sTuples: Semantic Tuple Spaces*

Deepali Khushraj
Nokia Research Center
5, Wayside Road
Burlington, MA 01803
deepali.khushraj@nokia.com

Ora Lassila
Nokia Research Center
5, Wayside Road
Burlington, MA 01803
ora.lassila@nokia.com

Tim Finin
University of Maryland
1000, Hilltop Circle
Baltimore, MD 21250
finin@cs.umbc.edu

Abstract

Tuple Spaces offer a coordination infrastructure for communication between autonomous entities by providing a logically shared memory along with data persistence, transactional security as well as temporal and spatial decoupling– properties that make it desirable in distributed systems for e-commerce and pervasive computing applications. In most Tuple Space implementations, tuples are retrieved by employing type-value matching of ordered tuples, object-based polymorphic matching, or XML-style pattern matching. In a heterogeneous environment, this can pose several limitations. This paper discusses the architecture and implementation of a prototype semantic infrastructure, which uses Semantic Web technologies to represent and retrieve tuples from a Tuple Space. Semantic Tuple Spaces (sTuples) overcomes limitations of the JavaSpaces Tuple Space implementation, by making use of a web ontology language and RACER, a description-logic reasoning engine. The sTuples infrastructure extends and integrates with Vigil, a secure framework for communication and access of intelligent services in a pervasive environment. Specialized agents, such as the tuple-recommender agent, task-execution agent and publish-subscribe agent, which have a better understanding of the environment, reside on the Tuple Space and play an important role in providing user-centric reasoning.

1. Introduction

The original vision of Pervasive Computing (PerCom) was formulated by Mark Weiser[28] more than a decade ago. Since then, there have been significant developments in wireless technology and device capabilities; however, there are significant research challenges that are yet to be completely addressed. A few important research challenges in such an environment include auto-configuration of entities, context-sensitive behavior and the creation of unobtrusive services. We believe that the Semantic Web[2] can help realize these challenges by providing *semantic interoperability*. Semantic interoperability here refers to a common understanding of the semantics associated with both data and services in a pervasive environment.

To enable semantic interoperability, we need *semantic infrastructures* and *semantic gadgets*[18] that would allow heterogeneous entities to work together in a pervasive environment. It is important that semantic gadgets and semantic infrastructures both have built-in capabilities to process and infer meaningful information from semantic meta data associated with data and services. In addition to this, semantic infrastructures should also cater to factors such as network QoS, device limitations, security, transactional integrity, synchronization, context-based factors etc. The *Semantic Tuple Spaces* (sTuples) work aims at providing one such semantic infrastructure by extending the Tuple Space model.

The Tuple Space model was initially conceived for parallel computing in David Gelernter's Linda system[6]. In AI parlance, a Tuple Space is similar to a blackboard system. It can be viewed as a logically shared memory, where producers add tuples to the space, while consumers read or extract tuples from the space using a search template. The look up for tuples on the space is based on content; hence it can also be viewed as an associative memory. Although Tuple Spaces were not originally designed to be used as mobile middleware, they provide several features, which make them an interesting model for the same[20]. One major shortcoming of Tuple Space implementations that impedes it from being a suitable infrastructure for PerCom is the way tuples are represented and retrieved from space. In most implementations, tuples are retrieved by employing type-value matching of ordered tuples (e.g. Linda), object-based polymorphic matching (e.g. JavaSpaces), or XML style pat-

* This work was partially supported by DARPA contract

tern matching (e.g. Ruple). The shortcomings in these approaches are discussed in subsequent sections.

In this paper, we present the architecture and implementation of the sTuples infrastructure. The sTuples infrastructure overcomes limitations of the JavaSpaces Tuple Space implementation, by using the web ontology language, DAML+OIL[12] and RACER[9], a description-logic reasoning engine. The sTuples infrastructure also extends and integrates with Vigil[16], a secure framework for communication and access of intelligent services in a pervasive environment.

The key focus of our work is to:

Enhance Tuple Representation and Searches: Representation is enhanced by introducing the concept of a Semantic Tuple, and extending it to represent data and service-descriptions in a pervasive environment. Tuple template matching is enhanced by using a semantic match on top of object-based matching.

Provide User Centric Reasoning: This is achieved by using agents on the space that provide unobtrusive data and services, that execute atomic or composite tasks on behalf of the user; and that dynamically deliver/notify data or events to the user.

Serve as a Semantic Infrastructure in PerCom: This is achieved by integrating with Vigil, employing constructs of the Tuple Space and by incorporating semantics in Tuple Spaces.

The rest of the paper is organized as follows: Section 2.1 provides a discussion on Tuple Spaces, existing implementations and its attractiveness and shortcomings as a middleware. Section 2.2 provides a description of the Vigil framework. Section 2.3 briefly discusses the role of AI in PerCom and the Semantic Web. Section 3 motivates the use of sTuples. Section 4 presents how tuples are represented and retrieved in sTuples and provides an example of its use. Section 5 presents enhancements made to the Vigil framework. Section 6 discusses the purpose and implementation details of specialized agents. Section 7 provides implementation insights, points out possible future work and concludes the paper.

In this paper, we do not take into account issues related to scalability and security of Tuple Spaces. We also do not touch upon issues concerning distributed Tuple Space implementations. These issues have been discussed in [23] and [10].

2. Background

2.1. Tuple Spaces

Why Tuple Spaces? The Tuple Space paradigm renders itself as a suitable middleware because it provides properties like: data persistence, temporal & spatial decoupling, transactional security, synchronization constructs and associative memory lookup. Temporal decoupling is useful because entities involved in communication do not have to exist at the same time. The persistent nature of tuples enables the consumer to utilize data, regardless of the connection state of the producer. Spatial decoupling provides for group interactions and implicit fault tolerance [24]. Tuple spaces provide transactional integrity [22] [29], which is an important requirement for m-commerce applications. Recent implementations of Tuple Spaces allow entities to exchange executable code; this facilitates the creation of mobile agents. There are several implementations of lookup services like Jini's Reggie service[27]. We believe that the functionality of a lookup service augments rather than replaces the functionality of a Tuple Space; thus, they should be used together.

Existing Tuple Spaces and their Limitations: Several implementations of Tuple Spaces exist, which essentially differ in the degree of extensions that they provide. A tuple in Linda [6] is defined as an ordered set of values (or fields). Tuples can contain actual fields (the value itself) and formal fields (a wild card). In order to match a tuple in Linda, the template and tuple should have the same arity and ordering of fields. The actual matching is done by comparing field values (actuals or formals) and field types of the template and candidate tuples. TSpaces, JavaSpaces and GigaSpaces are all object-oriented Tuple Spaces. In JavaSpaces and GigaSpaces tuples have data as well as executable code, and tuples, in addition to the fields, are typed. The tuples are stored as serialized objects on the space to allow exchange of executable code. These systems support polymorphic type matching, where the returned result could be a subtype of the tuple template. The supported operations include *read*, *write*, *take* and *notify*. Features such as transactional security and support for multiple spaces are also provided[22]. IBM's TSpaces system is a combination of Tuple Space and relational database features. It supports simple types and objects as fields in a tuple as opposed to using only serialized objects[29]. Ruple and other XML Space implementations use a document-centric approach where XML documents are stored on the space and retrieved using XML query syntax (XQL)[26]. Lime extends the Linda model to support mobility in both wired and ad hoc networks[25]. MARS introduces reactivity in Tuple Spaces for mobile agent coordination [3].

In all these implementations, we see a general transition from type-value matching to OO based polymorphic type matching to XML type querying. There are shortcomings in all of these approaches. One, it provides only syntactic interoperability– XML representations can cater to syntactic interoperability; however, they cannot cater to semantic interoperability because the same XML tag can be interpreted differently across systems E.g. A field tag could re-

fer to magnetic field or a branch of knowledge. Two, existing implementations do not have representations that are expressive enough and that can be reasoned by machines. E.g. a radio service cannot advertise itself as service that does not play music that belongs to genre hard rock. Three, inexact matching– the *read* and *take* operations in existing implementations do not account for subsumption based or partial matches. E.g. a query for a monochrome printer on the space will not consider a color printer as a valid match. Four, need for common ontology– a common ontology to refer to common terms is required; however, object-based interfaces and other data-structures cannot be adapted as a standard to share ontologies, as they do not meet basic requirements of an ontology language[11].

2.2. Vigil

In sTuples, we integrate Tuple Spaces with the Vigil[16] framework to create clients and services. Vigil realizes the Smart Home scenario, in which mobile users can access devices like printers, lights etc. over low-bandwidth or short-range wireless networks. In our system, Vigil acts as a communication gateway by abstracting and translating communication protocols such as infrared and Bluetooth[5]. The core component of Vigil is the Service Manager(SM) component, which acts as a mediator between services and users in a particular Smart Home. In our implementation, the SM integrates with sTuples. SMs are arranged in a hierarchy, and form the core of the Vigil system. The SM provides the user with a list of available services and also acts as a broker to subscribe, unsubscribe or forward commands from users to the services. The main idea behind the hierarchical configuration of SMs is the relegation of services on the basis of domains such as buildings, floors, and particular rooms [14]. To share services across SMs, messages are routed to foreign SMs by following the tree hierarchy. Vigil also provides distributed trust, access control and authentication. Hardware and software services are not distinguished so that a mobile user can access them in the same way. The information flow in the system takes place using Centaurus Capability Markup Language (CCML)[15], an XML-based language for data exchange.

2.3. AI and Semantic Web

There are several aspects of AI that help enable semantic interoperability as mentioned in [17]. Aspects of AI that help achieve this include: knowledge representation and reasoning through the Semantic Web, automated planning to enable service composition, machine learning to bootstrap existing knowledge on the web and the use of software agents that act on behalf of the user and exhibit reactivity, pro-activity, autonomy and sociability[13] .

The vision of the Semantic Web[2] expands on the vision of the WWW by associating accessible formal semantics with content and services. An important characteristic of the Semantic Web is that it can address "anything", any object – virtual or *physical* – that has a URI. This allows us to overlay the Semantic Web on a Pervasive Computing environment, making it possible to represent and interlink devices, their capabilities, and the functionality they offer.

Motivated by the need to automate functions on the Web, the introduction of semantics offers greater interoperability of information systems: via the use of shared *ontologies*, semantics enable automated software (such as autonomous agents) to reason about content and services, and produce intelligent responses even to unforeseen situations. Ontologies ("specifications of conceptualization" [7]) are expressed using languages such as RDF [19] or its more expressive Description Logic (DL) -based extension DAML+OIL [12]. sTuples currently uses the DAML+OIL ontology language[1].

In DL parlance, the term TBox refers to terminological axioms used to define concepts in a domain and ABox refers to assertional axioms about individuals. RACER implements an optimized tableau calculi algorithm for DL to enable reasoning over multiple TBoxes and ABoxes. The use of a DL reasoner – in our case RACER [9] – offers attractive computational characteristics over other forms of logic and over other types of reasoners.

3. Motivation

Consider the scenario where a user is attending a conference.

Data Tuple: A registered user at a conference should be able to accesses data tuples on the space that has information about the proceedings and schedule for the conference. By making use of semantics attached to the data tuple, the user's personal agent can then synchronize the conference schedule with his trip's schedule and research interests to help create a personalized conference agenda.

Synchronous Access: The speaker should be able to write a data tuple that contains his PowerPoint presentation on the space, and other registered users should be able to access the presentation. Users should also be able to add annotations to slides, to share their opinion about the presentation with others. The synchronization constructs of the space help multiple users access the tuple.

Access Rights and Service Tuple: By using access rights provided by Vigil a registered user gets rights to access data and services in the conference space based on his registration type. For example, the user can enroll for academic, student or industrial track registration. Access rights

1 Eventually we plan to migrate to OWL [21]

are also governed by the user's role in the conference. For example, invited speakers and authors of accepted papers should be able to get a handle to service tuples on the space that would allow them to control the projected presentation or dim lights or control the speaker volume in the room.

Subscription to Tuples: Conference volunteers can access the room reservation service to reschedule or handle last minute changes to the schedule. If the conference schedule changes, then a user who has subscribed to the conference schedule tuple through the publish-subscribe agent should be notified of changes.

Semantic Search: Registered users should also be able to browse for presentations, or schedule of events on the Tuple Space based on the research track that is of interest to them. For, example a user should be able to search for all talks on the space that are related to the work on both Semantic Web and pervasive computing within universities in the US.

Task-Execution: Users should have the ability to schedule an atomic task or a set of associated tasks that access data and service tuples in the conference room E.g. by using the task-execution agent a speaker can get a control of the PowerPoint service and have the lights switched off when it is time for him to present the talk

Unobtrusive behavior: Users should only be presented with tuples that are of interest to them. E.g. advertisement tuples that represent hotel discounts should not be presented by the recommender-agent to a user who lives locally.

In order to implement such a scenario we need an intelligent middleware infrastructure that has built-in semantic capabilities. Tuples in such a space should be expressive enough, and should support both syntactic and semantic interoperability.

4. sTuples: Representation and Retrieval

The sTuples system extends Outrigger[22], Sun's implementation of the JavaSpaces service specification, to support intelligent matching of tuples.

A *Semantic Tuple* in sTuples extends JavaSpace's object-based tuple and acts a role marker in the system. For an object tuple to become a valid semantic tuple, it must contain an object field of type DAML+OIL Individual. This object field contains assertional axioms(ABox) about the service instance or data instance that is being shared. Alternatively, the object field can provide a URL from where assertional axioms about the shared data or service can be loaded. From an implementation point of view, a generic SemanticTuple interface that extends the JavaSpace's Entry[22] interface is introduced. System designers can choose to extend/implement this interface to create application specific tuples.

The *Semantic Tuple Manager* and the *Semantic Tuple Matcher* are two primary components that are incorporated into Outrigger. There are two extensions to the Semantic Tuple: Service Tuple and Data Tuple. A service tuple is used to advertise a service, whereas a data tuple is used to share data/information provided by a service/agent. Each of these tuple types have specific object fields, which are discussed in the following section. Figure 1 gives an overview of the components on the space and the interaction between entities.

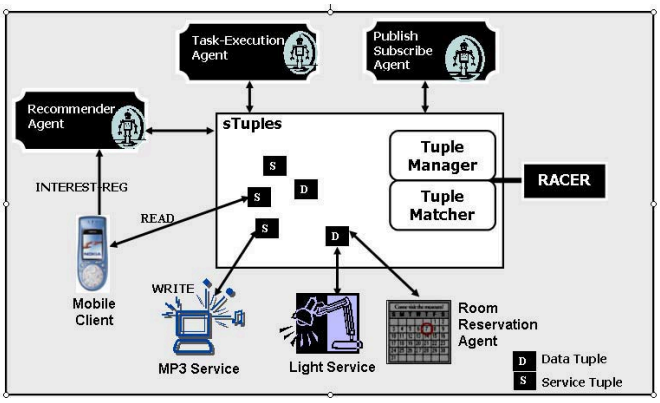

Figure 1. Semantic Space with Specialized Agents

Semantic Tuple Manager: This component manages the addition, removal and state changes of tuples. Every time a *write* operation is performed all ABoxes associated with the tuple are asserted into the knowledge base. There are two modes of operation: one in which the axioms associated with newly encountered resources and imports are asserted into the knowledge base, and the other in which only the specified axioms are asserted. Before new facts and rules are committed into the knowledge base, the tuples are validated and also a check on the consistency of the knowledge base is performed. If an inconsistency (using Racer macros abox-consistent? and tbox-coherent?) is detected by the reasoner, the description is retracted from the knowledge base and the *write* operation fails. Every time a remove operation is performed the facts are removed from the Tuple Space.

Semantic Tuple Matcher: This component implements the algorithm for matching templates and interfaces with Racer to draw additional inferences, which is achieved by classifying the TBox [9] and realizing the ABox. A tuple can be retrieved from space by performing a *read* or *take* operation. In order to invoke these operations, a semantic template that best matches the consumer's requirements is passed as an input. A semantic tuple template is a special

case of a semantic tuple where assertional axioms are based on terminological axioms provided by the TupleTemplate ontology. A snapshot of the template ontology is given in figure 2.

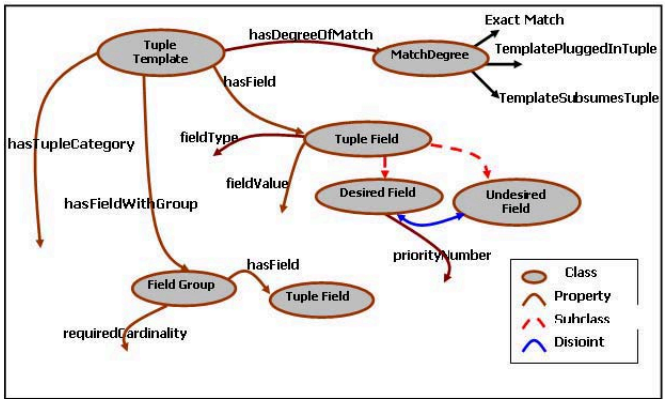

Figure 2. Semantic Tuple Template Ontology

The TupleTemplate class has several properties. The *hasField* property links a *TupleTemplate* with a *TupleField*. The *TupleField* class is an abstract class with two subclasses: *DesiredField* and *UndesiredField*, where both classes are disjoint. The *TupleField* class has two properties *fieldType* and *fieldValue*. The range of the *fieldType* property is a Property– this property makes it similar to the property *daml:onProperty* used to express restrictions on properties using *daml:Restriction*. The *fieldValue* property can point to any Resource. It also has a property called *priorityNumber*, which indicates how important a particular field is with respect to other fields. The tuple template has the *hasDegreeOfMatch* property, using which the user can specify the type of matches that are acceptable, which could be one of *ExactMatch*, *PluggedInMatch* or *SubsumptionMatch*. The tuple template's *hasTupleCategory* property refers to the possible classes that the tuple can be an individual of. The *hasFieldGroup* property is used to map a tuple template to a *FieldGroup*, which essentially is a collection of TupleFields with a *requiredCardinality* restriction.

Template matching is done by posing several queries to the reasoning engine. The matching algorithm works as follows: The matcher first validates the incoming tuple template instance.It then searches for candidate tuples. A semantic tuple becomes a candidate result, only if the *hasTupleCategory* field is satisfied. The TBox is classified and the Abox is realized, before the reasoner gets queried– this is done so that all hidden classes that an individual belongs to become visible. It then iterates over every *TupleField* in the tuple template to select matches from the list of candidate semantic tuples. Since the *fieldValue's* range is any resource, it could be of type instance, class or qualified value. Based on the type of the *fieldValue* property, and the acceptable degree of match, different cases are handled. The property name that the *fieldType* refers to, is used as an argument for queries about related-individuals and also to retrieve individual-role-fillers [8].

The following steps are performed for every TupleField of the tuple template:

1. The first step is to find an exact match, which occurs when a tuple and a template are equivalent. In our prototype, a template is considered equivalent to a tuple if all the *TupleField* properties (including the *DesiredField* and *UndesiredField* properties) specified by the template "exactly match" the description of a tuple.
2. Subsumption Matches– if there are no exact matches and if the preferred degree of match indicates subsumption as valid then queries are posed to find subsumption based relations.
3. Plugged-In Matches– if there are no exact matches and if the preferred degree of match indicates plugged-in as valid, queries are posed to extract plugged-in tuples.
4. If none of the above worked then the match has failed and no tuple is returned.

At each step a weight is assigned to every tuple that gets selected, based on its degree of match. The weights given for each of these matches is configurable. If an undesired field is present in the tuple, then there is a clash of interest due to which the tuple is assigned a negative weight. For *FieldGroups* the procedure remains the same except that the required cardinality should be satisfied in the group. After processing all the TupleFields in the template, the tuple with the highest weight gets selected.

Due to ontology modeling errors, the knowledge base can become inconsistent. The evaluation functions in RACER are used to detect such errors. The ABox querying mechanism enables us to detect the consistency of an ABox in context of a TBox. Other ABox and TBox retrieval queries are used to derive most specific instantiators of an individual, to query for related individuals, to determine concept/role descendants and ancestors, to ascertain equivalent concepts etc.

There are two inference modes that Racer supports to classify and realize the knowledge base. In the eager mode answering queries requires only a knowledge base lookup. The lazy mode avoids computations that are unrelated to a query. When a *write* operation is performed on the space, only a knowledge base coherency check is required. Since this can be done even when the knowledge base is not classified, we operate the reasoner in lazy mode. However, based on the system load, the knowledge base is classified at regular intervals– this is done to reduce the time required to perform operations such as *read* and *take*.

A simplified view of the modifications made to the Outrigger matching scheme is discussed below. All operations on the space use the *Entry* class to either store objects on the space or, as a template, to search for objects from the space. The *EntryRep* class, which represents the *Entry* object that gets marshalled from the client to the space, was modified to keep track of indexes corresponding to fields that contain semantic information. The space maintains a *TypeTree* object that keeps track of all the entry types and subtypes that the space has encountered thus far. It also maintains an *EntryHolderSet* for every entry type in the *TypeTree*. In the object-based match, an entry holder set is first selected from the tree based on type of the incoming template. A hash code is then computed for every entry in the holder set, and if the hash value of the template and the entry matches then the match is considered valid. The new matching scheme works as follows: If the template does not have any semantic content then we can do the regular object-based match, else we go further and try to find a semantic match for the template. Based on the *EntryRep* index the DAML content is extracted and the data is sent to the *Semantic Tuple Matcher* component. If there are any matches, then the *Semantic Tuple Matcher* returns a vector of all tuples that match the template description. The tuples in the vector are ordered in descending degree of match. After the semantic matches are retrieved, an object-based match is performed on the selected tuples to ensure that a tuple where both the semantic match and object-based match succeed gets selected. The first tuple in the vector that passes the object-based match gets selected and is passed back to the client.

4.1. An Example

To advertise service tuples on the space we created several service ontologies like Light Service, Printer Service, Room Reservation Service, Music Service etc. All our service related ontologies are sub-classed from the *TupleService* class.

A snapshot of the Light Service instance in the LAIT Lab at UMBC is shown in Figure 3. The LAIT Lab Light Service is an instance of a LightService that provides light from a source that is located on a table, which is used for reading and whose energy source in electric. The light service also happens to be a secondary light source in the room.

Alice enters the room and wants to get control of a light service so that she can read a book in the room. She searches for tuples on the space using the template as shown in Figure 4:

The template indicates that she is looking for an electric reading light that is located on a table or on the floor. It also indicates that the user wants control of only a non-primary light source in the room.

```
<LightService rdf:about="#LAITLabLightService">
 <serviceId>AllLaitLights</serviceId>
  <hasLightSource>
   <FluorescentLight>
    <hasLocation rdf:resource="#Table"/>
    <usedFor rdf:resource="#Reading"/>
  </FluorescentLight>
 </hasLightSource>
 <hasControlType
      rdf:resource="#SecondaryControlOfRoom"/>
</LightService>
```

Figure 3. Light Service Instance

sTuples searches for all tuples on the space that can provide the best match for Alice's query. Using the classification support provided by the reasoner, the space figures out that *LAITLabLightService* provides a light source that belongs to the class *FluorescentReadingLight*. From, the service ontology it also figures out that a *FluorescentReadingLight* had energy source *Electric* and therefore it also belongs to the class *ElectricReadingLight*. Since the *LAITLabLightService's* location, category and control type all match with the template the service tuple gets selected as a valid match.

5. sTuples with Vigil

The Semantic Tuple Space can be viewed as an extension layer to the Service Manager(SM) component in the original Vigil framework discussed in Section 2.2. The Service Manager still acts as a mediator between the services and mobile users; however, it now uses constructs of the Tuple Space to incorporate features such as data persistence, leasing, transactional integrity, spatial decoupling etc. The persistent nature of tuples on the space, allows the data and its state to be preserved even when the system goes down. The leasing mechanism on the space is used to handle unwanted and outdated tuples. Transactional integrity is preserved by using constructs of the Tuple Space (this feature is not used by current implementations of Vigil Clients/Services). Producers and Consumers of data tuples do not have to be present in the vicinity at the same time to share information– this kind of spatial decoupling is very useful in a pervasive environment where network QoS can keep changing and where the communicating entities are constantly moving in and out of a given space.

The following enhancements were made to the Vigil framework:

1. Service Registration: A Vigil service can now advertise itself by registering a service tuple with the SM. A service tuple instance contains the service id, the DAML+OIL instance describing the service, a free-text message describing service functionality, a service

```
<tt:TupleTemplate rdf:about="#ReadingPreference">
  <tt:hasTupleCategory rdf:resource="&so;LightService"/>
  <tt:hasFieldWithGroup>
    <tt:FieldGroup>
      <tt:RequiredCardinality>
        <xsd:integer>
          <rdf:value>1</rdf:value>
        </xsd:integer>
      </tt:RequiredCardinality>
      <tt:hasField>
        <tt:DesiredField>
          <tt:fieldType rdf:resource="&so;hasLightSource"/>
          <tt:fieldValue>
            <so:LightSource>
              <so:hasLocation rdf:resource="&so;Table"/>
            </so:LightSource>
          </tt:fieldValue>
        </tt:DesiredField>
      </tt:hasField>
      <tt:hasField>
        <tt:DesiredField>
          <tt:fieldType rdf:resource="&so;hasLightSource"/>
          <tt:fieldValue>
            <so:LightSource>
              <so:hasLocation rdf:resource="&so;Floor"/>
            </so:LightSource>
          </tt:fieldValue>
        </tt:DesiredField>
        <tt:hasField>
    </tt:FieldGroup>
  </tt:hasFieldWithGroup>
  <tt:hasField>
    <tt:DesiredField>
      <tt:fieldType rdf:resource="&so;hasLightSource"/>
      <tt:fieldValue rdf:resource="&so;ElectricReadingLight"/>
    </tt:DesiredField>
  </tt:hasField>
  <tt:hasField>
    <tt:UndesiredField>
      <tt:fieldType rdf:resource="&so;hasControlType"/>
      <tt:fieldValue
          rdf:resource="&so;PrimaryControlOfRoom"/>
    </tt:UndesiredField>
  </tt:hasField>
  <tt:hasDegreeOfMatch
      rdf:resource="&tt;TupleSubsumesTemplate"/>
</tt:TupleTemplate>
```

Figure 4. Light Service Search Template

icon, a limit on the number of sharable instances of the service, an associated lease and an indication of whether the service is location dependent or not. The CCML service registration message is extended to include content of the service tuple. Upon receiving a registration request, the SM registers a service tuple on the space by forwarding the CCML message to the Tuple Manager component.

2. Object Registration: The CCML object registration message is introduced to publish data tuples on the space. An object tuple contains a unique id to identify the producer of the tuple, a DAML+OIL instance of the sharable data and a list of users who are subscribed to this object. The subscribed user's list is required to handle user movement across multiple spaces.

3. Leasing: The leasing mechanism on the space is used to maintain semantic tuples on the space. A timer thread is associated with Vigil's Service Client to renew objects and service registrations that have expired.

4. CCML Translator: The CCML translator utility module creates semantic tuples from CCML messages for service and object registration. The specialized agents make use of this module to understand incoming CCML messages for interest registration, object subscription, task execution etc.

5. Specialized Agents: Communication between the specialized agents and the service clients take place through the SM in the current implementation. New CCML message types such as *Interests Registration Request/Response/Update*, *Published Object Subscription Request/Response* and *Task Execution Request/Response* etc. are introduced in the system. The CCML Parser is updated to understand these new messages.

6. Inter-space Communication: The service managers in the Vigil system are arranged in a tree-like hierarchy. Every Service Manager uses its own Semantic Tuple Space and specialized agents. The specialized agents allow clients and services to do *read* operations on data and services across spaces and to do a *take* operation across spaces, when appropriate delegation and access rights are available.

6. sTuples: Specialized Agents

Specialized agents reside on the Semantic Tuple Space to off-load the user and to incorporate user semantics in delivering data and services to the user. Our prototype implements three specialized agents as shown in figure 1. These agents make use of service tuples and data tuples. Several concepts from ontologies like DAML Time (http://www.kestrel.edu/DAML/2000/12/TIME.daml) and Cyc (http://opencyc.sourceforge.net/daml/cyc.daml) were used to create ontologies in sTuples. For the sake of conciseness, the ontology modeling details are excluded.

Tuple Recommender Agent: In the original Vigil framework, the client's service list was created based on access rights only; however, in a pervasive environment since there can be a plethora of data and service tuples, customized recommendations is important. This agent pushes unwanted data and services into the periphery of user's attention and presents only services or data tuples that are of interest to the user.

There are several aspects of context: location, temporal information, user preferences, environmental factors (like light, temperature etc.), proximity of resources, resource availability, schedules etc.[4]. Our interest registration ontology takes into account only factors related to temporal information(like time and date), user preferences and location.

The client registers interests with the recommender agent by sending a CCML message of the type *Interests-Registration-Request* to the Service Manager. The Service Manager forwards this request to the recommender gent after verifying the validity of the CCML message using the digital signature attached to the message. Upon receiving the request, the recommender agent checks to see if the message is from a valid Service Manager, otherwise it drops the request. The client expresses his preferences in the message using a pre-defined ontology. This ontology captures the user's preferences (specified using the tuple template ontology) along with time and location factors. The incoming DAML+OIL instance is parsed using the Jena API to extract individual interest descriptions from the incoming DAML+OIL instance. An entry is created for every individual interest description; each entry contains: the client id, the DAML data describing the kind of service, or object that it is interested in, location preference, time preference and an expiration time associated with the interest registration request. All entries are added to the interested-entries table and are indexed using a unique id. Two lists are maintained: the *active-entries-list* and the *passive-entries-list*. Entries are moved from the passive list to the active list by checking the time preference of every entry in the passive list regularly. A non-blocking *readIfExists* operation is performed on every entry in the active list where the timeout is set to *NO_WAIT*. If the *read* operation does not return back any tuple from the space, then a *notify* is registered with the space, with the lease time set according to the time out of the interest registration request. When a matching tuple is acquired from space, a service description of the service tuple or an instance of object tuple is sent back to the client that initiated the request. To avoid too many *Interests-Registration-Response* messages from being sent back to the client the results are buffered for a short time span, after which a single *Interests-Registration-Response* message is sent.

The user can specify his time preference as: *all-the-time*, *morning*, *noon*, *night* etc. Or, he could specify a time range using *day*, *month*, *year*, *hour*, *minute* etc. The location preference can be specified to access services that do not belong to the same Service Manager. The Service Managers create an instance of the location class of the space ontology to specify where they are. A mapping of the location name to the Vigil handle is specified using this instance. Using this mechanism clients can register for all services or objects that are available on a particular *floor*, *building* etc. A few of the services or objects might not be accessible to the client; however, the client can request for permissions to access them.

Task-Execution Agent: The key idea of this agent is to off-load certain well-defined tasks that the user performs by acting as a proxy on behalf of the user. The client registers tasks with the task-execution agent the same way it registers tasks with the recommender agent. The CCML message type for task registration is *Task-Execution-Request*. The ontology used to express tasks includes two types of tasks: simple and composite. Every simple task specifies the time at which the task should get executed. The tuple template associated with a task is used to search for tuples on the space. The agent performs a *read* operation on the space to acquire a desirable tuple associated with the task. If the desired tuple is a service tuple, then the agent tries to subscribe to the task by sending a CCML message of type *Subscription-Request* to the Vigil service. If the subscription fails due to lack of access right and delegation, then the agent does another read operation on the space to look for alternate service tuples. If a match takes place, then a subscription request is sent out to the new service. If the *Subscription-Response* is a success, then the CCML Service Parameters[16] from the response are stored in a parameter-list. In Vigil, commands (like switching the light ON or OFF) are executed by modifying the value associated with parameters in the list and sending the updated parameter-list back to the service. The task ontology allows the specification of these commands. It also provides a wait time associated with every command in the task. The agent can execute the next command of the task only after the wait time for the last command has expired. After a command is executed an update is received from the service which contains a new CCML Service Parameter list. The task ontology also allows the specification of output parameters. The values associated with output parameters from the parameter-list are sent back to the user or might serve as an input for other tasks.
Example Scenario: A user might register a task with the execution agent to turn off the music service 5 minutes after all people in the room have left.

The client specifies the task by creating an instance of a SimpleTask or CompositeTask class. A composite task is composed of several atomic tasks. The ontology supports control constructs to execute *Sequential*, *Concurrent* and *Unordered* tasks. *Sequence* allows a sequence of atomic tasks to be performed one after the other; *Concurrent* allows the parallel execution of multiple atomic tasks; and *Unordered* allows the atomic tasks to be executed in some unspecified order. The user can also specify the start time and stop time of atomic task for tasks that do not require

immediate execution.

Publish-Subscribe Agent: This agent dynamically delivers data to subscribed users. A service/agent can publish data or events that are meant to be shared by multiple users by writing a data tuple on the semantic space. The subscription request to this agent is routed the same way as it is done for the other agents. The CCML message type for the request is *Published Object Subscription*. The description of the subscription object is provided by using the tuple template ontology. Upon receiving a subscription request from the user, the agent searches the space for published data tuples that best matches the user's subscription request. If such a data tuple exists, it does a *take* operation on the tuple, updates its list of subscribed users, and writes the updated tuple back to space. The actual data associated with the selected object tuple is sent back to the client using the response message. This agent registers a *notify* operation for every subscription request to get notified of new objects or changes to objects that might occur on the space. Example Scenario: As mentioned in the motivating scenario section, users can subscribe to the conference schedule tuple. If the schedule changes then the subscribed users get notified of this change. Users could also subscribe to specialized advertisements on the conference space that offer discounts or special deals. For example, registered users in a conference might be entitled to special discounts in shops located close to the venue of the conference. As part of future work, we want to support subscription requests that contain rules, based on which the user gets notified.

7. Conclusions

In order to verify the semantic matching that sTuples performs; ontologies were created for the printer service, music service, light service and room-recommender service. A vCard ontology was created to create data tuples. Several instantiations of these services and data tuples were written on the space and different types of tuple templates were created to see what service instances get picked up by sTuples. The reaping interval to eliminate unused entries on the space was set to about 3 minutes. The matching algorithm on sTuples, increased the time required to retrieve a tuple to a couple of seconds under low-load conditions (where there are about 2 to 3 candidate service instance matches). Under high-load conditions (where there are about 12 to 15 candidate service instance matches) the retrieval time increased to approximately 3 seconds. As part of future work several optimizations can be performed on the Tuple Space, to enable a faster retrieval of tuples. Further empirical evaluation is beyond the scope of this paper.

As part of current work, the recommender agent is undergoing enhancements to do semantic association rule learning. The recommender agent should be able to recommend tuples based on the access patterns of users on a particular space. Consider the scenario where a user wants to access the postal address of the current location from the space. It is likely that he will next try to look for a map quest agent that gives directions from the current location to the closest airport(or some other destination). It is also likely that he will next try to get a handle to the closest printer service on the space to print out directions. There are several such usage patterns of data and services in a space based on the context of deployment. Such patterns can be mined by making use of an association-based rule learning algorithm such as the Apriori algorithm[1]. The semantic knowledge about data and services can be used to deduce more that what a traditional association rule mining algorithm can achieve. For example, the recommender agent might mine an association rule, which indicates that every time a map quest service tuple is used a laser printer service tuple is also used. By looking at the description of the tuples it can infer that a laser printer is a type of printer; hence when a map quest service tuple is used there is a good chance that any printer service tuple is used(not just laser printers).

In the future, semantic descriptions can be introduced to express the functionality of methods/behaviors in addition to data in a tuple. This will enable heterogeneous mobile agents to use sTuples. sTuples should also migrate from DAML+OIL to OWL. Most constructs outside the scope of OWL DL such as qualified cardinality restrictions were not used while developing our ontologies. However, in some ontologies the type separation between Classes and Individuals was not always maintained. With minor modifications to our ontologies we should be able to migrate from DAML+OIL to OWL DL. Currently the Centaurus Client is made available to the user through a fairly primitive graphical interface. The user specifies preferences, registers tasks and subscribes to data services using this interface. In the future we would like to integrate the Centaurus Client/Service with the user's Personal Agent to better understand preferences and usage patterns. Also, the task-execution agent in our system does not currently use a planner to execute composite tasks. In the future, we would like to recommend composable tasks. In addition to the security provided by JavaSpaces and Vigil additional security can be enforced on tuples by using the policy ontologies.

In this paper, we have presented a Tuple Space based *Semantic Infrastructure* that overcomes shortcomings of existing Tuple Space implementations by introducing semantic matching of tuples on top of object based polymorphic matching. The sTuples system provides a generic framework to implement clients and services in a pervasive environment by introducing semantic tuples and its extensions, service tuple and data tuple. sTuples integrates with Vigil, which serves as a communication gateway and provides distributed trust and access control. Specialized agents cater to

unobtrusive behavior of data and services, task execution and subscription to data tuples. The use of semantics enable clients and services developed by heterogeneous systems to work together by making use of shared ontologies and generic tools to process web ontologies. sTuples uses Racer, a sound and complete description-logic reasoner to detect inconsistencies and inferred subsumptions. The OWA(Open World Assumption) that Racer makes is a desirable feature since, in a pervasive environment, information can be incomplete. The Tuple Space's *notify* operation is implemented by making use of Racer's publish-subscribe mechanism. The *Semantic Tuple Spaces* system helped us validate our ideas, and makes us believe that entering the Semantic Web can help realize the original vision of Pervasive Computing by providing *Semantic Interoperability*.

References

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, (VLDB)*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[4] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical report, Dept. of Computer Science, Dartmouth College, 2000.

[5] T. Finin, A. Joshi, L. Kagal, O. Ratsimore, V. Korolev, and H. Chen. Information Agents for Mobile and Embedded Devices. *Lecture Notes in Computer Science*, 2182:264–286, 2001.

[6] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[7] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Aquisition*, 5(2):199–220, 1993.

[8] V. Haarslev and R. Moller. *RACER User's Guide and Reference Manual*.

[9] V. Haarslev and R. Möller. RACER System Description. *Lecture Notes in Computer Science*, 2083:701ff, 2001.

[10] R. Handorean and G.-C. Roman. Secure Sharing of Tuple Spaces in Ad Hoc Settings. Technical report, Washington University in St. Louis, 2003.

[11] J. Heflin, R. Volz, and J. Dale. Requirements for a Web Ontology language. Technical report, 2002.

[12] J. Hendler and D. L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 2000.

[13] N. R. Jennings and M. J. Wooldridge. Software Agents. *IEE Review*, pages 17–20, 1996.

[14] L. Kagal and J. U. et al. Vigil: Providing Trust for Enhanced Security in Pervasive Systems. Technical report, University of Maryland, Baltimore County, 2002.

[15] L. Kagal, V. Korolev, S. Avancha, A. Joshi, T. Finin, and Y. Yesha. Centaurus: An Infrastructure for Service Management in Ubiquitous Computing Environments. *Wireless Networks*, 8(6):619–635, 2002.

[16] L. Kagal, J. Undercoffer, A. Joshi, and T. Finin. Vigil: Enforcing Security in Ubiquitous Environments. In *Grace Hopper Celebration of Women in Computing 2002*, 2002.

[17] O. Lassila. Serendipitous interoperability. In Eero Hyvönen, editor, *The Semantic Web Kick-off in Finland – Vision, Technologies, Research, and Applications*, HIIT Publications 2002-001. University of Helsinki, 2002.

[18] O. Lassila and M. Adler. Semantic Gadgets: Ubiquitous Computing Meets the Semantic Web. In D. Fensel, J. Hendler, W. Wahlster, and H. Lieberman, editors, *Spinning the Semantic Web*, pages 363–376. MIT Press, 2003.

[19] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, World Wide Web Consortium, February 1999.

[20] C. Mascolo, L. Capra, and W. Emmerich. Middleware for Mobile Computing. In *NETWORKING Tutorials*, pages 20–58. Springer, 2002.

[21] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Working Draft, World Wide Web Consortium, March 2003.

[22] S. Microsystems. JavaSpace Specification, 1998. Sun Microsystems. JavaSpace Specification, March 1998. http://java.sun.com/products/jini/specs.

[23] P. Obreiter and G. Graef. Towards Scalability in Tuple Spaces. In *Symposium of Applied Computing (SAC) Special Track on Coordination Models, Languages and Applications*, pages 344–350. ACM, 2002.

[24] S. Paul. An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments. Master's thesis, Lancaster University, 1999.

[25] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[26] P. Thompson. Ruple: an XML Space Implementation. In *In XML 2002 Conference and Exposition*, 2002.

[27] Waldo. Jini architecture overview. Technical report, SUNLABS, July 1998.

[28] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, 1991.

[29] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1988.