2

Architecture

This chapter describes the high-level architecture of the GAMBAS middleware. To clarify the architecture, the chapter first presents a static perspective that focuses on the identification and definition of entities that are operating different parts of the architecture (operational view), building blocks that constitute the architecture (component view) as well as types of information that are handled by the architecture (data view). After presenting the static perspective on the architecture, the chapter introduces a dynamic perspective that focuses on a description of the interaction of architectural components. To do this, the dynamic perspective provides details on the acquisition of data (acquisition view), the discovery of data and the respective processing of queries (processing view) and the usage of data for inferences (inference view). Finally, to clarify the interactions, the chapter discusses the interfaces between the different components.

2.1 Static Perspective

The static perspective introduces the entities that interact with each other in order to produce and consume services. Furthermore, it introduces a functional breakdown into a number of core building blocks. Finally, it discusses different characteristics of the data that shall be handled by these building blocks in order to facilitate the envisioned creation and usage of behavior-based autonomous services. The dynamic perspective, which is discussed later on, ties these entities and building blocks together by describing how the different types of data are exchanged among the building blocks in order to achieve different goals of entities.

2.1.1 Operational View

As basis for the further discussion of the functional building blocks, it is important to clarify the roles of different parties that may be involved in the operation of various parts of the architecture. It is worth mentioning that a

single entity may exhibit a number of roles simultaneously and the roles that it adopts may also depend on the specific type of data that might be processed by a service. Furthermore, it is also possible to look at the infrastructure from different angles. For example, we might take a data-oriented view and classify the entities as either data acquirers or data aggregators. In the following, however, we look at the operation of the infrastructure from a service-centric perspective, as this clarifies the entities and also highlights the innovative features that are targeted by the project.

As depicted in Figure 2.1, from a high-level service-centric perspective, the entities involved in the GAMBAS architecture can exhibit one of more of the following three roles:

- **Service operators:** A service operator is responsible for executing and maintaining a part of the software and hardware infrastructure that is required for a particular service or a set of services. The operator provides computing resources such as processing capabilities and storage capacities in such a way that they can be accessed remotely

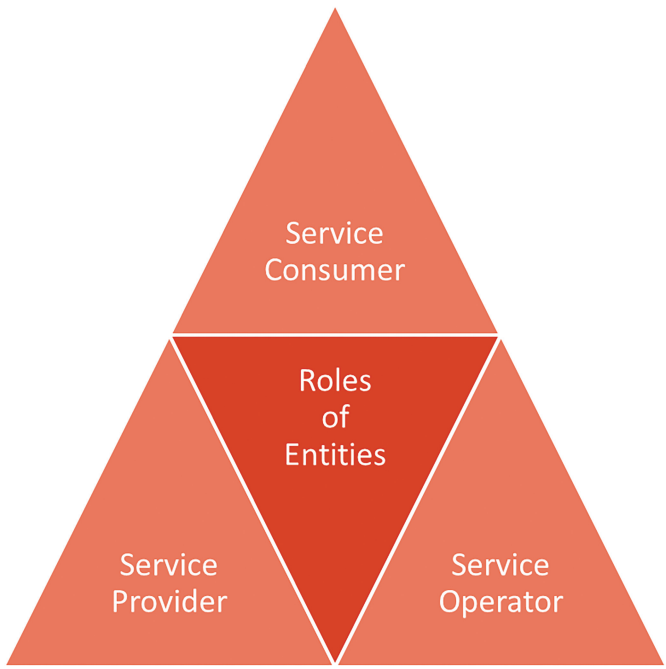

Figure 2.1 Roles of Entities.

via the network. The operator is not responsible for the actual functionality provided by the service. Instead, the service operator simply provides the basic infrastructure, possibly with service-level agreements on the performance of the system. To protect the privacy of sensitive data, we envision that some generic services such as the distributed stream processing between employees of the same enterprise or between groups of friends might be operated by an enterprise or by one or more citizens. To ease the provisioning of such generic services, we envision that service operators might offer them as pre-installed and managed bundles, similar to existing website or cloud computing offerings. Alternatively, technically advanced users might run the necessary software components on their own Internet-connected home server or wireless home router.

- **Service providers:** While a service operator is solely providing thetechnical basis for the execution of a particular service, the service provider is the actual responsible entity that offers the service to different users. In many cases, the service provider will be interested in offering a particular set of data to the consumers of the service. Furthermore, the service provider might also be interested in collecting some information from the service consumers which can then be used, for example, to improve the quality of the service. In this case, the service provider actually becomes the consumer (of parts) of the provided service. Besides service providers that want to share a particular data set, we also envision service providers that simply combine existing data sets (possibly offered by different service providers) in order to add value. For example, one service provider might combine the social graph of a set of persons with their travel behavior in order to provide recommendations for trip routes. To do this, the service provider might have to access the social graph and the trip routes from two services that are controlled by another provider. To enable such service mashups, the GAMBAS middleware uses an interoperable data representation that is based on linked open data principles.
- **Service consumers:** The last role foreseen by the architecture is the actual consumer of a service. The service consumer accesses the data and functionality offered by a service provider using the infrastructure of the service operator in order to ease their everyday tasks. In many cases, the consumer will be an end user that is accessing a particular service from a mobile Internet-connected device. Thereby, the end user might not have to initiate the interaction. Instead, the intent-aware user

interface might initiate interactions via the middleware at the right point in time without manual intervention. In order to improve the service quality available to them, the end users might be willing to opt-in to collect additional data that can help the service provider to improve the service. This mutually beneficial data collection and sharing forms the basis for the second group of service consumers. In addition to the end users, we envision that service providers can become the consumers of their own services. As a simple example, consider that a city might offer a service that enables users to share and report air quality measurements. The end users, i.e., the citizens, might then use the resulting air quality map to avoid polluted parts of the city. In addition to this, the service provider, i.e. the municipality, might use the service in order to dynamically adjust the road toll on different streets of the city in order to improve the minimum air quality.

Although these three roles are not new and are at the core of most service-oriented infrastructures, their interpretation in the context of the GAMBAS project is more dynamic. The basis for this is formed by the two key characteristics of the overall vision, namely the adaptive data acquisition and presentation as well as the dynamic and distributed data processing. Due to the former, the service consumers may also contribute to the provisioning of a service by collecting and sharing some data using their Internet-connected mobile devices. This, in turn, can result in mutual benefits for the service providers and the consumers. For some services, the providers may become the consumers of their own services. Due to the latter, new types of service providers may emerge in the network. Instead of providing their own data sets, they may simply link the existing data sets in novel ways – possibly enriching them with additional data. This will allow more tailored and specialized services and it should lead to a more thorough support for various types of service consumers that may exhibit different behaviors.

2.1.2 Component View

Intuitively, as a service-oriented architecture that is supposed to be capturing and delivering data, we can identify three main building blocks which provide data acquisition, data storage and distributed data processing. On top of that, in order to enable the limited sharing of data, we can furthermore identify a building block that is responsible for managing the data access. Finally, in order to remotely retrieve the necessary data in an automatic fashion, we can identify a building block that takes care of data presentation. In the following, we describe these building blocks in more detail:

- **Data Acquisition Framework (DQF):** A primary capability of the GAMBAS middleware is its ability to automatically capture data on behalf of the end user or a service provider. For this, the middleware encompasses a data acquisition framework that is capable of running on different types of devices. Based on the four device classes introduced in Section 1.5.1.4, the data acquisition framework primarily targets Constrained Computer Systems (CCS). In addition to that, the data acquisition framework provides support for Embedded Computer Systems (ECS) by means of connecting the embedded systems to a constrained system. The data acquisition framework provides generic and extensible support for virtual and physical sensors, and it optimizes the data acquisition with respect to energy consumption. Furthermore, in order to support applications, the data acquisition framework provides a number of example activities and intent recognition components that primarily deal with location information, movement modalities, bus routes and environmental information. The data that is captured using the acquisition framework can either be stored locally on the device or it can be forwarded automatically to a particular service that is connected to the Internet. The former approach can be taken in order to protect the user's privacy when dealing with privacy sensitive data, whereas the later approach can be taken with data that does not impact the user's privacy or which is explicitly shared on behalf of the user.
- **Semantic Data Storage (SDS):** To store the data of the user on a local device or at a particular service, the architecture introduces a semantic data storage component. Similar to the data acquisition framework, the semantic data storage is primarily targeted at device classes with more resources, such as Constrained (CCS), Traditional (TCS) and Backend Computer Systems (BCS). The data that is stored in a semantic data storage component follows the linked open data principles and uses the interoperable data representations that have been developed as part of the GAMBAS middleware. Furthermore, the data storage is able to interface with different types of query processors, depending on the resources available on the device. This implies that there may be different implementations of this component that are optimized for different device classes.
- **Legacy Data Wrapper (LDW):** The semantic data storage component is primarily targeted at the management of interoperable data that is following the linked open data principles. However, in the short term to mid-term, it is unrealistic to expect that all types of information that are interesting for a service consumer are modeled with this

approach. Consequently, it is necessary to integrate with data that is stored in an existing "data silo" using a proprietary data representation. In order to smoothen the transition from proprietary to interoperable representations, the architecture explicitly foresees legacy data wrapper components that transform the data and possible functionality provided by legacy services into an interoperable data representation. Intuitively, it is not possible to provide a generic legacy data wrapper that can handle all possible data representations. Instead, the GAMBAS middleware encompasses basic software that eases the development of an application-specific data wrapper. Thereby, the basic software primarily targets Traditional (TCS) and Backend Computer Systems (BCS) as these are commonly used to manage data and to provide services.

- **Query Processors (xQP):** In order to make the data stored in semantic data storages available to services and applications, the architecture introduces query processor components that are capable of executing queries on top of the storages. As described in detail in Chapter 4, the query language that is supported by the query processors is a subset of the SPARQL language that considers the limited resources available on Constrained Computer Systems (CCS). Due to the different dynamics of different types of data that is handled by the GAMBAS middleware and due to the different amounts of resources that are available on different classes of devices, the architecture divides the query processors into the following two components:
    - **One-time Query Processor (OQP):** The one-time query processor is targeted at the execution of queries that evaluate the current state of the data in the semantic data storages. It executes queries that produce a single result based on the current information and the specific query. Consequently, this query processor is targeted at static information that does not change frequently or at applications that only require a one-time view. From a resource perspective, the one-time query processor is designed to support a broad range of devices including Constrained (CCS), Traditional (TCS) and Backend Computer Systems (BCS). Due to resource constrains, one-time query processors in CCS are limited to process only data stored in semantic data storages belonging to the same system. OQPs in less constrained devices have access to remote semantic data storages to allow the combination of data from multiple sources. As explained later on, the provisioning remote access respects the privacy constrains.

  - **Continuous Query Processor (CQP):** In contrast to the one-time query processor, the continuous query processor is specifically targeted at dynamic data. It executes queries that can produce multiple results based on the changes to the underlying information and the specific query. Consequently, this query processor is suitable for services and applications that require continuous monitoring of events that might be captured by multiple data sources. However, in order to handle such queries, it is necessary to introduce buffers that can easily exceed the resources available on Constrained Computer Systems (CCS). Consequently, this type of query processor will be targeted at Traditional Computer Systems (TCS) and Backend Computer Systems (BCS). Yet, in order to evaluate continuous queries, Constrained Computer Systems may make use of continuous query processors that are provided as a service that is operated by a third party. Towards this end, the continuous query processor can be considered to be a generic component that can be deployed by different entities, provided that they have access to a suitable Internet-connected computer system. Similar to OQPs, access to remote data is also enabled in continuous query processors.

- **Data Discovery Registry (DDR):** To enable transparent distributed query processing, the query processors must be able to discover the data sources that are available on the network. To make the data discoverable, a device may announce the data available in the semantic data storage to the data discovery registry which in turn will typically use a semantic data storage component to manage the announcements. In case of personal mobile devices, the announcement may be limited or modified depending on the privacy preferences of a particular end user. To enable this, the semantic data storage and the data discovery registry must interface with the privacy framework.
- **Privacy Framework (PRF):** Given the above components, it is possible to acquire information using all types of systems. Furthermore, it is possible to access dynamic as well as static information using one-time and continuous queries. In principle, this is sufficient to enable the acquisition and sharing of data. However, as some data such as the end user location or the end user travel preferences might be sensitive from a privacy perspective, it is necessary to limit the data acquisition and in particular the data sharing such that it respects the privacy preferences of different entities. Achieving this is the primary task of

the privacy framework. Conceptually, the framework interacts with the semantic data storage as well as the data acquisition framework that is deployed on each personal device. In addition, the privacy framework may also be used to limit the access to information that is provided by a particular service. For this, it is integrated into the device that is offering the service.

Using a privacy policy that can be generated automatically by means of plug-ins that access proprietary data sources, the privacy framework takes care of exporting sensitive data in such a way that it can only be accessed by legitimate entities. Furthermore, depending on the user preferences, it can apply obfuscation in order to limit the data precision and it may anonymize the data in order to unlink the data from a particular user. Since the GAMBAS middleware targets the use of personal mobile devices as primary sources of data, the privacy framework not only supports Traditional Computer Systems (TCS) but also Constrained Computer Systems (CCS) as its execution platform.

- **Intent-aware User Interface (IUI):** As the last building block of the architecture, the intent-aware user interface is responsible for leveraging the remaining components in such a way that the end user ideally receives the right information at the right time. To do this, the intent-aware user interface executes queries against different services based on the behavior of the user and decides on how and when to present what information to the user. Since the past behavior of the user might not be sufficient to predict new user goals, the intent-aware user interface can also provide ways of allowing the user to modify the predicted behavior. Furthermore, as it is the primary component that is visible to the user, it has to support manual customization by the user. This encompasses, for example, the selection of layers that are interesting for a user or the manual tweaking of a generated privacy policy in a user-friendly way. Although we envision that the concepts behind the intent-aware user interface are applicable to different types of devices, we assume that in the short term and mid-term, they will be most useful for users when they are presented on their personal mobile devices. Consequently, the current implementation of the GAMBAS middleware focuses primarily on Constrained Computer Systems (CCS).

2.1.3 Data View

The GAMBAS architecture aims at supporting a broad range of services and applications whose data exhibits vastly different characteristics.

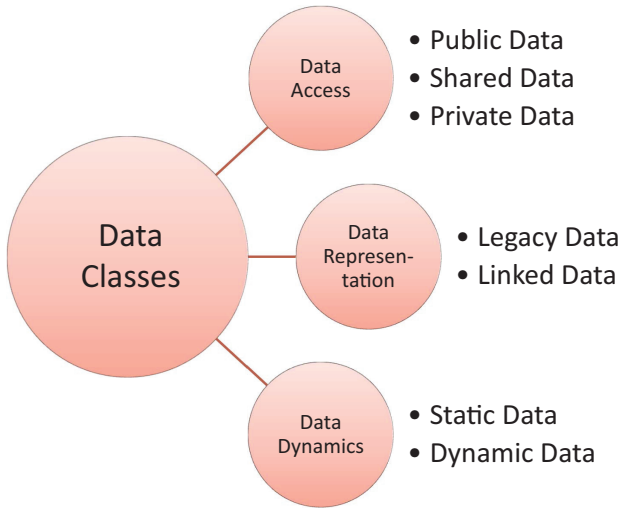

Figure 2.2 Classes of Data.

Depending on the point of view, it is possible to classify these characteristics along various orthogonal dimensions. As depicted in Figure 2.2, we focus on the level of data access, the type of data representation and the dynamics of the data. In the following, we take a closer look at these three dimensions and describe how the different classes of data are handled by the GAMBAS middleware architecture.

2.1.3.1 Data Access

Given the fact that GAMBAS aims at supporting the development of behavior-driven services that adapt autonomously to the user, it is clear that the GAMBAS architecture must be able to thoroughly support different levels of access to data, especially in cases where the collected data may be sensitive in terms of privacy. Based on the level of access, we can identify the following categories:

- **Public Data:** Public data may belong to an individual or an organization which makes the data available to third parties. Thereby, the entity that owns the data grants free access to all data for all other entities. Examples of such data could be stock prices, weather information, etc. We can assume that many applications will require public data to provide relevant and useful services. Although we can assume that most public data will be provided by services that are executed on resource-rich

devices connected continuously to the Internet, the GAMBAS architecture also allows the provisioning of public data by means of Constrained Computer Systems (CCS) such as a mobile phone. To enable seamless discovery of public data, however, the device responsible for the data must publish the metadata in the Data Discovery Registry (DDR).

- **Private Data:** In most cases, private data belongs to an individual person and it could be the user's personal data or data that the user is not willing to share with everyone. Examples of such data could be the user's contact information or the user's current location. In addition, private data may also reflect the internal data of an enterprise that is not supposed to be shared with other entities. For this type of data, the GAMBAS consortium made the deliberate decision to limit its distribution. Although it might be more practical to provide online access to private data, the GAMBAS architecture foresees the storage of private data exclusively on the devices that own it in order to prevent illegitimate access and processing through third parties. Consequently, the private data will remain on the devices that collected it unless the responsible entity makes a deliberate decision to share (parts of) it.
- **Shared Data:** In many cases, limiting the types of data to only private and public can be overly constraining. Depending on the user's preferences or on the business model of an enterprise, it might be more beneficial to share (parts of) the private information in order to get better services or to increase the revenue. For both cases, the GAMBAS architecture foresees support for shared data. In essence, shared data is a particular view on the private data. This view can be accessed by other entities that are authorized. In order to safely support shared data, it is necessary to enable trustworthy authentication among the different entities and there needs to be a policy that details who will gain access to which view on the data. Managing this process and the associated policies is done by means of the privacy framework that is an integral part of the architecture. The privacy framework thereby ensures that only legitimate entities will be able to access a shared view.

2.1.3.2 Data Representation

As hinted in the component view, in the short term and mid-term, we cannot assume that all types of data will be represented using the models and approaches developed by the GAMBAS project. Instead, we must ease the integration of existing data that may be represented using proprietary formats

by means of legacy data wrappers. Consequently, based on the level of integration, we can identify the following two classes of data:

- **Linked Data:** Linked data represents data that follows the linked open data principles that are the basis for the interoperable data representation used by the GAMBAS middleware. Using the semantic data storage component, it is possible to store linked data on any supported device. Furthermore, using the one-time and the continuous query processor, it is possible to query the static and dynamic data stored in one or more semantic data storage components. To implement interoperable services and to ensure that it is possible to easily create composed services, it is necessary that the information is represented using this data representation.
- **Legacy Data:** Although there are good reasons for picking up the interperable data representations promoted by the GAMBAS middleware, it is clearly unreasonable to assume that all data providers will immediately switch their data format. Consequently, the GAMBAS middleware provides ways to integrate legacy data that does not follow the linked open data principles. To do this, the GAMBAS middleware pursues a dual strategy. For frequently used personal data coming from different existing services such as Google calendar or Facebook, the GAMBAS middleware provides fully functional wrappers that allow the use of the stored information in order to compute privacy policies or to use them as sensor inputs. For public data coming from existing services such as the route information and time tables of public buses, the GAMBAS middleware provides support by simplifying the development of legacy data wrappers. Together, this allows the immediate use of frequently used data and it fosters extensibility with respect to more specialized existing services.

2.1.3.3 Data Dynamics

Finally, the last dimension categorizes the data on the basis of its dynamics. Intuitively, the dynamics of the underlying data can have a significant impact on the way it needs to be handled by the architecture. Clearly, there is a broad spectrum of possible dynamics and even data such as street names, which can be considered to be static, is subject to change. However, at both ends of the spectrum, we can identify the following categories:

- **Static Data:** Static data is data that never changes or changes rather infrequently. Examples for static data are geographic information such

as a map of a city or the route information of a public bus. Clearly, both examples can change over time. However, considering their update rate of months or years, it is usually possible to query the information once and then cache the results of the query for a significant amount of time. Aside from caching intermediate results, it is also possible to replicate the complete set of static data that is offered by one service at another service in order to trade-off storage for network bandwidth and latency. Given this optimization potential, the handling of static data is often less demanding than the processing of dynamic data.

- **Dynamic Data:** Dynamic data is data that changes frequently. Examples for dynamic data are the location of a particular user or a bus in the city. Although there might be periods in which updates are less frequent, like at night when the user is sleeping or the bus is parked at the depot, in many cases, it is not possible to apply similar optimizations as with static data. For example, the application of replication will require frequent synchronization and the introduction of caches for intermediate results may lead to significant imprecisions. Consequently, in many cases, dynamic data requires the execution of continuous queries, which are more resource-intensive to evaluate.

2.2 Dynamic Perspective

Given the introduction of the entities, building blocks and data types in the static perspective of the architecture, the dynamic perspective describes how they interact in order to achieve the different goals. Due to the technical objectives of the GAMBAS middleware, the dynamic perspective focuses on three main parts, namely the acquisition view, the processing view and the inference view. The acquisition view describes how different types of data are collected. The processing view describes how different types of data can be queried. The inference view describes how different data inferences can be drawn using the architecture.

2.2.1 Acquisition View

From the point of view of data acquisition, the GAMBAS middleware supports two different scenarios. The first scenario is targeting the personal acquisition of data that is used to capture the user's behavior on behalf of the user. The second scenario is targeting the collaborative acquisition of data from a large number of users that is used to improve or provide a particular service upon request of a service provider.

For the first scenario, the identity of the user is important to ensure that the resulting profile can be associated with the right user. Consequently, the acquired data may be highly sensitive from a privacy perspective. For the second scenario, the identity of the user is often not important but the service provider is rather interested in an aggregated view of the data. Consequently, by ensuring that the acquired data cannot be associated directly with a particular user, the resulting privacy issues of data collection can be minimized.

Independent of the type of acquisition, we assume that the user must be able to give an explicit consent to the data acquisition at least once in order to ensure that only the desired data types are acquired. To do this, the user must interact with the privacy framework by means of the intent-aware user interface to set the associated preferences. In the following, we outline how both scenarios are handled from an architectural perspective.

2.2.1.1 Personal Data Acquisition

A primary objective of the GAMBAS middleware is to enable the development of behavior-driven services. Intuitively, the realization of a behavior-driven service requires knowledge about the behavior of the service consumers. A key feature of the GAMBAS middleware is to provide support for the gathering of such knowledge automatically in the background.

In contrast to other approaches, the middleware focuses on the use of personal mobile Internet-connected objects such as tablets or smartphones as primary platforms for data acquisition. The reasons for this are manifold. First and foremost, many Internet-connected objects are self-contained and do not require additional infrastructure support. Secondly, the objects are often not utilized to their fullest capacity, leaving enough resources to perform context recognition. Thirdly, many Internet-connected objects have access to both physical and virtual data sources, which allows multi-modal context recognition with high precision. Lastly, the object's context is usually tightly correlated to the user's context and the recognition alone (i.e. without sharing) does not invade privacy.

While the former points are primarily underlining the technical suitability of personal mobile Internet-connected objects as acquisition platforms, the last point highlights a key feature of the approach taken by GAMBAS that is the explicit decision to focus on privacy. Given the possibly privacy-sensitive nature of a behavior profile, the data contained in it must be considered private unless a user actively shares it, e.g., in order to enable service adaptation. Consequently, the data should not be accessible directly by other parties

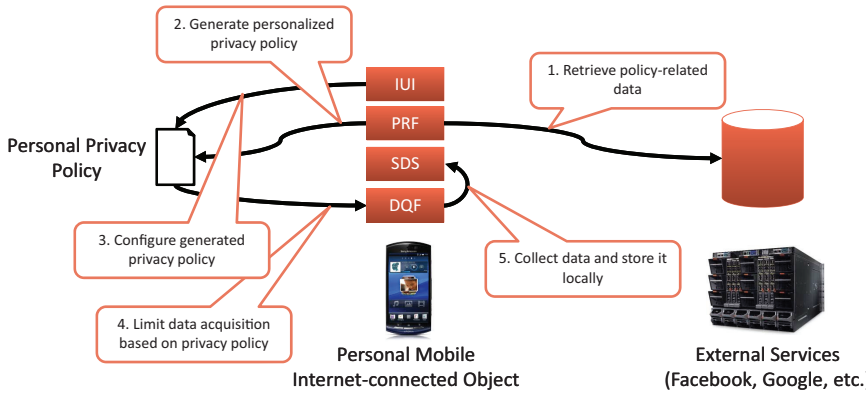

Figure 2.3 Personal Data Acquisition.

rather than the user. In order to achieve this, all personal data collected by the system is stored locally on the devices of the respective users.

The resulting component interaction for personal data acquisition is depicted in Figure 2.3. To reduce the configuration effort for the user, the privacy framework retrieves policy-related data from third-party services such as Facebook or Google, for example. Using this data, it computes an initial privacy policy. This policy can then be refined through the intent-aware user interface in order to enable manual control over all aspects of data acquisition and sharing. The resulting personal privacy policy is then used by the data acquisition framework, which limits the acquisition to those data types that are allowed by the user. In order to limit the access, the acquired data is stored locally in the semantic data storage of the mobile Internet-connected object.

2.2.1.2 Collaborative Data Acquisition

In addition to personal data acquisition, the GAMBAS middleware also supports the collaborative collection of data, for example, to enable the optimization of services based on aggregated usage information. Intuitively, this requires an alternative to the previously described personal data acquisition, since the local storage of data is not suitable for aggregating remote data. To support this, the GAMBAS architecture introduces the ability to remotely store information. Intuitively, this remote storage raises additional privacy concerns since a service provider might be able to associate the reported data with a particular user.

To mitigate this, the GAMBAS middleware enables fine-grained control over the collection process using the same procedure that has been introduced

for personal data acquisition. This enables a user to control the data that will be acquired on behalf of a service provider. In addition, the architecture also enables modifications to the data that is reported to a service. As a simple example, the middleware could refrain from sending unique identifiers or could replace them with (randomly) generated pseudonyms that change over time. In more complicated scenarios, the middleware might also apply obfuscation to reduce the data quality or it might refrain from reporting certain pieces of information at all. For this, the data acquisition framework provides a user with control over the data that is reported. This control can then be exercised to limit the sharing of data in such a way that it does not conflict with the users privacy requirements.

The resulting component interaction for collaborative data acquisition is depicted in Figure 2.4. Like in the personal data acquisition case, the privacy framework retrieves policy-related data from third-party services, which is used to compute an initial privacy policy. This policy can then be refined through the intent-aware user interface in order to enable manual control. The data acquisition framework uses the resulting privacy policy to limit the data acquisition to those data types that are allowed by the user and to modify the data accordingly before transmission. As a last step, the data acquired by the adaptive data acquisition framework is then sent to a remote device where it is stored or further processed.

2.2.2 Processing View

To describe the processing of queries, it is necessary to consider the different classes of data depending on the possible level of access. Intuitively, since

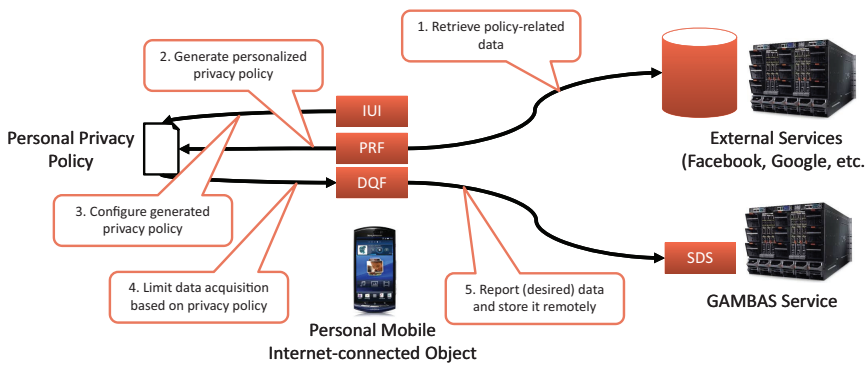

Figure 2.4 Collaborative Data Acquisition.

private data is only available to the device that collected it, distributed processing is not possible with private data. Instead, only local queries can be executed on it. However, as described previously, a user may share (parts of) his/her private data with other users or their devices. In order to ensure that shared data can only be accessed by legitimate entities, an associated access control mechanism is required. Furthermore, it is necessary to encrypt the underlying communication in order to avoid the overhearing of data over insecure network connections. For public data, access control and encryption are not necessary since the data is freely shared with everyone. Due to these differences, the GAMBAS architecture supports two possible data discovery and access mechanisms that are used depending on the level of access granted to the data. In the following, we describe both of them individually. Intuitively, it is possible to create queries that involve public as well as shared data by combining both approaches. Similarly, local queries follow the same idea but since they are targeting only data that is available locally, the associated discovery procedures are omitted.

2.2.2.1 Processing of Public Data

Overall, the processing of public data relies on the following generic three-step procedure that is frequently used in service-oriented architectures:

- **Export (Announcement, Publication):** In the first step, the availability of the data is indicated to other devices by means of exporting metadata (which describes the available data) to the data discovery registry. Depending on the architecture, this step is often referred to as export, announcement or publication. If the underlying data changes in such a way that the metadata is no longer valid, the changes must be reflected by an update to the exported metadata to avoid stale references.
- **Search (Lookup, Binding):** In the second step, which takes place before query execution, the data discovery registry is used by the query processor to find the relevant data sources. To do this, the query processor executes a query on the metadata that is stored in the data discovery registry. The query that must be executed on the metadata typically depends on the query that has been posted to the query processor from an application. Based on the result of the query against the data discovery registry, the query processor continues with the execution of the actual query against some of the retrieved data sources.
- **Execution (Usage, Invocation):** In the third step, the actual query is executed against the data sources. Depending on the capabilities of the

query processor, the query execution might be decomposed in further phases such as query planning and query execution. In the query planning phase, the query processor will typically select one of multiple possible query execution strategies in order to optimize certain goals such as decreasing the network load or decreasing the resource usage on certain types of devices.

Figure 2.5 shows an example for the execution of a query on two public data sources. Intuitively, steps one, two and three are decoupled in time, i.e. they must happen sequentially but the time period between them may vary.

As a first step, the public data sources announce their data by exporting associated metadata to the data discovery registry. Typically, this is done once the device starts up its semantic data storage and the announcement might be updated in cases where the data storage holds dynamic data that is reflected in the metadata. Intuitively, however, the update frequency of the metadata should be lower than the update frequency of the actual data in order to avoid scalability issues with the data discovery registry. Once a query is issued, for example, through the intent-aware user interface, the query processor receives it and interprets it. Based on its contents, it will then create and execute queries on the data discovery registry, which results in a set of possible data sources. Based on the strategy taken by the query engine, an appropriate query plan is generated and executed. For the execution, the query processor executes sub-queries against the necessary set of semantic data storages – via their local query processors – and returns the result to the intent-aware user interface.

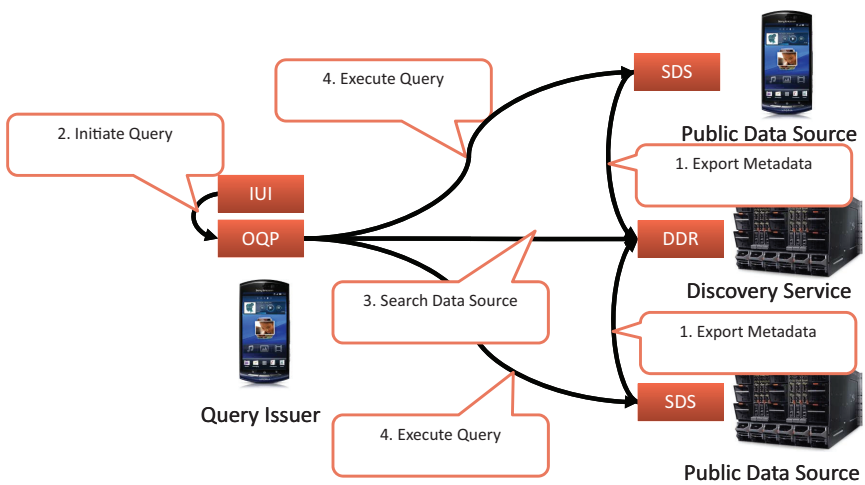

Figure 2.5 One-time Processing of Public Data.

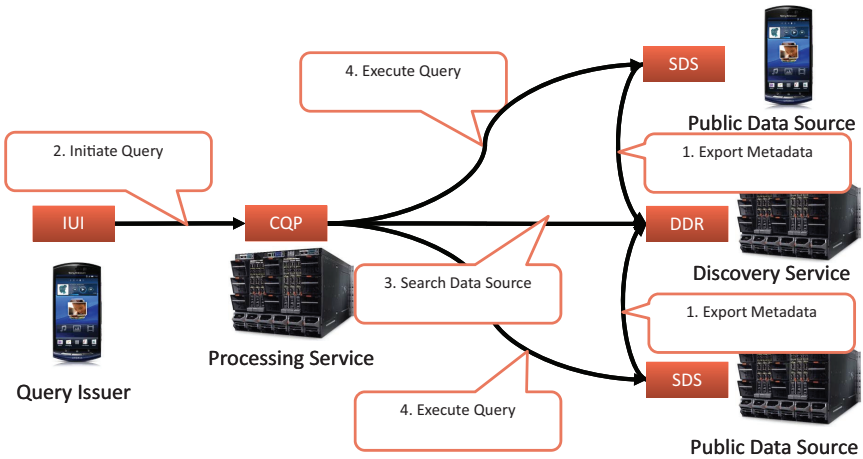

Figure 2.6 Continuous Processing of Public Data.

While the approach described above is sufficient to enable the processing of one-time queries, the execution of continuous queries over dynamic data raises additional issues. Due to the associated resource requirements, the GAMBAS middleware does not encompass a continuous query processor for all types of devices. Instead, the continuous query processor is only suitable for Traditional Computer Systems (TCS) and Backend Computer Systems (BCS). Consequently, it is necessary to handle continuous queries on other systems by means of a third-party system. For public data, this third-party system can be introduced easily. As depicted in Figure 2.6, the system simply acts as a proxy for query processing and there is no need to change the remaining interaction.

2.2.2.2 Processing of Shared Data

As indicated before, the processing of shared data cannot be handled in the same manner as the processing of public data due to the additional requirements on access control and encryption. Consequently, we need to modify and extend the previous interaction by introducing additional steps that take care of both. For this, the architecture foresees the following general process:

- **Export (Announcement, Publication):** As with public data, the first step is to announce the availability of data to other devices by means of exporting metadata. However, in contrast to public data, only the device identity will be exported in order to avoid privacy issues resulting from the export of private metadata. In cases where no privacy issues

result, other metadata could be exported as well in order to improve the performance of the query processor. Alternatively, it is possible to encrypt the metadata as described in Chapter 4.

- **Search (Lookup, Binding):** In the second step, which takes place before query execution, the data discovery registry is used by the query processor to find the relevant data sources by means of querying their identities.
- **Preparation:** The third step takes care of the creation of a view of the remote data that shall be shared with the device that executes a query. The view creation itself consists of a number of sub-steps. First, the identity and data requirements are forwarded to the privacy framework of the device issuing the query. Second, the privacy framework contacts the privacy frameworks on the devices hosting the shared data. For this purpose, the privacy framework performs a mutual authentication. Furthermore, the privacy framework executing on the devices hosting the shared data performs access control, which will eventually result in the creation of a view that represents the data that shall be visible to the requester. Thereby, it is noteworthy to mention that this view may modify the original data based on the level of access. For example, the device hosting the data might decide to generalize parts of the data or to make parts of the data inaccessible. Once the view is created, a secure token is generated, which can then be used to access the view. This token is returned back to the query processor.
- **Execution (Usage, Invocation):** In the last step, the actual query is executed against the view provided by the devices hosting the shared data. In order to access the view, the query processor provides the token to the shared data source and it uses an encrypted channel to transmit both the query and the result.

Figure 2.7 depicts this process with one device that is issuing a one-time query on two sources providing shared data. As described previously, the time between export and access of the device's identify information in the data discovery registry may be high since the device providing shared data will usually export its identity as well as other public and optionally encrypted metadata that does not raise privacy issues upon startup.

Following the general process described above, the devices hosting the shared data export their identity and non-privacy-critical metadata in a similar fashion, as public data sources will share their metadata. As depicted in Figure 2.7, the processing is then initiated by means of a query issued by

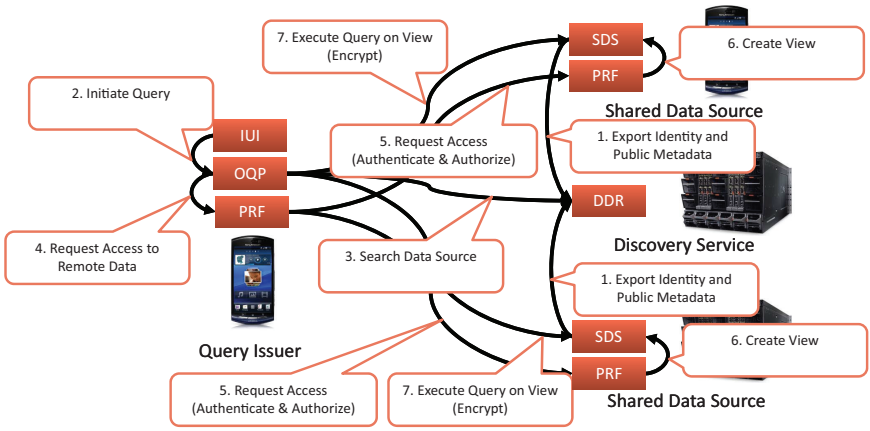

Figure 2.7 One-time Processing of Shared Data.

the intent-aware user interface. In order to access shared data, the query will typically specify the identity of one or more data sources whose connectivity information must be retrieved by searching through the data discovery registry in step two. Once the information is retrieved, the query processor will request the creation of the view through the local privacy framework. To prepare the view, the privacy framework on the query issuer contacts the devices hosting the shared data. Thereby, the privacy framework components on the devices will jointly perform request authentication and authorization. If this is successfully completed, the devices hosting the shared data will create the view on the data that shall be exposed to the query issuer. Thereby, they may perform arbitrary operations on the data such as generalizing information or removing information from the view on a per-request basis. Once the view is prepared, the privacy framework on the query issuer device will receive an access token enabling it to access the newly created view. This token is then passed back to the query processor, which will then issue the respective sub-queries to each of the data sources (again via the local query processors). Thereby, the whole transaction is encrypted and authenticated using the token. Once the sub-queries have been executed, the views on the devices hosting the shared data will be disposed and the result will be returned to the intent-aware user interface.

To enable stream processing on Constrained Computer Systems (CCS), the architecture mimics the proxy-based approach taken for public data where a remote processing service provides the hardware and software resources to perform queries. As shown in Figure 2.8, the primary difference between

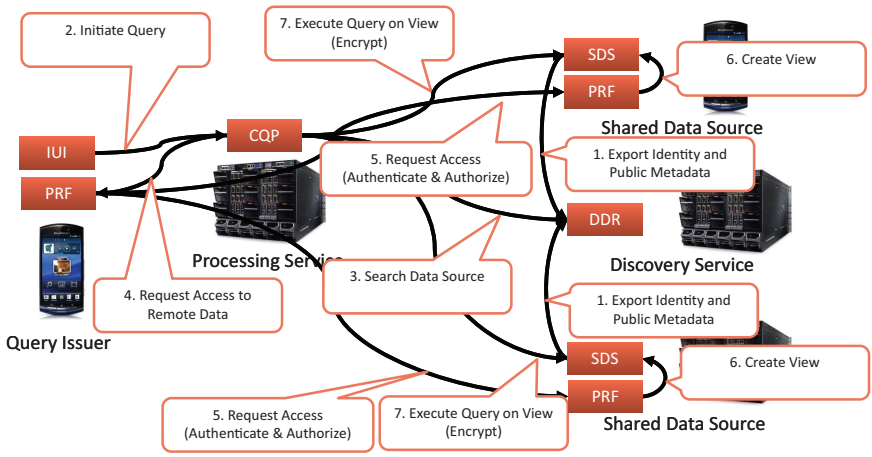

Figure 2.8 Continuous Processing of Shared Data.

continuous and one-time processing originates from the fact that a third party will be involved in data access. Consequently, this third party will have to be trusted by the shared data sources or they have to deny the request. In order to make the associated decision, the access request will not only have to authenticate and authorize the query issuer but it also has to decide upon the trust of the processing service. To do this, the request must identify the processing service that will be used during the processing. Furthermore, the remote communication between the actual query issuer and the continuous query processor must be secured accordingly. However, as described in Chapter 5, this can be done using regular cryptographic methods.

2.2.3 Inference View

Another key usage scenario of the GAMBAS middleware is the gathering of data from multiple sources in order to derive additional information. Based on the scenarios described in Section 1.3 and considering the acquisition and processing views described in the previous sections, we can identify two main classes of inferences, namely local and distributed inferences.

Local inferences are primarily based on information that is available to a single device. Thereby, the term local does not exclude the use of static public or shared data. A device can opt for locally storing a view of the static remote data to simplify the processing.

For dynamic data, however, such an approach is usually ill-suited – at least for devices that cannot host a continuous query processor – since the remote data would have to be refreshed continuously. Consequently, from an architectural perspective, the GAMBAS middleware realizes (simpler forms of) distributed inferences by means of executing continuous queries. Intuitively, the degree to which such inferences are possible depends on the capabilities of the query language. In the following, we briefly outline the approach taken to support local and distributed inferences.

2.2.3.1 Local Inferences

As indicated previously, local inferences encompass inferences on local data that is held by a device as well as public or shared static data that is available remotely and can be accessed by the device. Given the local availability of data, it is then possible to derive additional data using custom software that is executed by the device. Based on the application scenarios described in Section 1.3, the GAMBAS middleware supports two types of such local inferences which we outline in the following.

- **Personal inferences:** Personal inferences are inferences over the behavior data collected using personal data acquisition. They may entail the derivation of aggregated information from multiple sensors or the prediction of future behavior based on traces of the past behavior. In order to enable such an aggregation or prediction, the device itself may require additional static information. For example, in order to determine the typical bus stops that are used by a user, the personal mobile object of the user may have to retrieve the GPS coordinates of the bus stops. Similarly, in order to predict future trip destinations, it may be necessary to retrieve the street address of previously visited locations. From an architectural perspective, the GAMBAS middleware provides support for such personal inferences mainly by means of the adaptive data acquisition framework, which provides extensible support for data aggregation, behavior tracking and prediction. In addition, it is also possible to support this type of inferences at the user interface level in cases where the required inferences are highly application-specific.
- **Service inferences:** In addition to personal inferences which affect the data stored on the personal mobile objects, there is also a need to support service-specific inferences, which may be a result of the aggregation of data acquired collaboratively. As a simple example, it may be necessary to aggregate collaborative sensor readings in order to

derive an environmental map. Similarly, it might be necessary to assign the occupancy information collected by different users to different buses on a route in order to predict their remaining capacity. Intuitively, such aggregations should be handled by the service that collects the data since they may require the combination of data reported by multiple sources. However, since such aggregations are often highly application-specific, the GAMBAS middleware primarily supports them by providing data access by means of a generic query processor, which is detailed in Chapter 4. For aggregations that exceed the capabilities of this query processor, it is necessary to implement custom application logic.

2.2.3.2 Distributed Inferences

In contrast to local inferences which are limited to local data and static remote data, distributed inferences combine the dynamic data of multiple remote data sources. Considering the application scenarios targeted by the GAMBAS middleware, such inferences are necessary, for example, in order to detect the collocation of friends in the same public bus. Since we can assume that many users are not willing to publicly share their current location, such information is typically only shared among a specific and user-dependent set of people. To protect the privacy of the users, the GAMBAS middleware only stores this information on the personal mobile objects of the associated user. Thus, given that the user is willing to share this information, it must be retrieved from there.

As indicated previously, however, due to the resource constraints of many personal mobile devices, the GAMBAS middleware does not encompass a continuous query processor for these types of devices. Thus, there are only two ways of supporting distributed inferences. The first and simplest way is to perform the inference by means of a continuous query that is executed on some third-party system that is jointly trusted by the group of users. The second way to realize such inferences is to implement a custom service to perform the inference.

Both approaches have different benefits and limitations. The former approach does not require any custom implementation and thus, is easier to realize. However, given that the query language applied in GAMBAS may not support arbitrary application-specific functions, it is limited to the set of operators supported by the language. The latter approach does not suffer from this problem since arbitrary code can be used during the service implementation. However, in contrast to the use of an existing middleware component, it is much more complicated as it requires the development of

custom software. To realize the applications described in Chapter 6, it was sufficient to use a continuous query processor in order to perform the desired distributed inference; however, in a broader context, it may be necessary to provide additional custom services.

2.3 Interface Perspective

Given the static and dynamic perspective of the high-level architecture described in the previous sections, it is possible to identify the interfaces between the core building blocks that constitute the GAMBAS middleware. To do this, we take the different building blocks detailed in the component view of the static perspective, namely the data acquisition framework (DQF), the semantic data storage (SDS), the one-time and the continuous query processor (xQP), the data discovery registry (DDR), the privacy preservation framework (PRF) and the intent-aware user interface (IUI). Using these building blocks, we step through all interactions described in the dynamic perspective, namely the acquisition view, the processing view and the inference view. Consequently, we get the following interactions:

- **DQF–SDS:** In order to store contextual information for personal as well as collaborative data acquisition, the data acquisition framework needs to interact with the semantic data storage.
- **xQP–SDS:** To execute one-time as well as continuous queries over public or shared data, the one-time and continuous query processor needs to interact with the semantic data storages hosting the data.
- **xQP–xQP:** To execute distributed queries, the query processor needs to communicate with remote instances of itself to place subqueries there and receive results back.
- **xQP–DDR:** In order to determine the appropriate data source before executing a distributed query, the one-time as well as the continuous query processor needs to interact with the data discovery registry.
- **xQP–PRF:** When executing a distributed query over shared data, the one-time as well as the continuous query processor needs to interact with the privacy framework in order to gain access to the shared data.
- **PRF–DQF:** To protect the user from unwanted data collection, the access to the data acquisition framework is guarded by the privacy preservation framework. Thus, performing data acquisition requires interaction between these components.
- **PRF–PRF:** In order to gain access to shared data, the privacy framework must be able to interact with other instances of the framework remotely

to negotiate the appropriate access levels and to prepare the necessary views for querying.

- **IUI–xQP:** To retrieve data from the local storage or from remote services, the intent-aware user interface must execute distributed one-time or continuous queries over the associated data storages using the one-time and the continuous query processor.
- **IUI–PRF:** Although, the privacy preservation framework is supposed to enable the automated generation of a privacy policy, some users may want to control their sharing more tightly. In order to enable this, the intent-aware user interface enables manual configuration of the privacy framework.

In the following subsections, we describe the required functionality and the resulting interfaces in more detail. Thereby, we try to refrain from describing low-level implementation details. Instead, we rather focus on an architectural perspective by describing the types and flows of data that is interchanged through the interfaces. However, where appropriate, we also give some initial ideas on how this flow can be realized.

2.3.1 Storage Interfaces

DQF–SDS Interface: As detailed in the acquisition view, there are two basic types of data acquisition foreseen in the GAMBAS architecture, namely personal data acquisition (on behalf of the user) and collaborative data acquisition (voluntary, on behalf of a service). For both types of data acquisition, it is necessary to store the sensed data – either on the mobile device or on a remote device hosting the service. To perform the data acquisition, the GAMBAS architecture introduces the data acquisition framework (DQF) which is responsible for recognizing different types of context in an energy-efficient manner. To store data following the open linked data principles that are forming the core of the interoperability mechanisms provided by the GAMBAS, the architecture foresees a semantic data storage (SDS). In order to enable the persistent storage and the later retrieval of recognized context, it is necessary to introduce an interface between these two components.

Based on this rationale, it is necessary to support the insertion of data into the semantic data storage. In addition, in order to support the storage of transient states, it is also necessary to support the deletion of data from the storage. This allows, for example, the removal of stale entries. As described in detail in Chapter 4, the storages are using RDF as their internal data format.

Thus, the insertion and deletion functions or the storage use this format as well. As a result, the interface consists of the following two functions that can be called either locally (to support personal data acquisition) or remotely (to support collaborative data acquisition):

- **Insert (RDF Triple) Success :: Local & Remote:** Enables the insertion of an RDF triple into the semantic data storage by placing either a local or remote call and indicates whether the insertion has been completed successfully.
- **Delete (RDF Triple) Success :: Local & Remote:** Enables the deletion of an RDF triple from the semantic data storage by placing either a local or a remote call and indicates whether the deletion has been completed successfully.

As indicated previously, the architecture foresees the usage of the interface through the data acquisition framework by means of a special storage component that enables the application developer to define the data storage that will receive the insertion (or deletion) as well as the graph (i.e. the set of triples) that shall be generated (or updated). Once the data acquisition framework generates a new result and transmits it to the storage component, the storage component will perform a deletion of previously inserted triples (if desired) and execute an insertion of the newly created graph. If the insertion or deletion shall be executed on a remote system (to perform collaborative data acquisition), the same set of procedures shall be executed. To improve the overall performance of the interface, in particular, when executing insertions and deletions remotely, it is beneficial to support batch insertions and deletions. This can significantly reduce the latency of updates, especially when multiple triples have to be removed and inserted into a remote storage over a low-bandwidth connection (such as a GPRS link, for example).

2.3.2 Query Interfaces

xQP–SDS Interface: As explained in the components view, there are two types of query processors. One-time query processors (OQP) aim at executing one-time queries, i.e. queries that are evaluated against the current state of the data. One-time query processors are focused on more static data. For the dynamic data, continuous query processors (CQP) are in place. CQPs can monitor the input data coming from streams, and as soon as a new data item is generated, the CQP will evaluate the query and if new results are produced, they are then forwarded to the query initiator. In both cases, the data is stored

in semantic data storages and the goal of the query processors is to make the data from an SDS available to services and applications. Therefore, an interface between xQP and SDS is needed. Query processors should be added to retrieve data from an SDS that matches a query. Additionally, an xQP also servers as an interface to add or delete data in an SDS, for example, the data generated from the user intention interface.

For query optimization purposes, the query processor might make use of what we call "temporary data". Temporary data is used only during the query execution and can be discarded afterwards. As an example, the query processor might decide to temporarily store the public data in a local SDS to avoid remote calls during the processing. To keep the storage costs low, this data would be removed after the query is executed. For this, the query processor need to provide functions to add, retrieve and delete temporary data from an SDS.

Based on this rationale, it is necessary to support retrieving data from a semantic data storage that matches a query, as well as data insertions and deletions. This applies for both persistent data and temporary data. As described in detail in Chapter 4, the data retrieval is done via SPARQL queries. As a result, the interface consists of functions described below. Since the query processing supports the aggregation of data from different sources, the functions can be called either locally or remotely:

- **Insert (RDF Triple) Success :: Local & Remote:** Enables the insertion of an RDF triple into the semantic data storage by placing either a local or remote call and indicates whether the insertion has been completed successfully.
- **Delete (RDF Triple) Success :: Local & Remote:** Enables the deletion of an RDF triple from the semantic data storage by placing either a local or a remote call and indicates whether the deletion has been completed successfully.
- **Insert Temporary (RDF Triple) Success :: Local:** Enables the insertion of an RDF triple into a temporary graph in the local semantic data storage and indicates whether the insertion has been completed successfully.
- **Reset Temporary () Success :: Local:** Deletes all entries in the temporary graph.
- **Retrieve (SPARQL Query) result set :: Local:** Enables to query the local SDS for data items that match a given search pattern. To specify the search pattern, an SPARQL query can be used. Matching data items

are returned as a result set, containing all suitable bindings for each requested variable.

- **Retrieve Temporary (SPARQL Query) result set :: Local:** Enables to query the temporary graph of the local SDS for data items that match a given search pattern. To specify the search pattern, an SPARQL query can be used. Matching data items are returned as a result set, containing all suitable bindings for each requested variable.

The architecture foresees the usage of the interface as in most query processing systems. Since the SDS uses a graph data structure to store the RDF triples, the retrieval function works by finding sub-graphs on the SDS that matches the input query. The query processor takes care of parsing the input query to generate the execution plan. The processor also contains a component that inserts and deletes data from a data storage. To improve the overall performance of the interface, as in the case of the DQF–SDS interface, batch insertions and deletions are a useful optimization, especially on remote calls. Optimizations on the query execution plan, with the use of temporary data are also possible.

xQP–xQP Interface: In the GAMBAS architecture, some queries can only be answered by combining data from multiple sources. One solution would be to gather all data from the relevant sources in a single device and execute the query locally on that device. However, there are many problems with this approach. For starters, it would create a lot of data traffic, since it is not possible to know a priori which data is needed, therefore each source would ship all its data to a single device. Scalability would also be an issue, since the device executing the query would become a bottleneck. Finally, this approach does not preserve privacy and therefore becomes unsuitable for the GAMBAS framework.

Our solution is to equip each device hosting data with a query processor. Each query processor can execute queries locally over the device's data, and it can also aggregate results from multiples sources. For executing a distributed query among the devices, the query initiator first identifies the relevant sources using the DDR. It then breaks the query into subqueries. Each subquery is sent to the device that contains the data for it. The query processor on each device will then execute the subquery locally and only forward the relevant results to the query initiator (as opposed to all data). The query initiator merges the results of all subqueries and creates the final query result.

To execute distributed queries, the query processor needs to communicate with remote instances of itself to place subqueries there and receive results back. This is done by implementing an interface that allows query processors to post queries to remote query processors and retrieve the results, as shown below.

- **Retrieve (SPARQL Query) result set :: Remote:** Enables to query a remote xQP for data items that match a given search pattern. To specify the search pattern, an SPARQL query can be used. Matching data items are returned as a result set, containing all suitable bindings for each requested variable. This is used to place subqueries that are part of a distributed query.

The middleware architecture foresees the usage of the interface during the execution of distributed queries. Since the local execution is done over RDF triples, the interface between query processors is done using a language suitable for RDF, in our case SPARQL.

xQP–DDR Interface: To answer queries which involve remote data, the query processor must be able to discover the data sources that are available on the network. Once a query is issued, the query processor receives and interprets it. This allows the processor to identify which data is needed to answer the query (for example, the location of friends of a user). The data discovery registry contains the meta information about the data sources, not the data itself. This is to preserve the privacy of shared data. By consulting the registry, the query processor can obtain the list of sources that contain the data in question. For instance, the registry can return the list of semantic data storages from the friends' devices.

Based on this rationale, an interface between query processors and registry is needed. The interface must allow the processor obtain a list of remote SDSs (or endpoints) that contains a particular type of data. This interface requires only one functionally, which is given below:

- **Resolve (data source specification) endpoints :: Remote:** Enables the discovery of SDS endpoints that can be contacted for a specific kind of data, e.g. whom to contact to get information about a user's location. To do so, a data source specification is given, e.g. specifying the user for which data is searched (for instance, a friend). This request is sent to the remote discovery server and a set of matching endpoints is returned.

The interface functionality detailed is used during the processing of queries that involve remote data. The query processor identifies which data sources

are needed (e.g. the sources containing the location of friends), and request them to the registry. The registry then performs a lookup on the metadata it stores and returns the list of remote storages that matches the request. To improve the performance of the interface, and the performance of the query processing in general, the metadata stored in the registry can be enhanced in order to provide more accurate and results sets. However, storing more metadata might lead to privacy issues, so this needs to be handled carefully.

2.3.3 Privacy Interfaces

xQP–PRF Interface: During the query execution, the query processor identifies the sources needed to answer the query and then sends a request to the registry. The registry resolves the sources and sends the list of endpoints (remote storages) that contain that data in need back to the processor. For shared data however, before the query processor can access the data on the remote source, a privacy control is performed to check if the query initiator has the rights to access the data. A view of the data matching the privacy rules in place is created and shared with the query processor. This is done in the preparation phase explained in Section 2.2.2.2. The query processor forwards the identity and data requirements to the privacy framework, which in turn checks with the privacy framework of the device hosting the shared data. A view of the data is created based on the access control. The view can reflect the original data, or it can modify the original data according to the privacy in place. For example, it can aggregate or hide parts of the original data. Once the view is created, a secure access token is generated and sent to the query processor. If a remote endpoint is trying to access the shared data, the secure access token will allow transferring the shared data securely over the chosen communication channel.

Based on this rationale, an interface between xQP and PRF is needed to check whether the query initiator is allowed to access the data. Additionally, if the xQP is executing a remote query, the communication must be properly secured. For this, the user and data access credentials are sent over a secure proxy that is part of the communication subsystem of the middleware to provide a secure data connection between the two endpoints. As discussed in Chapter 5, the secure proxy manages the secure communication transparently. Thus, the interface does not include a method that enables the exchange of security tokens or start the encryption. In contrast to that, the access to data must be checked through an interface. The interface consists of one function that checks if the query initiator (i.e. the user requesting the data) is allowed

to access the data. The data being requested also needs to be specified. The PRF looks at these two input items and decides whether the query is allowed or not. Each request is handled by the privacy framework of each semantic data storage and therefore this function is performed locally. The function supported by this interface is given below:

- **Check (Set of Ontology Classes, Requester) allowance :: Local:** Enables to check with the PRF, if executing a received query is allowed according to the currently active privacy policies. To do so, the query processor hands the PRF (1) a set of classes in the GAMBAS ontology that specify what data types the query will access and (2) the origin of the query, e.g. if it was a local query or a query from a remote user. The PRF returns whether this query is allowed or not.

The architecture foresees the usage of the interface described above in the privacy-preserving query execution mechanism, when shared data is involved. The query processor must first interact with the privacy framework, which is responsible to allow or deny data access and responsible for data encryption/decryption.

PRF–DQF Interface: The Adaptive Data Acquisition Framework (DQF) enables the collection of data using various sensors built into the user's mobile device. The collected data can then be used personally (i.e. by the device, in the case of personal data acquisition) or collaboratively (i.e. by a remote service, in the case of collaborative data acquisition) to optimize services in a behavior-driven manner. Clearly, the data acquired by means of sensors built into the device of a user may raise privacy concerns. Furthermore, the preferences with respect to privacy may vary drastically from user to user. In order to empower users to exercise control over which data can be collected, the access to the data acquisition framework is guarded by the Privacy Preservation Framework (PRF). Thereby, all accesses made to the data acquisition framework are checked against the user's privacy preferences with respect to data collection. This allows the user to limit the data types that can be collected at all. In extreme cases, a user may limit the collection of all data through the GAMBAS middleware. In less extreme cases, the user may limit the collection of a particular type of context information, such as location-related information or audio information.

The PRF–DQF interface enables the data acquisition framework to check whether the user has given consent to the acquisition of a particular type of contextual information. To do this, the DQF performs calls to the PRF in order to verify that the data types that shall be captured are permissible under

the user's current preferences. Furthermore, since the user's preferences may change at any point in time, it is necessary that the PRF provides functionality to signal a change to the DQF whenever the user's preferences with respect to a particular data type change. Consequently, the interface must be composed of the following two functions:

- **Check (Datatype) authorize :: Local:** The PRF checks the data type that is about to be captured against the preferences of the user and returns a Boolean to indicate whether the user permits the acquisition of the specified data type. If the access is denied, the acquisition is aborted. If access is granted, the acquisition task can be started.
- **Signal (Datatype) void :: Local:** The PRF signals a change to the preferences with respect to a particular data type such that the DQF can check all currently executed data acquisition tasks against the updated set of preferences. If a data acquisition task is no longer permitted by the user, it must be aborted.

In order to guarantee that all data acquisition tasks continuously conform to the user's preferences, the architecture foresees the continuous and gapless usage of this interface for all calls to the DQF. This means that all tasks that are started within the DQF need to pass through the check method of the PRF with the associated data types. In addition, as long as the DQF is executing any tasks, it needs to react to changes indicated by the signal method. If a signaled change affects a data type that is currently acquired, the check for the associated (set of) task(s) needs to be reevaluated, possibly aborting any conflicting tasks. The check of the DQF against the policy managed by the PRF may entail some slight overhead, which may become significant if data acquisition tasks are started and stopped very frequently. In this case, it makes sense to cache the user's preferences in memory to reduce the associated overhead. However, in most usage scenarios, the overhead can be neglected.

PRF–PRF Interface: The PRF allows the transfer of data between two devices. The data that is transferred should be encrypted. The reason for this is twofold. At first, the data might contain private information that should not be shared with unauthorized users or devices. Additionally, the shared data might be transferred over an insecure communication channel (e.g. the Internet or an insecure WiFi network). To enable encrypted communication, it is necessary for both communication endpoints to use a cryptographic key. Using the efficient concept of symmetric encryption, the key must be identical

and exchanged before the secure communication can take place. During the exchange of a cryptographic key, the communication endpoints show that they are eligible to access the data that should be transferred by authorizing themselves. After the authorization process, both endpoints possess a shared cryptographic key that allows them to transfer data securely.

The PRF–PRF interface allows the authorization of communication endpoints. The successful authorization can be performed in two different ways. The first way uses asymmetric cryptography and is based on certificates, similar to the implementation of SSL in the Internet. This allows an ad-hoc identification of devices that belong to a certain domain. If the domain root is trusted, the authorization will be successful. Also, the access rights depend on the trust in this root. For authentication, the device's certificate is transferred together with a challenge that proves that the device is in possession of the certificate's private key. Together this data forms the device's credentials that are checked at the other endpoint. The alternative of using compute intense asymmetric cryptography is symmetric cryptography. Using symmetric cryptography, a key can be attached to a connection between two endpoints. The first half of this shared key allows the identification of the other endpoint. The other half can be either directly used for the secure communication or used to exchange a new session key securely. For efficiency reasons, both of these checks (i.e. for asymmetric and symmetric cryptography) are performed directly during the communication. The local interface is designed as follows:

- **Check (credentials, user pseudonym) authorize :: Local:** The PRF checks the security credentials of a user and returns a Boolean that shows if the user was authorized successfully.

The middleware architecture foresees the usage of the interface described above for every secure transmission of data. The communication endpoint must first authorize each other at the remote privacy preservation framework, before a key for the secure communication is computed. Intuitively, the authorization that is performed by the privacy-preserving framework incurs some overhead during the data transfer. However, without the authorization, the communication partner is unknown to another device and this contradicts the privacy of the transferred data. While the authorization therefore is a crucial mechanism, it is may be possible to use more lightweight security mechanism, resulting in a decrease of the security level, in application scenarios that permit this.

2.3.4 Control Interfaces

IUI–xQP Interface: The Intent-Aware User Interface (IUI) is connected to the GAMBAS middleware through a query processing interface, which provides access to local and remote data sources. Local data pertains, for example, to personal travel information, which may include the user's travel history for making predictions to adapt the IUI to his future travel behavior. Remote data could include transport information hosted by third-party services such as a city's local transport agency (e.g. estimated time of arrivals), time tables and information about the travel habits from the user's friends in the social network as stored on their mobile devices. Since all this data is represented based on linked data principles using RDF triples, it can be queried in a uniform manner by means of a powerful graph-based query language irrespective of what specific kind of data is requested and where this data is located.

When the IUI needs to access data, it uses an interface from xQP to connect to external services and read information objects. In particular, this interface is a facade from xQP that calls methods and translate the received data into an understandable format for IUI. As a result, the interface consists of a single power query processor, which allows us the IUI to specify generic queries over data stored on local and remote SDS:

- **Select (SPARQL Query) result set :: Local & Remote:** Enables the retrieval of bindings for requested variables. To specify the variables as well as conditions that bindings for them must match, an SPARQL select query can be given. A query can be executed on a single data source or multiple ones, allowing to query and integrate information from multiple users at once. Matching data items are returned as a result set, containing all suitable bindings for each specified variable.

Frequent data access may be a critical factor for the IUI, especially when the queries need to be forwarded to remote data storage over cellular network connections. The low bandwidth of these connections and high variance in quality of service may slow down the query process and cause significant delays in information delivery that can negatively affect the user's experience. In order to improve upon this, the design of IUI foresees a caching strategy, where some static data (e.g. routing information, bus coordinates, time tables) is kept on the mobile device so that no repeated updates are required. This is especially useful for transport network data, which is not expected to change very often. For dynamic data (e.g. arrival time or crowd level of a vehicle), possible optimization strategies include primarily pre-fetching,

where the data is retrieved before it is requested by the user. As the data is already available on the device prior to the access to the information, the delay experienced by the user can be minimized.

IUI–PRF Interface: The goal of the privacy preservation framework (PRF) is to protect the user's privacy by providing security mechanisms that enable the secure and authentic interaction between different devices. Thereby, access to different types of data is controlled by the privacy preservation framework on behalf of the user. To do this, the privacy framework relies on a policy that defines the user's preferences with respect to the sharing of data with other users. Although the privacy framework attempts to minimize the configuration effort for the user by deriving a suitable policy from the policies that a user is already applying on different social services, there might be cases where the user wants to exercise full control over the sharing of data. To do this, the privacy preservation framework exposes a configuration interface to the intent-aware user interface (IUI) that provides manual control over the sharing.

To exercise manual control over the sharing of information, the privacy preservation framework enables the intent-aware user interface to (re-) configure the privacy policy. Given that the main entities contained in polices are users and permissions on different data types that express that a particular user may access a particular type of data, it makes sense to expose functionality to manipulate these two entities. To enable the development of a visual representation of the user's current privacy policy, the functionality required to manipulate the policy is additionally augmented with functionality to simply retrieve the current policy. In summary, this results in the following six functions that are available only locally and that are only accessible to the intent-aware UI in order to avoid unwanted modifications.

- **ListUsers() usernames :: Local:** This function enables the intent-aware user interface to list the names of users that have been configured on a particular device. The resulting list of user names can be pruned or extended using the following two functions.
- **AddUser(username) void :: Local:** This function enables the intent-aware user interface to add another user to the list of users that have been configured for a device. If the user is already configured, the method simply returns. If the user does not yet exist, it will be added to the list.
- **RemoveUser(username) void :: Local:** This function enables the intent-aware user interface to remove a previously configured user from the list of configured users. If the user is not configured, the

method simply returns. If the user was configured, the user and all its permissions to access data on the device will be removed.

- **ListPermissions(username) datatypes :: Local:** This function enables the intent-aware user interface to view all the permissions that have been configured for a previously configured user. The list will include all data types to which the specified user will have access. If the user has not been configured, the list of data types will be empty.
- **AddPermission(username, datatype) void :: Local:** This function enables the intent-aware user interface to add a permission for a previously configured user such that the specified user will be able to access data of the specified type. If the user is currently not configured or the user already exhibits a permission to access the data type, the method simply returns. Otherwise, the permission will be added for the specified user.
- **RemovePermission(username, datatype) void :: Local:** This function enables the intent-aware user interface to remove a previously added permission on a specified data type for a specified user. If the permission or the user does not exist, this method simply returns. Otherwise, the permission will be removed and the user will no longer be able to access the specified data type.

The primary intended usage of this interface is the manual manipulation of the privacy policy through a graphical user interface on the device of the user. Thereby, it is important to mention that the access to this interface is intended to be restricted to an intent-aware user interface component that ships together with the GAMAS middleware and that it cannot be accessed through other components in order to avoid unwanted manipulations. Consequently, we envision the creation of one or more list views that show which user has access to what type of data as well as controls that enable the injection of changes to these lists.