space has been split in:

TABLE III. Space occupation of the SIBs (in KiloBytes)

Disk usage	pySIB	RedSIB	OSGi SIB
SIB package	25	88	13824
Dependencies	640	10832	21504
Interpreter/VM	197	0	171
Interpreter Deps	12518	0	97224
Total	**13380**	**10920**	**137723**

- disk space required by the core of the Semantic Information Broker;
- disk space required by the dependencies of the SIB;
- amount of space required by the interpreter or virtual machine (if any);
- amount of disk space occupied by the dependencies of the interpreter / virtual machine.

The results of this comparison, visible in Table III, show that pySIB requires a quantity of disk space comparable with the one of the RedSIB and is, by far, more suitable than the OSGi SIB to run on constrained devices.

E. Memory Usage

The implementation presented in this paper has been compared with the other two main SIBs of the Smart-M3 interoperability platform, even in terms of memory requirements. The SP^{2}B has been used to load 50k triples into each broker. The Resident Set Size (RSS) for every SIB has been calculated after the start-up and then after each insertion of 5k triples in order to trace the memory usage of the analyzed brokers.

Fig. 8 shows that pySIB requires less than half the memory used by the OSGi implementation of the SIB. On the other hand, pySIB underpeforms if compared with RedSIB requiring approximatively 1.7 times its amount of RSS.

The evident difference between the memory usage and disk space occupation of pySIB and OSGi SIB is justified by the different targets of the two implementations: the first is oriented to support small devices with low computational power (e.g. System on Chips, also known as SoC, like Raspberry or Hummingboard), while the second is designed to run on gateways.

Despite the higher resource usage of pySIB, if compared with RedSIB, our broker can be still considered a valid solution for resource-constrained devices (i.e. like System on Chip microcomputers) due to the internal modular software architecture. Advantages, however, will be appreciable only when new features will be necessary because the development of such functionalities will be easier with respect to the monolithic software architecture.

VI. Conclusion and future work

In this article a novel modular implementation of the SIB for the Smart-M3 framework has been presented. In cases where a lightweight solution, characterized by good performance is needed (e.g. on SoC microcomputers) pySIB reveals to be a valid choice. In fact, section V highlights a good

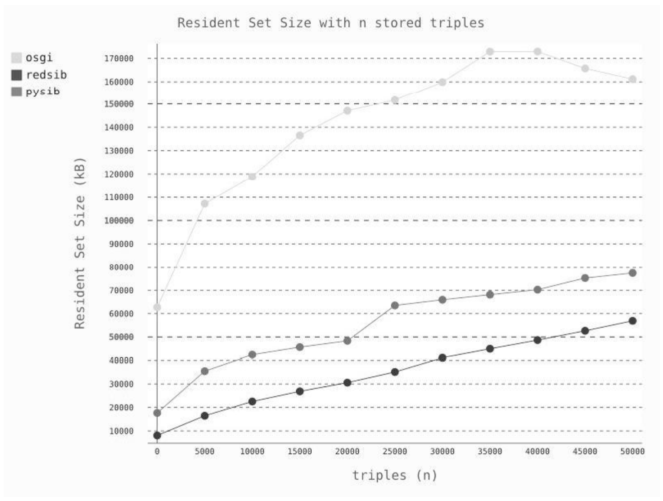

Fig. 8. Memory usage varying the size of the knowledge base

behaviour of pySIB, but at the same time reveals the direction for future improvements (i.e. in terms of memory and disk space requirements, as detailed in subsections V-E and V-D).

Unexpectedly pySIB performs updates of the knowledge base faster than the other considered implementations of the SIB. Even retrieving data with RDF queries reveals high performance of this new software component. However, the SP^{2}B benchmark revealed the existence of particular SPARQL queries that hinder the SIB from providing a correct reply. Further investigations will be performed in order to enhance the SIB.

pySIB currently does not provide support for data persistency. This is planned to be introduced in the next release and with minor effort thanks to the modular structure of the broker. The next release, as mentioned in section III-B, will also include a security module through which it will be possible to protect data with SPARQL rules.

References

[1] 18.2. json json encoder and decoder. https://docs.python.org/2/library/json.html.

[2] Arrowhead ahead of the future. http://www.arrowhead.eu/.

[3] dblp: computer science bibliography. http://dblp.uni-trier.de/.

[4] Fruct - open innovations framework program fruct. http://www.fruct.org/.

[5] pysib github repository. https://github.com/desmovalvo/pysib.

[6] python-cjson 1.1.0. https://pypi.python.org/pypi/python-cjson.

[7] simplejson 3.8.1. https://pypi.python.org/pypi/simplejson/.

[8] ujson 1.35. https://pypi.python.org/pypi/ujson.

[9] M. Albano, L. L. Ferreira, L. M. Pinho, and A. R. Alkhawaja. Message-oriented middleware for smart grids. *Computer Standards & Interfaces*, 38:133 – 143, 2015.

[10] S. Balandin and H. Waris. Key properties in the development of smart spaces. In *Universal Access in Human-Computer Interaction. Intelligent and Ubiquitous Interaction Environments*, pages 3–12. Springer, 2009.

[11] T. Berners-Lee. Notation 3 (n3): An readable language for data on the web, 2005.

[12] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[13] H. Chen, T. Finin, and A. Joshi. Semantic web in the context broker architecture. Technical report, DTIC Document, 2005.

[14] I. V. Galov, A. A. Lomov, and D. G. Korzun. Design of semantic information broker for localized computing environments in the internet of things. pages 36–43. IEEE, Apr. 2015.

[15] P. Gearon, A. Passant, and A. Polleres. Sparql 1.1 update. *Working draft WD-sparql11-update-20110512, W3C (May 2011)*, 2012.

[16] A. Gyrard, S. Datta, C. Bonnet, and K. Boudaoud. A semantic engine for internet of things: Cloud, mobile devices and gateways. pages 336–341, July 2015.

[17] J. Honkola, H. Laine, R. Brown, and O. Tyrkko. Smart-m3 information sharing platform. In *The IEEE symposium on Computers and Communications*, pages 1041–1046. IEEE, 2010.

[18] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. In *2007 Future of Software Engineering*, pages 244–258. IEEE Computer Society, 2007.

[19] J. Kiljander, A. D'elia, F. Morandi, P. Hyttinen, J. Takalo-Mattila, A. Ylisaukko-Oja, J.-P. Soininen, and T. S. Cinotti. Semantic interoperability architecture for pervasive computing and internet of things. 2:856–873.

[20] D. G. Korzun. Service formalism and architectural abstractions for smart space applications. In *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR '14, pages 19:1–19:7, New York, NY, USA, 2014. ACM.

[21] D. G. Korzun, S. I. Balandin, and A. V. Gurtov. Deployment of smart spaces in internet of things: Overview of the design challenges. In *Internet of Things, Smart Spaces, and Next Generation Networking*, pages 48–59. Springer, 2013.

[22] D. Manzaroli, L. Roffia, T. S. Cinotti, P. Azzoni, E. Ovaska, V. Nannini, and S. Mattarozzi. Smart-m3 and osgi: The interoperability platform. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 1053–1058. IEEE, 2010.

[23] R. Masuoka, B. Parsia, and Y. Labrou. Task computing–the semantic web meets pervasive computing. In *The Semantic Web-ISWC 2003*, pages 866–881. Springer, 2003.

[24] B. McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 51–65. Springer Berlin Heidelberg, 2004.

[25] F. Morandi, L. Roffia, A. DElia, F. Vergari, and T. S. Cinotti. Redsib: a smart-m3 semantic information broker implementation. In *Proc. 12th Conf. of Open Innovations Association FRUCT and Seminar on e-Tourism*, pages 86–98. SUAI, 2012.

[26] M. Palviainen, J. Kuusijrvi, and E. Ovaska. A semi-automatic end-user programming approach for smart space application development. *Pervasive and Mobile Computing*, 12:17–36, June 2014.

[27] P. F. Patel-Schneider, P. Hayes, I. Horrocks, et al. Owl web ontology language semantics and abstract syntax. *W3C recommendation*, 10, 2004.

[28] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454, 2014.

[29] D. Pfisterer, K. Römer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kroller, M. Pagel, M. Hauswirth, et al. Spitfire: toward a semantic web of things. *Communications Magazine, IEEE*, 49(11):40–48, 2011.

[30] E. PrudHommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[31] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp^ 2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[32] J. Suomalainen, P. Hyttinen, and P. Tarvainen. Secure information sharing between heterogeneous embedded devices. page 205. ACM Press.