The Smart-M3 Semantic Information Broker (SIB) Plug-in Extension: Implementation and Evaluation Experiences

Paolo Bellavista, Veronica Conti, Carlo Giannelli
DISI – University of Bologna, Italy
{paolo.bellavista, veronica.conti, carlo.giannelli}@unibo.it

Jukka Honkola
Innorange Oy, Helsinki, Finland
jukka.honkola@gmail.com

Abstract— **Smart spaces are gaining relevance as promising deployment environments for novel classes of applications stemming from the dynamic discovery and interaction between smart objects and the resources available in their physical localities, e.g., seamlessly exploiting smartphones to control embedded home equipment. The Semantic Information Broker (SIB) of the Smart-M3 platform can well support interoperability of statically unknown devices and service/application components in a smart space based on lightweight semantically-tagged data sharing. The paper focuses on the experience made and the lessons learned from the work of design, implementation, deployment, and experimental validation/evaluation of the SIB "Plug-in" extension. This extension allows SIB administrators to personalize and extend SIB installations simply and with a well-defined methodology, by cleanly adding plug-in extensions that can support domain- and deployment-specific facilities, thus opening new market opportunities for Smart-M3 exploitation. The reported results show the feasibility and effectiveness of the proposed approach: in particular, we report the experience made with a notable example of plug-in component that can measure the runtime-offered SIB quality of service, expressed as a set of concise performance indicators.**

Keywords- Smart spaces, ubiquitous computing, lightweight semantic technologies, interoperability, dynamic extensibility

I. INTRODUCTION

The enormous market of smartphones with different forms of wireless connectivity, coupled with the increasing computing capabilities of embedded devices deployed in a pervasive way, is pushing for smart space solutions, e.g. to support the seamless discovery and invocation of locally available services seamlessly. For instance, let us consider a home environment where an Android smartphone can dynamically join the user's home smart space and allow to remotely monitor the oven temperature and to dim the light intensity while on the sofa. Or, as another simple example, imagine tourists who can dynamically gather information related to the city they are visiting by interacting with a smart traffic light, where they can possibly post their own geographically-tagged comments that will be made available to future visitors when in traffic light proximity.

However, notwithstanding that the smart space concept has been around for several years, the actual industrial exploitation of smart spaces has been limited by several factors, first of all, the difficulties of interoperating among heterogeneous devices in an open and statically unpredictable deployment environment. It is recognized that smart applications should be able to dynamically join smart spaces without any pre-configuration and invoke/provide services without any a-priori knowledge of the hosting hardware/software infrastructure. To this purpose, the Smart-M3 middleware proposes a relevant approach supporting smart space resource interworking at the data level (interoperability at the information level), i.e., by enabling information sharing among heterogeneous devices through a standardized access protocol and lightweight semantic techniques for data self-description. In particular, the Semantic Information Broker (SIB) component maintains shared data in a smart space by exploiting Resource Description Framework (RDF) triples to describe them according to a common ontology [1]. Knowledge Processors (KPs), possibly running on different devices participating to the smart space, interact each other by publishing and reading data to/from the SIB via the Smart Spaces Access Protocol (SSAP). The idea is that developers can easily and rapidly provide new smart applications by creating and deploying KPs that access SIB shared data via SSAP. Prior work has already demonstrated that Smart-M3 can be effectively adopted in several heterogeneous scenarios, ranging from remote monitoring of sensor data provided by constrained devices [2] to multimedia provisioning [3].

Within the framework of the EU SOFIA project [4] and while working in different vertical domains, it was recognized the opportunity to define and run special management-oriented third-party components internally to the SIB, in order to smoothly and effectively extend the SIB functionality when needed, e.g., in given deployment scenarios or for specific and application-oriented management purposes. One of the goals is to easily obtain differentiated and personalized versions of the same SIB, with extended built-in functionality, in a clean and well-disciplined way. Management-oriented third-party components are expected to be designed and implemented only by SIB developers and administrators, who are responsible for SIB personalization, deployment, and management. Their typical use could include discarding "old" triples in the SIB and saving them on persistent storage to reduce current memory occupation, reconciling possibly conflicting SIB data (also by removing some of them), and aggregating SIB data in more concise indicators. This interface, called plug-in extension, has been realized at the SIB adaptation layer and offered for the inclusion of internal third-party management components implemented in the C

language. The plug-in extension bypasses the standard SSAP SIB access and runs locally to the SIB, thus enabling the Smart-M3 platform to effectively host privileged (also in terms of achievable performance) and easily implementable internal third-party components for runtime management.

The paper reports about the experience made and the lessons learned from the work of design, implementation, deployment, and experimental validation/evaluation of the SIB plug-in extension. This extension, originally presented for the first time in this paper, allows SIB developers and administrators to personalize and extend SIB installations in a simple way, by dynamically adding plug-in components that can realize domain- and deployment-specific facilities. In particular, the paper presents primary design/ implementation insights about our recent implementation of the plug-in extension for the Smart-M3 SIB version and quantitative experimental results about its performance evaluation. The presented results show the feasibility and effectiveness of the approach. In addition, several plug-in components have already been realized on top of our plug-in interface, also to validate the solution in practical case studies. Among them, it is worth mentioning a plug-in component for the metering of the SIB-offered quality of service in terms of a set of concise performance indicators. The results of the periodic operation of this plug-in component are stored in the SIB itself; they contribute to the performance profiling of a given SIB installation, by offering performance-related data that can be usefully exploited by "regular" application-level KPs to dynamically select their most proper SIB, among the available ones, during the discovery phase and based on the performance results it was recently able to achieve.

Section 2 provides some details of the Linux-based Smart-M3 SIB implementation, which we extended with the plug-in interface, whose design and implementation is presented in Section 3. Section 4 focuses on how we have implemented the SIB metering plug-in component based on both "regular" KPs and the introduced plug-in extension. Finally, related work and conclusive remarks end the paper.

II. SMART-M3 SIB IMPLEMENTATION: STRENGTHS AND LIMITATIONS

Smart-M3 is the Linux-based SIB reference implementation proposed and developed by Nokia. It supports information interoperability via decoupled interaction and standard information representation. On the one hand, KPs can publish and gather information that is shared on Smart-M3, without any direct mutual interaction; on the other hand, information is stored based on common ontology models and common data formats. In this way, Smart-M3 gets the notable advantage of being device-, domain-, and vendor-independent: users are free to use the preferred device to exploit for accessing their smart space regardless of the manufacturer (multi-vendor); applications can seamlessly perform operations that involve a set of devices (multi-device) available in the smart space; application developers and their companies can focus on consumers' interests by adopting the same concept and technology in different application domains (multi-domain).

While Smart-M3 actually provides interoperability among heterogeneous devices and applications in a domain-independent fashion, its current implementation still exhibits some weaknesses, thus contributing to prevent Smart-M3 from being widely adopted in the mass market of nowadays smart space users. For instance, Smart-M3 proponents are extending the current Smart-M3 implementation to support access control and security management (secure SSAP), autonomous discovery and composition to create complex services based on simpler ones automatically, and SIB federation to easily access multiple SIBs as if they were a unique smart space entity. Furthermore, there are several additional features that Smart-M3 could provide to increase its value and efficiency, e.g., garbage collection to delete obsolete data from the RDF store and profiling features to publish SIB capabilities in order to facilitate SIB selection when multiple options are available in the smart space (SSAP version, memory availability, CPU load, and so on).

In its current implementation SIB capabilities are defined at compile-time: it is not possible to dynamically add/remove or enable/disable features in a selective manner, e.g., in relation to specific requirements depending on the deployment environment and the targeted applications. At the same time, there is the need of keeping Smart-M3 very lightweight, in order to make it suitable for a wide set of devices with heterogeneous capabilities, ranging from desktop PCs to low-end mobile-phones. For this reason, SIB improvements up to now have been limited to few paramount features every SIB should provide [5], e.g., secure SSAP and service composition, that is, mandatory features for the new release of the technology. The only possible method to dynamically add features is deploying KPs; however, they can interact with the RDF store only via SSAP, with the non-negligible negative effects of imposing communication overhead and limiting RDF management capabilities to what statically included in the SSAP API.

III. PLUG-IN EXTENSION FOR DYNAMIC SIB CUSTOMIZATION

In this paper we propose to modify the SIB architecture to allow the execution of new components inside the SIB, with the objective of enabling the well-disciplined customization of Smart-M3 with additional features that overcome KP limitations while maintaining its lightweight architecture. In this way it will be possible to customize the SIB by dynamically adding or removing features only if and when required, also at runtime, as detailed in the following.

In particular, the idea is that smart space developers and SIB administrators can register new software components (e.g., third-party services developed and compiled independently from Smart-M3) to run inside the SIB and access the RDF store directly (see the Adaptation layer in Fig. 1 [1]). Management operations at the RDF store level may improve SIB efficiency, e.g., a software component that operates as garbage collector to remove obsolete and useless data, or that performs complex reasoning to aggregate/infer additional data, or that translates data in response to modifications in the associated description ontology, or that performs context pre-processing and information filtering. In

this way, it is possible to achieve the notable benefit of modifying the SIB behavior requiring neither to re-compile nor to restart it; each SIB can be tailored de/activating and adding/deleting features in relation to specific objectives.

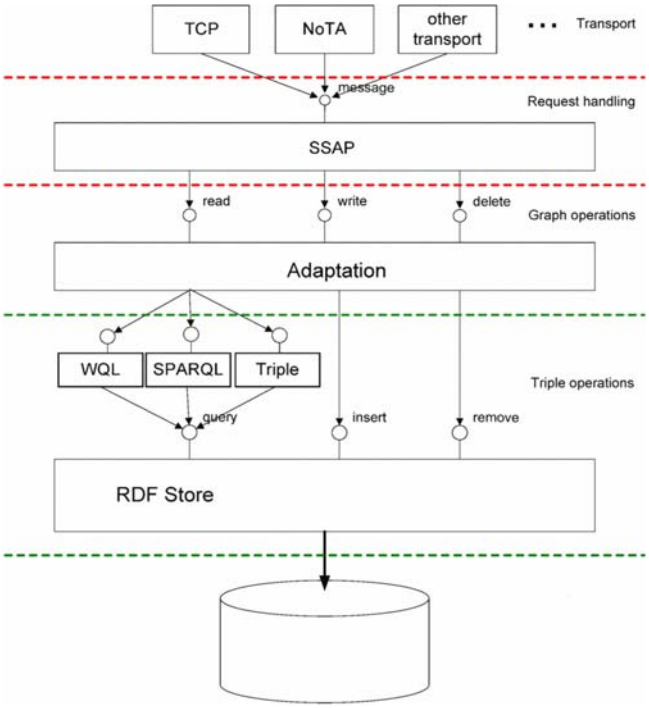

Figure 1. The SIB architecture in the Smart-M3 reference implementation.

A. *Plug-in Template*

Mainly for performance and efficiency motivations, we have designed SIB extensibility in order to allow plug-in extensions to directly interact with the RDF store without any specific restriction. The only requirement is that plug-in extensions adhere to a well-defined but general interface suitable for the development of a wide set of features. Moreover, since plug-in extensions may have exclusive and privileged access to the RDF store (see the following), the plug-in interface should be used by skilled and trusted developers, e.g., SIB administrators with deep knowledge of Smart-M3 details. In particular, we have designed the plug-in interface so that extensions must offer the following features:

1. **evaluateState**, which exploits any available information (internal to the SIB or more usually saved in the RDF store) to dynamically determine whether there is the need to activate the plug-in extension. For instance, it is possible to specify to activate a garbage collector extension component only if a given amount of RDF triples have been added/modified/deleted or if a certain time period has passed since the last garbage collection operation;
2. **run**, which includes the operational behavior of the plug-in extension. Tasks should be performed in multiple cycles, to provide the capability of periodically checking whether the above activation condition is still verified, by possibly stopping the execution of the extension component (see also the feature below);
3. **stop**, which forces the termination of the extension component. That may be useful to guarantee fair use of local resources, e.g., interrupting a plug-in extension after a given period if it is overloading the local node.

Note that plug-in extension developers are in charge of correctly implementing the above functionality based on a well-known template, described in the following, by adhering to a specific programming discipline and registering extension components to the SIB. Let us point out also that the proposed extensibility model is independent from the specific implementation of the SIB and can be applied regardless to the underlying operating system and exploited programming language. However, to provide a proof of concept implementation, we have specifically considered the Smart-M3 SIB implementation, written in C and running on Linux, which imposed us some constraints and influenced how we have decided to effectively realize the plug-in interface previously described. In particular, our plug-in extensions are implemented as Linux dynamic linked libraries, i.e., Shared Objects: in this way, plug-in extensions can be easily loaded/unloaded at runtime without stopping the SIB execution; moreover, it is possible to clearly separate the code of the SIB and of its extension components (the SIB footprint per se does not change, while plug-in extensions can be unregistered and deleted in order to reduce overhead and occupied disk space, which is sometimes required in low-end mobile devices with limited hardware resources).

In the following, to show a practical example of usage, we report a simple case of implementation of **evaluateState**, **run** and **stop** features as C functions:

```
enum boolean FALSE=0, TRUE;
…
boolean stop = FALSE;

void stop(){
  // developers should implement this method
  stop = TRUE;
}

boolean evaluateState(){
  boolean active;
  active = TRUE;  // in case it is required...
  active = FALSE; // in case it is NOT required...
// to activate this component in the next iteration
  return active;
}

void run(){
  while( (stop!=TRUE) && (have more work) ){
    // actual task of the component
  }
  // reset the value of the stop variable
  stop = FALSE;
}
```

B. *Plug-in Runtime Management*

As already stated, plug-in management is performed at the Adaptation Layer, with the objective of enabling direct access to the RDF store without exploiting SSAP. In this way, plug-in extensions can use low-level and highly-efficient read, write, and query operations: other entities

(either plug-in extensions or KPs) cannot access concurrently the RDF store when one plug-in has started its execution, to avoid any possible interference with the active plug-in operation. To achieve these goals, we have designed out plug-in runtime management solution structured into three primary components: Plug-in Entry Point, Plug-in Manager, and Plug-in Timer, as depicted in Fig. 2).

Plug-in Entry Point offers an API to register and unregister plug-in extensions. plug-in developers can add/remove extension components very easily, either copying their Shared Object files into a given directory or remotely transferring them via TCP/IP. In both cases, Plug-in Entry Point checks the compliance of the plug-in extension to be installed with the standard template, by verifying that the extension actually provides `evaluateState()`, `run()`, and `stop()` functions.

Plug-in Manager periodically activates the currently registered plug-in extensions by invoking their `run()` function. The Smart-M3 architecture has been slightly modified to execute plug-in extensions inside the SIB daemon scheduler; Plug-in Manager executes registered extensions in the SIB scheduling loop, after insert operations and before query ones; the rationale is to make query operations to work on the already modified RDF store as soon as possible.

Plug-in Timer plays the role of ensuring fairness in terms of plug-in extension execution time. Whenever Plug-in Manager starts the execution of an extension, Plug-in Timer monitors its execution time and invokes its `stop()` function in case a given threshold has been passed. Thus, plug-in extensions are forced to gracefully release access to the RDF store, e.g., by possibly storing state related to their task execution prior to interrupting themselves and recovering it when restarting their task at the following activation. Additional details and the source code of the plug-in extension prototype can be found at the SOFIA Web site [4] and at the associated SourceForge repository - http://sourceforge.net/projects/smart-m3/.

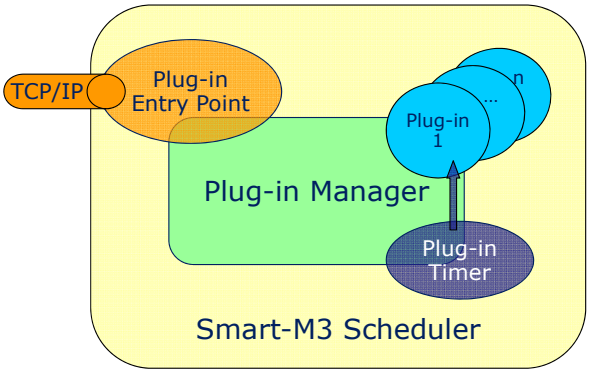

Figure 2. Modifications to the Smart-M3 SIB to support plug-in extensions.

IV. A Notable Example of Plug-in Extension Component: SIB Performance Profiling

We have tested the plug-in architecture presented in Section 2 and its implementation integrated with Smart-M3 by designing and deploying a SIB performance profiling feature, with the two-fold objective of validating the soundness of the proposed plug-in architecture and of offering an additional smart space functionality of relevance for smart application developers. In fact, we claim that SIB profiling can allow KPs to select the SIB that dynamically best fits their performance requirements: KPs may discover multiple SIBs even during service provisioning (nodes hosting SIBs can abruptly join and leave the network), heterogeneous not only in terms of supported features, e.g., with a different set of plug-in extensions, but also considering computing capabilities (varying depending on installed hardware and current computational load). Let us stress that anyway the plug-in architecture can be exploited to develop a wide set of other features that can benefit from direct and exclusive access to the RDF store, e.g., ranging from garbage collection to ontology-based inferencing.

A. *SIB Profiling Performance Indicators*

Also in order to well evaluate the efficiency and effectiveness of our plug-in extension implementation, we have prototyped two different options for SIB profiling, one based on the operations of a regular KP and the other based on an extension component that exploits our plug-in interface. The two options are implemented, respectively, based on the following components:

- **Profiling KP**, i.e., an SSAP-based regular KP accessing the RDF store, in competition with other KPs, as usual. The performance achievable by a regular KP depends on the executing node current conditions, in terms of load, network traffic, … The SSAP protocol is expected to increase latency and overhead to access the RDF store if compared with native query operations at the adaptation layer. Moreover, SIB access is shared among multiple concurrent KPs, by possibly adding a source of further delay in insert/delete/query execution;
- **Profiling Plug-in**, i.e., a plug-in extension component with direct and exclusive access to the RDF store. SIB profiling based on the plug-in extension allows to test/assess the capabilities of the SIB-hosting node more accurately since, in this case, other KPs and plug-in extensions cannot interfere with the execution of the profiling tasks included in the plug-in.

It is worth noting that both approaches can produce results useful to compare the performance expected to be achieved by different available SIBs. In fact, the former represents the performance results that a regular KP can observe when accessing the SIB. Instead, the latter provides an ideal upper bound that KPs are not able to exceed: this value is particularly suitable for long-standing KPs, willing to compare SIBs performance not only based on recent monitoring results but also in relation to their potential capabilities, e.g., after temporary and current overload.

We propose the adoption of two indicators to quantitatively evaluate SIB performance by regular KPs and plug-in extensions (the lower, the better):

- **CP** = (KP insert + 10 * KP query + KP delete / 10) / 3;
- **BP** = (plug-in insert + 10 * plug-in query + plug-in delete / 10) / 3.

The rationale is that both Current Performance (CP) and Best Performance (BP) evaluate performance results by performing several insert/query/delete operations repeatedly: triples are generated in a random way, inserted in one transaction, and queried/deleted one by one. The same operations are repeated 100 times, split in 10 different cycles to adhere to the plug-in programming discipline. The query and delete values are multiplied and divided by 10 respectively, because we experimentally observed that they take about 1/10 and 10 times the insert execution time. In this way, after that normalization, it is possible to get a homogeneous value easier to manage for comparison purposes.

In addition, for the sake of simplicity, we propose a third indicator that is a simple combination of the two above:

- **RP** = CP / BP

By definition, the Relative Performance (RP) value is in the [1, ∞] range, where 1 is the best value since it means that current performance is equal to the best performance it is possible to achieve.

In other words, CP plays the role of the relevant indicator to estimate the current SIB performance, BP to know the best performance a SIB can achieve independent on current load conditions, and RP to know how much the SIB is currently loaded. For instance, a KP interested in quickly retrieving some information is likely to exploit the SIB with best (lowest) CP value, while a KP aiming at repeatedly accessing a SIB for a long time period may prefer the SIB with best (lowest) BP since it provides the best upper-bound performance. Finally, RP can be used to properly balance workload on available SIBs: a KP not interested in high performance could select the SIB with best (lowest) RP despite CP/BP values in order to avoid to further load an already largely occupied SIB. Only to mention a very simple example, the RP evaluation could be useful in a smart space including two SIBs, one hosted on a high-performance server already serving many KPs and one on a laptop not serving any KP. While the former provides greater BP and CP values, a KP without strict performance requirements could select the latter by achieving the benefits of not further loading the server and of accessing a less loaded SIB (see Section 4.2.2 for additional details).

B. Evaluating the SIB Profiling Components

We have implemented the Profiling Plug-in and the Profiling KP respectively in the C language and in Python+Java. In our implemented prototype, the Profiling Plug-in executes only once a day since it measures rather static performance that is unlikely to vary very often (its value mainly depends on hardware characteristics). Instead, the Profiling KP runs every two hours to monitor the daily performance trend and executes in the same node hosting the SIB, in order to avoid network traffic overhead. As a result of its operation, the Profiling Plug-in inserts in the RDF store the triple "*http://sofia.org/sib_internal#sib_properties"*, *":best_performances"*, *BP_value* representing the BP value. Similarly, the Profiling KP inserts the triples *"http://sofia.org/sib_internal#sib_properties"*, *":current_performances"*, *CP_value* and *"http://sofia.org/sib_internal#sib_properties"*, *":relative_performances"*, *RP_value* in the SIB RDF store.

We have performed several tests to assess the effectiveness and correct working of the proposed plug-in architecture and performance indicators. The used testbed consists of two Linux nodes (Ubuntu distribution 10.10 and 11.04) with only one KP running on each node (to avoid excessive delays due to processor scheduling); for briefness sake, we identify the two nodes as:

- **High**: Intel Core2 Duo P8400 2.26GHz, 3GB RAM;
- **Low**: Intel Pentium M processor 1,10GHz, 500MB RAM.

The Profiling Plug-in and KP gather CP/BP/RP values when the SIB alternatively resides in one of these nodes. To achieve significant CP values, the daily trend of each single SIB has been evaluated while imposing different loads by means of workload KPs specifically designed to emulate some common tasks, composed by cycles with a heterogeneous mix of insert/delete/query operations (see Table I). Each cycle includes 8 insertions and 2 deletes, while at the end of each cycle one triple is added in a query list; finally, triples in the query list are queried one by one.

TABLE I. DAILY WORKLOAD TO SIMULATE SIB WORKLOAD

Daily time	**Workload Conditions**
1:00, 3:00. 5:00, 7:00, 23:00	No workload KPs
9:00	1 workload KP, 18 cycles
11:00, 13:00	1 workload KP, 100 cycles
15:00	2 workload KPs from different nodes, 20 cycles
17:00	2 workload KPs, from different nodes, 40 cycles
19:00	2 workload KPs, from different nodes, 20 and 40 cycles
21:00	1 workload KP, 40 cycles

Fig. 3 shows the daily trend of CP and BP values related a single node: the SIB alternatively runs on High (left) and Low (right); BP values are 0.86 for High and 1.24 for Low. The reported results clearly show that the adopted performance indicators depend on node workload and hardware capabilities: High has lower execution time than Low, since the former is equipped with a more powerful processor and greater memory resources.

Then, we have tested our SIB profiling solution exploiting two KPs with very different behavior and performance requirements:

- **FastKP**, executing few operations with strict delay requirements (representative of the operations of a CPU-bound non-interactive KP);
- **SlowKP**, executing several operations, but without strict delay requirements (representative of the operations of an IO-bound KP, e.g., interacting with a user).

In our envisioned profiling-enabled scenario, first of all, KPs look for and join the available SIBs to gather CP/BP/RP values; then, KPs evaluate the obtained performance indicators and keep connected only with the SIB best fitting their requirements, i.e., the SIB with best CP in case of FastKP, the SIB with best RP in case of SlowKP. Note that KPs can exploit CP/BP/RP values and combine them as they prefer to choose their "best" SIB; in the following, for the sake of simplicity, we focus on CP and RP to underline their suitability to achieve a good tradeoff between best performance and workload balancing.

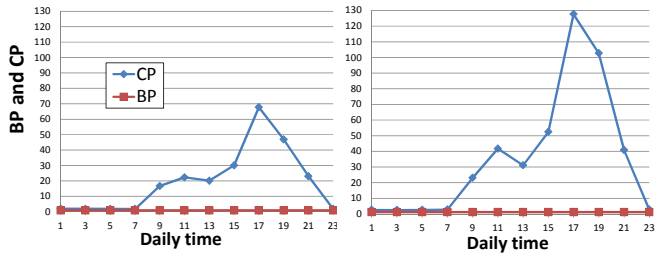

Figure 3. CP and BP when the SIB runs on High (left) and Low (right).

Secondly, Fast/Slow KPs work by performing insert, query, and delete operations. In particular, FastKP operates 10 inserts, 5 queries, and 1 delete: each operation type is performed in an aggregated manner, e.g., 10 inserts correspond to only one SIB interaction, in order to optimize operation processing. SlowKP performs 100 inserts, 50 queries and 10 deletes, thus emulating a greater workload in terms of SIB requests. In addition, each SlowKP operation is performed separately, e.g., 100 SIB interactions to perform 100 inserts, to mimic inter-operation activity, e.g., waiting for data provided by other KPs.

In order to evaluate how CP/BP/RP can give relevant indications for the SIB choice, in the following we provide two different comparisons:

1. FastKP chooses the SIB randomly vs. the one with best CP (only FastKP executes in the smart space);
2. SlowKP chooses the SIB with lowest CP vs. the one with best RP (SlowKP executes while FastKP already accesses the SIB with best CP).

Fig. 4 shows the average execution time (10 runs for each case) of FastKP (left) and SlowKP (right). When FastKP exploits CP to properly evaluate and select the most suitable SIB, its execution time relevantly lowers from 0.63s (left column) to 0.35s (right column). Instead, when SlowKP exploits RP, the execution time rises from 13.62 to 23.08s. It is worth noting that time execution growth for SlowKP is not a crucial issue since SlowKP has not strict latency requirements. In addition, the exploitation of RP allows to better consume smart space resources because SlowKP selects the SIB not already accessed by FastKP. In fact, it is thus possible to distribute the workload on the available SIBs more evenly (see Table II): by adopting RP, SlowKP can exploit the less loaded (more idle) SIB.

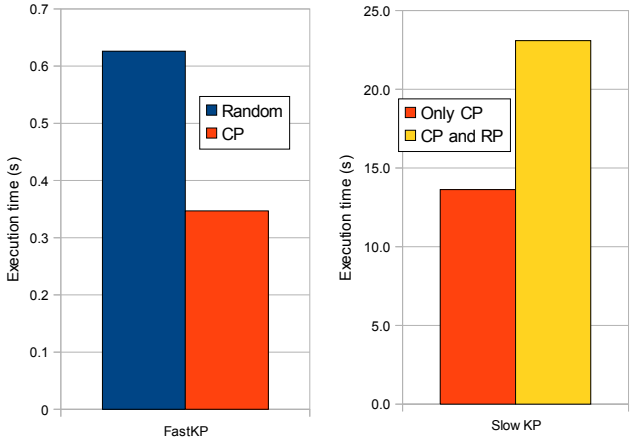

Figure 4. KP execution time when adopting CP and RP for SIB selection.

TABLE II. CPU AVERAGE CONSUMPTION OF SIB WHEN BOTH FASTKP AND SLOWKP RUN (SIBD IS THE MAIN SMART-M3 PROCESS; THE SIB-TCP PROCESS PROVIDES TCP-BASED ACCESS TO SMART-M3).

Node	**High**		**Low**	
Component	***sibd***	***sib-tcp***	***sibd***	***sib-tcp***
Both Fast/ SlowKP use CP	3.62%	1.02%	0.00%	0.00%
FastKP uses CP, SlowRP uses RP	1.00%	0.01%	2.56%	0.56%

Finally, Table III and Fig. 5 show the overhead that KPs have to pay for SIB performance-based comparison before selection, i.e., to join to available SIBs, query CP/BP/RP indicators on them, and evaluating their suitability. Table 3 points out that comparison overhead is greatly lower than execution time, while Fig. 6 underlines that SIB joining and performance indicator gathering have the greatest impact on overhead. In other words, comparison overhead grows almost linearly in relation to the number of available SIBs to interrogate and compare. However, even in the challenging case of FastKP requiring small execution delays, the steps of joining, gathering, and comparing impose delay that are largely lower than execution time, thus demonstrating that the proposed solution imposes very little overhead. In fact, the SIB selection overhead may be well balanced by the capability of choosing the proper SIB, thus allowing the reduction of overall execution time in the smart space.

TABLE III. EXECUTION TIME AND OVERHEAD COMPARISON.

	FastKP		SlowKP	
	Execution (s)	***Comparison (s)***	***Execution (s)***	***Comparison (s)***
Both FastKP and SlowKP use CP	0.35	0.02	13.62	0.04
FastKP uses CP, SlowRP uses RP	0.35	0.02	23.08	0.03

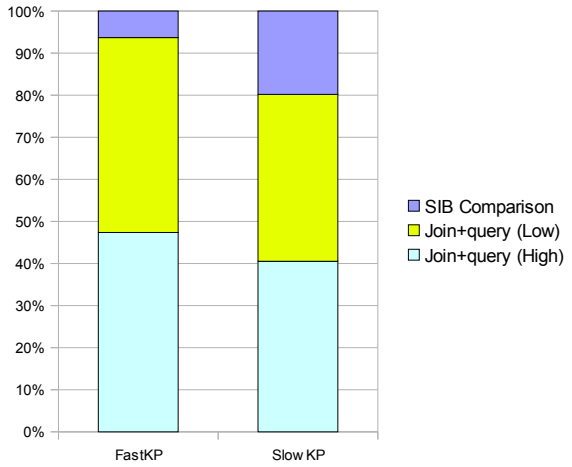

Figure 5. The different elements composing the SIB comparison overhead.

V. RELATED WORK

Several contributions in the literature recognize the importance of supporting heterogeneous device interoperability to actually push the spread of open and flexible smart spaces. For example, [6] presents standardization efforts to develop a universal interaction solution with diverse devices, by specifically focusing on user interaction. Possible approaches can be categorized in either universal user interface languages, allowing to describe user interfaces based on an abstract language, or user interface remoting, exploiting service discovery and agreed presentation protocols to interact with remote devices. Space Integration Services (SIS) achieve device interaction by supporting the dynamic mapping of locations, e.g., physical spaces or devices, related to different smart spaces. In this way, the data published in a location of a smart space are made visible also to other smart spaces to which the location is mapped [7].

By focusing on interoperability among smart space software components, [8] exploits standard communication protocols such as SOAP Web Services and agreed presentation common metadata schemas based on XML/RDF. [9] exploits a RDF-based Resource Information Base (RIB) as the resource sharing environment. RIB contains information that devices are willing to share, thus allowing remote heterogeneous devices to easily discover available service components and to dynamically compose them, e.g., to provide a new virtual service stemming from the mash-up of available and more basic components.

Other contributions propose the exploitation of ontologies and reasoners to further increase interoperability among heterogeneous devices [10]. For instance, SocioSpace exploits the IP Multimedia Subsystem (IMS) standard specification to integrate Smart Spaces and social networks, by adopting Friend of a Friend (FOAF) as the common syntactic vocabulary to exchange and represent social network data [11]. Shared Ontologies for Pervasive Computing (SO4PC) proposes to adopt i) a core ontology (SO4PC Core) defining common generic vocabularies and ii) additional ontologies (SO4PC Extensions) extending the core one to support specific types of applications [12]. Finally, Smart Space Event Ontology (SSEO) uses semantic reasoning to compose different semantic events triggered by heterogeneous devices [13]. To this purpose, SSEO exploits a semantic adaptor to translate sensor-gathered events into a shared primitive format and a semantic reasoner to infer from them human-readable and machine-processable semantic events, easily sharable among heterogeneous devices. However, these solutions do not provide the capability of interoperating at the data level in a dynamic way as Smart-M3 can do by including metadata descriptions in its RDF store; in addition, these solutions do not provide any capability of adding novel features to the smart space support platform at runtime in order to further increase its flexibility and to tailor it for a specific deployment environment and application domain.

VI. CONCLUSIONS

Our proposed SIB extensibility solution based on the plug-in architecture allows to customize and personalize the SIB behavior dynamically in relation to smart space objectives and administrator requirements. In particular, we have modified the Linux-based Smart-M3 implementation to allow the dynamic de/activation of extension components, possibly developed by third parties, implemented as Shared Objects. In this way we achieve the twofold objective of keeping the SIB architecture very lightweight (suitable even for low-end and resource-constrained mobile phones) and of enabling the possibility to add new features to the SIB behavior dynamically, depending on the targeted application and deployment environment.

The presented SIB profiling service test-case not only provides performance indicators useful to select the SIB best fitting the current KP requirements, but also demonstrates that regular KPs and Plug-in extensions can be used together to collect performance data under different perspectives. In both cases, the reported performance results show the feasibility of the approach in terms of limited overhead, for all most practical smart application cases of common interest. We truly believe that this design/implementation work, freely available for download, can relevantly help the community of researchers/practitioners working on smart space interoperability at the information level and/or on Smart-M3-based smart spaces to extend the available SIB implementations with ad-hoc functionality of specific interest for their application scenarios, thus leveraging the

adoption of the proposed approach in very differentiated application domains and growing the associated community of developers.

ACKNOWLEDGMENT

This research activity, accomplished within the framework of the Artemis JTI SOFIA project, has been possible also thanks to the support of EIT ICT Labs (Helsinki node) and Nokia. Our thanks also go to Petri Liuha and Vesa Luukkala from Nokia for the fruitful and enjoyable discussions we had the opportunity to have with them, in order to improve and refine the architecture and prototype of the Smart-M3 SIB plug-in extension interface.

REFERENCES

[1] J. Honkola, H. Laine, R. Brown, O. Tyrkko, "Smart-M3 Information Sharing Platform", IEEE Symp. on Computers and Communications (ISCC), pp.1041-1046, IEEE Press, 2010.

[2] J. Kiljander, M. Etelapera, J. Takalo-Mattila, J. Soininen, "Opening Information of Low Capacity Embedded Systems for Smart Spaces", 8th Workshop on Intelligent Solutions in Embedded Systems (WISES), pp. 23-28, 2010.

[3] S. Balandin, I. Oliver, S. Boldyrev, A. Smirnov, N. Shilov, A. Kashevnik, "Multimedia Services on Top of M3 Smart Spaces", Int. Conf. on Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), pp. 728-732, 2010.

[4] "SOFIA Project – Smart Objects For Intelligent Applications", funded through the European Artemis programme, http://www.sofia-project.eu/

[5] A. D'Elia, J. Honkola, D. Manzaroli, T. Salmon, "Access Control at Triple Level: Specification and Enforcement of a Simple RDF Model to Support Concurrent Applications in Smart Environments", Int. Conf. Smart Spaces and Next Generation Wired/Wireless Networking, LNCS 6869, pp. 63-74, 2011.

[6] Choonhwa Lee, S. Helal, Wonjun Lee, "Universal Interactions with Smart Spaces", IEEE Pervasive Computing, vol.5, no.1, pp. 16- 21, 2006.

[7] D. Bernini, D. Micucci, F. Tisato, "A Platform for Interoperability via Multiple Spatial Views in Open Smart Spaces", IEEE Symp. on Computers and Communications (ISCC), pp. 1047-1052, 2010.

[8] B. Simon, S. Sobernig, F. Wild, S. Aguirre, S. Brantner, P. Dolog, G. Neumann, G. Huber, T. Klobucar, S. Markus, Z. Miklos, W. Nejdl, D. Olmedilla, J. Salvachua, M. Sintek, T. Zillinger, "Building Blocks for a Smart Space for Learning", Sixth Int. Conf. on Advanced Learning Technologies, pp.309-313, 2006.

[9] K- Sohn, Ki-Hyuk Lee, Taehyun Kim, Sangshin Lee, Jaeho Kim, "Resource Sharing Using RDF in Ubiquitous Smart Space", Int. Conf. on Convergence Information Technology, pp. 2062-2065, 2007.

[10] X. Wang, J.S. Dong, C.Y. Chin, S.R. Hettiarachchi, D. Zhang, "Semantic Space: an Infrastructure for Smart Spaces", IEEE Pervasive Computing, vol.3, no.3, pp. 32- 39, 2004.

[11] A. Hasswa, H. Hassanein, "SocioSpace: an Adaptive Service-oriented Architecture that Integrates Smart Spaces and Social Networks through the IP Multimedia Subsystem", IEEE Symp. on Computers and Communications (ISCC), pp. 85-90, 2011.

[12] Jun-Feng Man, Qing Chen, Xiao-Heng Deng, Yin-An Qiu, "The Design and Implementation of Shared Ontologies for Smart Space Application", Proc. of 2005 Int. Conf. on Machine Learning and Cybernetics, vol.1, pp. 125-131, 2005.

[13] Zang Li, Chao-Hsien Chu, Wen Yao, R.A. Behr, "Ontology-Driven Event Detection and Indexing in Smart Spaces", IEEE Fourth Int. Conf. on Semantic Computing (ICSC), pp. 285-292, 2010.