Offline Business Objects: Enabling data persistence for distributed desktop applications

Pawel Gruszczynski, Stanislaw Osinski, Andrzej Swedrzynski

Poznan Supercomputing and Networking Center
Poznan Noskowskiego 10, 61-704, Poland
{grucha,stachoo,kokosz}@man.poznan.pl

Abstract. Despite the overwhelming popularity of web-based distributed systems, in certain areas the traditional desktop applications still seem to be a better choice. In this paper we examine the challenges involved in developing a secure and fault-tolerant data persistence layer for distributed desktop applications. We analyse the currently existing persistence frameworks and show why they do not meet the requirements set by certain distributed applications. The body of this paper concentrates on the Offline Business Objects framework which aims to fill this gap. The framework introduces the offline operation paradigm, whereby client applications operate without a permanent server connection and only periodically synchronise their data with the central database. We successfully deployed a distributed information system based on this paradigm and the Offline Business Objects for five major cities in Poland, which confirmed the practical value of our approach.

1 Introduction

The increasing availability of broadband networking infrastructure is an attractive incentive for software architects to abandon the traditional desktop application model of software delivery in favour of the thin client Application Service Provision [7]. One reason for this is that a great majority of today's distributed application frameworks, e.g. data persistence frameworks, have been designed with web-based systems in mind and cannot be directly used in desktop software.

In areas, however, where important are such factors as high security standards, fault-tolerance or responsive user interfaces, the desktop application model is still a viable approach [1]. Unfortunately, with the lack of adequate software frameworks, the main challenge involved in developing a secure and fault-tolerant distributed desktop application becomes the implementation of a suitable data access layer.

In this paper we describe the design and implementation of the Offline Business Objects (OBO) framework, which aims to fill the gap in data persistence for distributed desktop applications. In section 2 we briefly describe the High School On-line Admission System called *Nabor*—an application that defined the requirements for OBO. In section 3 we analyse the currently available data persistence frameworks and explain why they cannot be directly integrated with distributed desktop applications. Section 4 presents the design and implementation of Offline Business Objects, emphasising problems that are not addressed in the currently available frameworks, while section 5 briefly evaluates the framework. Finally, section 6 concludes the paper and gives directions for future work.

2 Requirements

The requirements for Offline Business Objects framework were driven to a large extent by the requirements for the High School On-line Admission System called *Nabor*[1], which we developed and deployed for five major Polish cities in 2003 and 2004 [13]. The main objective of *Nabor* was to provide high enough a level of co-ordination between all high schools in a city as to reduce the time and effort involved in the admissions process. The key idea was to gather all necessary data (e.g. candidates' marks, admission limits set by schools) in a central database and design an algorithm that would automatically generate admission lists for all schools in the city.

To comply with legal regulations governing high school admissions, the architecture of *Nabor* had to meet a number of technical requirements:

High security standards A great majority of data processed by *Nabor* (e.g. candidates' marks) was of a confidential character and had to be carefully protected from unauthorised access and modification. Also, the system had to be designed in such a way as to minimise the possibility of manipulating the admission results and to enable tracing possible manipulation attempts.

Fault-tolerance High school admissions procedures divide the whole process into several phases and impose strict deadlines on each of them. For example, one of the final phases is entering the candidates' marks to the system, which must be completed by all schools within a time span of two or three days. Therefore, *Nabor* had to be able to operate even in case of a temporary failure of the central database or the network infrastructure.

Varied user group An important characteristic of *Nabor* was also the fact that it required active involvement of all parties of the admissions process. High school candidates used the system to find out about the offered schools, fill in and print out the application form. School administration used *Nabor* to enter the candidates' data and download the admission lists once the admissions had been closed. Finally, for the local education authorities the system provided a range of analytical reports, which aided global planning.

Having analysed the possible architectural designs [13], we decided that in order to fulfill all the requirements, *Nabor* should be implemented as a client-server system, with client nodes running offline desktop applications. In this setting the client application would normally stay disconnected from the central server and

[1] *Nabor* is the Polish word *admissions*

perform all operations on locally cached data. Occasionally, a connection would be established to synchronise the local data cache with the central database.

We also made a decision to implement the *Nabor* system using Java technology. Employing Java Swing [19] on the client side would make the desktop application platform-independent, while using non-proprietary Java technologies on the server side would reduce the cost of the whole undertaking.

As a result of the above requirements and architectural decisions, the underlying data persistence layer would have the following responsibilities:

Object-relational mapping Performing mapping between the relational and object-oriented representation for all business objects in the system.
Offline operation Supporting remote reading and editing of business objects on client nodes in the absence of the server link.
Data synchronisation Synchronising client's local changes with the database maintained on the server side.
Logging Automatic logging of all client operations on the server side.
Security Ensuring high standards of data security.

In section 3 we explain why none of the currently available Java frameworks was suitable as a persistence layer for *Nabor*, and in section 4 we describe how Offline Business Objects implements the desired mechanisms.

3 Existing persistence frameworks

3.1 Introduction to Object Relational Mapping

Contrary to earlier expectations and predictions, relational databases are still the primary technology for data persistence. Although the alternative approaches, such as object database systems, are gaining more and more popularity, it is still relational or object-relational databases that prevail.

Implementing a large information system based on simple relational database manipulation interfaces, such as JDBC [20] for Java, can prove a daunting and error-prone task. This is especially true in case of object-oriented technologies, where a significant amount of application code would deal not with implementing business logic, but with object-relational conversions.

This is one of the reasons why Object Relational Mapping (ORM) technologies started to emerge [11]. ORM can be defined as a mechanism for automated persistence of objects (e.g. Java objects) to the tables of a relational database. Essentially, ORM can be thought of as a means of transforming data from the object-oriented to the relational representation and *vice versa* [3, 10]. Most modern ORM tools perform the transformation fully transparently, i.e. any changes made to a persistent business object, such as setting a new value for its instance field, are automatically reflected in the database without a single persistence-related line of code on the business object's side.

The most popular Java ORM tools are currently Enterprise Java Beans (EJB) [17], Java Data Objects (JDO) [18] and Hibernate [2]. In the remaining part of this section we will use Hibernate as a representative example of an ORM tool. The reason for this is that JDO is in essence very similar to Hibernate and the recently released EJB 3.0 draft specification is also largely influenced by the experiences with Hibernate.

3.2 ORM and desktop applications

As we pointed out earlier, the majority of currently available data persistence frameworks, including Hibernate, have been designed and implemented with web-based applications in mind. For this reason integrating ORM tools with a desktop application, such as a Java Swing application in case of *Nabor*, is not a straightforward task.

A naive approach could be to use the ORM framework on the client side. This would require that the database manipulation interface (e.g. JDBC in case of Java) be directly exposed so that it can be accessed by client nodes. For obvious reasons this approach introduces high security risks, as the database connection could be used in a trivial way to read, modify or delete data. Although in many scenarios, such as an intranet application with trusted users, the naive scheme would be perfectly sufficient, it was not acceptable for the purposes of *Nabor*.

The alternative approach is to integrate the ORM framework into the server software and create a special layer of communication between the client nodes and the server. In this way, the direct database connection would not need to be exposed. On the other hand, the introduction of an additional communication layer would mean losing or re-implementing a number of useful features the ORM tools can offer. In case of Hibernate these features would be e.g.:

Unambiguous representation Hibernate guarantees that querying the database for the same object many times within the same session will always return the same instance of a Java class.

Modification detection Hibernate automatically detects which objects have been modified and sends to the database only the data that needs updating.

Yet another possibility would be to use ORM on the client side with a local off-line database and periodically synchronize the database with the central server. One major difficulty with this approach is the lack of freely available software frameworks for database synchronisation. Moreover, in some applications each client should have access to only a small part of the system's business objects, which can be enforced more flexibly by synchronising and authorising individual operations rather than the whole database.

3.3 ORM and the offline operation model

As we mentioned earlier, the key characteristic of an application implemented in the offline operation model is that it does not maintain a permanent link to the server. Instead, all client-side operations are performed on locally cached data which is then periodically synchronised with the server. The possible advantages of the offline model are better fault-tolerance and lower bandwidth usage. The biggest problem with this model, on the other hand, is that none of today's ORM frameworks fully supports offline operation. Therefore, in order to achieve the offline functionality, additional components must be built including a local data cache and some data synchronisation layer.

The responsibility of the local data cache component would be to ensure read and write access to business objects in the absence of the server link. Ideally, the data cache would have properties similar to those known from relational databases, namely: atomicity, consistency, isolation and durability, together referred to as ACID [12]. One way of achieving these properties is implementing the local cache using a lightweight embedded relational database engine, such as the Java-based HypersonicSQL [16]. An alternative approach would be based around the concept of Prevayler [22], which implements persistence by keeping all data in memory, logging every operation before it is executed, storing periodic snapshots of all data and using these snapshots together with operation logs to recover system state.

The task of the data synchronisation component is to maintain consistency between the contents of the client's local cache and the global database managed by the the server [4]. The main problem here is to properly resolve conflicting modifications of the same business object performed locally by disconnected clients. A general solution would require taking into account the complex semantics of business objects and thus might be quite costly to implement. Alternatively, specific characteristics of a concrete application can be exploited in order to simplify the problem. In case of *Nabor*, for example, for a large number of business objects there would be only one client authorised to modify them. Later in this paper we show how we took advantage of this property. Another important requirement for the data synchronisation component is making the communication efficient in terms of the volume of data transferred and guaranteeing that data consistency will not be violated as a result of e.g. a communication layer failure.

Although Hibernate does not explicitly support the offline operation mode, it offers a mechanism of *detached objects* which can prove helpful in implementing the local cache and data synchronisation components. With this mechanism, every persistent data object (i.e. a Java object stored in a database by Hibernate) can be temporarily detached from Hibernate's data persistence layer, serialised and transmitted between different system layers, and then re-attached back to Hibernate.

In late 2003, when the development of the *Nabor* system was already underway, a specification of a standard called Service Data Objects [6] was announced, which aimed at simplifying and unifying the ways in which applications handled data. Among others, SDO defines a *disconnected programming model*, which seems to be the counterpart of the offline operation model required for *Nabor*. Unfortunately, no implementations of SDO were available at that time, hence the need for the Offline Business Objects we describe in detail in the next section.

4 Offline Business Objects

The purpose of Offline Business Objects is to provide a reliable and secure persistence layer for offline desktop applications. Its main responsibilities are:

- object-relational mapping of business data
- offline access to business data
- client-server synchronisation
- authorisation of client operations
- logging of client operations
- ensuring high standards of data security

In the following subsections we describe how client- and server-side components of OBO shown in Figure 1 cooperate to implement the required mechanisms.

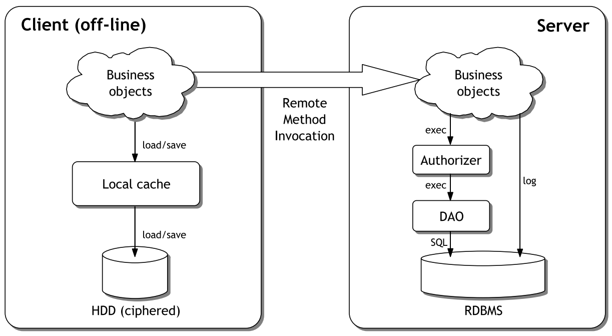

Fig. 1. The architecture of Offline Business Objects

4.1 Object-relational mapping

The task of the object-relational mapping component is to translate business data between the relational and object-oriented models. As this part of OBO would reside on the server side, it's implementation could almost entirely rely on one of the existing ORM tools. We decided to implement the object-relational mapping component using the Torque framework[2] [8].

Torque is based on the idea of so-called Partial Class Generators [15], where a declarative XML specification of object-relational mapping is used as an input for automatic generation of base classes for business objects and data access

[2] The alternative choice was Hibernate [2], but at the time of designing OBO (mid 2003) Hibernate was not as popular as it is now and we felt we should not yet use it on production scale.

objects (DAO). Specific behaviour of business objects can be implemented by subclassing the automatically generated classes.

For the following example specification:

```
<table name="PERSON">
  <column name="PERSON_ID" required="true"
          primaryKey="true" type="INTEGER"/>
  <column name="FIRST_NAME" required="true"
          size="50" type="VARCHAR"/>
  <column name="LAST_NAME" required="true"
          size="50" type="VARCHAR"/>
</table>
```

the following artifacts will be generated:

BasePerson A business object base class with appropriate getter and setter methods. Business-specific behaviour can be implemented by subclassing this class.

BasePersonPeer A class that handles reading and writing of the business object's data to a relational database.

SQL DDL statements SQL Data Definition Language statements that can be used to create all database structures (e.g. tables, sequencers) required for persisting the business objects.

4.2 Offline operation and client-server synchronisation

The offline operation and client-server synchronisation component was by far the most difficult to design and implement part of OBO. It involved both the client-side local cache and the server-side synchronisation and persistence mechanisms.

Local cache The purpose of the local cache is to provide read and write access to business objects in the absence of the server connection. To that end, business data is stored on the client's workstation in an encrypted form. The storage mechanism is similar to this used in Prevayler [22]. All data is kept in memory, with each client's operation being logged before execution. Additionally, the system takes periodic snapshots of all data and stores them on the local disk. Local system state recovery is based on the freshest snapshot and the log of operations performed after that snapshot was taken.

An interesting design problem related to the local cache was how and when to assign unique identifiers to business objects. There are two possibilities here:

- a unique identifier is created on the client side at the time of creating or storing of the business object in the local cache
- a unique identifier is assigned on the server side during client-server synchronisation

In a naive implementation of the latter method, the client application could transactionally synchronise its local changes with the server and as a result receive the identifiers of newly created business objects. Such an approach, however, has a subtle flaw connected with fault-tolerance. Suppose a transaction has been successfully committed on the server side, but due to a failure of the client-server link, the client node did not receive the identifiers of newly created objects. In this situation, the client application will have no way of knowing that a number of its local business objects that do not yet have identifiers correspond to some server-side objects for which unique identifiers have already been assigned.

One way of solving this problem would be to extend the communication protocol with additional acknowledgment messages, e.g. a "ready to commit" message sent by the client application after the identifiers have been received and applied to the locally cached data. However, in order to simplify the communication, we have decided that the identifiers should be generated on the client side. Global uniqueness of locally assigned identifiers is achieved by dividing the global pool of identifiers into extendable subsets used exclusively by one client. If a client reports that it has used a certain number of identifiers from its set (e.g. 80%), the server will extend the identifier set for that client.

Business object lifecycle Designing a data persistence framework requires a decision as to how the lifecycle of business objects should look like, i.e. in what states an individual object can be and how they relate to the client-server communication. A starting point here can be the object lifecycle defined in one of the existing persistence frameworks, such as Hibernate or JDO. With the additions required by the offline processing model, the list of states of an individual business object can be the following (Figure 2):

Transient An object is Transient right after it has been created, e.g. by a call to a constructor method.

Persistent New A Transient object becomes Persistent New when it gets stored in the local cache, and has not yet been sent to the server database.

Persistent Clean A business object becomes Persistent Clean after it has been synchronised with the server.

Persistent Dirty A Persistent Clean object becomes Persistent Dirty after it has been locally modified.

Deleted A business object that has been locally deleted.

The state of an individual business object determines the actions required to synchronise this object with the central database. During data synchronisation the client application sends to the server the contents of objects that are Persistent New or Persistent Dirty, and also identifiers of objects that are Deleted. On the server side, the object's state can be used to choose the appropriate SQL operation to perform: *insert* for Persistent New objects, *update* for Persistent Dirty objects, and *delete* for Deleted objects.

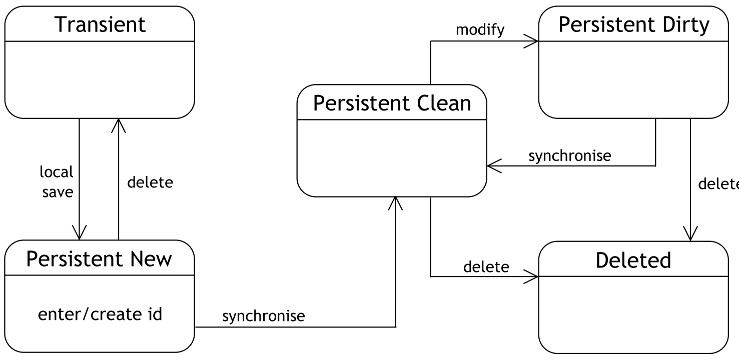

Fig. 2. Offline Business Objects initial life cycle

It turns out, however, that for enhanced fault-tolerance we need to introduce one more state to the lifecycle of a business object. To illustrate this, let us assume that the client application has created a new business object and stored it in the local data cache as Persistent New. During the next client-server synchronisation, the new object should be sent to the server and its state should become Persistent Clean. The major problem here is that committing changes on the server side and changing object's state on the client should be one atomic operation. With the initially proposed lifecycle, the client could change the object's state to Persistent Clean either "right before" it sends the changes to the server, or "right after" the server acknowledges that the data has been successfully committed to the database.

Neither of these variants, however, is perfect with respect to fault-tolerance. If a communication layer failure occurs, the client's information about the states of locally cached business objects may become inconsistent with what is stored on the server side. With the "right before" strategy, the client application may mark an object as Persistent Clean even though the server did not commit a corresponding change to the database. With the "right after" variant, on the other hand, if the server acknowledgment is not received on the client side, the state of some objects will not be changed to Persistent Clean even though it should. As we can see, neither of the above strategies guarantees correct information about the objects' state on the client side. Consequently, the server cannot rely on this information received from the client during the subsequent synchronisations either.

A solution to the above problem can be enhancing the business object's lifecycle with an additional intermediate state called Persistent Possibly New shown in Figure 3. In this way, during the client-server synchronisation a Persistent New object is sent to the server and then its local state is changed to Persistent Possibly New. Only after the server acknowledges that the transaction has been successfully committed, a Persistent Possibly New object becomes Persistent Clean. Otherwise, if a communication failure occurs, the object remains Persistent Possibly New and will be sent in that state to the server during the next data synchronisation. On the receipt of a Persistent Possibly New object, the server will need to do an additional *select* operation to check if corresponding data is already present in the database.

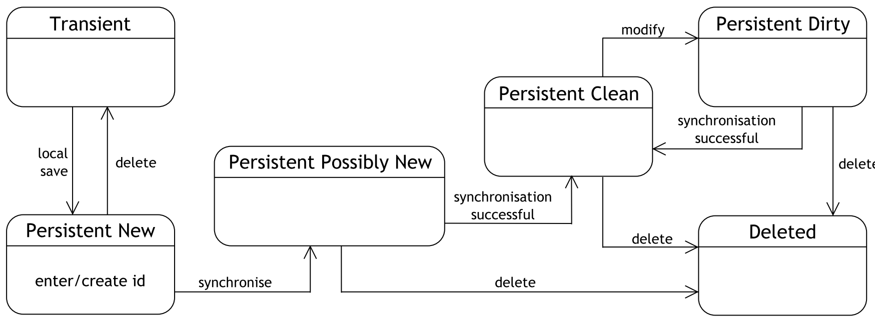

Fig. 3. Offline Business Objects modified life cycle

A similar type of problem can arise with transitions between the Persistent Dirty and Persistent Clean states. In this case, however, there is no need for intermediate states and the transition can take place right after the server acknowledges a successful completion of the transaction. If the client application does not receive the acknowledgment because of a communication layer failure, it will send the modified object to the server once again during the next synchronisation.

Data synchronisation The purpose of data synchronisation is to ensure consistency between the desktop application's local cache and the global database on the server. While designing OBO we decided that data synchronisation should be performed in two phases. Phase one is applying client's local changes to the global database while phase two is updating client's local cache with modifications made by other clients.

In the first phase the client application sends to the server a log of locally cached operations. Based on the current state of a business object (see section 4.2), different actions will be taken:

Create object For a Persistent New object, the client application will send the contents of that object (values of attributes, identifiers of collection members) to the server. The server will add corresponding data to the database

Delete object For a Deleted object, the client application will send the identifier of that object to the server. The server will delete the corresponding data from the database.

Update attribute value For a Persistent Dirty object with modified attributes, the client application will send to the server the identifier of that object and name/new value pairs for all modified attributes. The server will modify the corresponding values in the database.

Update collection For a Persistent Dirty object containing a modified collection of business objects, the client application will send identifiers of objects added and removed from that collection. The server will make corresponding changes to the database. Noteworthy is the fact that this operation modifies the *relationship* between business objects, but not the objects themselves.

Crucial to the second phase of data synchronisation is versioning of business objects [21]. OBO implements this mechanism by assigning an integer version number to each of the business objects managed by the system. Every time the server processes an attribute update operation, the version number of the involved business object is incremented. For a collection update increased is only the version number of the business object that contains the collection, version numbers of collection members remain unchanged.

The client applications initialises the second phase of data synchronisation by sending to the server the identifier/version number pairs of all locally cached objects. Then, for each business object the server compares its version number V_S of that object with the corresponding version number V_C sent by the client. Based on that comparison, the server can take the following actions:

When $V_C < V_S$ The client's version of the business object is older than the server's version. The up-to-date object needs to be sent to the client.
When no V_C corresponds to V_S The client has not yet received any information about the business object. The object needs to be sent to the client.
When no V_S corresponds to V_C The client has an object that has been deleted some time after the last synchronisation. Information that the object has been deleted needs to be sent to the client.
When $V_C = V_S$ The client's version of the business object is up-to-date. No data needs to be sent. This is the case where versioning can bring a substantial decrease in the volume of data transferred during synchronisation.

An interesting problem related to data synchronisation is how the system should deal with concurrent modifications of business objects. Situations in which the same version of an object is modified by two or more clients are far more probable in an offline processing model. With Offline Business Objects this problem can be solved in two different ways. One method is to accept the first modification of a certain version of an object and reject all subsequent modification attempts related to that version. Users whose modifications have been rejected will be notified of the fact, and will need to explicitly cancel either all local operations performed since the last the synchronisation or only the conflicting ones. In this way, the first accepted modification will effectively overwrite the remaining concurrent modifications[3]. The alternative method is to execute all

[3] The exact sequence of operations would be the following: the server accepts the first modification of an object and advances the object's version number from $V_S = i$ to $V_S = i + 1$. On the receipt of another modification related to version $V_S = i$, the server rejects the modification and instructs users to cancel the corresponding operations. Upon the next synchronisation, the client will receive the $V_S = i + 1$

modifications in the order they were received by the server. This method, on the other hand, means that the last concurrent modification will overwrite the earlier ones. In this case, no special interaction with the user is needed. The choice as to which of these methods OBO should use can be made for each database table separately.

Using the former conflict resolution strategy (*first wins*) in *Nabor* would require the end users to make decisions related to the application's underlying communication protocols, which might be both misleading and frustrating for them. On the other hand, due to the characteristics of *Nabor*, lost updates would occur fairly rarely and would not result in any major inconsistencies. For this reason, in our real-world application, we used the *last wins* strategy for all kinds of business objects.

4.3 Authorisation

One of the requirements for the Offline Business Objects framework was that each modification of data attempted by the client's desktop applications should be authorised. OBO implements a model whereby the client's credentials are verified on the server side during data synchronisation. For maximum flexibility, OBO performs authorisation on the level of Java code before business objects are read or written to the database (see Figure 1).

The details of authorisation can be specified separately for each database table managed by OBO in a declarative way, e.g.:

```
<table name="TEAM" authorizationMethod="hasCoachPrivileges"/>
```

With the above declaration, before a modification of a Team object is committed to the database, the `hasCoachPrivileges` method (available in a globally accessible credentials object) is called to verify if the client is allowed to perform that modification.

More interesting is the problem of authorising operations involving interrelated tables, e.g. master-detail tables:

```
<table name="PLAYER" authorizationMethod="hasCoachPrivileges">
  <references foreignTable="TEAM" authorizationPropagates="true"/>
</table>
```

In this case, before the server applies any modifications to the Player object, it will check whether the client is allowed to modify the Player's Team. Into account will be taken both the current Team of the Player and also the Team to which the Player is to be transferred. Moreover, OBO will group operations involving different Players, so that the data of all required Teams can be fetched with a single database query.

version of that object, which will effectively overwrite the (deleted) local change made to version $V_C = i$

4.4 Data security

The security model we implemented in Offline Business Objects is based on the Public Key Infrastructure (PKI) and the RSA cryptography [23]. During the design phase we decided that each client operation, such as creating or modifying a business object (see section 4.2), should be transmitted to the server as a separate encrypted packet.

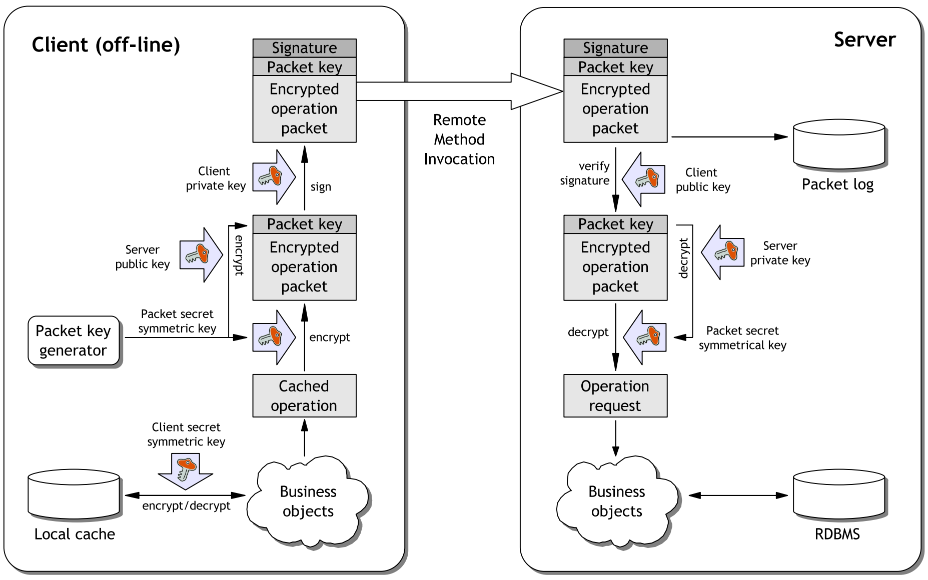

Fig. 4. Offline Business Objects security model

Figure 4 schematically shows all cryptographic operations involved in transmitting a single operation packet from the client's desktop application to the server. In the absence of the server connection, all application data is stored in an encrypted form in the local data cache. For the purposes of local cache protection OBO uses symmetric cryptography with a client-specific secret key distributed together with the desktop application.

During data synchronisation, for each cached client operation a separate packet is created. Each such packet is then encrypted with a dedicated symmetric key generated only for that packet. The symmetric key itself is encrypted with the server's public key and appended at the beginning of the packet. Finally, the packet is signed with with the client's private key and the signature along with the client's identifier is appended at the beginning of the packet.

Upon the receipt of a packet, the server verifies its signature. If the signature is invalid, the packet is logged in the "suspicious" packets log and removed from further processing. Otherwise, the server uses its private key to decrypt the packet's symmetric key, which is then used to decrypt the contents of the packet. Finally, the client operation encoded in the packet is applied to the server's database. Communication in the reverse direction is analogous, with private/public keys swapped appropriately.

4.5 Operation logging

The purpose of operation logging in the OBO framework is twofold: to provide a reliable client operation history and to ensure a data recovery mechanism in case of a database failure.

As we describe in section 4.4, each client operation, such as creating or modifying a business object (see section 4.2), is sent to the server as a separate encrypted packet. Before the server executes the operation, it logs the corresponding packet. Any such packet can be fetched from the log later on and executed once again when needed. The packet log mechanism can therefore be used to recreate a snapshot of the system's business objects from any time in the past.

The primary purpose of packet logging, however, is to implement a reliable client operation history mechanism. To this end, client packets are signed with the client's private key (see section 4.4), so that fraudulent operations can be (probably with diligent manual analysis) detected on the server side.

4.6 Implementation considerations

It is in the very nature of a distributed offline information system that a significant proportion of a business object's implementation code deals with system-level concerns, such as persistence, communication or security, and not the business-specific behaviour. Moreover, most of the system-level functionality would be very similar across different business objects. To avoid problems caused by manual implementation of this functionality in every single business object[4], Offline Business Object employs a software development technique called active source code generation [14].

At the heart of source code generation lies a declarative specification of business objects created by the software developer. Based on that specification, the generator can automatically produce different artifacts, such as source code of Java classes or SQL-DDL statements. An important property of active code generation is that the automatically generated code is not intended for the programmer to modify. Instead, all customisations should be implemented with the use of e.g. subclassing or delegation. In this way, should the declarative specification change, none of the customisations will be lost or overwritten.

In case of OBO, the declarative specification of a business object describes such its elements as object-relational mapping information (i.e. which database tables and columns the object maps to), authorisation methods, security levels (e.g. whether the object's data should be encrypted or not) or concurrent modification resolution strategy. Given the following example specification of a business object representing a school unit:

[4] In case of the *Nabor* project, the number of business object classes exceeded 80

```
<table name="UNIT" concurrentModification="lastWins">
  <references foreignTable="SCHOOL" authorizationPropagates="true"
              primaryKey="false" onDelete="cascade"/>
  <column name="UNIT_ID" primaryKey="true"/>
  <column name="NAME" type="STRING"/>
  <column name="DESCRIPTION" type="STRING"/>
</table>
```

Offline Business Objects will generate the following artifacts:

BaseUnit.java A base class for implementing the business behaviour of a school unit
BaseUnitPeer.java A base class that handles the object-relational mapping of the business object
BaseRightChecker.java A base class for implementing access control
Data synchronisation Java classes supporting data synchronisation and communication
Local cache Java classes implementing the offline caching of the business object
SQL-DDL Table and sequencer definitions for the business object

In Table 1 we show the size of Offline Business Objects expressed as the number of lines of Java source code. Included are also Java code templates in the Velocity [9] format.

Artifact	*Size*
OBO code generator	2 750 LOC (Java)
	938 lines (Velocity templates)
OBO security and utility classes	4 326 LOC (Java)
TOTAL	8 014 lines
Source code generated by OBO	53 059 LOC (Java)
Total number of business object instances	about 500.000

Table 1. Size of the Offline Business Objects source code

4.7 Limitations

Finishing our discussion of the Offline Business Objects framework, we need to emphasise that the offline processing model is not suitable for all kinds of distributed applications. In particular, OBO is not the best choice for systems which perform large numbers of concurrent modifications of the same data. In such cases, with too many concurrent modifications being rejected, the system may become inconvenient or even impossible to use. Choosing the alternative strategy, on the other hand, would increase the risk of losing data coherence.

The assumption of the *Nabor* system, however, was that for a great majority of business objects there would be only one client with the rights to modify them.[5] Consequently, the number of concurrent modifications would be low enough for the offline processing model to be efficient. For the same reason we did not consider implementing more complex conflict resolution mechanisms, such as non-repudiable information sharing [5].

Another problem related to OBO, but also to the ORM frameworks in general, is different operation semantics. To illustrate this, let us consider two concurrent transactions each of which increases the balance of a bank account by 10; the starting balance is 15. In a relational database, both transactions would execute an SQL statement similar to *update account set balance = balance + 10*, which would finally result in the balance being changed to 35. With an ORM framework, however, a similar semantics is very difficult to achieve. Assuming the "last committer wins" strategy, both transactions would have to execute a statement similar to *account.setBalance(account.getBalance() + 10)*, which in both transactions would effectively translate to *account.setBalance(25)* and an incorrect final account balance. The only way of dealing with this problem is rejecting concurrent modifications, which may prove inconvenient for the end users.

5 Evaluation

In 2004 the *Nabor* system, which we built based on the Offline Business Objects framework, was deployed in five major cities in Poland on over 200 independent client workstations and handled almost 30.000 high school candidates. Throughout the operation period, the system performed smoothly and reliably. No attempts to manipulate the admission results were reported, no major usability issues were discovered. We therefore feel that this is the best proof that the offline processing model is feasible and can be implemented in practice.

In Table 2 we summarise the size of the *Nabor* system source code. An interesting observation is that the size of code generated by OBO is more than six times bigger that the source code of OBO itself (see Table 1). This might be considered as a sign that our investment in OBO was a profitable one. It must be borne in mind, however, that programming a source code generator is far more complex than writing regular code, which makes the accurate comparison very difficult. The real benefit can come from reusing OBO in similar projects in the future.

6 Conclusions and future work

Creating a secure distributed desktop application supporting offline operation is a non-trivial task. Even more so is designing and implementing a general framework that would facilitate the development of such applications.

[5] For example, only one of the schools a candidate is applying to has the right to enter and further modify the candidates data.

Artifact	*Size*
Hand-written source code	92 051 LOC (Java)
Source code generated by OBO	53 059 LOC (Java)
Source code generated by JAX-B and others	35 957 LOC (Java)
TOTAL	181 107 LOC (Java)

Table 2. Size of the *Nabor* system source code

In this paper we have presented the Offline Business Objects framework whose aim is to enable data persistence for distributed desktop applications. We identified the requirements for OBO and its main components. We also discussed a number of specific design and implementation problems we faced. Finally, we emphasised the limitations of the offline operation paradigm that must be considered before adopting this approach in real world software.

Our future work on Offline Business Objects will concentrate on looking into the possibilities of extending the existing Java-based persistence frameworks, such as Hibernate, with offline processing capabilities. Service Data Objects implementations can be used as a communication layer for OBO.

Although OBO has been successfully used as a building block for a real-world application, we are aware that detailed performance analyses and benchmarks are required. Another interesting research area is adapting the OBO framework for mobile distributed applications.

References

1. IBM Workplace Client Technology: Delivering the Rich Client Experience. White paper, IBM Corporation, 2004.
2. Christian Bauer and Gavin King. *Hibernate in Action.* Manning Publications, 2004.
3. Luca Cabibbo. Objects meet relations: On the transparent management of persistent objects. In *CAiSE*, pages 429–445, 2004.
4. Paul Castro, Frederique Giraud, Ravi Konuru, Apratim Purakayastha, and Danny Yeh. A programming framework for mobilizing enterprise applications. *WMCSA*, pages 196–205, 2004.
5. Nick Cook, Paul Robinson, and Santosh K. Shrivastava. Component middleware to support non-repudiable service interactions. In *DSN*, pages 605–. IEEE Computer Society, 2004.
6. IBM Corporation and BEA Systems. Service Data Objects Specification v1.0. 2004.
7. Jens Dibbern, Tim Goles, Rudy Hirschheim, and Bandula Jayatilaka. Information systems outsourcing: a survey and analysis of the literature. *SIGMIS Database*, 35(4):6–102, 2004.
8. Apache Software Foundation. Torque Persistence Layer. *http://db.apache.org/torque*, 2004.
9. Apache Software Foundation. Velocity Template Engine. *http://jakarta.apache.org/velocity*, 2004.

10. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
11. Mark L. Fussell. Foundations of object-relational mapping. *ChiMu Corporation*, 1997.
12. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
13. Pawel Gruszczynski, Bernard Lange, Michal Maciejewski, Cezary Mazurek, Krystian Nowak, Stanislaw Osinski, Maciej Stroinski, and Andrzej Swedrzynski. Building a large-scale information system for the education sector: A project experience. *Accepted for the 7th International Conference on Enterprise Information Systems*, 2005.
14. Pawel Gruszczynski, Stanislaw Osinski, and Andrzej Swedrzynski. Should we jump on the cross-cutting techniques bandwagon? *Submitted for the 7th National Conference on Software Engineering, Poland*, 2005.
15. Jack Herrington. *Code Generation in Action*. Manning Publications, 2003.
16. HypersonicSQL Lightweight Java SQL Database Engine. *http://hsqldb.sourceforge.net*, 2005.
17. Sun Microsystems. Enterprise JavaBeans. *http://java.sun.com/products/ejb*, 2005.
18. Sun Microsystems. Java Data Objects Specification v1.0.1. *http://www.jdocentral.com*, 2005.
19. Sun Microsystems. Java Foundation Classes. *http://java.sun.com/products/jfc*, 2005.
20. Sun Microsystems. JDBC. *http://java.sun.com/products/jdbc*, 2005.
21. Panos A. Patsouris. A formal versioning approach for distributed objectbase. In *ICPADS '97: Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 686–693, Washington, DC, USA, 1997. IEEE Computer Society.
22. Prevayler: Free-software prevalence layer for Java. *http://www.prevayler.org*, 2004.
23. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.