4

Data Processing

This chapter describes the data processing supported by the GAMBAS middleware. Towards this end, the chapter first describes the formalisms and ontologies for the data and query models. The formalisms and ontologies provide a unified view of the heterogeneous data produced by the different players in the targeted applications. Such a unified view, based on semantic descriptions of the data and the data sources, is in line with the linked data paradigm, and it not only facilitates data understanding, but also improves data discovery and integration between both objects and persons, and other sources of data that follows the same paradigm, such as the Web of Data. Based on the data and query models, the chapter introduces the general data discovery mechanisms that are used to make data available to others. Finally, the chapter describes the architecture and implementation of the distributed data storage and processing system that allows devices to cooperate with each other in a seamless and interoperable way.

4.1 Focus and Contribution

The data representation and the associated query processing infrastructure are key to allow data interoperability between the devices and services targeted by the GAMBAS middleware. This is particularly important given that behavior-driven services often base their decision on data coming from multiple sources. Descriptions of the data and the data sources should be available to all devices. Such descriptions can include the features of interest, accuracy, measuring condition, time point, location, etc., and they are essential for search and discovery when an Internet-connected object is confronted with a large number of data sources. The query processing needs to account for the dynamic nature of some of the generated data, and it should be done in a distributed fashion, whenever possible, to improve scalability and also to increase privacy-level of data processing.

4.1.1 Data Representation

There have been a lot of efforts in employing Semantic Web technology to semantically enrich sensor data [WZL06], [BFL+07], [SHS08], [RMLM09], [PHS10]. In order to allow easy integration with other data sources available in a Linked Open Data (LOD) cloud, [Lin12] suggests that sensor data sources should be published following the Linked Data principles [BHBL09] – a concept that is known as Linked Stream Data [SC09]. The advantages of such an approach are manifold. Not only would it support the direct integration of sensor data with the large amounts of already available web and enterprise data, but it can also benefit from a large body of work and infrastructure from existing research areas such as LOD, Web and Data Base Management Systems (DBMS). One example scenario is the case where GPS locations streamed as Linked Data are combined in real time with a Cocitation Collection Service available in the LOD cloud. The service can then notify an author if there is any other author in the same location whose papers he cites. However, the state of the art in Semantic Web technologies is inadequate for sensor-generated data, due to the highly dynamic and temporal aspects of this data. Moreover, the data representation suggested by Semantic Web technologies typically are not suitable for devices with limited data storage.

Stream elements of Linked Stream Data are usually represented as RDF triples with temporal annotations. A temporal annotation of an RDF triple can be an interval-based [LPSZ10] or point-based [GHV07] label. An interval-based label is a pair of timestamps, which commonly are natural numbers representing logical time. The pair of timestamps, [start, end], is used to specify the interval that the RDF triple is valid. The point-based label is a single natural number representing the time point that the triple was recorded or received. Both approaches have their advantages and disadvantages. The point-based label looks redundant and less efficient in comparison to the interval-based one. Furthermore, the interval-based label is more expressive than the point-based label because the latter is a special case of the former, i.e. when start = end. However, a point-based label is more practical for streaming data sources where triples are generated unexpectedly and instantaneously.

4.1.2 Query Processing

The state of the art in query processing of Semantic Web data can also not be directly applied to the context of data generated by smart mobile devices. There has been work on extending Semantic Web technologies for stream data. StreamingSPARQL [BGJ08] has rules for translating continuous

queries, common in stream processing scenarios, to SPARQL algebra, the standard query processing language for Linked Data. Streaming SPARQL extends the SPARQL 1.1 query language for representing continuous queries on RDF Streams.

CSPARQL [BBCG10] combines triple stores with data stream management systems (DSMS). When a continuous query arrives, it is first split into static and dynamic parts, and both parts are executed independently and results are combined at the end. EP-SPARQL [AFRS11] translates the processing into logic programs. The execution mechanism of EP-SPARQL is based on event-driven backward chaining (EDBC) rules. EP-SPARQL queries are compiled into EDBC rules, which enable timely, event-driven and incremental detection of complex events (i.e., answers to EP-SPARQL queries). EDBC rules are logic rules and hence can be mixed with other background knowledge (i.e. domain knowledge that is used for reasoning).

CQELS (Continuous Query Evaluation over Linked Streams) provides a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data [LPDTXPH11]. The query executor is able to switch between equivalent physical query plans during the lifetime of the query. The CQELS engine employs both efficient data structures for sliding windows and triple storages, to provide high-throughput native access methods on RDF datasets and RDF data streams. Similar to other systems, the CQELS engine extends SPARQL 1.1 for continuous queries. However, it also supports updates in RDF datasets as well as variables for stream identifiers, allowing queries that continuously discover streams that contain a certain property. Despite the progress in Linked Stream Data processing, currently none of the approaches consider a distributed solution for resource-constrained devices.

4.1.3 Contribution

Data representation and query processing of Linked Stream Data is an active research area with many open challenges. The GAMBAS middleware addresses the problem of data interoperability among dynamic heterogeneous data sources, where data storage is limited. It provides an infrastructure supporting the discovery of dynamic linked data sources that runs on resource-constrained devices. Thereby, it provides solutions for important aspects of continuous query processing over heterogeneous Internet-connected objects to create a scalable system that can react to changes in the network and in the data being produced.

Data interoperability is achieved by means of a unified representation of the heterogeneous data and their data sources, following the Linked Open Data principles. The unified view consists of basic vocabularies and ontologies that cover all aspects of the data required to realize the application scenarios. Special care is taken to represent dynamic and temporal aspects. The goal is to enable the devices themselves to store their generated data locally in the form of Linked Data, by using the vocabularies and ontologies provided as part of the middleware. Therefore, special care is taken to limit the amount of data that needs to be stored, since storage in connected objects is limited. To do this, the descriptions applied by GAMBAS are complete, yet compact.

To allow data discovery, the infrastructure constructs and maintains a directory of descriptions, which are accessible to every device and are constantly updated to incorporate changes in the network, whilst respecting the communication cost for each device. The directory complies with the privacy rules, by having the devices to publish only information they wish to make it public and by supporting the encryption of metadata.

To support both data interoperability and discovery, the data processing framework of GAMBAS provides Linked Data storage capabilities for all connected objects. This improves scalability and also privacy, since each device can take on the responsibility of storing its own data and it can therefore decide which data can be disclosed to which devices. There are many Linked Data storage frameworks available but none of them are designed for resource-constrained devices. The GAMBAS middleware encompasses a data storage framework based on the state of the art approaches that also complies with limitations imposed in terms of memory, processing power, battery life, etc. On top of the data storages, a query processing framework is offered that follows the same guidelines. Even though the query processing capability at each device is limited, distributed query processing techniques are integrated in order to provide a more powerful processing framework among the devices.

4.2 Data Model

As basis for interoperable distributed data processing, this section introduces the data definitions and query specifications integrated into the GAMBAS middleware. The data definition is based on an ontology that has been developed with the goal of supporting the internal mechanisms of the middleware as well as the application scenarios targeted by the middleware. The ontology and query examples are described using free text descriptions and UML-like

diagrams to clarify ontological relationships among concepts and groups of concepts. These diagrams are used to facilitate the comprehension of ontological concepts and their relationships. Along with that, example instances are used to illustrate how to populate ontology instances in RDF/Turtle [W3C12d]. For the description of the example queries, GAMBAS uses a subset of SPARQL query semantics and syntaxes rather than creating a new query language. In order to enable the processing of streams of data, GAMBAS leverages the CQELS query language.

4.2.1 Data Definition

Figure 4.1 shows the GAMBAS ontologies, its classes, the dependencies among the classes as well as the external ontologies from which the ontology extends concepts and properties. The external ontologies include PIMO [Sem12], SPT [SPI12], GoodRelations[Goo12] , Ordered List[Ord12] and Vehicle Sales [Mar12]. The PIMO Ontology provides a vocabulary for describing calendaring data (events, tasks, meetings). The SPITFIRE Ontology (SPT), developed within the SPITFIRE project, aligns already existing vocabularies – such as DOLCE [CNR12], WGS84 [W3C12f] and FOAF [FOA12] – to enable the semantic description of not only sensor measurements and sensor metadata, but also the context surrounding them. In particular, the activities sensed by sensors are modeled and related with social domain vocabularies and complex event descriptions. The GoodRelations ontology is widely used to describe business and product offerings. We take advantage of the Ordered List Ontology to represent a sequence of steps. An OrderedList is a list of slots with indexes to each slot and pointers to the next and the previous slot. The Vehicle Sales ontology is a web vocabulary for describing cars, boats, bikes and other vehicles for e-commerce, and it is useful in the context of GAMBAS to generalize the means of transport of a user.

The GAMBAS ontology consists of a number of sub-classes, the generic classes being **User**, **Place** and **Activity**. In addition, the ontology contains the classes **Journey**, **TravelMode** and **Bus** that are motivated by the mobility scenario as well as **Jogging** and **Shopping** that are motivated by the environmental application scenario. In the following, we describe these classes in more detail.

4.2.1.1 User Class

The User class is used to describe users of the GAMBAS middleware. In GAMBAS, users play the roles of both data consumer and provider. As a

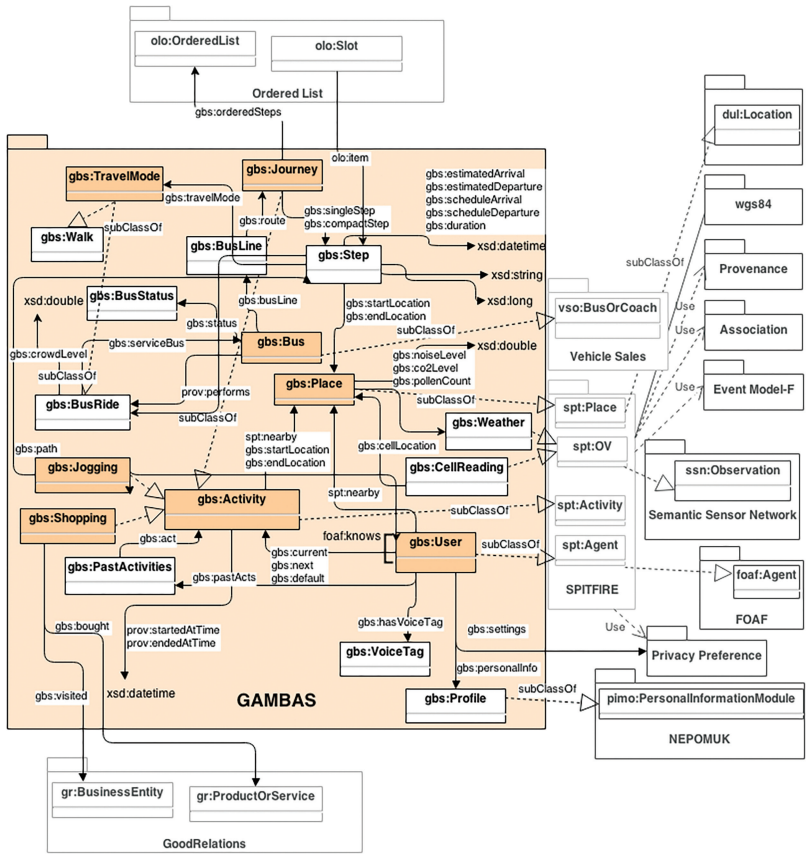

Figure 4.1 The GAMBAS Ontologies.

consumer, a user is accessing services provided through some user interface such as suggestions of bus routes or jogging areas. As a data provider, users allow GAMBAS to acquire personal data such as location and activities (e.g. traveling in a public transport, jogging, shopping, etc).

Figure 4.2 shows the User class in the GAMBAS ontology. The user class is a subclass of the spt:Agent class from the SPITFIRE ontology, which allows us to describe the user's profile such as name, email and addresses. Privacy settings are crucial in GAMBAS. To model them, we rely on the Privacy Preference vocabulary given by the Privacy Preference Ontology (PPO) [DER12]. However, during the implementation of the application prototype, it became apparent that the PPO was not suitable to describe users' shared keys and permission settings, which are needed in the privacy-preserving

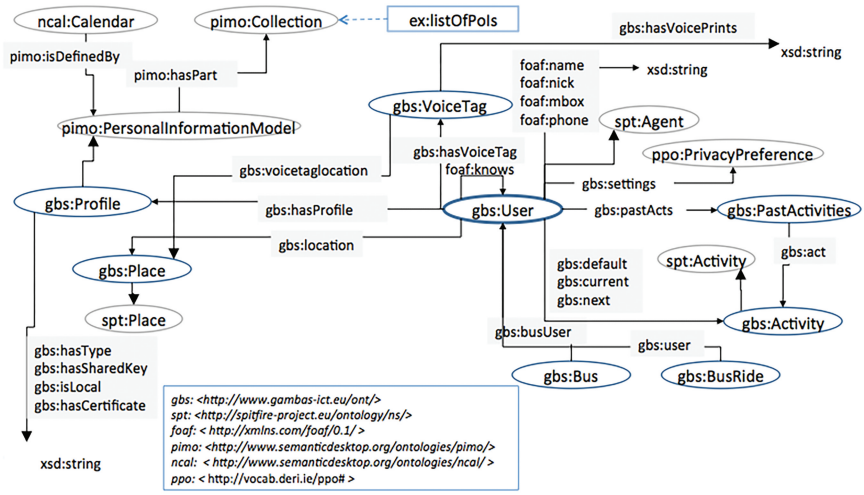

Figure 4.2 User Class.

data exchange mechanism of GAMBAS. Therefore, we added privacy-related properties to the user profile. More specifically, we extended the Profile class to include the sharedKeys and certificates used by the mechanisms described in Chapter 5.

The user's calendar information, which is used as input for the user's intent analysis, is described by creating a PIMO (Personal Information Model) instance. Users are connected to other users via the "foaf:knows" property, which allows us to list the friends of a user. The location of a user is also available and can be represented with the Place class.

Users in GAMBAS perform activities, for instance, commuting in a bus or shopping. The GAMBAS ontology provides a vocabulary to represent the user's activities, including the past, future and current ones. Past and current activities are used in combination to determine which are the user's next activities. This is done by the user's intent analysis.

Listing 4.20 shows an example of how to use the above concepts to describe a user within the GAMBAS scope, using the Turtle syntax. The example shows, among other things, how users can set access levels to other users. In this particular example, the user "John" is giving the user "Paul" access to his location. Note that the access is restricted to read-only, therefore Paul cannot modify or create instances of location for John.

To preserve the user's privacy, instances of the User class are stored in the mobile devices of the respective users. The user's location, current and next activities are dynamic properties. All remaining properties are expected to change less often and are therefore considered to be mostly static.

Listing 4.1 User Instance Example

```
ex:john a gbs:User, pimo:Agent;
foaf:nickname ``userid''^^xsd:string ;
ex:john gbs:current ex:activity1 ;
ex:john foaf:knows ex:paul ;
gbs:Profile ex:johnProfile ;
gbs:pastActs ex:archive1 ;
gbs:settings ex:ppoJohn ;
.
ex:archive1 a gbs:PastActivities ;
gbs:act ex:activity2 ;
gbs:act ex:activityn;
.
ex:activity2 a :Journey ;
prov:wasAssociatedWith ex:user ;
prov:startedAtTime ``..''^^xsd:datetime ;
prov:endedAtTime ``..''^^xsd:datetime ;
.
.
ex:johnProfile a gbs:Profile;
gbs:hasSharedKey ``B8C382391061E449CE51B29C2549BB1F'';
.
ex:ppoJohn a ppo:PrivacyPreference;
ppo:hasCondition[ ppo:classAsObject gbs:Place ];
ppo:hasAccess acl:Read;
ppo:hasAccessSpace[ ppo:hasAccessAgent ex:Paul>; ].
.
ex:activity23 a :Jogging ;
ao:mood ex:friendly ;
prov:wasAssociatedWith ex:john ;
gbs:runWith ex:paul ;
prov:startedAtTime ``2012-04-03T10:00:00Z''^^xsd:
dateTime ;
prov:endedAtTime ``2012-04-03T11:00:00Z''^^xsd:date
Time ;
gbs:path ex:runningLeg ;
.
```

4.2.1.2 Place Class

The location of a user in GAMBAS can be captured by different sensors (e.g., GPS, WIFI, GSM). The GAMBAS Place class, shown in Figure 4.3, provides different properties for the different representations. The Place class is built upon the spt:Place class, which already provides a vocabulary that includes

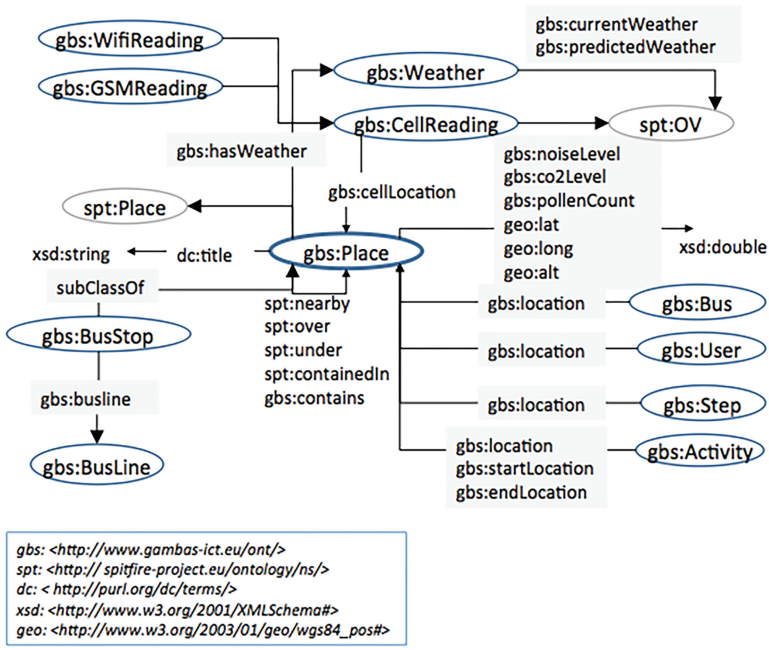

Figure 4.3 Place Class.

concepts like, city, street and GPS coordinates. The Place class extends spt:Place by enabling the representation of bus stops and cell location.

The CellReading class extends the spt:OV class, which provides the vocabulary to describe sensor observations. A noise level can be associated with every location, which can be used, in combination with the user's preferences, to suggest optimal travel routes. In addition, the place class adds properties related to the environmental scenario, such has CO2 levels and pollen count.

It is important to note that locations can be described by the set of locations it contains. This allows us to aggregate information from smaller areas, to generate a broader view. Lastly, as bus stops are a very relevant type of place in the mobility application scenario of GAMBAS, we introduce a subclass of Place, called BusStop, to specifically model them. In addition, we can have a property associated with a bus stop that lists all the bus lines that serve that stop.

A directory of locations is made available via external servers. For privacy reasons, the users' current location is dynamically stored on the mobile device.

4.2.1.3 Activity Class

A user may perform different activities, e.g. visiting a location, shopping, taking the bus or train, jogging, etc. The GAMBAS Activity class, shown in Figure 4.4, provides the properties to describe an activity. Every activity can have a start/end location and start/end time. Locations are represented as instances of the Place class. For representing the time, we use the xsd:datetime description from the OWL Time ontology [W3C12e]. Different activities, such as traveling in a bus or jogging on a particular route, are modeled as subclasses.

4.2.1.4 Journey Class

The journey class models special activities that represent general location changes of the user. A journey can involve a trip by a bus or other modes of transportations (e.g. walk between two bus stops to switch buses). A journey consists of a series of segments, or steps, and these steps are described using the class Step, which is also part of the GAMBAS ontology.

In each Step, we can specify a number of properties, such as arrival/departure times (both scheduled and estimated), duration, distance covered and start/end locations. Moreover, we can specify the travel mode used in each instance of Step, which will be described later on.

In some cases, we are interested in recording every segment between two consecutive bus stops, i.e. to check whether a user might meet a friend or not. By using the gbs:singleSteps property, we can model this case, and each Step will correspond to two consecutive points in the journey. However, we might also be interested in a more compact version of the journey, where steps in

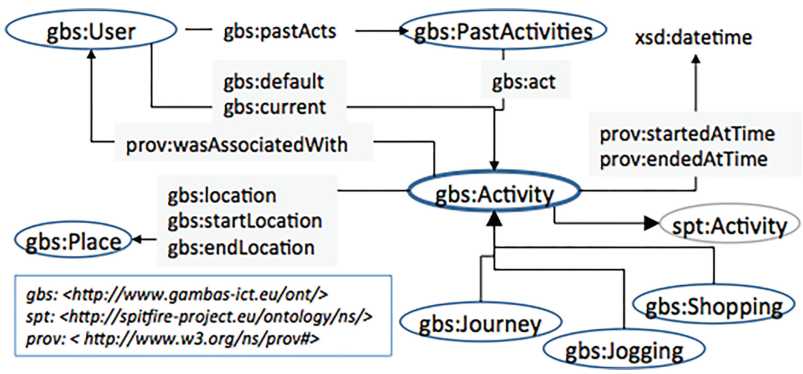

Figure 4.4 Activity Class.

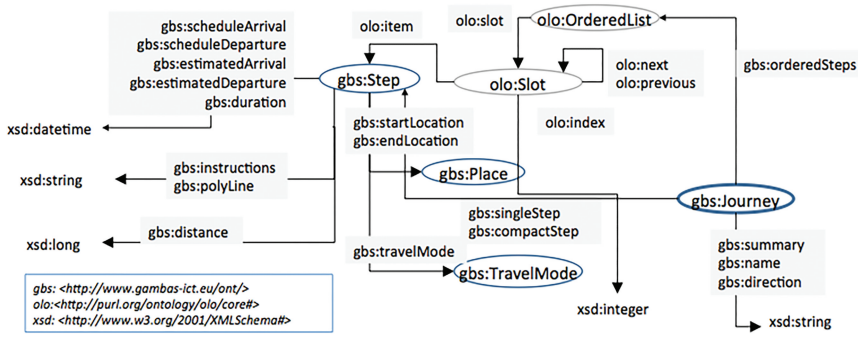

Figure 4.5 Journey Class.

which the travel model has not changed can be represented by one single step. This provides a shortcut to determine when a user entered or left a bus, for instance. For this, we have created a gbs:compactSteps property. Note that this compact version can be created at any time from the list of single steps. While it provides some redundant information, it greatly improves the performance of some queries. In addition, we also introduce a mechanism to keep track of the order in which the steps were performed during the journey. We take advantage of the Ordered List Ontology [Ord12] to represent a sequence of instances of the Step class. An OrderedList is a list of slots with indexes to each slot and pointers to the next and the previous slot. In our case, each slot contains an item of type Step. Figure 4.5 illustrates the Journey class, and an example is given in Listing 4.2.

The instances of the Journey class can be stored in the user's mobile device or a trusted external server. Information regarding the schedules is static, while the estimated departure/arrival times are usually updated dynamically.

4.2.1.5 TravelMode Class

As we mentioned in the previous section, a journey is composed of multiple steps, and each step can be performed by a different travel mode. To model this, we introduce an abstract class that represents the different travel modes. At the moment, there are two possible subclasses: BusRide and Walk, but it is straight forward to extend this by adding other means of transport, e.g. car or subway. Figure 4.6 illustrates the TravelMode class, as well as its subclasses.

For steps where a bus ride was used, we can specify further properties, like the bus used and the crowd level of the vehicle. We can also attach the

Listing 4.2 Journey Instance Example

```
ex:itinerary1 a gbs:Journey
gbs:orderedSteps ex:list1 ;
gbs:singleStep ex:step1 ;
gbs:singleStep ex:step2 ;
.

ex:list1 a olo:OrderedList ;
olo:slot ex:slot1 ;
.
ex:slot1 a :Slot
olo:item ex:step1 ;
olo:next ex:slot2 ;
.

ex:slot2 a :Slot
olo:item ex:step2 ;
.

ex:step1 a gbs:Step ;
gbs:startLocation ex:PlazaMayor ;
gbs:endLocation ex:stop2 ;
gbs:distance ``10'' ;  #distance between the two
stops.
gbs:scheduleArrival ``21:13:54Z''^^xsd:time ;
gbs:scheduleDeparture ``21:23:00Z''^^xsd:time ;.
gbs:travelmode ex:walk ;
gbs:instructions ``walk from Plaza Mayor to stop2'' ;
.
ex:step2 a :Step ;
gbs:startLocation ex:stop2 ;
gbs:endLocation ex:stop3 ;
gbs:distance ``15'' ;  #distance between the two
stops.
gbs:scheduleArrival ``21:30:00Z''^^xsd:time ;
gbs:scheduleDeparture ``21:35:00Z''^^xsd:time ;.
gbs:travelmodel ex:busride ;
```

information about the user performing the bus ride directly to this class, which can be beneficial for some types of queries.

4.2.1.6 Bus Class

A bus ride is performed by a bus, and this is also represented in the GAMBAS ontology. Figure 4.7 shows the Bus class. A bus can be associated with a stream of crowd levels to describe the number of passengers that are traveling

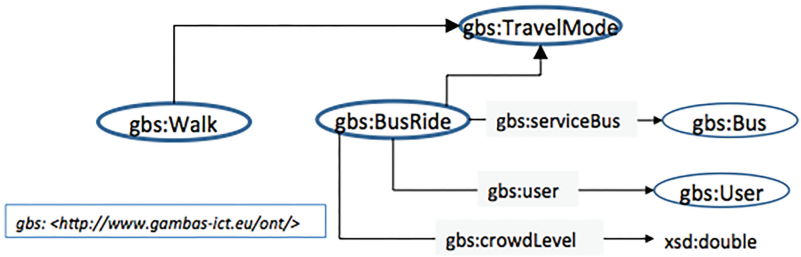

Figure 4.6 TravelMode Class.

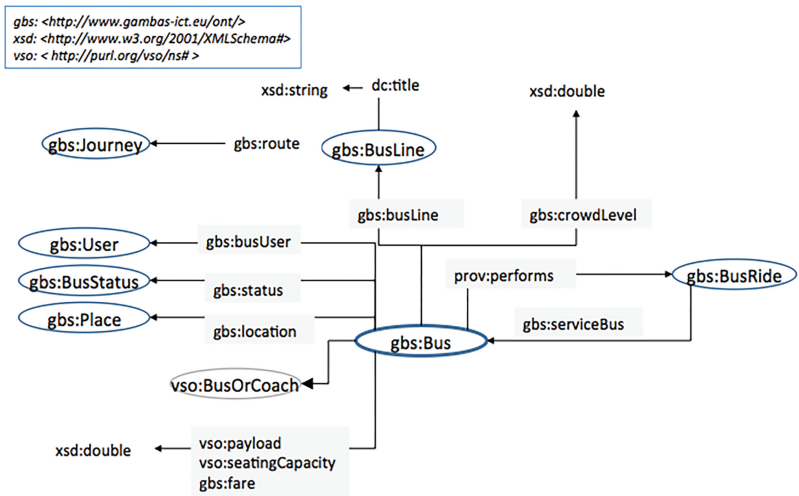

Figure 4.7 Bus Class.

on the bus. Aggregated values can be recorded and stored in instances of the BusRide class, to compute statistics of the crowd levels in the different bus routes. In addition, we can represent the route of a bus line by reusing our Journey class. Other properties include the bus line name, the bus status (in service or not) and the bus' current location.

The information about buses is provided by the transport layer and it is usually stored in an external semantic data storage. The bus location, crowd levels and its status are constantly updated.

4.2.1.7 Jogging Class

The Jogging class is a subclass of the Activity class, and it can record the path followed during the jog, the distance covered, the aggregated CO2 and pollen

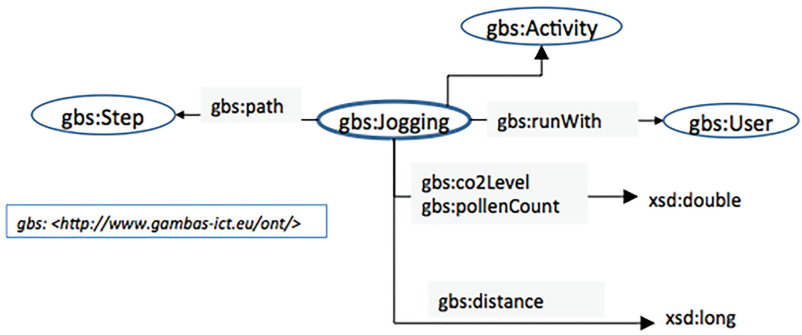

Figure 4.8 Jogging Class.

levels and the friends met during jogging. Since we do not expect changes regarding transportation mode during a Jogging activity, we can model the path taken as one single instance of the Step class, which already provides all the required properties (start/end location, polyline, duration). Figure 4.8 shows the Jogging Class.

The jogging activities are recorded in the mobile device of the user that performed the activity. However, in order to support coordination, they may be shared explicitly, e.g. with friends.

4.2.1.8 Shopping Class

In addition, the ontology includes a Shopping class, which is also a subclass of the Activity class, to describe the user's shopping. Instead of proposing a new class to model stores and their products, we use the GoodRelations ontology [Goo12], which is well known and widely used. The Shopping class allows us to enlist the products bought by the user during this activity as well as shops visited. Figure 4.9 shows the Shopping Class that are typically stored on the user's mobile device.

4.2.2 Query Specification

The data instantiated from the GAMBAS ontology is represented as RDF [W3C12a]. SPARQL [W3C12b] is the most widely used RDF query language, and therefore it has been chosen as a query language in the GAMBAS context. However, some of the data in GAMBAS is available as a stream of RDF data, or RDF streams. This is the case for the dynamic information, like the location of a user. For handling RDF streams, GAMBAS relies on an extension of the SPARQL query language, called the CQELS query language

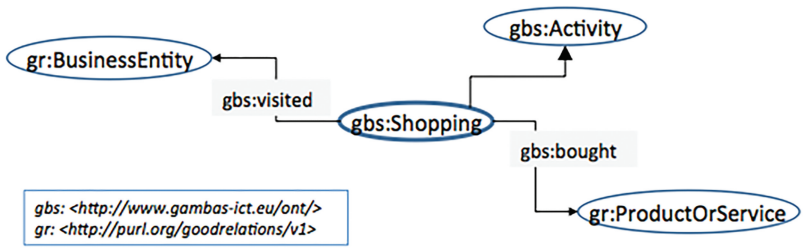

Figure 4.9 Shopping Class.

[LPDTXPH11]. The full specification of the SPARQL query semantics and syntaxes are defined by the W3C and can be found in [W3C12b]. In the RDF data model, each instance must have a globally unique URI. An RDF instance has properties that have values as literals or other instances. A literal can have text or numeric value.

In the context of GAMBAS, the SPARQL-SELECT and CQELS-SELECT queries are sufficient for all realized applications. The output of these queries is results sets in tabular form of literal and URI. Query results can be easily serialized, for example, in XML [W3C12c] or JSON [W3C13a] format. In the following, we present a number of examples for queries against the data definitions contained in the GAMBAS ontology. The main purpose of these examples is to clarify how the ontology and the definitions can be accessed using SPARQL and CQELS, respectively.

4.2.2.1 Queries on Users

For retrieving the list of all users registered at the system, we can use the query shown in Listing 4.3.

Listing 4.3 Query All Users

```
PREFIX : http://www.gambas-ict.eu/ont/
SELECT *
WHERE ?x a :User.
```

To determine the current activity of the user with a specific user identifier, we could use the query shown in Listing 4.4. Similarly, we could retrieve the user's calendar entries or friends.

For analyzing the users' intent, we can access information like the activities where a bus ride on a particular bus line was involved. Especially for the case where we want to discover whether two users have been on the same bus, we can ask for activities with a particular bus line and via a certain step.

Listing 4.4 Query Current Activity

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?activity
WHERE ?x foaf:nick ``userid'' .
      ?activity :current ?x .
FILTER ( ?endtime > NOW ) .
```

A step, in this case, corresponds to the route between two consecutive bus stops given by the URIs of the start and end locations. In both cases, we can narrow the search to a time interval. Listing 4.5 shows an example for this.

Listing 4.5 Query Bus Rides of a Line for a Segment within an Interval

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?busride
WHERE ?x foaf:nick ``userid'' .
      ?activity a :Journey ;
       prov:wasAssociatedWith ?x ;
      :singleStep ?step. ?step :startLocation
      <startLocURI>;
      :endLocation <endLocURI> ;
      :travelMode ?busride.
      ?busride a :BusRide ;
      :serviceBus ?bus .
      ?bus  gbs:busLine <buslineURI> .
      ?activity prov:startedAtTime ?starttime ;
      prov:endedAtTime ?endtime .
FILTER (?endtime < ``2012-04-03T00:00:00Z''^^xsd:date
Time) .
FILTER (?starttime > ``2012-04-02T00:00:00Z''^^xsd:
dateTime) .
```

The examples show that the GAMBAS ontology is flexible whether you are looking for a journey specified by start and location or other properties, such as the bus line taken. The travelMode property allows us to filter out activities where a bus was not involved.

For the user intention mining, it is important to analyze the historical information associated with buses. The query shown in Listing 4.6 retrieves all recorded bus traces of a user in a given bus.

Note that we can use the compact representation of the journey to retrieve the full segment of the user in a bus, rather than the individual steps.

Listing 4.6 Query Ride History of a User

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?step
WHERE ?x foaf:nick ``userid'' ;
      :pastActs ?acts. ?acts :act ?journey ;
      :compactStep ?step. ?step
      :travelMode a :BusRide .
```

In the environmental domain, we can look for journeys in which some of the steps had a CO2 level above a given threshold. This is shown in Listing 4.7.

Listing 4.7 Journeys with CO2 Level above Threshold

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?journey
WHERE ?x foaf:nick ``userid'' . ?activity a :Journey ;
      prov:wasAssociatedWith ?x ; :singleStep ?step.
      ?step :startLocation ?startLoc ;
      :endLocation ?endLoc.
      ?startLoc gbs:co2Level ?startco2.
      ?endLoc gbs:co2Level ?endco2
      OR{?startco2 > <threshold>. ?endco2 >
      <threshold>} .
```

For the above query, we need to retrieve all the start and end locations and check for their CO2 levels. We iterate over every single step on the journey to make sure we retrieve all locations visited in that journey.

Another interested query is to retrieve a list of users who had gone jogging with a particular user shown in Listing 4.8. This could be used, for instance, to indicate a stronger friendship level between the two users.

Listing 4.8 Query Users Jogging with a User

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?user
WHERE ?x foaf:nick ``userid'' .
      ?activity a :Jogging ;
      prov:wasAssociatedWith ?x ;
      :runWith ?user.
```

As we mentioned earlier, GAMBAS extends the query set by supporting queries that involve dynamic information. For this, it uses the CQLES query language that resembles SPARQL. The main difference is the introduction of the STREAM command that allows us to specify a window of data within the stream. The query shown in Listing 4.9 retrieves the current location of a user.

Listing 4.9 Continuously Query the Latest User Location

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?location
WHERE ?x foaf:nick ``userid'' .
STREAM <streamURI> [NOW] {?x :location ?location}.
```

In this query example, <streamURI> refers to the URI from where the stream with the data in question can be accessed. The parameter [NOW] extracts the latest location streamed. CQELS is a very flexible language, allowing an easy integration of static and dynamic data. For example, for suggesting bus stops near the user, we can write the query shown in Listing 4.10.

Listing 4.10 Continuously Query Near by Bus Stops

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX prov: http://www.w3.org/ns/prov#
PREFIX spt: http:// spitfire-project.eu/ontology/ns/
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?nearby
WHERE ?x foaf:nick ``userid'' .
STREAM <streamURI> [NOW] {?x  :location ?location}.
?nearby a :BusStop ; spt:nearby ?location.
```

It is noteworthy to highlight that CQELS queries are continuous queries, which means they are registered in the system and whenever new data is generated in the stream, the query is evaluated and results are pushed to the output. For example, we can imagine a scenario of a user walking around and getting notifications of nearby bus stops as he changes location.

4.2.2.2 Queries on Buses

This section presents a subset of queries about buses, bus stops and bus lines. For instance, to get bus stops near a particular GPS location, we can query as shown in Listing 4.11.

Listing 4.11 Query Bus Stops at GPS Location

```
PREFIX : http://www.gambas-ict.eu/ont/
PREFIX g: http://www.w3.org/2003/01/geo/wgs84_pos#
PREFIX spt: http:// spitfire-project.eu/ontology/
ns/
SELECT ?place
WHERE ?place a :BusStop ; spt:nearby ?location.
      ?location a :Place ; g:Lat ''50.0'' ; g:long
      ''3.0''.
```

Similarly, we can also retrieve the bus route for a particular bus line. The corresponding query is shown in Listing 4.12.

Listing 4.12 Query Bus Stops of a Bus Line

```
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?busroute
WHERE ? busline a :BusLine ; :route ?busRoute
```

To retrieve the list of stops covered by a bus line in the correct sequence, we can use the ordered list to iterate over the different steps as shown in Listing 4.13. Note that the query might return duplicates if start/end locations overlap. However, this can be easily fixed by a simple scan over the results list.

Listing 4.13 Query Bus Stop Sequence of a Bus Line

```
PREFIX : http://www.gambas-ict.eu/ont/
PREFIX olo: http://purl.org/ontology/olo/core#
SELECT ?start ?stop
WHERE { ?busline a :BusLine ; :route ?busRoute.
        ?busRoute :orderedSteps ?list.
        ?list olo:slot ?slot .
        ?slot olo:item ?step ; olo:index ?index .
         ?step :startLoc ?start ; :endLoc ?end
      }
ORDER BY ASC(?index).
```

With the Place ontology, we can easily query for all bus lines that run on a stop. Moreover, we can also query for bus lines that run on a given date on that stop as shown in Listing 4.14. To do this, the query looks at the routes of the bus lines and filters them by the date.

For a user waiting at a bus stop, we want to send notifications of possible delays. We can first retrieve all the bus lines that run on the stop and check their timetables against the stream of estimated times. In the query shown in Listing 4.15, we can specify a threshold (e.g., 5 minutes), and if the current

Listing 4.14 Query Bus Stop Sequence of a Bus Line

```
PREFIX : http://www.gambas-ict.eu/ont/
PREFIX prov: http://www.w3.org/ns/prov#
SELECT ?busline
WHERE <busstopURI> :busLine ?busline .
      ?busline :route ?route .
      ?route prov:startedAtTime ?start ; prov:
      endedAtTime ?end.
FILTER( ?start ><date>). FILTER (?end <<date>).
```

Listing 4.15 Query Delayed Buses

```
PREFIX foaf: http://xmlns.com/foaf/0.1/
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?estimateddeparture
WHERE ?x foaf:nick ''userid'' ; :location ?stop.
      ?stop :busline ?line .
      ?line :route ?route .
      ?route :singleStep ?step .
      ?step :startLocation ?stop ;
      :scheduleDeparture ?scheduleDeparture
STREAM <streamURI> [NOW]
      { ?step :estimatedDeparture ?estimated
      Departure }.
FILTER (?estimateddeparture >
          ?scheduleDeparturel +threshold).
```

live departure time estimation is over the threshold, then the system will notify the user.

The last query examples are related to the crowd-level information available for different public transit vehicles. To access the latest status and crowd-level information of a particular bus, we can use the query depicted in Listing 4.16.

Listing 4.16 Query Latest Crowd Level of Bus

```
PREFIX : http://www.gambas-ict.eu/ont/
SELECT ?crowdLevel ?status
WHERE ?bus a:Bus
STREAM <streamURI> [NOW] {?bus :crowdLevel ?
crowdLevel}.
STREAM <streamURI> [NOW] {?bus :status ?status}.
```

Using the GAMBAS ontology, we can store an aggregated value of crowd levels recorded for a particular step of a journey. This value can be, for instance, the maximum crowd level at any stage of that step or the average

value. In the query depicted in Listing 4.17, we show how to extract the maximum crowd level of a step.

Listing 4.17 Query Latest Crowd Level of Bus

```
PREFIX : http://www.gambas-ict.eu/ont/
SELECT MAX (?crowdLevel)
WHERE  ?step a :Step ; :estimatedDeparture ?start ;
       :estimatedArrival ?end ; :travelMode ?busride .
       ?busride :serviceBus ?bus .
STREAM <streamURI> [RANGE 30min]
       {?bus:crowdLevel ?crowdLevel[timeStamp]}.
FILTER (?start < timeStamp < ?end).
```

When processing data streams, we can extract windows of data, by specifying the window parameters. In the previous queries, we used [NOW] to extract the latest value. Here, we select all the data of the last 30 minutes. Note that it is not possible to specify a start/end time interval for the window operators. Nevertheless, we can take advantage of the fact that every stream data can have a timestamp associated with it. In the case of this query, we assume that the start time did not occur before 30 minutes ago, and we select the valid crowd levels during the step in the filter condition.

4.3 Data Discovery

To enable the distributed execution of queries across multiple data stores, the query processors must be able to discover the available data stores. The GAMBAS dynamic data discovery system is responsible for providing this functionality. From an architectural perspective, it is realized as a central registry service that offers two distinct interfaces: (1) a GAMBAS-based registration interface to export metadata and search for data sources and (2) a web-based administration interface that allows to configure the discovery system, check its state and browse current registrations. The discovery system is developed using the Google Web Toolkit (GWT), a toolkit for the development of web-based client/server applications, and deployed in a Java servlet container such as Apache Tomcat. Figure 4.10 shows a screenshot of the administration interface of the discovery registry.

Besides a central registry instance for normal system operation, application developers can run their own private instances of the discovery system in their local networks. This allows using separate discovery systems for development work or prototyping and isolates the development systems from each other and the central discovery system used for normal system operation.

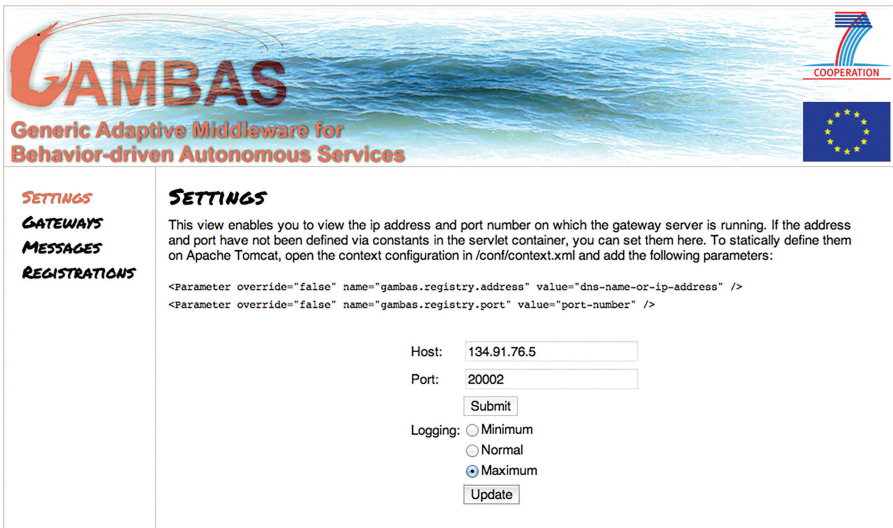

Figure 4.10 Dynamic Data Discovery Registry Administration Interface.

4.3.1 Architecture

The architecture of the GAMBAS dynamic data discovery is shown in Figure 4.11. The system is deployed as a servlet in a regular servlet container. It builds upon the GAMBAS communication system to realize remote communication and lease management as described later. The data registration is co-located with a communication gateway component that is used by the communication system to enable multi hop routing and connectivity in peer-to-peer environments with firewalls or networks with native address translation (NAT).

The co-location of the registry with the gateway allows to easily locate the registry and thus simplifies the bootstrapping of the system. The dynamic data registration contains all functionalities needed to publish metadata describing data sources, to update this information and ensure its freshness. The web-based administration interface depicted in Figure 4.10 allows to configure the discovery system (as well as the communication gateway) and to browse current metadata as well as exchanged messages.

4.3.2 Metadata Management

Metadata is used to describe data sources such that clients can easily select semantic data stores that contain data that is relevant for their queries.

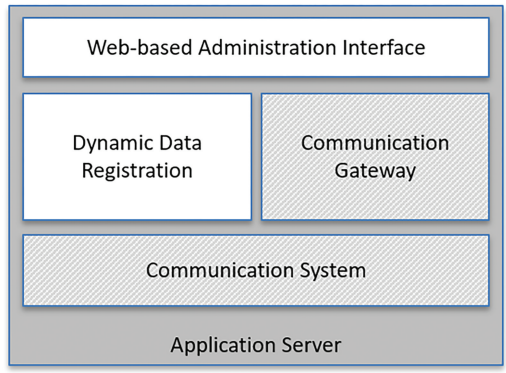

Figure 4.11 Dynamic Data Discovery Registry Architecture.

The metadata published in the registry follows the linked data paradigm to describe the data provided by devices. Listing 4.18 and Listing 4.19 show different examples of metadata information for a service providing environmental information about places and a service providing information about bus schedules, respectively.

Listing 4.18 Register a Service for Environmental Information

```
<place1> a gbs:Place ;
         gbs:noiseLevel ``-1'' ;
         gbs:co2level ``-1'' ;
         gbs:pollenCount ``-1'' .
```

Listing 4.19 Register a Service for Bus Schedules

```
<line1>  a gbs:BusLine ; gbs:route <route1> ;
         dc:title ``some_line'' .
<route1> a gbs:Journey ; gbs:singleStep <step1> .
<step1>  a gbs:Step ; gbs:startLocation <p1>;
         gbs:endLocation <p2> ;
         gbs:scheduleArrival ``00:00:00Z''^^xsd:time ;
         gbs:scheduleDeparture ``00:00:00Z''^^xsd:time .
```

It is important to note that the registry only keeps the data structure (ontologies classes and properties), but not the actual instances and property values. As the purpose of the directory is to allow discovery, it only needs to store the shape of the RDF graph, which are then used to match against user queries. An example query looking for providers of GPS coordinates is shown in Listing 4.20.

Listing 4.20 Finding Services Providing GPS Coordinates

```
SELECT distinct ?g
WHERE { GRAPH ?g
        { ?p geo:lat ?lat ; geo:lon ?lon . }
      }
```

4.3.2.1 Publishing Metadata

To make a data source available, the discovery service offers remote methods using the underlying communication system to register a new data source, to update a registration and to remove a registration. To do so, data sources send their metadata description to the registry. This metadata is then stored at the registry and made available for clients to find suitable data sources. The signature of the registration method is:

- *DeviceRegistration register(DeviceDescriptor)*

The method takes a device description that specifies the metadata to describe a data source and returns a new registration object that can be used to maintain the registration. In case a description changes, data sources can update a registration by calling the update method:

- *Boolean update(DeviceDescriptor, DeviceRegistration)*

This method takes a new descriptor as well as an existing registration (obtained by an earlier call to register) and returns a Boolean specifying if the update was successful. If the registration cannot be found in the registry, the update will fail.

4.3.2.2 Unpublishing Metadata

At some point of time, a data source might want to stop offering data or it may become unavailable. To stop offering data, a data source can deregister itself from the registry using the remove method:

- *void remove(DeviceRegistration)*

This method takes a registration and removes it from the registry. If the registration cannot be found in the registry, the method fails silently, i.e. no error notification is given. In any case, after the method finishes, the registration is no longer available for clients.

In addition to this explicit removal, the discovery service also employs a lease mechanism to ensure freshness of registrations in cases where a data source becomes unavailable without being able to deregister. To do so, the discovery service uses an existing component of the communication system.

For every registration, it starts a lease process that checks the availability of registered data sources periodically. In case a data source is not available several times, a lease manager integrated into the communication system notifies the discovery service, which eventually removes the registration.

4.3.3 Querying Data Sources

To find suitable data sources for a specific data need, clients can issue data source queries at the discovery system. To do so, they can call the find-method of the registry:

- *DeviceResult find(DeviceQuery)*

This method takes a query (implemented as an *DeviceQuery*) that specifies the intended data sources and returns a query result (implemented a *DeviceResult*) possibly including a set of suitable data sources.

4.3.4 Security and Privacy

In addition to support for public services, a secure version of the Dynamic Data Registry (DDR) provides privacy guarantees for users who may wish to limit sharing of their data to specific users or groups of users. To do this, the secure version of the registry integrates an encryption scheme known as IPHVE. This scheme not only ensures that only users with access to a particular data item are able to discover the location of the item in question, but it also ensures that the registry itself cannot become a security or privacy liability, since the registry itself also cannot read the stored metadata.

IPHVE is an attribute-based encryption scheme, which extends the Hidden Vector Encryption scheme [IP08]. IPHVE uses the Dual Pairing Vector Spaces (DPVS) framework [OT08]. Some of the main operations are:

- **Setup** – Generates a Secret Key (**SK**) and Public Key (**PK**).
- **Encryption** – Generates a Ciphertext (**Ct**) given a Message (**M**), **PK** and a Vector of Attributes (**Vx**).
- **Key Generation** – A Decryption Token (**DTk**) is generated given **SK** and another Vector of Attributes (**Vy**).
- **Decryption** – Given **Ct** and **DTk,** generates a Plain Text (**Pt**) if the **PK** used to generate **DTk** corresponds to the **SK** used to generate **Ct,** if **Vx** and **Vy** correspond to the HVE definition.
- **Test or Verification** – Returns true if, given **Ct** and **DTk**, **Vx** and **Vy** correspond to the HVE definition.

As an extension to IPHVE, a Generic Decryption Token (**GDTk**) can be generated, which allows users to set provider-defined attribute values. The **GDTk** can then be modified by the users with a Random Session Key (**RSK**), which prevents the registry to decrypt a message.

The resulting interaction with the secure DDR is shown in Figure 4.12. The message exchange remains similar, i.e. data providers publish metadata for users to discover. The novelty lies in the addition of a message from the data provider to the user with a decryption token that enables discovery. This token needs to be included in the message to the registry in order to get the results.

4.3.5 Client-side Caching

Since discovery is a mandatory step in execution of remote queries, the discovery process increases the latency experienced by applications. To mitigate this, the GAMBAS middleware provides a client-side cache that enables clients to store information about remote data providers to reuse this information in case there is another request for the same data. This is a standard approach for remote directory systems that is also used by DNS, for example. When executing a query, the mechanism first checks if it already has information about the requested data provider in a local cache. If that is the case, then this information is returned. Otherwise, a standard discovery request is issued. Freshness is provided by using standard techniques, i.e. leases and data invalidation in case of unsuccessful communication requests.

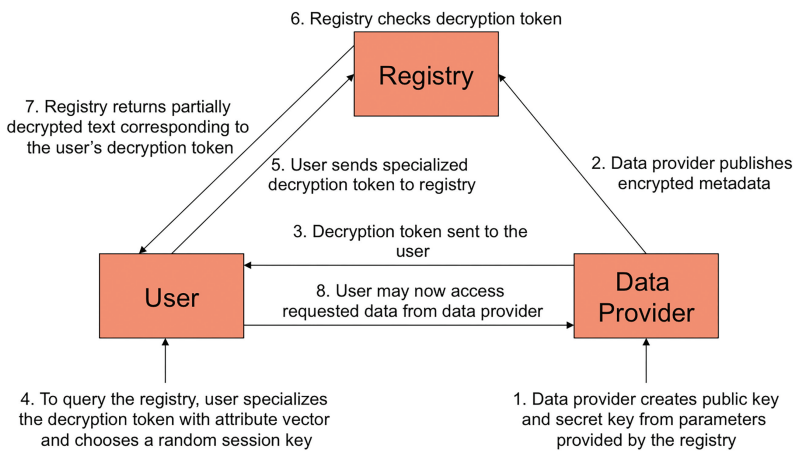

Figure 4.12 Secure Data Discovery Registry.

4.4 Data Processing

Using the Dynamic Data Discovery Registry, it is possible to discover the systems that are storing data that might be relevant for the execution of a query. However, the Data Discovery Registry only stores metadata. In order to provide security and privacy guarantees, the data itself is stored in a semantic data storage that can be queried using a query processor. In the following, we discuss the details of these two remaining components.

4.4.1 Data Storage

The semantic data storage (SDS) component provides the ability to store and retrieve RDF [W3C12a] data on devices equipped with the GAMBAS middleware. These devices range from constrained to back-end computer systems. To cope with these different device classes, two different versions of the SDS are provided: one for Android and one for J2SE environments. Both versions rely on a common (i.e. platform independent) base implementation as far as possible. To further reduce the development effort, both versions use a basic triple store for actually storing data and extend this triple store with GAMBAS-specific functionality, e.g. a remote storage interface or handling of intermittent query results (used for distributed queries).

As no established triple store exists for both J2SE and Android, we decided to use different triple stores for them and to provide a unified interface on top of them through the GAMBAS middleware. For J2SE, we use Apache Jena [Apa13], a well-established, efficient and powerful implementation that supports many additional functions such as full support for SPARQL 1.1. For Android, we use rdf-on-the-go [NUI12], a triple store implementation that is derived from Jena. On top of the triple stores, GAMBAS adds additional support for formatting query results as JSON strings according to [W3C13a]. Finally, to support formatting RDF data as N-Triple strings [W3C04], the semantic data storage contains bindings to a custom but generic N-Triple parser, called YANTRIP (Yet Another N-TRIple Parser) that is based on the JavaCC parser generator to minimize development effort and to allow for easy extensibility.

In the following, we discuss the optimization techniques applied to the semantic data storage components in order to increase their scalability. The focus of the optimizations lies on memory consumption and data indexing techniques of the storage on mobile devices. Consequently, the optimization primarily apply to the Android version of the SDS, since this version faces the most restricting constraints.

4.4.1.1 Data Storage Optimization Techniques

Reducing the memory footprint is one of the critical key targets to improve performance of the SDS [Nor07], especially when running on mobile devices. Although random access memory on mobile devices has improved, the heap size of an Android application is still limited. For example, the system RAM of an ASUS NEXUS 7 tablet is approximated 1GB, but the default memory heap size for an application running on it is only 64MB. There are a couple of reasons for this limitation. First, Android is a multi-tasking operational system with many applications stored in memory concurrently. If an application occupies too much memory, it might impact other applications or bloat the whole system. Second, Android uses the mark-sweep algorithm to perform garbage collection. Thus, an application will be paused while being garbage collected and bigger heap sizes lead to longer pause times [MNP+10]. This reduces the performance of an application significantly.

To reduce memory footprint, the GAMBAS SDS for Android employs dictionary encoding which is similar to the implementations of Jena TDB or Sesame. In contrast to solutions for standard computers, we use a compact integer format that is optimized for millions rather than billions of RDF nodes. We believe this is the common scale of most mobile personal information applications. Existing RDF stores for mobile devices are restricted to smaller data sizes of approximately one order of magnitude less [ZS12]. Each RDF node is processed and mapped to a node identifier before it is loaded into main memory. A node identifier is 32 bits in size, where 9 bits are used for encoding the node type and the remaining 23 bits for encoding a string identifier. Most operations on nodes, e.g., matching during a query execution, can be performed on these node identifiers without accessing the actual string representation. Thus, only one integer must be kept in memory for each node, while string representations can be stored on flash memory. This leads to a memory footprint of just up to 12 bytes per triple, while memory profiling reported about 450 bytes per triple for the Jena memory model. Note that despite this large memory footprint reduction, we do not restrict our system to keep all triples in main memory. Instead, our RDF store can store triples in flash memory as discussed next.

For efficient access, all RDF triples are indexed with a schema we already presented in [LPPRH10]. It consists of three triple indexes with different node orders with respect to subject (S), predicate (P) and object (O): SPO, POS and OSP. The indexes are stored in flash memory to reduce the required amount of main memory and to make data persistent. We also operate a triple cache in main memory, which contains currently used parts of the indexes.

Flash memory has a great impact on the design of an efficient DBMS for mobile platforms [LNK+07]. For example, well-known B-Tree indexing techniques were shown to be not suitable for flash memory [LHLY09]. Therefore, we have built a special lightweight key-value database. This database is optimized for flash memory and allows us to fully control I/O blocking and block caching. This way we can better manage memory access and minimize the impact of Android's garbage collection due to erase-before-write limitations of flash devices [JS10].

Flash I/O is based on memory blocks. Instead of reading or writing individual bytes, the I/O unit always reads/writes a whole block. The size of a block depends on the individual devices. Thus, in order to write a single byte in a block, the whole block must be read, modified and written again. This makes random access writing very inefficient. Our aim is to reduce the number of read and write accesses as much as possible. To do so, we partition each index into individual blocks, which have the same size as the flash I/O blocks of the device. The individual blocks are stored in flash memory. A metadata structure specifies the triples contained in each block, given as lowest and highest node identifier in the sorted block. The triple cache contains a number of index blocks. If a new triple is added, it must be added to the indexes. To do so, the system loads the required index blocks into the cache. Then, the triple must be included at the right position in the index. This is trivial if the triple should be added at the end of an existing block that still has open space. Otherwise, we would need to move all triples by one position, resulting in a large number of writes. To reduce this overhead, we do not change the original block. Instead, we slice the block into two parts: an old, original block and a new one. The old one is not changed at all. The new one contains all triples starting with the newly added one. Then, the metadata structure is updated to specify that the new block contains all parts including the new triple, while the old one only contains parts before that.

As an example, imagine that a block contains three triples for subject nodes with identifiers 1, 5 and 7. The metadata will specify that this block contains triples for subjects 1 to 7. To add a triple starting with a subject node with identifier 6, we read the original block if it is not already in the cache and create a new block containing the triples starting with identifiers 6 and 7. Then, we update the metadata to specify that the old node contains triples for subjects 1 to 5, while the new one contains triples for subjects 6 to 7. We did not have to modify the original block in any way. The new block is still in the cache and hopefully will get additional triples for the same subject before writing it onto flash later. This way, we will only need to perform one write

access to flash memory. To further reduce the number of read/write accesses, when we need to remove a block from the cache and write it back to flash, our strategy chooses a block that has a high chance of not being changed in the future. Together, these optimizations reduce the overhead of using flash memory considerably.

4.4.1.2 Data Storage Optimization Results

To evaluate the performance gains when applying the optimization techniques to a Semantic Data Storage, we have implemented them as part of the SDS for Android. Using this implementation, we compare the new version with the old version, which used Berkeley DB as underlying database (RDF-BDB). We also compare against the Android version of Jena TDB (TDBoid).

Figure 4.13 shows that the throughput of the improved version of the SDS (RDF-OTG) is four times higher than TDBoids and is roughly seven times higher then the original version (RDF-BDB). Moreover, besides having much better update throughput, RDF-OTG also consumes considerably less memory than other systems (see Figure 4.14). Especially, while the previous version crashed at 200,000 triples due to memory overflow error (i.e. the application consumed more than 64MB heap size), the improved version only needs 20MB heap size for the same amount of triples.

A similar trend can be seen when analyzing the response times of queries and the scalability of the optimized implementation. There, we can measure a performance increase of 20 to 200 times, depending on the query complexity. Similarly, while the original version of rdf-on-the-go was only able to handle

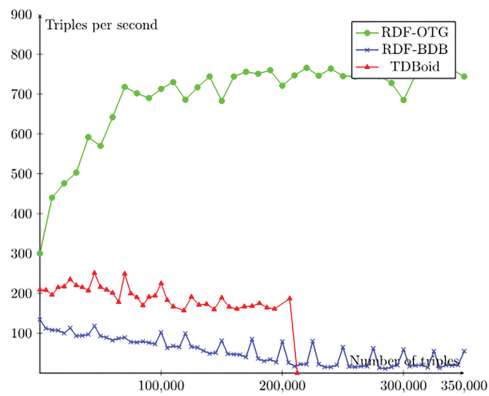

Figure 4.13 SDS Throughput Comparison.

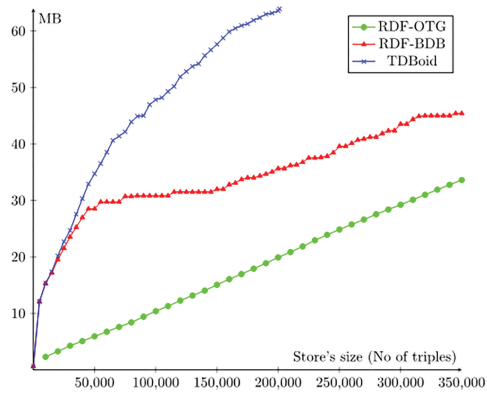

Figure 4.14 SDS Memory Comparison.

200000 triples, using the optimization techniques, it is possible to scale up to 4 million triples while still achieving response times in the order of seconds.

4.4.2 Query Processor

The query processing (QP) component enables clients to execute SPARQL [W3C12b] data queries on data sources, including queries on the local SDS, on remote SDS or on a combination of both. The GAMBAS architecture contains two different components for this: the one-time query processor (OQP) and the continuous query processor (CQP). The QP relies on the SDS to store RDF data and to execute local SPARQL queries and retrieve results for them. To enable this, the SDS provides a special interface to the QP. This interface allows direct access to the SDS in order to increase system performance. In addition to this, the interface is also used to store intermediate results of distributed queries. In the following, we discuss how the two main functional parts of the query processor, the privacy analysis and the distributed query support, are realized.

4.4.2.1 Access Control

When a query is received, the query processor has to check if this query can be executed within the specified privacy policies of the user. To do so, each query has an accompanied CallerContext to identify the sender of a query and to distinguish local and remote queries. Each query can be authorized or denied by an implementation of a so-called PrivacyManager that is described in more detail in Chapter 5. The authorization process is based on an analysis

of the received query, more specifically on the kind of data (models with data classes in the ontology) that the query will affect, e.g. a user or a location. The privacy manager can block the query, if the remote user is not allowed to access those classes. This allows a very fast authorization phase even on low-end Android devices, because it requires no result filtering. The actual privacy authorization workflow is handled by the so-called PrivacyQueryVerifier, which coordinates several internal classes to:

1. Extract the predicates for each subject in the query.
2. Derive the most probable class for each subject in the query.
3. Ask the privacy manager if the querying user might access the derived classes.

In order to support the ontology class derivation on Android devices, the ontologies are preprocessed and only an index is included inside the middleware. This avoids the overhead for parsing the ontologies, reduces the memory requirements and speeds up the analysis.

4.4.2.2 Distributed Queries

Dynamic distributed queries in GAMBAS are realized via a partial implementation of the recommendation for SPARQL 1.1 federated queries [W3C13b]. A query may contain one or more SERVICE keywords, each one specifying a sub-query on a remote data source. Following the linked data principles, data sources are identified by URIs. In principle, SPARQL 1.1 allows SERVICE sub-queries with unbound data sources. The QP does not support such queries since they can lead to a high communication overhead and may easily overwhelm restricted computer systems.

The core functionality for distributed queries consists of a generic distributed query processor and an intermediate result storage. The latter is implemented using semantic data storage. The distributed query processor receives a query and checks if it can be handled locally or contains remote parts. In the first case, it executes the query on the local SDS. In the second case, it forwards the query execution to the intermediate result storage. The result storage sends each SERVICE sub-query to the specified data source, collects intermediate results from them in the local SDS and joins them into an integrated result set. The question remains how the query issuer knows the right data sources for its query. For this, the QP uses the data discovery registry (DDR) described in Section 4.3. The general approach can be summarized as follows:

1. A query issuer wants to retrieve data from data sources of multiple remote users.
2. To do so, the query issuer first places a local query for the URIs identifying these users, e.g. based on their names or pseudonyms. Thus, the query issuer must know these users before sending them queries. Due to privacy, we do not allow users to send queries to other users that they do not know.
3. The query issuer then constructs a distributed query by adding one SERVICE sub-query for each remote user, which contains the user's URI as the URI of the remote data source.
4. This query is then placed at the QP.
5. When the QP finds SERVICE sub-queries, it accesses the local SDS and retrieves the pseudonyms of all users, whose URIs are contained in SERVICE queries.
6. With this information, the QP then contacts the DDR and requests contact information for all data sources that are bound to these pseudonyms.
7. After retrieving these, it uses this information to contact these data sources and place their SERVICE sub-query at them.

Note that to reduce the complexity for the application developers, the QP contains utilities that can be used to construct a query with all necessary SERVICE parts from a query template, in case that the remote query is identical for all receivers, e.g. querying location information for a set of users.

4.4.2.3 Continuous Queries

In addition to one-time queries, the GAMBAS data processing system also supports continuous query processing over streaming data. Similar to the one-time query processor, the continuous module also follows the Linked Data paradigm. This allows data integration among different data sources, being stream or static. Stream data is represented by Linked Data Streams [SC09], whereas the processing is supported by an instantiation of the CQELS framework for Linked Data Stream processing [LPDTXPH11].

The architecture of the module for stream processing is shown in Figure 4.15. It consists of an application client and an application server. For the full-duplex client–server communication, the system uses Websockets, which are supported by the client–server framework Netty [Net14]. In the client application for Android devices, the system uses the SDS as the Semantic Web framework, which provides an API to extract data from and write data to Linked Data Streams. The Client Publisher Handler manages

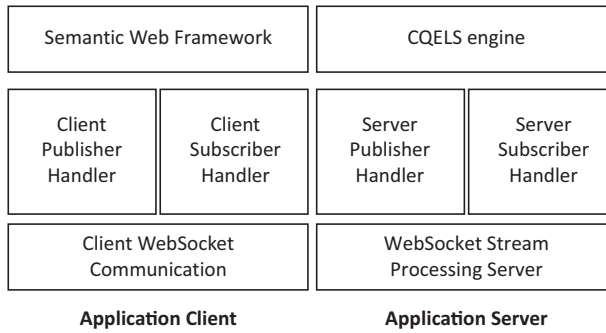

Figure 4.15 Stream Processing Module.

the upstream, which pushes RDF-triples from clients to server. To subscribe for the stream data from a particular server, the Client Subscriber Handler registers the queries to the server and manages the results listeners. Each listener listens to the results from the server through a downstream corresponding to the registered query. In the server application, the Linked Data Stream management and continuous query processor are provided by the CQELS engine. The physical streams are handled by the Server Publisher Handler and the Server Subscriber Handler. The Server Publisher Handler is tightly connected to the input manager of CQELS, in order to get the data from the clients. The Server Subscriber Handler registers the subscribed queries to the CQELS executor and routes the results to the corresponding subscribed channels.