A Semantic Event Notification Service for Knowledge-Driven Coordination

Martin Murth and eva Kühn
Vienna University of Technology, Institute of Computer Languages
Space Based Computing Group
Argentinierstraße 8, 1040 Vienna, Austria
{mm,eva}@complang.tuwien.ac.at

ABSTRACT
The need for cooperation between an ever increasing number of distributed information clients has led to the development of a broad number of tools and theories in the field of the semantic web. As a consequence, also several middleware systems have been extended to support these semantic data formats and knowledge integration techniques. However, all these middleware systems implement semantic extensions of their original communication model, but they do not employ the concept of knowledge as an integral part of the interaction metaphor. This often necessitates writing unnatural programme code, results in redundantly transferred data and leads to inconsistent interpretation of knowledge. In this paper, we present a semantic event notification service that addresses this problem by defining an event as the change of knowledge. We describe the communication mechanisms of the system and show how they are employed for implementing knowledge-driven coordination tasks. We also provide an architecture overview of our implementation and present first performance and scalability results.

Keywords
Knowledge-driven coordination, collaboration system, event processing, semantic data processing, middleware.

1. INTRODUCTION
Providing a mechanism for sharing knowledge between information clients is a core requirement on modern middleware systems. Data that is distributed over many information clients and the conclusions that can be drawn from this data are essential for realising many important tasks:

- **Knowledge-based coordination:** workflows need to be controlled (i.e. started, interrupted, postponed, aborted, etc.) depending on knowledge about the workflow context and changes/extension of this knowledge
- **Knowledge-based decision making:** decisions have to be made based on existing knowledge about the decision domain; domain knowledge may be available explicitly (as pure data) or implicitly (inferable by algorithms, rule engines, etc.)
- **Processing of semantic data streams:** data and event streams have to be analysed for patterns that are representing particular knowledge; clients need to be notified about the (un)availability of knowledge
- **Semantic queries, knowledge extraction:** parts of the stored knowledge need to be extracted for further processing or presentation to the end user
- **Semantic correlation:** events need to be correlated via semantic dependencies, i.e. dependencies that are not explicitly visible but can be inferred from a knowledge base
- **Metadata management:** context information and data about data semantics needs to be stored and made accessible to applications and users

For consolidation, storage, and processing of knowledge from different data sources, a number of tools and concepts have been developed in the field of the semantic web. Over the last years, existing middleware systems have been extended with support for these developments, e.g. data stores provide support for semantic data types, coordination spaces have been complemented with matchmakers for semantic data, and messaging systems allow for annotating event channels with metadata (see Section 5 Discussion and Comparison with Related Work). However, all these approaches are only semantic extensions of the original interaction model, but none of them truly integrates the concept of knowledge with its own interaction metaphor. This often results in redundantly stored data and leads to reduced data quality and inconsistent interpretations of the same knowledge fragments.

In this paper, we present conceptual model and implementation of Semantic Event Notification Service (SENS), an event processing system that avoids redundant processing by introducing the concept of knowledge events, i.e. events that indicate the change of knowledge rather than the change of state or the transfer of data.

The paper is structured as follows: In Section 2 we introduce the conceptual model of the semantic event notification service SENS. Section 3 describes how SENS' interaction mechanisms can be employed for implementing knowledge-driven coordination. Section 4 presents architecture and implementation results of the SENS prototype and discusses first performance and scalability observations. In Section 5 we compare our approach to related work. Conclusions and future work are presented in Section 6.

2. SENS – SEMANTIC EVENT NOTIFICATION SERVICE
Semantic Event Notification Service (SENS) is realised as a publish/subscribe middleware and shall ease the development of semantically enabled applications following the event driven

architectural style (EDA) [25]. Generally, all interactions with the system are modelled as events. However, the essential difference to ordinary event processing systems is that SENS works with *knowledge events*, i.e. events that contain knowledge (fragments) about a certain domain. Clients can subscribe for changes or extensions of domain knowledge by registering a description of this knowledge at SENS. When a knowledge event is sent to SENS, the event's content is added to the internal knowledge base. Then SENS tries to infer additional knowledge by reasoning about the available data and adds the new data to the knowledge base. If the new knowledge is relevant for any of the registered subscribers, these are notified and provided with the new parts of the affected knowledge. This is at the same time the most important advantage of SENS over other approaches, as the middleware itself decides whether newly inserted data is relevant for the subscriber. The subscriber only tells the middleware in which kind of information it is interested in.

Example: The SENS knowledge base states that "Randy is 8." and "Tim is a parent of Randy.". Client A wants to get notified about persons with siblings and registers the according subscription at SENS. If client B sends a knowledge event stating that "Mark is child of Tim.", this would cause SENS to generate a new knowledge event containing knowledge about Randy and Mark[1].

Note that sending the same knowledge event twice would not result in another notification, since SENS detects that the subscribers have already received this knowledge. The same is true for sending knowledge that was already inferred from the knowledge base before. This means that the subscribers are really notified about new knowledge and not about the availability of particular data structures.

2.1 SENS API

In SENS, knowledge is represented in RDF. RDF [27] describes both data and metadata as directed graphs which are created by making statements about resources in the form of ⟨subject, predicate, object⟩ triples (corresponding to a directed, named edge *predicate* from node *subject* to node *object*).

When a SENS client wants to subscribe for changes of knowledge, it needs to define the parts of the knowledge it is interested in using a SPARQL CONSTRUCT statement. The SPARQL CONSTRUCT [35] query form is defined to identify a single RDF graph that matches a given graph template. The result graph is formed by substituting variables of the graph template by the query solutions found.

The SENS API is shown in Listing 1. Knowledge can be added to SENS in form of a single RDF triple or as a graph data structure defined by a set of triples (*publish*).

The subscription mechanism of SENS (*subscribe, unsubscribe*) allows a client to be notified about changes or extensions of certain parts of the stored knowledge. Whenever data is added, SENS checks whether a re-evaluation of the SPARQL statement would return additional triples (i.e. new knowledge) and transfers new results to the subscriber using a callback interface.

[1] We assume that the reasoning engine is aware of the required relations.

Listing 1. SENS API (Java)

```
public interface SENS {

    // adds a triple to the knowledge base
    void publish(Triple triple);

    // adds a set of triples to the knowledge base
    void publish(TripleSet tripleSet);

    // subscribes for all knowledge events that match the provided
    // knowledge description
    SubscriptionID subscribe(Subscriber s, String sparqlDescr);

    // removes the subscription
    void unsubscribe(Subscriber s, SubscriptionID id);

    // receives the desired knowledge; blocks if no result is found
    TripleSet receive(String sparqlDescr);

    // like receive but returns null if no result is found
    TripleSet tryReceive(String sparqlDescr);

    // registers a continuous insertion
    InsertionID registerInsertion(String sparqlDescr);

    // unregisters a continuous insertion
    void unregisterInsertion(InsertionID id);

    // removes the result graph from the knowledge base; blocks if
    // no result is found
    TripleSet remove(String sparqlDescr);

    // like remove but returns null if no result is found
    TripleSet tryRemove(String sparqlDescr);

}
```

Example: The registration of a subscription with the following SPARQL description would notify the subscriber every time new knowledge about persons with at least one child is available. Notice that the construct part of the query defines the operation's result, which is then compared to previous results of the query.

```
CONSTRUCT {              // CONSTRUCT part defines
  ?s    :name ?n;        // the result graph
        ?p    ?o.
}
WHERE {                  // WHERE part describes
  ?s    :name ?n;        // the knowledge of
        ?p    ?o;        // interest by means of
        a     :Person;   // a graph pattern
        :hasChild ?c.
}
```

The subscriber receives the new triples together with a triple containing the name of the concerned person. Implementing this example with a traditional event processing system would require to (1) define an extra event channel for modifications of data of persons with children, (2) register for events on this channel, (3) determine the concerned person, when an event is received, and (4) query a database for the required context information. With SENS, this can be implemented within one single subscription.

A client can also wait for the availability of certain knowledge (*receive*). If the requested knowledge is not available, the request is blocked until it can be answered. If the client only wants to test for the existence of certain knowledge, SENS offers a non-blocking variant of this API primitive (*tryReceive*).

Another mechanism frequently employed in event processing is continuous insertion (cf. [8]) (*registerInsertion, unregisterInsertion*). Continuous insertions are descriptions of event patterns that are evaluated each time an event occurs. If the given pattern is found, a new event is generated. This mechanism is usually applied to generate high-level events from a number of low-level events. In SENS, an insertion is also described by a SPARQL CONSTRUCT statement. When the described knowledge is available in SENS, the triples of the constructed result graph are added to the knowledge base.

Finally, we also added two primitives for the removal of semantic data (*remove, tryRemove*). These primitives have been introduced as it turned out that in some cases it is necessary to make SENS "forget" particular parts of its knowledge. This is especially useful for controlling memory and storage requirements of SENS and it simplifies the implementation of certain knowledge-centric coordination patterns (e.g. request/response, produce/consume).

3. KNOWLEDGE-DRIVEN COORDINATION

Malone et al. [26] define *coordination* as "managing dependencies between certain activities". Accordingly, we define *knowledge-driven coordination* as coordination that is driven by knowledge about the coordination scenario, e.g. knowledge about the involved entities, dependencies between entities, consequences of coordination activities, and the current state of an interaction. SENS implements several typical interaction primitives that can be employed for the coordination of multiple clients (subscriptions, blocking receive, blocking consume). While the use of such primitives has already been studied extensively in other work (e.g. [13][15][16][24][40][44]), this section describes how *ontologies* and *continuous insertions* are employed for implementing knowledge-driven coordination with SENS.

3.1 A Use Case Scenario

A governmental health organisation wants to develop a system that allows for quickly finding blood donors for people with very special forms of blood incompatibleness. The system should therefore collect knowledge about patients of general practitioners and hospitals and analyse specific relations between the blood types and other blood properties. Multiple information systems provide the required data which is then used to coordinate a number of clients. Currently, a commercial enterprise service bus is used to connect all involved information systems following the SOA approach. The new system has to be integrated with the existing infrastructure and shall use the existing communication mechanisms for interacting with the employed business process execution engine. Figure 1 shows an overview of the system infrastructure and presents a small example fragment of the patient data that will be managed by SENS.

In the example, only the most important properties (*age, blood type*) and known relationships (*isParentOf, isChildOf)* between the blood donors are shown (a textual description of the RDF graph can be found in Appendix A).

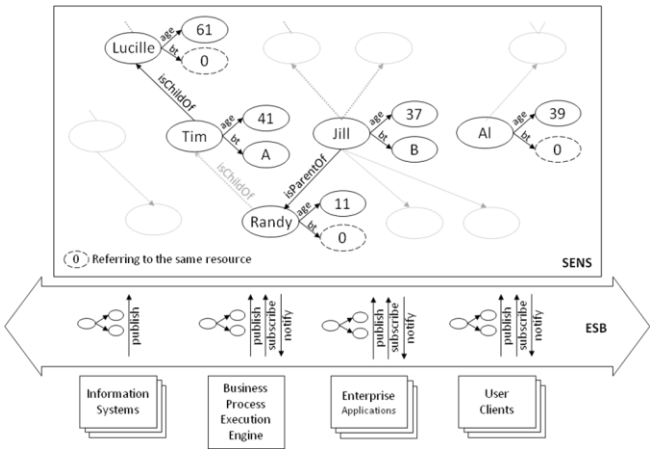

Figure 1. Use case scenario

In the following, we briefly summarize four use cases of this scenario:

- **UC1 – Data Import**: After the deployment of the new system as well as each time a new information system of a hospital or a general practitioner is integrated with the system, patient data of existing databases, knowledge management systems, or experts systems has to be imported into the SENS knowledge base.
- **UC2 – Data Analysis**: For finding appropriate blood donors, the knowledge base has to be analysed with respect to the given blood properties, health situation, relational dependencies, etc. of the donors. In emergency situations, this process must not exceed a critical time limit.
- **UC3 – Knowledge-driven Coordination**: While some clients will act as pure data providers, others need to be notified about suitable blood donors. SENS has to correlate semantic data published by different clients and notify the corresponding subscribers so that they can continue their current work unit. The notification mechanism has to be implemented using the existing IT infrastructure.
- **UC4 – Knowledge Extraction**: For the purpose of statistical analyses in medical research, the system must allow for the extraction of particular parts of the stored knowledge. Appropriate query mechanisms have to be provided.

3.2 Ontologies

The use of ontologies for driving a coordination process is the most distinguishing feature of SENS compared to existing event processing and space-based middleware. An ontology is a model that describes a set of concepts within a domain and the relations between them. It can be used to reason about data and information within that domain. SENS supports ontologies for RDF based knowledge descriptions. Whenever new data is added, the internal reasoning engine infers new knowledge and adds it to the knowledge base in the form of additional (virtual) RDF triples.

Example: SENS supports ontologies defined in OWL [29], a formal language for the specification of relations between certain classes and properties of resources. The following OWL ontology describes certain relationships between the data objects stored in SENS (OWL can be represented as RDF triples itself).

```
:Person a       owl:Class .

:Man    a       owl:Class ;
        rdfs:subClassOf :Person .

:Woman a        owl:Class ;
        rdfs:subClassOf :Person .

:isAncestorOf
        a       owl:TransitiveProperty ,
owl:ObjectProperty .

:isParentOf
        a       owl:ObjectProperty ;
        rdfs:domain :Person ;
        rdfs:range :Person ;
        rdfs:subPropertyOf :isAncestorOf ;
        owl:inverseOf :isChildOf .

:isChildOf
        a       owl:ObjectProperty ;
        rdfs:domain :Person ;
        rdfs:range :Person ;
        owl:inverseOf :isParentOf .
…
```

The ontology specifies three classes (*Person*, *Man*, *Woman*), the latter two being sub-classes of the former, and three properties. The property *isAncestorOf* is defined to be transitive; its sub-properties *isParentOf* and *isChildOf* are defined as relationships between two persons and to be the inverse of each other.

We now assume that a young person (Randy) has been injured during a car accident and requires a blood transfusion. Due to special blood incompatibleness, the person can receive blood from a blood-relative (ancestor) only. While it is not possible to express recursions with SPARQL descriptions, the OWL ontology allows for modelling such relations using a transitive property (*isAncestorOf*). After registration of the above ontology at SENS, we can use a subscription with the following simple SPARQL description to get notified each time a potential blood donor is found.

```
CONSTRUCT {
  :Randy :canReceiveBloodFrom ?p.
}
WHERE {
  ?p :isAncestorOf :Randy.
  ?p :hasBloodType :O.
}
```

Addition of the triple <:Randy, :isChildOf, :Tim> (cf. Figure 1) to the SENS knowledge base would trigger the reasoning engine inferring the following knowledge:

- ⟨:Tim, :isParentOf, :Randy⟩, as *isParentOf* is the inverse property of *isChildOf*
- ⟨:Tim, :isAncestorOf, :Randy⟩, as *isAncestorOf* is a super-property of *isParentOf*
- ⟨:Lucille, :isAncestorOf, :Randy⟩, as *isAncestorOf* is a transitive property

Consequently, the subscriber is notified about the existence of the triple ⟨:Randy, :canReceiveBloodFrom, :Lucille⟩. In this case, not the addition of the triple but the reasoning process has triggered the event. The defined relations have directly triggered a coordination step.

The use of formal, logic-based ontology languages like OWL allows for solving highly complex coordination problems for which obvious algorithmic solutions are hard to find. Ontologies should be used for modelling generally valid or at least in the context of the application permanently applicable relations and taxonomies. These are read frequently and hardly change over time. For the definition of temporary relations, SENS provides the mechanism of continuous insertion.

3.3 Continuous Insertion

The term *continuous insertion* originates from the field of event processing, where it is evaluated after each event, whether a certain (potentially complex) event pattern can be detected and whether a new high-level event indicating the occurrence of this pattern shall be generated. In SENS, we adopted this concept for the notion of knowledge. A continuous insertion describes a graph pattern to search for and a graph data structure to be added to the knowledge base, when the pattern is found or can be inferred. The description is registered at SENS and is evaluated each time data is added or removed.

Example: This time we assume that the injured person can only receive blood from a male adult also having blood type '0'. The following SPARQL description defines these requirements.

```
CONSTRUCT {
  :Randy :canReceiveBloodFrom ?p.
}
WHERE {
  ?p a :Man.
  ?p :hasBloodType :O.
  ?p :hasAge ?a.
  FILTER (?a >= 18)
}
```

Registration of a continuous insertion with this SPARQL description has the effect that SENS explicitly generates triples for all persons who can be blood donors for Randy. Addition of data about Al would cause the generation of the triple ⟨:Randy, :canReceiveBloodFrom, :Al⟩. This may in turn allow for inferring new knowledge which triggers the notification of subscribers or unblocks pending receive requests.

Consequently, also continuous insertions may directly trigger a coordination step. They provide the client with a means to define rules for special relations between the data objects using the employed description language. Since continuous insertions can be registered and unregistered dynamically, they allow a client to define these rules temporarily, which can be advantageous in many application scenarios. While this is not possible for ontologies, they provide a more powerful means for defining complex relations and concepts.

4. ARCHITECTURE AND IMPLEMENTATION

In this section we present architecture and implementation of SENS and discuss first performance and scalability results.

4.1 SENS Architecture and Implementation

SENS consists of two main components: a semantic storage and inference layer for RDF data and an event processing layer that implements the SENS API primitives (see Figure 2).

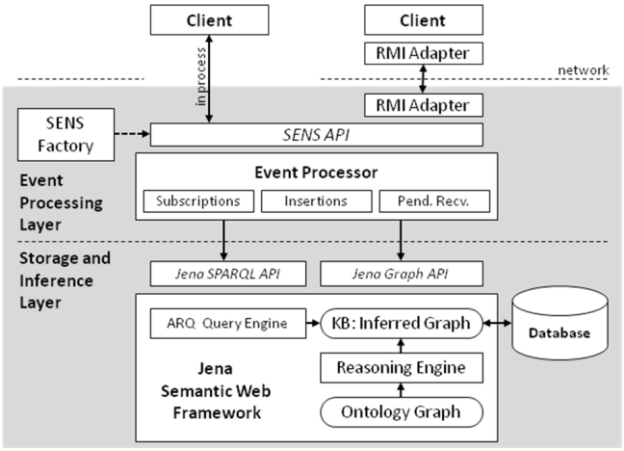

Figure 2. SENS architecture

The Jena Semantic Web Framework (Java) [20] is employed for storing and querying RDF data. When SENS is initialised, the used ontology model is loaded into the *Ontology Graph* and the Jena built-in reasoner creates an initial *Inferred Graph*. Both insertion and removal of data are performed on the inferred graph. Hence, the inferred graph contains explicitly inserted as wells as implicitly available data, i.e. data inferred by applying the provided ontology.

If the inferred graph is changed, this may trigger further rule firings of the reasoning engine. The RETE-based forward reasoning engine [11] works incrementally and only the consequences of the added or removed triples are explored. The current version of SENS supports a subset of the ontology languages supported by Jena, namely *NONE*, *RDFS*, and *OWL*.

The Jena framework offers two interfaces to access the inferred graph. The *Graph API* provides access to the contained triples through explicit references to resources and properties of the RDF model. More complex queries can be formulated using the *SPARQL API*.

The main component of the event processing layer is the *Event Processor*, which implements the *SENS API*. It synchronises concurrent access and manages subscriptions, continuous insertions, and pending receive/consume requests:

- **Subscriptions:** Whenever new data is added, the event processor checks whether the result graph of any subscription's request has changed. If so, the according subscriber is notified and provided with the newly availably result triples.
- **Continuous insertions:** At each write operation, the event processor checks whether the knowledge base matches any of the registered SPARQL descriptions for continuous insertions and adds the corresponding new triples to the knowledge base.
- **Pending receive and consume requests**: If receive or consume requests cannot be answered immediately, they are stored and re-evaluated when new data is added to the space. The result graph is provided to the client as soon as it consists of at least one triple.

The current version of SENS can be run in-memory or in persistence mode with an HSQLDB 1.8 [19] or MySQL 5.0 [31] database backend. Clients can instantiate SENS in-process or access it via an extensible adapter mechanism. Currently, we provide an RMI adapter for remote access to a SENS server installation. This RMI adapter can also be used for connecting SENS to an enterprise service bus.

4.2 Performance and Scalability

With the goal to get a first impression of performance and scalability of SENS, we implemented two test scenarios for the previously described use cases. All tests were run on a Pentium IV HT 3,2GHz, 4GB RAM, Windows Vista PC.

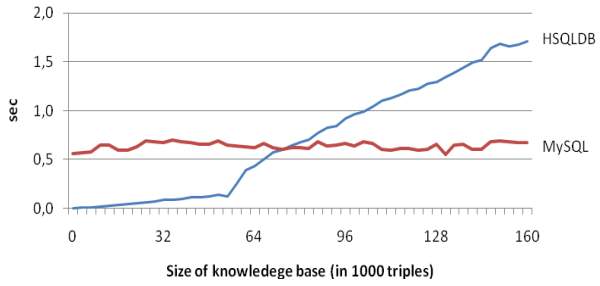

Fig. 3: Data load times for one patient record (32 triples) at different sizes of the SENS knowledge base (persistence mode)

For the evaluation of data load times (UC 1), we loaded chunks of 32 triples (one patient description) into SENS. While the in-memory mode performed well (~2.5sec/10.000 triples), both configurations with data persistence did not show satisfactory results (see Figure 3). The high load times of up to 1.7sec for one record can be ascribed to the high number of database connections that are opened by the Jena framework. While this leaves much room for optimizations, it makes it difficult to draw further conclusions about the performance of the persistence mode of SENS at the current stage of development. An interesting observation is that with HSQLDB, the insertion times increased with the size of the knowledge base. This may indicate bad scalability and could cause severe performance problems with bigger knowledge bases.

In the described use case scenario, SENS is initialised with the ontology presented in Section 3.2. Generally, we can say that the reasoning engine did not cause any significant delay in any of the performed tests (UC 2). Apart from the instantiation of the reasoning engine and the initial reasoning process, the processing overhead caused by the reasoning engine never exceeded 5% of

the total processing time. However, the time required for reasoning strongly depends on the number of rule firings that are triggered after adding a triple, which correlates with the size and complexity of the employed ontology as well as the size and internal structure of the stored knowledge. A more comprehensive evaluation of the reasoning engine is beyond the scope of this paper (the reader may refer to [12]).

For the evaluation of query processing (UC 3 & UC 4), we defined two SPARQL descriptions and registered them at SENS: the first describes one specific property of a particular patient description (1 triple); the second describes more complex relations between three different patient types (~100 triples). Again, the in-memory mode performed best (~1,1ms for 1 triple; ~3,5ms for 3 descr.), but for the simple query, HSQLDB exhibited almost the same processing times as the in-memory configuration (see Figure 4). In this test scenario, all queries showed constant processing time. While HSQLDB being significantly faster than MySQL, it is important to note that HSQLDB does not support full ACID transactions.

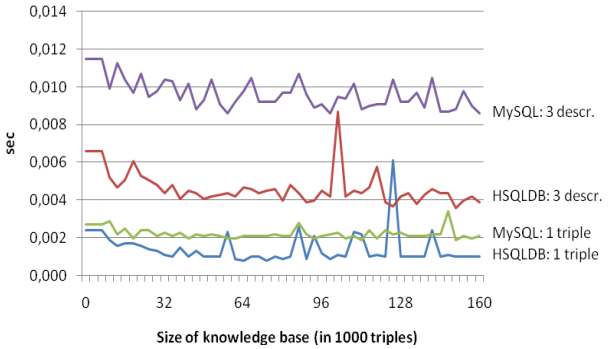

Fig. 4: Query times for read operations at different sizes of the SENS knowledge base (persistence mode)

The first tests demonstrated that SENS has the potential to coordinate clients based on larger knowledge bases. However, especially the data load mechanism still requires substantial optimisation. As a next step, we are going to investigate a number of possible performance improvements. Special indexing schemes (e.g. [6][18][28][39]) could be employed for efficient storage and retrieval of RDF triples. Furthermore, highly optimised algorithms for scalable matching of graph based data structures have been proposed for semantic publish/subscribe systems (e.g. [34][43]). Although these algorithms were developed for the comparison of rather small graphs, they could be adapted and employed for the processing of subscriptions and receive operations for an entire knowledge base. Further performance improvements could be achieved by leveraging query engines capable of result set caching and incremental query execution. Performance of write operations may be further improved by employing reasoning engines optimised for different requirements on data sizes, responsiveness, platforms, and distribution topologies (e.g. [17][38][41]; for a comparison see [12]).

More comprehensive measurements of performance and scalability for an optimised implementation of SENS as well as a comparison with alternative implementation approaches are subject to future work.

5. DISCUSSION AND COMPARISSON WITH RELATED WORK

In the past few years, several databases and frameworks for the management of semantic data have been developed (e.g. Redland [2], Sesame [3], Yars [18], Jena [20], Oracle@Spatial RDF [33]). While these systems are optimised for storing, querying, and reasoning about large amounts of semantic data, they do not offer a coordination mechanism for controlling interactions between multiple clients.

Traditional event processing systems [25] are a practical means for implementing simple coordination scenarios. However, the event models of these systems define an event as a simple data object that can only be received via a certain event channel. Consequently, the data structures of events and the hierarchy of event channels need to be defined in advance. Knowledge, in contrast, is inferred from arbitrarily structured and connected data. Techniques of complex event processing (e.g. Coral8 [7], Esper [8]) [30] allow for detecting multiple related events based on relations that do not need to be known in advance, but they still rely on the events' data structures.

RDF based publish/subscribe systems (e.g. GToPSS [34], OPS [43]) extend the event matching algorithm with semantic matching capabilities. Every time a message is published, it is verified whether the message meets certain semantically defined matching criteria. If this is the case, the message is sent to the subscriber in its original form. While this is a useful improvement of content based subscription, it is still based on the exchange of single messages. In contrast, our approach aims at collecting and distributing knowledge, i.e. (fragments of) the consolidated contents of all exchanged messages.

Semantic coordination spaces take an approach that aims at realising space-based coordination [23] with technologies from the semantic web. Based on the Linda model [13], coordination is implemented as reactions to insertion or removal of tuples that contain semantic data. sTuples [22] and Semantic Web Spaces [32], for example, allow for formulating more expressive templates than the original Linda model, but they still limit template matching to tuples. The TSC [10] prototype offers a query primitive for extraction of arbitrary parts of the entire stored knowledge. However, this primitive just passes the read request to the underlying database, which makes it difficult to implement more complex coordination patterns. TripCom [37][5] also provides access to the entire knowledge that is stored in the space. The *rd* and *in* primitives of TripCom behave like the original versions of Linda, which (in contrast to SENS) do not reliably report all occurrences of events. TripCom's notification mechanism also works differently as it only evaluates whether the inserted tuples match a given template. TripCom follows this approach, since the developed system is targeted at becoming a web scale infrastructure for the storage and retrieval of RDF data. Therefore, its interaction primitives are defined to allow for maximal scalability. In SENS, triple patterns are matched against the entire knowledge that can be inferred from the currently as well as from previously inserted tuples. Although scalability is also a key requirement for SENS, we introduced this more processing intensive matching process in order to allow for a more expressive subscription mechanism.

Furthermore, the concept of continuous insertion for extending the system with user-defined rules is not available in any implementation of a semantic coordination space.

For distribution of semantic data, there exists interesting work on optimisation of storage and retrieval of RDF [27] data. PAGE [42] and RDFCube [36] define indexing schemes that use multi-dimensional hash indices to provide efficient query processing for RDF triples. For the implementation of simple subscriptions, peer-to-peer and distributed hash table based approaches such as RDFPeers [4] and MDV [21] promise to offer query times that are logarithmic to the number of participating network nodes.

6. CONCLUSION

In this paper we presented SENS, a semantic event notification service for the implementation of knowledge-driven coordination. By allowing a client to register for changes of knowledge, typical knowledge integration problems such as redundant storage, inconsistent interpretations, and repeated processing of the same knowledge fragments are avoided.

Using SENS, the stored knowledge can directly drive a coordination process. The use of ontologies allows for describing highly complex coordination problems which can be automatically resolved and processed by SENS. Continuous insertion provides a flexible means for generating new knowledge when clients are interacting with SENS. This new knowledge can then trigger subsequent steps of a coordination process. While continuous insertions can be employed more dynamically, ontologies provide a more powerful means for defining complex relations and concepts within the knowledge domain. Both coordination mechanisms can be combined to best meet the requirements of a specific coordination problem.

Finally, the implementation of a SENS use case scenario gave a first insight into performance and scalability of the proposed system. While the read operation is efficient enough for querying large amounts of semantic data, the measurements showed that the write operation still requires optimisation.

Performance of data import is at the same time the first problem that we are going to address in future work. Furthermore, we are going to define a formal model of semantic event processing and to employ this model in the specification of the SENS API semantics.

7. ACKNOWLEDGEMETNS

This work was supported by the FP6 project TripCom (IST-4-027324-STP).

8. REFERENCES

[1] Beckett, D. Turtle - Terse RDF Triple Language, 2006. Available at: http:// www.dajobe.org/2004/01/turtle/

[2] Beckett, D. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[3] Broekstra, J., Kampman, A., and van Harmelen, F. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the International Semantic Web Conference (ISWC)*, pp. 54–68. 2002.

[4] Cai, M. and Frank, M. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer network. In *13th International Conference on World Wide Web*, 2004.

[5] D. Cerzza, E. Della Valle, D. Foxvog, R. Krummenacher, and M. Murth. Towards European Patient Summaries based on Triple Space Computing, *Proc. of 1st European Conf. on eHealth*, Fribourg, Switzerland, 12-13 October, 2006.

[6] Christophides, V., Plexousakis, D., Scholl, M., and Tourtounis, S.: On labeling schemes for the semantic web. In *Proceedings of the twelfth international conference on World Wide Web*, ACM Press, 2003.

[7] Coral8: *Coral8 Complex Event Processing Technology Overview*. Available at: http://www.coral8.com/system/files/assets/pdf/Coral8TechWP.pdf

[8] Esper. *Esper Reference Documentation Version 1.8.0*. (2007) Available at: http://esper.codehaus.org/esper-1.8.0/doc/reference/en/pdf/esper_reference.pdf

[9] Fensel, D.: Triple-space computing: Semantic Web Services based on persistent publication of information, In *Proc. of IFIP International Conf. on Intelligence in Communication Systems*, Bangkok, Thailand (2004)

[10] Fensel, D., Krummenacher, R., Shafiq, O., Kühn, e., Riemer, J., Ding, Y., and Draxler, B. TSC - Triple Space Computing, In Special issue on ICT research in Austria, Journal of Electronics & Information Technology (e&i Elektrotechnik & Informationstechnik), January-February, 2007.

[11] Forgy, C.L. RETE: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence, 1982.

[12] Gardiner, T., Tsarkov, D., and Horrocks, I. Framework for an Automated Comparison of Description Logic Reasoners. In *Proceedings of 5th International Semantic Web Conference*, Athens, GA, USA, November 5-9, 2006.

[13] Gelernter, D. Generative Communication in Linda, In *ACM Transactions in Programming Languages and Systems (TOPLAS)*, 7(1), (1985): 80-112.

[14] Gelernter, D. and Carriero, N. 1992. Coordination languages and their significance. *Commun. ACM* 35, 2, 1992, 97-107.

[15] Greifeneder, M. *The Request/Answer Coordination Design Pattern*, Diploma Thesis, Institute of Computer Languages, Vienna University of Technology (2001)

[16] Grossberger, G. *The Publish/Subscribe Coordination Design Pattern*, Diploma Thesis, Institute of Computer Languages, Vienna University of Technology (2000)

[17] Haarslev, V. and Möller, R. RACER system description. *In Proceedings of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, pages 701–705. Springer, 2001.

[18] Harth, A. and Decker, S. Optimized index structures for querying rdf from the web. In *Proceedings of the 3rd Latin American Web Congress*. IEEE Press, 2005.

[19] HSQLDB. HSQLDB 1.8.0 – 100% Java Database, http://hsqldb.org/

[20] Jena. Jena – A Semantic Web Framework for Java. Available at: http://jena.sourceforge.net/, last accessed: Sept. 2007.

[21] Keidl, M., Kreutz, A., Kemper, A. and Kossmann, D. A publish and subscribe architecture for distributed metadata management. In *Proceedings of 18th International Conference on Data Engineering*, San Jose, CA, USA, 2002.

[22] Khushraj, D., Lassila, O. and Finin, T.W. sTuples: Semantic Tuple Spaces. In *1st Ann. Int'l Conf. on Mobile and Ubiquitous Systems*, August 2004.

[23] Kühn, e. Fault-Tolerance for Communicating Multidatabase Transactions. In *Proc. of the 27th Hawaii Int. Conf. on System Sciences (HICSS)*, ACM, IEEE, 1994.

[24] Lederer, A. *The Database Replication Coordination Design Pattern*, Diploma Thesis, Institute of Computer Languages, Vienna University of Technology. (2004)

[25] Luckham, D. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, 2002.

[26] Malone, T.W. and Crowston, K. The Interdisciplinary Study of Coordination, *ACM Computing Surveys 26*, 1994.

[27] Manola, F. and Miller, E. *RDF Primer W3C-Recommend*., 2004. Available at: http://www.w3.org/TR/rdf-primer/

[28] Matono, A. Amagasa, T., Yoshikawa, M., and Uemura, S. An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays. *Transactions of Information Processing Society of Japan*, Vol. 45, No. 4, pp. 50-62, 2004.

[29] McGuinness, D.L. and van Harmelen, F. OWL Web Ontology Language, W3C Recommendation, 2004. Available at: http://www.w3.org/TR/owl-features/

[30] Murth, M. and Kühn, e. Complex event processing with a semantic tuplespace approach. Technical report, E185/1, Vienna University of Technology, 2007.

[31] MySQL. MySQL 5.0. http://www.mysql.com

[32] Nixon, L.J.B., Paslaru Bontas Simperl, E., Antonenko, O., and Tolksdorf, R. Towards Semantic Tuplespace Computing: The Semantic Web Spaces System. In *22nd Ann. ACM Symposium on Applied Computing*, March 2007.

[33] Oracle® Spatial. Resource Description Framework (RDF) 10g Release 2 (10.2). Available at: http://download.oracle.com/docs/cd/B19306_01/appdev.102/b19307.pdf, last accessed: Jan. 2008

[34] Petrovic, M., Liu, H., and Jacobsen, H. G-ToPSS: fast filtering of graph-based metadata. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)*, ACM Press, New York, NY, USA, 2005.

[35] Prud'hommeaux, E., and Seaborne, *A. SPARQL Query Language for RDF*. W3C Working Draft, 2007. Available at: http://www.w3.org/TR/rdf-sparql-query/

[36] RDFCube. Database grid grid – RDFCube. Available at: http://projects.gtrc.aist.go.jp/dbwiki/pukiwiki.php?RDFCub, last accessed: Sept. 2007.

[37] Simperl, E., Krummenacher, R., and Nixon, L. A Coordination Model for Triplespace Computing. In *Proc. of the: 9th Int'l Conf. on Coordination Models and Languages*, 2007.

[38] Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A. and Katz, Y. Pellet: A practical OWL-DL reasoner. *In Proceedings of Web Semantics: Science, Services and Agents on the World Wide Web*, Vol. 5, Issue 2, 2007.

[39] Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of 13th International World Wide Web Conference, New York*, pages 631–639, 2004.

[40] Tolksdorf, R.: Coordination patterns of mobile information agents. In *Cooperative Information Agents II, proceedings of the second international workshop CIA'98*, Paris, France (1998)

[41] Tsarkov, D. and Horrocks, I. FaCT++ description logic reasoner: System description. In *Proceedings of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of Lecture Notes in Artificial Intelligence, pages 292–297, 2006.

[42] Della Valle, E., Turati, A., and Ghioni, A. PAGE: A distributed infrastructure for fostering RDF-based interoperability. In *DAIS*, pages 347–353, 2006.

[43] Wang, J., Jin, B., and Li, J. An Ontology-Based Publish/Subscribe System. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware,* Canada, 2004.

[44] Wernhart, H., Kühn, e., Trausmuth, G.: The Replicator Coordination Design Pattern, In *Journal on Future Generation Computer Systems*, Elsevier. (1999)

Appendix A. Content of SENS Knowledge Base[2]

```
:Tim      a       :Man ;
          :hasAge "41"^^xsd:int ;
          :hasBloodType :A .

:Al       a       :Man ;
          :hasAge "39"^^xsd:int ;
          :hasBloodType :0 .

:Jill     a       :Woman ;
          :hasAge "37"^^xsd:int ;
          :hasBloodType :B ;
          :isParentOf :Randy .

:Randy    a       :Man ;
          :hasAge "11"^^xsd:int ;
          :hasBloodType :O ;

:Lucille  a       :Woman ;
          :hasAge "61"^^xsd:int ;
          :hasBloodType :O ;
          :isParentOf :Tim .

...
```

[2] in Turtle RDF format [1]