Co-ODB: Integrating Support for Collaborative Information Systems into an Object Database

Alexandre de Spindler  
Center for Information Systems  
Zurich University of Applied Sciences  
CH-8400 Winterthur, Switzerland  
alexandre.despindler@zhaw.ch

Stefania Leone  
Semantic Information Research Laboratory  
Computer Science Department, USC  
Los Angeles, CA, 90089-0781, USA  
stefania.leone@usc.edu

Moira C. Norrie  
Institute for Information Systems  
ETH Zurich  
CH-8092 Zurich, Switzerland  
norrie@inf.ethz.ch

Abstract—Many modern information systems require some form of collaboration among peers. To address these needs, existing approaches tend to build a layer on top of traditional database technologies that implements a specific distribution model. In contrast, our aim was to integrate concepts into a database that would provide native support for data sharing while allowing developers the flexibility of configuring their own distribution models and collaboration logic. As proof of concept, we present Co-ODB, an object database that can be used to support a wide variety of architectures and modes of sharing. We describe how Co-ODB has been implemented and illustrate how it can be used to implement applications with different collaboration logics.

I. INTRODUCTION

Nowadays, there is a rich variety of information systems that require some form of collaboration among peers to support information sharing in distributed environments. These range from enterprise systems to novel forms of mobile services based on opportunistic information sharing [1]–[3]. Platforms and frameworks designed to support application development tend to be built as a layer on top of traditional database technologies and often focus on a particular architectural variant or model of information sharing.

Our aim was to investigate how a more general and flexible platform could be provided by integrating fundamental concepts for information sharing within an object database. In the resulting system, Co-ODB, applications consist of collaborating peer databases where peers can take on various client/server roles, and data can be exchanged and synchronised in flexible ways according to application requirements.

Co-ODB provides developers with simple, high-level abstractions for object persistence and sharing as well as an event-driven programming model. Through a clear separation of concerns between these three aspects of collaboration, it is easy for developers to define their own collaboration logic tailored to an application. Collaboration is based around the notions of shared collections and shared objects. The approach builds on previous research [4] that focused on supporting the development of mobile peer-to-peer applications through a shared collection construct. In this work, we have generalised and extended this concept to cater for more general variants of information sharing and we show by means of examples how it can support a rich variety of applications and architectures.

We discuss the background in Sect. II before presenting the main features of Co-ODB in Sect. III. Sect. IV provides details of shared collections and different modes of sharing. To show the generality and flexibility of the approach and how it can support more innovative forms of applications in addition to more traditional distribution models, we present two example applications with dynamic and variable forms of collaboration in Sect. V. Details of the implementation of Co-ODB are given in Sect. VI and concluding remarks in Sect. VII.

II. BACKGROUND

Although information systems may use different data models, sharing activities, network topologies and architectures, what they typically have in common is the fact that they realise some form of domain-specific collaboration protocol among the various system components [5]. Recently, there have also been a number of mobile applications emerging that offer a variety of novel services based on support for opportunistic information sharing [1], [2] as well as social context-awareness [6], [7]. Regardless of whether it is an enterprise information system or a mobile social application, these systems are based on the common abstraction of peers connecting to each other and sharing selected data. The collaboration logic of a system determines how and when data is shared by specifying whether data transmissions are triggered explicitly by user actions or implicitly as part of the system or network behaviour as well as whether shared data should be treated as independent or affiliated copies, similar to copy and reference semantics in programming languages. In addition, the network architecture and topology specifies whether some of the participating peers take on particular roles, such as clients or servers, and how they are connected to form a predefined or spontaneous, and durable or transient, network. A general platform or framework designed to support the development of collaborative information sharing would need to provide simple ways of defining the collaboration logic as well as high-level abstractions of lower-level facilities such as persistent data management, network or wireless peer discovery and data transmission.

Existing approaches to information sharing are generally implemented as a layer on top of traditional database technologies. For example, distributed databases provide a distribution layer on top of databases [8], but particular instances tend to be designed for specific architectures and modes of collaboration. Component-based [9] and service-oriented architectures [10] allow collaborative information systems to
be developed through the composition of system components or services, but this means that the collaboration logic is implemented at the architectural rather than the conceptual level and, as a consequence, it is more difficult to define it at design time and adapt it at runtime.

Tuple spaces were designed to support the development of collaborative information systems by providing a simple, but powerful sharing model [11]. In essence, data is represented as keys associated to values of arbitrary types. A tuple space is a collection of data that hides its physical storage and provides operations for inserting, retrieving and deleting tuples. Physically, tuple spaces may abstract from one or more nodes, which is transparent to the developer. More recently, a tuple space was used as a distributed data model in mobile ad-hoc networks [12]. This work highlighted the limitations of tuple spaces since the asynchronous nature of mobile collaboration required complex event specification to be encoded as tuples alongside the tuples representing the application data. This mix of data and metadata hinders a clear separation of concerns.

Object databases such as Objectivity/DB\(^1\), ObjectStore\(^2\), Versant\(^3\) and db4o\(^4\) integrate application development and persistent data management with the aim of reducing developer effort by eliminating the need for them to manage a mapping between the application model and the storage model. Underlying these systems is the principle of orthogonal persistence [13] which requires that any type of object can be made persistent and objects are manipulated uniformly independent of whether they are persistent or transient. In line with the principles of orthogonal persistence, we define orthogonal collaboration as the principle of uniform treatment of objects independent of if and how they are shared.

In summary, existing development platforms focus on providing the means to specify the structural and operational components of a system. Collaboration is realised by means of extended data type definitions and additional manipulation operations which does not allow for a clean separation of collaboration aspects from data definition and manipulation aspects. Moreover, the structural and operational components are defined at design time, and their definition typically remains static during runtime. Therefore, the specification of collaboration logic cannot be adapted dynamically at runtime, unless developers themselves implement dynamic adaptivity.

III. Co-ODB

Like all object databases, Co-ODB provides developers with an integrated programming environment with facilities for managing persistent application objects. Co-ODB has specifically been developed for Java applications, although the concepts could be adopted for other object-oriented language environments.

The basic facilities offered by an object database are the opening/closing of databases together with mechanisms for making objects persistent and retrieving objects from the persistent store. In Co-ODB, a database can be created by first creating a database management object and then creating a database. The following code shows how this is done and opens the database.

```java
DatabaseManager dbms = new DatabaseManager();
dbms.createDatabase("LinksDatabase");
Database db = dbms.open("LinksDatabase");
```

Assume that a class to represent links is defined as:

```java
class Link { URL url; String title; String description; }
```

Link objects can be made persistent by creating a database collection, with the name “Links” and member type Link.class, that will be used as a container.

```java
Collection links =
    db.createCollection("Links", Link.class);
```

The following code then creates a link object and makes it persistent by adding it to the database collection before committing the database.

```java
Link adele =
    new Link("http://youtu.be/hLQl3WQQoQ0");
adele.setTitle("Adele - Someone Like You");
adele.setDescription("Music video by Adele" + "performing Someone Like You");
links.add(adele);
```

In line with the notion of persistence by reachability [13], any objects referenced by objects that are added to a database collection will themselves be made persistent. Note that when a database is opened, the database collections that act as the roots for persistence can be retrieved either by name or using the query facility. Generally, queries are defined by constructing a query tree which is then executed against a collection. This is similar to query facilities offered by other object databases and it is beyond the scope of this paper to describe the query facilities in detail, but the following example illustrates the approach:

```java
QueryNode<Link> q = new Selection<Link>(
    "title", "Adele", Relation.CONTAINS);
Collection<Link> result = links.retrieve(q);
for (Link current : result) {
    System.out.println(current.getTitle());
}
```

Finally, a database is closed using `db.close()`.

The transaction mechanism in Co-ODB is a workspace abstraction. A database instance always has an ongoing transaction which ends when committed or rolled back and a new transaction is started automatically.

The features presented so far are also supported by current commercial object databases such as Versant, Objectivity/DB, ObjectStore and db4o, although there are variations in the details of the mechanisms used and how they are made available to the programmer. Most object databases also integrate support for distribution with a view to supporting scalability and fault tolerance by allowing a single logical database to be physically distributed. In contrast, our aim is to support cooperation between multiple, autonomous databases.

\(^1\)http://www.objectivity.com
\(^2\)http://www.progress.com/objectstore
\(^3\)http://www.versant.com
\(^4\)http://www.db4o.com
by providing high-level concepts for data sharing that allow developers to configure the collaboration logic according to application requirements, and even update these dynamically at runtime for maximum flexibility.

Co-ODB supports data sharing by means of shared collections which are database collections with the same name and set available for sharing in multiple databases. For example, assume a simple client-server scenario where link objects created and stored locally in client databases are shared with other clients through a server database. Then a collection “Links” could be created on each client as well as the server with each set available to receive shared data objects:

```java
links.setAvailable();
```

Members of a client collection filtered by the query above could be sent to the server and added to its corresponding collection using the following code:

```java
Peer server = new Peer("abe.ethz.ch", 3141);
links.share(result, server);
```

Clients willing to receive shared links would need to register with the server, providing their host name and a port number. Given these values, the following code executed on the server would share all members of the server collection with the specified client:

```java
Peer client = new Peer("caw.ethz.ch", 3141);
links.share(client);
```

Note that if the share method is used with only a peer as argument then all members will be shared, but optionally a subset can be specified as in the previous example. Both arguments may be single- or multi-valued.

To automate sharing between databases, a neighbourhood is created for a collection and an event handler added to automatically transmit data to members of the neighbourhood when the associated event is triggered. For example, the server can be added to the neighbourhood of a client collection using

```java
links.neighbourhood(server);
```

and then a handler created and registered to handle the event of a link object being added to the collection:

```java
class AddHandler implements Handler {
    Collection links;
    AddHandler(Collection links) {
        this.links = links;
    }
    void handle(Object member) {
        this.links.share(member);
    }
}
```

The handling consists of sharing the newly added link object with all peers in the collection neighbourhood, which is the server in this case. On the server, a neighbourhood of all clients could be created and a handler to automatically forward all received link objects:

```java
class ReceiveHandler implements Handler {
    ...
    CollectionEvent handleCollectionEvent() {
        switch (event) {
            case ADD:
                this.links.forward(member, source);
                break;
            case REMOVE:
                this.links.unforward(member, source);
                break;
        }
    }
...
```

... // as above

The forward method is similar to share, but a second argument specifies members of the neighborhood to be excluded. In this case, we exclude the source client that shared the link object with the server as specified by the Peer parameter to handle. Moreover, since Co-ODB collections distinguish the addition and receipt of data objects, links received by the clients from the server will not be sent back to the server.

The definition of shared collections, their neighbourhoods, handlers as well as queries specifying members to be shared form parts of what we refer to as the collaboration logic. While the business logic uses shared collections as regular bulk data structures, the collaboration logic is specified using an extended interface unknown to the business logic, thereby ensuring a clear separation of concerns. Figure 1 provides an overview of the extended collection API.

![Image](Fig. 1: Collection API supporting collaboration)

Not all methods are shown, but we indicate the various components that make up the API by grouping methods into boxes. The top component represents the standard collection API consisting of addition, removal and member traversal operators. The second component represents an event handling service. The third component is used to specify that a collection is persistent. The bottom component is used to specify the collaboration logic. In the next section, we will examine the different sharing modes in detail before presenting examples to demonstrate how they can be used to configure a wide variety of applications in Sect. V.
IV. SHARED COLLECTIONS

As seen in the previous section, Co-ODB is essentially an object database extended with the notion of shared collections to support the configuration of collaboration logic at the conceptual level. We will now examine in more detail the different sharing modes that it supports.

![Figure 2: Collaboration driven by collection memberships and member updates](image)

Figure 2 shows two peers A and B, a shared collection \(C(T)\) residing on each of them, and an indication of the collection members in curly brackets. Four situations \(t_1, \ldots, t_4\) occurring consecutively are aligned from left to right. At all times, the peers are connected to each other, which is indicated by the grey area between them. Situation \(t_1\) depicts the initial period. The collection residing on peer A contains the members \(\text{oid}(3), \text{oid}(5), \ldots\) and the one on B the members \(\text{oid}(4), \ldots\). In \(t_2\), a new object \(\text{oid}(6)\) is added to the collection residing on A. This triggers the sharing of this new member with B as indicated by the annotated arrow in the grey area. Note that the fact that a collection is shared between peers does not mean that they necessarily have the same members as the collaboration logic may specify a filter so that only certain objects will be shared.

Assume there is an update of \(\text{oid}(6)\) on B in \(t_3\) and the collaboration logic specifies that the instances on the different peers should be synchronised. Then an update propagation to A is triggered resulting in the updated version of \(\text{oid}(6)\) on both A and B as indicated in \(t_4\).

![Figure 3: Collaboration driven by connections and disconnections](image)

Fig. 3 depicts another form of collaboration where the establishment of connections and disconnections drive the process. Note that a third peer C participates in this scenario. Again, the grey areas between peers indicate the state of their connection. In \(t_1\), none of the peers are connected. In \(t_2\) and \(t_3\), the peers A and B are connected, and in \(t_4\) they are disconnected. The peers B and C are connected in \(t_3\) and \(t_4\).

We first explain the collaboration between peers A and B. Due to the establishment of a connection in \(t_2\), we assume that all members of the collection residing on A are transmitted to the collection residing on B and vice versa. As a result, the transmitted members appear in the collections residing on the receiving peers as seen in \(t_3\). In \(t_4\), the connection between A and B is closed. In such a situation, it is possible that the sharing was transient and transmitted objects are deleted, or, persistent in which case they remain in the receiving collections. Assume the collaboration logic is such that sharing from A to B is persistent, while sharing from B to A is transient. Then the objects received from A remain in the collection on B, while those received from B are no longer in the collection on A.

The collaboration between B and C follows the same pattern. As soon as a connection is established, all collection members are shared. Since C started with an empty collection, no members are sent to B, while all members of B appear in the collection residing on C, including the ones previously received from A.

Shared collections are characterised by their name and member type. We will use the form \(\text{Name}\) to refer to a collection by its name and \(\text{Name}(T)\) if the member type is of importance. On all peers, the collaboration configuration for the collection \(C(T)\) specifies that it is connected to collections with the same name \(C\) and compatible member type \(T\) residing on other peers. The member types are compatible if they are the same or if \(T\) is a supertype of \(T\). Shared collections provide \(\text{share}\) operations for the sharing of a single, multiple or all members, each with a single or multiple peers. Also, \(\text{setAvailable}\) operations are used to turn on the availability to receive members, which can be unconstrained or restricted to a single or set of peers.

Table I summarises the collaboration configurations and operations, their arguments and example values. Data filters use queries to specify which members should be shared. A trigger specifies when sharing processes are initiated. Sharing processes may be triggered by user actions, collection updates or peers appearing in physical proximity. The sharing mode specifies whether shared members should be transferred with copy or reference semantics, and if data should persist after disconnection. Finally, each shared collection is associated with a set of peers that form its neighbourhood. When members are shared, they may be shared with one particular peer or be broadcast among all peers in the neighbourhood. This neighbourhood supports the definition of application-specific network architectures and topologies such as client-server and peer-to-peer, simply by connecting collections in terms of neighbourhood relationships. In the case of a mobile

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Arguments</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Filter</td>
<td>Query</td>
<td>Members created locally</td>
</tr>
<tr>
<td>Trigger</td>
<td>Event, Condition, Action</td>
<td>User action, Data creation or manipulation</td>
</tr>
<tr>
<td>Sharing Mode</td>
<td>Transmission Semantics</td>
<td>Connection state, Copy, Reference</td>
</tr>
<tr>
<td>Neighbourhood</td>
<td>Peers</td>
<td>Peer B on A and C, Peers A and C on B</td>
</tr>
</tbody>
</table>

Table I: Shared Collection Configurations and Operations
environment with ad-hoc connections, another set of peers containing all peers currently in physical proximity serves as a global neighbourhood for data broadcast or as another trigger for data transmission.

All of these configurations may be set when a collection is made available and applied automatically whenever members are sent and received. They may also be set or updated at runtime. It is also possible to set them for a single share process, overriding previous settings in the scope of a single transfer process.

V. FORMS OF COLLABORATION

We now demonstrate the flexibility of shared collections by means of two example application which implement different forms of collaboration. These forms differ in terms of the roles taken by participating Co-ODB instances such as clients and server or equal peers. They also differ in terms of the sharing mode including combinations of persistent or transient and copy or reference semantics. Finally, in the first application, sharing processes are triggered by data creation or manipulation operations, while in the second, the sharing processes are triggered by connections being established.

A. Data-Driven Sharing

Consider the example shown in Fig. 4 where a collection “Travel News” contains members representing news about underground stations or lines in London and resides on a travel server.

News
Oxford Circus
"Reduced escalator service until late June 2011"

Neighbourhood
Mary
Fred

Travel Server

Fig. 4: Travel News Application

The News class and its subclasses StationNews and LineNews would be created in the same way as is normally done in Java. However, for the sake of brevity, we will only show attribute declarations.

class News { String news; }
class StationNews extends News { Station station; }
class LineNews extends News { Line line; }

The attribute news represents news items such as “Reduced escalator service until late June 2011”. Using the subclasses, such news can be related to underground stations or lines.

Next, a collection travelnews is created and set available.

SharedCollection<News> travelnews =
    db.createCollection("TravelNews",News.class);
    travelnews.setAvailable();

The sharing of members is triggered by the addition of a news object in the server collection. This is implemented as an AddHandler registered with the travelnews collection for member addition events similar to the handler shown in Sect. III. Consequently, as soon as a news object is added to the collection, it is automatically shared with all peers in the collection neighbourhood.

A user registers with the travel server in order to receive news items on their mobile device. As a result of the registration process, a peer object representing the registered user is added to the collection neighbourhood shown in Fig. 4 as described in Sect. III.

Shared objects should be transmitted to client peers with copy semantics, since there is no need for the sharing service to track and propagate updates. This is specified with the invocation of the mode method and the corresponding Mode.COPY argument.

travelnews.mode(Mode.COPY);

News objects are created and added to the server collection as shown below. Note that we assume the existence of an object representing the underground station, referred to by the variable oxfordCircus in this case.

News item = new StationNews(oxfordCircus, "Reduced escalator service until" + "late June 2011");
    travelnews.add(item);

As a consequence of being added to the collection, the news object will be propagated among all registered users.

On the client-side, a collection with the name “Travel News” is also created, using the same code as for the travel server. However, members of collections residing on client peers are not shared with the travel server. Therefore, there is no need to specify a sharing trigger or a neighbourhood. Since members received on the client peer should remain in the collection when disconnected, the sharing mode is set to be persistent as follows:

news.mode(Mode.PERSISTENT);

As a result, the collections residing on client peers contain the news objects they receive as indicated in Fig. 4. The user interface component of the application is registered to handle the collection member additions in terms of a pop up message containing data extracted from the news object, hence the users will see the news objects when they are received.

We now show how the application is extended to provide users with up-to-date travel itineraries. Fig. 5 shows itinerary objects which are managed in a collection travelplans residing on the travel server and client peers. The Itinerary class and travelplans collection are created as follows:
class Itinerary { Station start; Station end; Map<Station, Line> itinerary; }
SharedCollection<Itinerary> travelplans = db.createCollection("Travel Plans", Itinerary.class);
travelplans.setAvailable();

Users create itinerary objects on their mobile device by specifying start and end stations. The collaboration logic of the travelplans collection is configured so that newly created itinerary objects are shared with the travel server which executes a query to generate the itinerary and updates the itinerary attribute accordingly. This update is propagated back to the client peer that originally created the object. By saving the itinerary on the server, any disruptions to the network will cause the queries to be re-executed on the server and automatically propagated to the clients as updates to the itinerary object. In this way, users will always have access to the latest route information.

On the client device, the sharing process is triggered by itinerary objects being added to its “Travel Plans” collection. The handling consists of invoking the share method with which the newly added member is sent to the travel server. Since objects are not updated by the client peers and client collections do not receive members from the server, the collection is simply configured to share members with copy semantics. Since clients do not receive objects, there is no need to specify whether objects received should be kept or removed upon disconnection.

On the server peer, the addition of received itinerary objects triggers the execution of a query to generate the itinerary. A handler is created for additions with the following handle method implementation. Note that we assume an external service Planner.compute which generates the itinerary represented as a sequence of intermediary stations and associated lines.

void handle(Object member) {
  Map<Station, Line> itinerary = Planner.compute(member.getStart(), member.getEnd());
  member.setItinerary(itinerary);
}

The collection residing on the server peer is configured to share its members with reference semantics so that itinerary updates are propagated back to the clients. The sharing mode is also set to keep objects persistently. The reason for persistence is that updates to itinerary objects may be triggered by future travel news affecting them. Therefore, a second handler for addition events is registered on the travelnews that selects all affected itinerary objects and re-executes the queries. The resulting updates will then be propagated to the client peers.

B. Connection-Driven Sharing

To demonstrate the generality and flexibility of Co-ODB, we will show how more novel forms of opportunistic information sharing based on ad-hoc network connectivity can also be supported. Applications include the idea of attaching information to locations as virtual post-its [14] and we will consider the specific example of tourists posting their reviews of restaurants, museums etc. for later access by other visitors to the same place. The application is implemented in a purely distributed way by sharing information between the mobile devices of the tourists and computers at the participating places of interest. Data transmission is triggered by the establishment of connections between peers, in this case a mobile device and a fixed computer, based on physical proximity. Figure 6 illustrates two consecutive situations in a scenario involving a restaurant peer Bistro X and two mobile peers Mary and Fred.

void handle(Object member) {
  Map<Station, Line> itinerary = Planner.compute(member.getStart(), member.getEnd());
  member.setItinerary(itinerary);
}

A collection reviews resides on all participating peers and is defined as follows:

class Review { Peer user; String review; Peer place; }
SharedCollection<Review> reviews = db.createCollection("Reviews", Review.class);
reviews.setAvailable();

The collaboration logic on the mobile peers differs from the stationary peers hosted by places of interest. As shown in Fig. 6a, when Mary appears in physical proximity to the restaurant, all reviews contained in the collection residing on the restaurant peer are sent to her collection as indicated by the arrow labelled a. This sharing is based on the system-defined “Vicinity” collection which resides on all peers and contains currently nearby peers at any point in time. Proximity detection can be realised in different ways including the use of short-range Bluetooth and WiFi connections or by periodic

Fig. 6: Shared collections driven by connection states
comparisons of absolute geographical positions. When a peer is added to the “Vicinity” collection, a handler registered for addition events will cause the reviews to be shared with this peer. Since reviews authored by the recipient do not need to be shared, the following query is defined as the data filter:

```java
Predicate user =
    new AttributeValue(Review.class, "user", Actor.RECIPIENT);
Predicate not = new Not(user);
Query select = new Selection(reviews, not);
reviews.filter(select);
```

Since data is shared with any tourist peer appearing in proximity, there is no need to specify a collection neighbourhood. Review objects are persistent on the restaurant server. We assume that tourists may change their opinion over time and, therefore, reviews are shared with reference semantics. Whenever a review is modified, updates will be propagated to all peers where it is contained in the local collection.

```java
reviews.mode(Mode.PERSISTENT);
reviews.mode(Mode.REFERENCE);
```

Before Mary leaves the restaurant, she creates a review "Great Menu!" on her mobile device as shown in Fig. 6a. We assume that a variable named place has been assigned to the peer object representing the restaurant currently in physical proximity to Mary.

```java
reviews.add(new Review(Actor.THIS, "Great Menu!", place));
```

The review will be added to the collection and sent to the restaurant peer, as indicated with the arrow labelled b. This behaviour is realised by means of an addition handler registered with the reviews collection.

Figure 6b shows the situation where Mary is no longer connected to the restaurant peer, and Fred has appeared. Fred receives all reviews including the one created by Mary. Since Mary has left the restaurant, all reviews from there are removed from her collection. This is achieved by setting the sharing mode of the collection residing on the mobile peers to transient. To support the propagation of modifications to review objects, the collection is additionally set to share its members with reference semantics.

The application that we have described allows tourists to share reviews by attaching them to physical locations, but note that it would also be possible to allow tourists to share reviews based on their proximity. We have proposed such a recommender system previously [3]. The important thing to note in this context is that such a system could also be implemented in Co-ODB by defining an alternative collaboration logic to the one defined above.

VI. IMPLEMENTATION

Co-ODB was implemented by integrating the concept of collection services [15] together with metamodel extension modules [16] into an object database OMinho [17]. Consequently, shared collections are provided as a native database facility rather than a layer on top of it. We will describe metamodel extensions and how they were used to implement Co-ODB.

The key enabler of the metamodel extension mechanism is the collection API shown in Fig. 7. This API enriches common object database metamodels where classes declare attributes and methods, while objects contain values for these attributes. The enrichment consists of the notion of a collection, which triggers events for actions such as the addition or removal of members, and which can be bound to services extending the collection functionality.

```java
Collection<T>
```

```java
add(T)
remove(T)
iterator(): Iterator<T>
addService(Class<V>): V
getService(Class<V>): ... Collection<T>
CollectionEvent
handle(T)
Handler
```

Fig. 7: Collection API

Services are implemented in terms of regular classes whose instances can be attached to collections. A service may handle collection events and access collection members. Services may be invoked explicitly through user or programmatic interaction, or implicitly as part of event handling. Moreover, services may be invoked in the context of a single or multiple members, which are passed to the service as invocation arguments.

For example, data persistence is implemented with a service class Persistence which encapsulates all aspects of storing, updating and retrieving members using the database. A collection can be bound to the persistence service and made persistent as follows.

```java
Collection<...> collection =
    new Collection<...>("...");
collection.addService(new Persistence(db));
collection.getService(Persistence.class).setPersistent();
```

In general, services can be attached to a collection by means of an addService operation, which results in an extension of the collection API. Once attached, the service can be retrieved using the getService operation, in order to make use of the API extension. In the case of the persistence service, such an API extension consists of a method setPersistent, which turns a collection into a persistent collection.

Internally, the persistence service has the ability to add, retrieve, manipulate and remove collection members in persistent storage. When this service is attached to a collection, the service is automatically registered for addition and removal events, as well as for attribute update events on each collection member. Since services can be attached and detached dynamically at runtime, collections may be made persistent and transient dynamically.
In order to make a collection a shared collection, it is extended with a sharing service using code analogous to the one shown above. Note that the Shared Collection class shown in Fig. 1 is a subclass of the Collection class shown in Fig. 7. Instances of this subclass are returned by the createCollection method of the database and they simplify programming by encapsulating the persistence and sharing services and providing the API extension as part of its own API.

Information management facilities often require multiple classes and collections in order to be realised. For example, the information sharing facility demonstrated in Sect. V requires the domain class Peer, the collection Vicinity(Peer), the service class Sharing and additional service classes for the detection of peers in physical proximity and the implementation of the sharing modes. In general, information management facilities, such as data persistence and information sharing are grouped into modules, that consist of domain classes, collections and service operators. A module $M_k$ is defined as a triple $M_k = \langle DC_k, SC_k, C_k \rangle$ of service domain classes $DC$, service operator classes $SC$ and collections $C$. Typically, collections in $C$ manage instances of the domain classes in $DC$ by means of high-level operations provided by the classes in $SC$ and hiding the low-level operations required internally.

OMinho is an object database that makes use of this module concept in order to provide a modular extension mechanism. Standard object data management facilities are provided by a core module defined as:

$$M_{core} = \{\{\text{object, class, collection, query, event, handler, module}\},$$

$$\{\text{persistence, querying, event handling, module management}\},$$

$$\{\text{Objects, Classes, Collections, Modules}\}\}$$

The core module features a Modules collection extended with the Module Management service. This supports the dynamic definition, deployment and usage of additional modules at runtime, where each module extends the core module with additional classes and collections. Co-ODB was implemented with the sharing module providing the following domain classes, service classes and collections:

$$M_{sharing} = \{\{\text{peer, actor, mode}\},$$

$$\{\text{sharing, vicinity}\},$$

$$\{\text{Shared Collections, Available Collections, Vicinity}\}\}$$

Details about this module, its services and collections are provided in [4], [18].

The resulting collections and services are shown in Fig. 8 with collections shown as rectangles and the attached service classes relevant in the scope of this paper as oval shapes.

VII. CONCLUSIONS

We have shown how information sharing between peers may require many different kinds of collaboration logic depending upon the application requirements. To facilitate application development, we made the case for general frameworks and platforms capable of defining the collaboration logic orthogonally to other aspects of information management. As proof of concept, we have developed Co-ODB, an object database with integrated support for developing a wide variety of collaborative information systems ranging from more traditional distributed systems to novel mobile social applications. We have shown how this was achieved through the concept of shared collections to which various services for sharing and persistence can be attached dynamically. The generality and adaptivity of the approach has been demonstrated by showing how two very different application scenarios could be implemented using Co-ODB.

REFERENCES


