A Notification Model for Smart-M3 Applications

Ivan V. Galov[1] and Dmitry G. Korzun[1,2]

[1] Department of Computer Science, Petrozavodsk State University (PetrSU)
33, Lenin Ave., Petrozavodsk, 185910, Russia
{galov,dkorzun}@cs.karelia.ru
[2] Helsinki Institute for Information Technology (HIIT) and
Department of Computer Science and Engineering (CSE), Aalto University
P.O. Box 15600, 00076 Aalto, Finland

Abstract. Smart-M3 platform supports development of applications consisting of autonomous knowledge processors that interact by sharing information in a smart space. In this paper, we introduce a notification model for ontology-based design and programming of interactions in such applications. Our model is based on the two Smart-M3 fundamentals: subscription operation and RDF representation. The applicability is demonstrated on the case study of SmartScribo system for multi-blogging and on simulation experiments for performance evaluation.

Keywords: Smart spaces, Smart-M3, Publish/Subscribe, RDF.

1 Introduction

The smart spaces paradigm aims at development of ubiquitous computing environment that acquires and applies knowledge to adapt services to the inhabitants [1]. Smart-M3 interoperability platform [2] provides means for creating and deploying smart spaces. (M3 stands for Multidevice, Multidomain, and Multivendor.) Examples of Smart-M3 applications are SmartScribo system [3] for personalized semantic multi-blogging and SmartRoom system [4] for collaborative work in a spatially localized digital environment. In Smart-M3, a smart space realizes a shared knowledge base for use by applications [2, 5]. Each application consists of knowledge processors (KP) that interact in the smart space using blackboard [6] and publish/subscribe [7] interaction models. The information representation is RDF-based, employing Semantic Web technologies [8].

Application developers are faced with problems of design and programming of interacting KPs. In addition to blackboard-based read/write primitives, advanced interaction is based on publish/subscribe: whenever one KP publishes data in the smart space some other KPs are notified about the changes due to subscription. When many KPs participate and much data are shared such interaction becomes complicated for implementation. In this paper we analyze this design and programming problem. We introduce a notification model that systematizes the interaction part on the application level and provides properties to implement on each individual KP. Notification model allows construction of

S. Balandin et al. (Eds.): NEW2AN/ruSMART 2014, LNCS 8638, pp. 121–132, 2014.

information flows coupling a publisher KP with its subscriber KPs. Notification model is ontology-based, enhancing the ontology of the whole application.

The rest of the paper is organized as follows. Section 2 states the problem of design and programming of knowledge processors interaction in a Smart-M3 application. Section 3 describes the notification model and its design properties. Section 4 analyses the applicability of the model using SmartScribo system as a case study. Section 5 evaluates the performance using simulation experiments. Section 6 concludes the paper.

2 Smart-M3 Application: Interaction in Smart Space

Each smart space provides a shared information store for its participants. Blackboard model [6] is used for data exchange: participants write/read data to/from the shared information store. The model is extended with publish/subscribe [7]: participants subscribe on specific content and receive updates made by others.

Smart-M3 platform [2] is open-source platform for implementing smart spaces. The key architectural component of Smart-M3 is Semantic Information Broker (SIB) that provides access to the shared content. Participants are knowledge processors (KPs); they are software agents running on devices of the environment. Publish/subscribe operation supports advanced interaction of KPs when subscriber KPs make persistent queries and react on asynchronously incoming updates [9]. The idea of such interaction is shown in Fig. 1.

Shared content is represented using Resource Description Framework (RDF) of the Semantic Web [8]. An RDF model consists of a set of RDF triples, each has the form of "subject–predicate–object". Such a representation forms a directed graph with subjects and objects as nodes and predicates as links.

Every KP has its own description of content the KP shares participating in the smart space. This content forms a partial RDF graph stored in the smart space (maintained by SIB). An RDF model is machine-oriented and does not provide human-oriented mechanisms to describe semantics of the content nor determine problem-aware content representation structure. For this purpose, KP developers apply ontologies at the design phase. Application ontology declaratively describes the application domain. Overlapping of individual KP ontologies

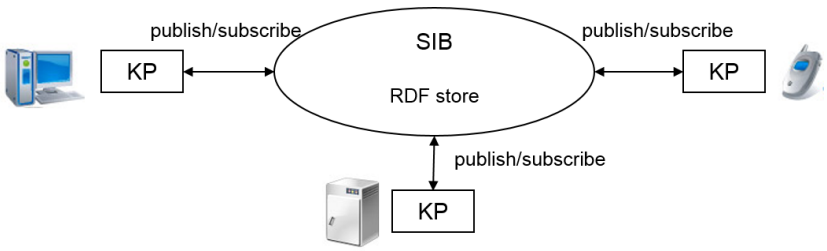


Fig. 1. Smart-M3 concept

makes interaction between KPs possible: each KP tracks changes in shared content if the latter is described within the KP's ontology.

Ontologies can be described with web ontology language (OWL). On one hand, an OWL ontology is serializable to RDF triples. On the other hand it allows describing application domain data and processes in terms of classes and their properties, which provides a high abstraction level for Smart-M3 application developers [10, 11].

From the OWL point of view smart space content consists of linked objects, called individuals. Any individual is an instance of particular ontological class of the application ontology. From the RDF point of view, an individual is a set of RDF triples with the same value of triple subject (i.e., an RDF subgraph). This triple subject is called the identifier of individual. We shall use descriptions in RDF as it allows to see what is actually stored in the smart space. (Note that SIB always operates on the RDF level an RDF triple store is used on the bottom.) Nevertheless we still exploit the term "individual" for simpler intuition.

One KP (or interaction of several KPs) constructs a service. There are KPs involved into service provision and KPs acting as service consumers. An example with two KPs—a client KP and a service KP—is shown in Fig. 2. Each user runs her/his client KP, which changes some properties of different individuals. The service KP has the constraint: to process modified individuals correctly the KP needs to receive the whole updated individual, not just updated properties separately. With subscription on separate properties of individuals, however, service KP subscription would run several times (each time with one edited property). It is not obvious how to estimate a moment when individual modification is finished and it is ready for processing on the service KP. The situation becomes more complicated when several individuals are modified and needed to be processed at the same time. In this case it is more convenient when one KP notifies other KPs about the need of processing specific individuals when they are already modified in the smart space. Thus we prevent reading information that is not ready for processing. It can be considered as some kind of access control mechanism. Another approach of access control is presented in [12] where undesirable changing of information is prohibited.

Let KP sender and KP receiver need to interact. The KP sender needs to pass a portion of information to the KP receiver to attain a required action.

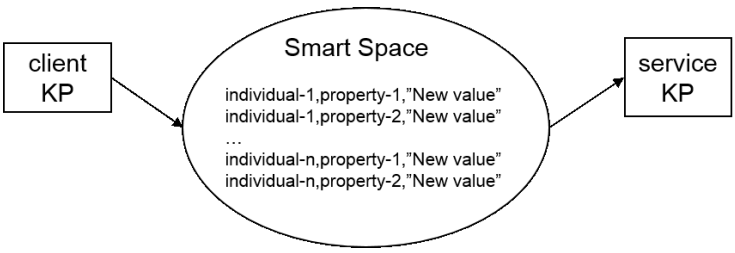


Fig. 2. Schematic example of subscription on several properties

This information can be represented either text data or individuals from the application ontology. The result of interaction is information transfer or service delivery. We consider the following problems where interaction of KPs appears: 1) service provision, 2) service composition, 3) service information dissemination.

In service provision, one KP consumes a service from another KP, and the former usually runs on a personal user device. Service composition implies that a service provides input for another service, forming a new composed service. Besides, services can disseminate some information between each other.

In general, several KPs are involved into interactive activity. To solve this design problem of interacting KPs we introduce a notification model. It focuses on implementation of basic pair-wise interaction between the involved KPs.

3 Notification Model

Given a Smart-M3 application, its notification model describes possible situations for KPs to interact with each other. We consider the following classes of situations where one KP (receiver, denote $\mathrm{KP_{rcv}}$) or more ones are involved into interaction by another KP (sender, denote $\mathrm{KP_{snd}}$).

Request: $\mathrm{KP_{rcv}}$ performs a given operation (service) based on data provided by $\mathrm{KP_{snd}}$.

Event: $\mathrm{KP_{rcv}}$ reacts on a particular event (informational fact) that $\mathrm{KP_{snd}}$ is disseminating.

These situations defines two functional viewpoints on $\mathrm{KP_{rcv}}$ in a Smart-M3 application: a data processor or a reacting unit. The former is closer to procedure call programming and the latter follows event-driven programming.

Consider the following pairwise interaction where asynchronous communication applies the publish/subscribe model

$$\mathrm{KP_{snd}} \leftrightarrow \mathrm{KP_{rcv}}. \tag{1}$$

The following properties are achieved for interacting KPs.

One-to-many: A single $\mathrm{KP_{snd}}$ can affect many $\mathrm{KP_{rcv}}$ at once.
Decoupling: Sending and receiving do not block $\mathrm{KP_{snd}}$ and $\mathrm{KP_{rcv}}$.
Anonymity: $\mathrm{KP_{snd}}$ and $\mathrm{KP_{rcv}}$ do not need to know each other.

We define a *notification* an abstract informational message to be sent by $\mathrm{KP_{snd}}$ and to be received by $\mathrm{KP_{rcv}}$ if these KPs need to attain required interaction (performing operation, returning the result, informing on event). In our model, a notification is represented as an RDF triple or a linked set of them. Note that the basic subscription mechanism of Smart-M3 operates on the RDF level, though OWL-aware extensions are available for application developers [9].

Notification process in interaction (1) is depicted in Fig. 3. The steps are implemented in application logic of both $\mathrm{KP_{snd}}$ and $\mathrm{KP_{rcv}}$. Before the process, the KPs subscribe for the notifications defined in the model.

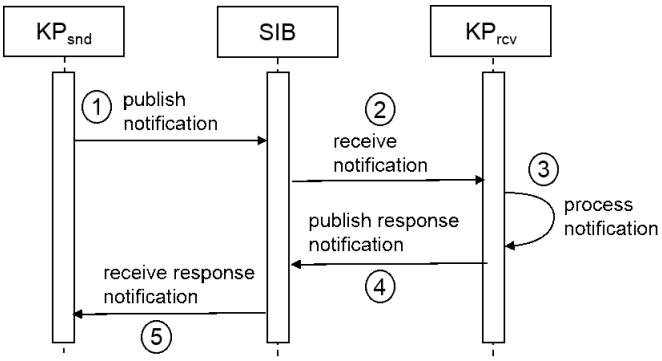


Fig. 3. Notification process for interacting KP_{snd} and KP_{rcv}

1. KP_{snd} publishes the notification in the smart space to start the required interaction with all appropriate KP_{rcv}.
2. KP_{rcv} recognizes the act of publishing due to the subscription and receives the notification data.
3. KP_{rcv} performs a corresponding operation based on the notification data.
4. KP_{rcv} publishes a response notification with the operation result.
5. KP_{snd} receives a response notification due to subscription mechanism.

Steps 1–3 are mandatory for interaction (1) since they implement forward actions in the interaction loop. Steps 4–5 implement feedback actions for the response notification, which are omitted if not needed by the application logic.

We distinguish the following design properties of a notification: representation, activation, function, response, clearing, performance. The KP developer implements these properties when programming the logic of a given KP.

Representation: A simple notification is represented with a single RDF triple. The general form is

$$\langle \text{notification\_id} \rangle, \langle \text{notification\_name} \rangle, \langle \text{value} \rangle$$

where ⟨notification_id⟩ is the identifier of notification individual, ⟨notification_name⟩ is the name of operation or event, ⟨value⟩ is the value of parameter (string data or identifier of some individual from the application ontology). The parameter holds data needed to pass to the notification receiver.

A compound notification consists of several RDF triples. It allows passing several parameters within the notification. The general form is

$$\langle \text{notification\_id} \rangle, \langle \text{notification\_name} \rangle, \langle \text{individual\_id} \rangle$$
$$\langle \text{individual\_id} \rangle, \langle \text{parameter1} \rangle, \langle \text{value1} \rangle$$
$$\ldots$$
$$\langle \text{individual\_id} \rangle, \langle \text{parameterN} \rangle, \langle \text{valueN} \rangle$$

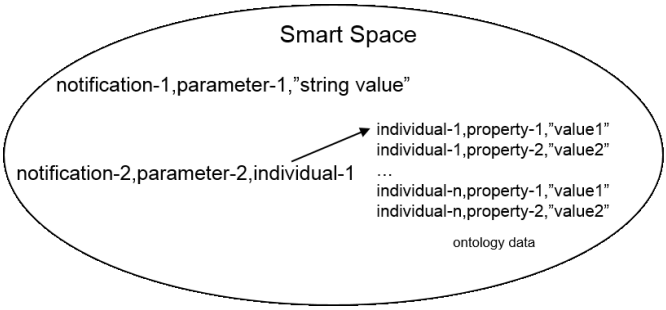


Fig. 4. Relation between a notification and the application ontology

where ⟨notification_id⟩ is the identifier of notification individual, ⟨notification_name⟩ is the name of operation or event, ⟨individual_id⟩ is identifier of the additional notification individual which stores other notification parameters, ⟨parameterN⟩ is the notification parameter name, ⟨valueN⟩ is the notification parameter value.

For a given application, its notification model implements an extension to the application ontology. Recall that every notification has parameter(s) and the value is stored as an object of notification RDF triple(s). The value can be either string data or ontology data. Ontology data are represented as a set of individuals in the smart space. Therefore, a notification can be linked with individuals in the smart space as schematically shown in Fig. 4. Thereby, interaction between KPs is performed due to changes in application ontology extension and not ontology data themselves. Otherwise, passing information from one KP to another requires to change individuals properties directly, which is not appropriate for data transferring. Moreover notifications can act as information exchange protocol between KPs operating on completely different ontologies.

Activation: Notifications are divided into two types depending on the reactive and proactive styles of activation at the KP sender side.

Reactive notification is used when a particular command (e.g., user action) directs $\mathrm{KP}_{\mathrm{snd}}$ to call $\mathrm{KP}_{\mathrm{rcv}}$ to perform operation (service). In this case, $\mathrm{KP}_{\mathrm{snd}}$ typically awaits for a response notification with the operation result.

In proactive notification, $\mathrm{KP}_{\mathrm{snd}}$ sends the notification with no explicit command from the user. Typically, $\mathrm{KP}_{\mathrm{snd}}$ does not wait for response. The user context forms implicit dependence on user activity: $\mathrm{KP}_{\mathrm{snd}}$ analyzes the context in the background (in parallel with primary user activity) and then activates events for other KPs.

Consider an example of reactive notification. A user consumes a service from her/his client KP. The service is constructed by another KP (service KP). When the user makes a control action (e.g., pushing a button) then the client KP sends reactive notification to the service KP and waits for the result. In this case, user is aware that a specific operation is going to be executed and it starts after the particular control command.

An example of proactive notification uses two KPs: client KP and recommender KP. The latter analyzes the data the user is working with and suggests related data from available sources. On the client KP the user browses service information and in parallel the KP sends a notification to the recommender KP (no explicit user action). The recommender KP reacts and finds a list of recommendations. In this case, user can be unaware that something is happening during her/his primary activity. A recommender KP can be even turned off: the system continues with a reduced service set. Proactive notification are usually used in the event-based interaction between KPs.

Function: A notification for interaction (1) can be a request notification or event notification. Request notification is used for another KP to perform certain operation. Event notification is used for informing another KP about an event. The previous examples applied request notifications: the client KP requests the service KP to perform some operations. Event notification is used to notify service KPs about the current user activity, e.g., a notification is sent when the user is reading. Then the service KP provides additional information (as a recommender KP). Also, the fact of reading can be used to block the service KP to deliver its service (e.g., reading is non-interrupted activity).

Response: Depending on the notification destination its KP sender is waiting or not for a response notification (contains operation result). While each client KP notifies a service KP to perform an operation, waiting for a response notification can be needed to show user whether operation was succeed or not. On the contrary, a KP client sending notification about user reading event does not wait for a response as it is possible that no KP will receive and process it. Request interaction between KPs most likely includes response notification while event interaction does not.

Clearing: A request notification is removed by its receiver after completion of required operation. An event notification is removed by its sender, which determines itself the time the notification is kept published.

Performance: Subscription is resource-consuming, thus every KP needs to minimize the number of subscriptions. Notification processing can be implemented within one subscription using RDF triple template with a fixed subject:

$$\langle \text{notification\_id} \rangle, *, * \tag{2}$$

where $*$-mask represents any value. Each KP has its own unique $\langle$notification_id$\rangle$.

In summary, our notification model is a universal solution to make interaction implementation between KPs easier. Note that 100% delivery of subscription updates to subscribers is not guaranteed. For instance, some packets are lost in the communication network. Each KP may assume the best-effort delivery only. Additional resilience mechanisms should be built into application KP logic.

4 Case Study: SmartScribo System

Let us consider a particular Smart-M3 application to show how our notification model can be applied. SmartScribo system [3] aims at semantic mobile multi-blogging: mobile users interact with multiple blogs at many blog services simultaneously. The smart space stores data related to user blogs and personal information. The architecture is shown in Fig. 5. There are three types of KPs: KP client, KP blog processor, and KP mediator. Each KP client is installed on a personal mobile device. KP blog processor implements interaction with a particular blog service. For each blog service the system has a separate KP blog processor. KP mediator is responsible for additional processing of smart space content (e.g., personalized blog recommendation).

The user workflow is organized as follows. On the KP client its user specifies information about her/his blogs. The KP blog processors receive information about posts from that blogs. Then the user can read/edit existing posts or create new ones. When a new post is created or existed one is updated the blog processors reflect these changes at the blog service.

The smart space can keep a lot of blogs and posts of many users. For a blog processor it becomes difficult to detect which posts were updated using subscription on selected properties of individuals. For example, a post individual has such properties as title, text, tags. Blog processor can subscribe on triples: ⟨post_id⟩–title–∗ and similarly for other properties. It can even subscribe on triple ⟨post_id⟩–∗–∗. Both cases have the problem: the KP receives subscription updates separately on each property.

When can updated post individual be transferred to blog service? Moreover, posts are created and deleted dynamically in the smart space, and it is not easy to set/unset subscription on each individual every time. Although a blog

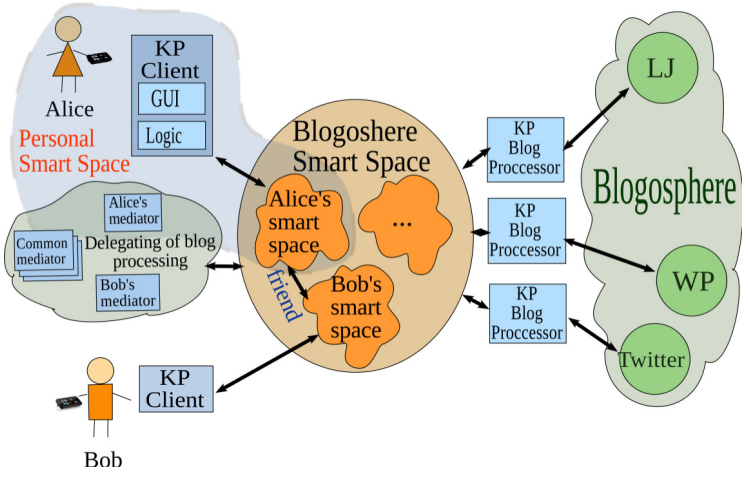


Fig. 5. SmartScribo system architecture, from [3]

Table 1. The set of notifications in SmartScribo system

Notification	Parameters	Description
refreshAccount	account individual identifier	receive blog account information
refreshPosts	account individual identifier	receive posts for given account
sendPost	account individual identifier, post individual identifier	send post to account
editPost	old post individual identifier, new post individual identifier	update old post to a new one
delPost	account individual identifier, post individual identifier	delete post
refreshComments	account individual identifier	receive comments for all posts of given account
sendComment	account individual identifier, comment individual identifier, parent individual identifier	send comment to parent message (post or comment) of given account
delComment	account individual identifier, comment individual identifier, parent individual identifier	delete comment for given parent message

processor can subscribe only on properties related to a post individual, i.e. $*$–title–$*$. However it does not solve the problem: subscription will inform about property changes separately and such updates will be mixed among several posts. Therefore a solution is to inform a blog processor about the need of performing particular action with a particular post.

SmartScribo applies the notification model in interaction between KP client and KP blog processor. The list of all notifications is presented in Table 1. The notifications are request and reactive as they are sent after user actions and require blog processor to perform operations. Every notification requires a response notification which informs the user about operation result. As parameters every notification has individual identifier, i.e., subject of triple to represent a particular account, post, or comment.

Notification "refreshPosts" has the form

$$\text{Notification-}\langle\text{service}\rangle, \text{refreshPosts}, \langle\text{account\_id}\rangle$$

where $\langle$service$\rangle$ is a blog service type (e.g., "LJ" for LiveJournal), and $\langle$account_id$\rangle$ is an account individual identifier. Different service identifiers are used to distinguish notification for different blog processors. According to (2) each blog processor subscribes on a triple template where the subject is the service identifier while predicate and object are of any value. For LiveJornal blog processor the template is Notification-LJ–$*$–$*$.

A SmartScribo user specifies her/his blog account credentials using KP client and presses button to refresh posts list on that account. KP client publishes the account individual with all necessary properties for accessing it on the blog service and publishes "refreshPosts" notification. A KP blog processor receives notification with subscription, extracts account properties from the smart space,

receives a posts list from the blog service and removes the notification. Then the blog processor publishes the posts list in the smart space and sends a response notification to KP client to inform about the operation result.

Notification "sendPost" has the form

$$\begin{aligned}&\text{Notification-}\langle\text{service}\rangle, \text{sendPost}, \langle\text{notif\_ind}\rangle\\ &\langle\text{notif\_ind}\rangle, \text{postAcc}, \langle\text{account\_id}\rangle\\ &\langle\text{notif\_ind}\rangle, \text{postId}, \langle\text{post\_id}\rangle\end{aligned}$$

where ⟨service⟩ represents a blog service type, ⟨notif_ind⟩ is an identifier of additional notification individual storing all notification parameters, ⟨account_id⟩ is an account individual identifier and ⟨post_id⟩ — post individual identifier. A user (from KP client) creates a new post and then publishes this post and its notification to the smart space. A KP blog processor receives notification with subscription, and extracts account and post identifiers from the individual. Then KP extracts post properties using post identifier and sends this post to the blog service. Then the blog processor removes the processed notification and publishes a response notification for the KP client.

5 Performance Evaluation

Let $t(n)$ be the time elapsed since sending a notification from $\mathrm{KP}_{\mathrm{snd}}$ until receiving the result at $\mathrm{KP}_{\mathrm{rcv}}$. We consider two types of experiments: 1) all n parameters are retrieving in one subscription query based on (2), 2) each parameter $i = 1, 2, \ldots, n$ is received with a separate query (n iterations query) with the triple template

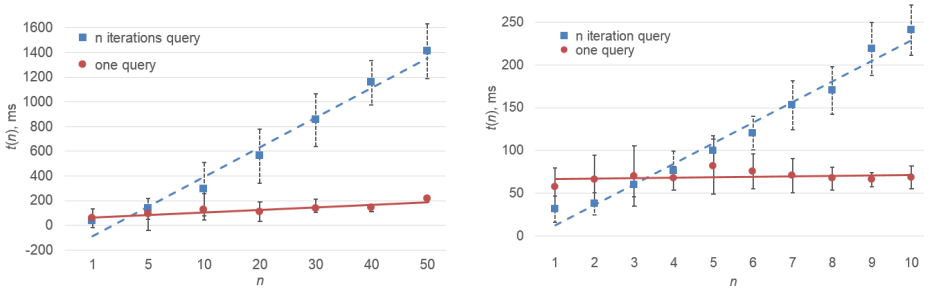
$$\langle\text{individual\_id}\rangle, \langle\text{parameter}_i\rangle, *$$

The first type represents the best way for KP to subscribe to n-parameter notification. The second type shows the worst case: $\mathrm{KP}_{\mathrm{rcv}}$ executes a loop to query value of each of n parameters. These two types represent lower and upper bounds on the performance. Let $t_{\mathrm{bst}}(n)$ and $t_{\mathrm{wst}}(n)$ be the time metrics to estimate experimentally.

The experimental setup includes SIB running on a server machine. Another computer hosts $\mathrm{KP}_{\mathrm{snd}}$ and $\mathrm{KP}_{\mathrm{rcv}}$. Both KPs are implemented in Python. The use of the same host computer for the KPs simplifies the time measurement since no synchronization is needed between different computers. The server machine and KP computer are located in different LANs with RTT = 3 ms on average.

Measurement cycle consist of 1) $\mathrm{KP}_{\mathrm{snd}}$ sends an n-parameter notification to SIB, 2) $\mathrm{KP}_{\mathrm{rcv}}$ receives the notification from SIB, retrieves values of all n parameters, and removes the notification. There are 100 samples for each n.

The plot in Fig. 6 (a) shows the average values for $t_{\mathrm{bst}}(n)$ and $t_{\mathrm{wst}}(n)$. The values fit well to linear regression, which is constructed for n up to 100. The shown plot is a truncated version for $n \leq 50$, since $t_{\mathrm{wst}}(n)$ for $n > 50$ behaves similarly and the growing difference $t_{\mathrm{wst}}(n) - t_{\mathrm{bst}}(n)$ hides the details of $t_{\mathrm{bst}}(n)$.



(a) (b)

Fig. 6. Experimental behavior of $t_{\mathrm{wst}}(n)$ and $t_{\mathrm{bst}}(n)$

The linear grows $t_{\mathrm{wst}}(n)$ is essentially faster compared to $t_{\mathrm{bst}}(n)$. It is a clear consequence of n iterations query implemented in a loop. Notably that it also leads to higher variability (for each average value its standard deviation is shown as a vertical bar). The observed linear grows of $t_{\mathrm{bst}}(n)$, although low, is due to more processing at the SIB side and more data to transfer when n increases. Therefore, we conclude that our notification model preserves reasonable performance even for large n.

In Smart-M3 applications, the typical case is relatively small n, see Section 4. The plot in Fig. 6 (b) shows the finer-grained measurements for $n = 1, 2, \ldots, 10$. The behavior is again linear, though with less slopes in linear regression. We conclude that $t_{\mathrm{bst}}(n)$ is low and almost constant for this typical case.

The observed grows of slopes of linear regression indicates that there is some non-linear effect in the performance when n increases. We expect that the effect is due to 1) search algorithm complexity at the SIB side and 2) data transfer resources the network provides to KP.

6 Conclusion

The paper presented our notification model for use in Smart-M3 applications. The model supports coordination of interacting KPs, which are autonomous distributed entities communicating via information sharing in the smart space. The RDF nature of the model makes it applicable for almost any Smart-M3 application. The model extends the application ontology with possible requests and events that KPs use in interaction. We classified the key design properties of notification, and a KP developer can consider them as programming patterns in interaction logic of KPs. Our case study—SmartScribo system—shows the feasibility of the model. Further steps of this research are development of resilience and performance optimization mechanisms related to the underlying subscription operation of Smart-M3.

Acknowledgment. This research is financially supported by project # 1481 (basic part of state research assignment # 2014/154) of the Ministry of Edu-

cation and Science of the Russian Federation. The reported study was partially supported by RFBR, research project # 14-07-00252. The work is a part of project 14.574.21.0060 of Federal Target Program "Research and development on priority directions of scientific-technological complex of Russia for 2014–2020".

References

1. Cook, D.J., Das, S.K.: How smart are our environments? an updated look at the state of the art. Pervasive and Mobile Computing 3(2), 53–73 (2007)
2. Honkola, J., Laine, H., Brown, R., Tyrkkö, O.: Smart-M3 information sharing platform. In: Proc. IEEE Symp. Computers and Communications (ISCC 2010), pp. 1041–1046. IEEE Computer Society (June 2010)
3. Korzun, D.G., Galov, I.V., Balandin, S.I.: Proactive personalized mobile mutliblogging service on SmartM3. Journal of Computing and Information Technology 20(3), 175–182 (2012)
4. Korzun, D., Galov, I., Balandin, S.: Smart room services on top of M3 spaces. In: Balandin, S., Trifonova, U. (eds.) Proc. 14th Conf. of Open Innovations Association FRUCT, SUAI, pp. 37–44 (November 2013)
5. Korzun, D.G., Balandin, S.I., Gurtov, A.V.: Deployment of Smart Spaces in Internet of Things: Overview of the design challenges. In: Balandin, S., Andreev, S., Koucheryavy, Y. (eds.) NEW2AN 2013 and ruSMART 2013. LNCS, vol. 8121, pp. 48–59. Springer, Heidelberg (2013)
6. Corkill, D.D.: Collaborating Software: Blackboard and Multi-Agent Systems & the Future. In: Proc. the Int'l Lisp Conference (October 2003) (invited paper)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35, 114–131 (2003)
8. Chen, H., Finin, T., Joshi, A., Kagal, L., Perich, F., Chakraborty, D.: Intelligent agents meet the semantic web in smart spaces. IEEE Internet Computing 8, 69–79 (2004)
9. Lomov, A.A., Korzun, D.G.: Subscription operation in Smart-M3. In: Balandin, S., Ovchinnikov, A. (eds.) Proc. 10th Conf. of Open Innovations Association FRUCT and 2nd Finnish–Russian Mobile Linux Summit, SUAI, pp. 83–94 (November 2011)
10. Palviainen, M., Katasonov, A.: Model and ontology-based development of smart space applications. In: Pervasive Computing and Communications Design and Deployment: Technologies, Trends, and Applications, pp. 126–148 (May 2011)
11. Korzun, D., Lomov, A., Vanag, P., Honkola, J., Balandin, S.: Generating modest high-level ontology libraries for Smart-M3. In: Proc. 4th Int'l Conf. Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2010), pp. 103–109 (October 2010)
12. D'Elia, A., Honkola, J., Manzaroli, D., Cinotti, T.S.: Access control at triple level: Specification and enforcement of a simple RDF model to support concurrent applications in smart environments. In: Balandin, S., Koucheryavy, Y., Hu, H. (eds.) NEW2AN 2011 and ruSMART 2011. LNCS, vol. 6869, pp. 63–74. Springer, Heidelberg (2011)