System Design Document

Database Team

PAID Project

Informatik XII

WS 1998/1999

Technische Universität München

February 7, 1999

Preface:

This document addresses the requirements of the PAID system. The intended audience for this document are the designers and the clients of the project.

Target Audience:

Clients, Developers

PAID Members:

Project Management:	Bernd Brügge, Guenter Teubner
Team Coaches:	Ralph Acker, Stefan Riss, Ingo Schneider, Oliver Schnier, Anton Tichatschek, Marko Werner
Architecture Team:	Asa MacWilliams, Michael Luber
Authentication & Security Team:	Klaas Hermanns, Thomas Hertz, Guido Kraus, Gregor Schrägle, Tobias Weishäupl, Alexander Zeilner
Database Team:	Osman Durrani, John Feist, Florian Klaschka, Johannes Schmid, Florian Schönherr, Ender Tortop
Learning Team:	Burkhard Fischer, Jürgen Knauth, Andreas Löhr, Marcus Tönnis, Martin Uhl, Bernhard Zaun
Network & Event Service Team:	Henning Burdack, Jörg Dolak, Johannes Gramsch, Fabian Loschek, Dietmar Matzke, Christian Sandor
STAR NETWORK Integration & User Interface Team:	Daniel Stodden, Igor Chernyavskiy, Inaki Sainz de Murieta, Istvan Nagy, Stefan Krause, Stefan Oprea
Testbed Team:	Bekim Bajraktari, Bert van Heukelom, Florian Michahelles, Götz Bock, Michael Winter, Sameer Hafez

Revision History:

Version R1.0 01/31/99 - All DB Subsystem Members, Created and released.
Version R1.1 02/01/99 - Johannes Schmid, some minor modification and restructuring
Version R1.2 02/02/99 - Johannes Schmid, changes on Class Diagrams & other changes
Version R1.3 02/03/99 - John Feist, major changes to text passages
Version R1.4 02/03/99 - Florian Schönherr, modified sequence diagrams and descriptions
Version R1.5 02/07/99 - Johannes Schmid, updates and layout modifications

Table of Contents

1 Goals and Trade-offs

Provide an easy to administer and efficient system (see Architecture Team).

2 System Decomposition

The database subsystem provides access to all databases that support JDBC. Normally other subsystems should not access a database directly, instead events such as requesting or changing data have to be posted on the event bus. They are executed on either a local database or, if the subset is not available locally, sent to a parent server that has to answer this request. The result of every database query event is posted back to the event bus, so the requesting subsystem has to listen for the response event.

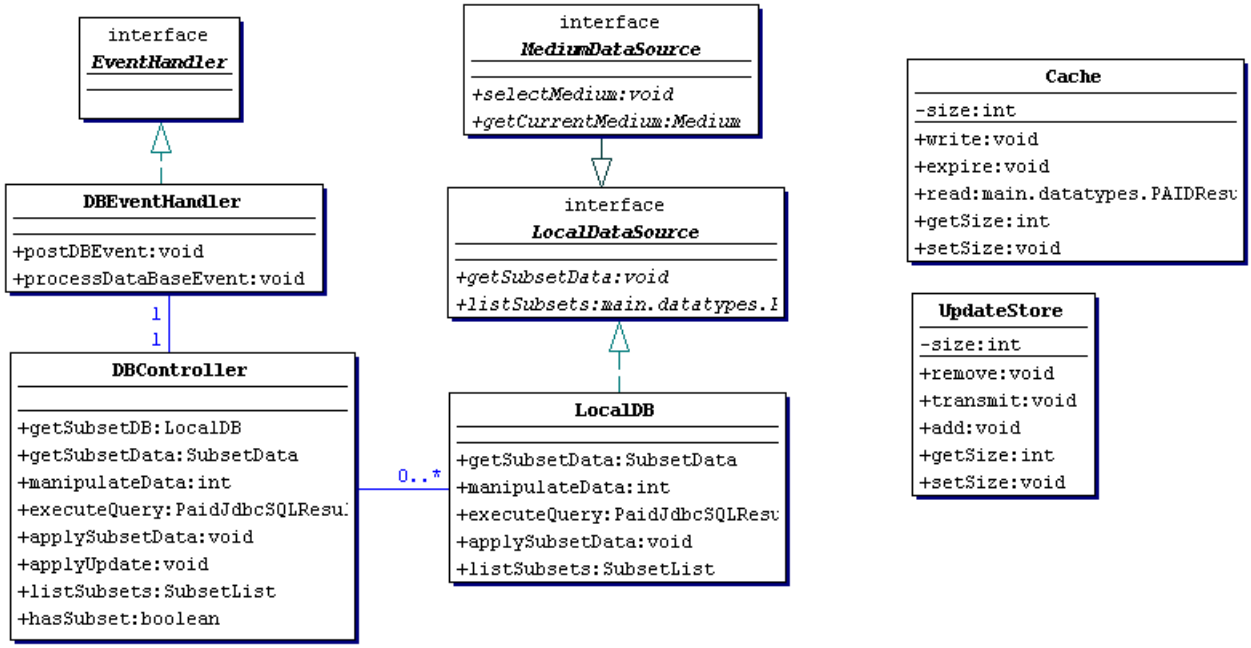
The database subsystem is divided into three main parts:

1. **The Datamove Part** which contains the classes of Ingo Schneiders diploma thesis. These classes are used to compare two databases, extract the differences (that's what Ingo's analyzer tool will do), and create objects, the so called SmartUpdates, which are able to update a local database in a way that after executing them, the local database contents are up to date.

2. **The Database Handler Part** which holds the main classes for database actions within PAID. The most important classes may be the DBEvents, because we decided to give each event a method called „handle“, which carries out the right actions that have to be done for this event. Therefore our DBEventHandler simply calls this method on receipt of an event and passes the reference to the local databases that are represented by the class LocalDB. Furthermore there is a Cache class, that holds the cached queries previously retrieved from network.
   The DB Part itself is devided into the DB Main Package (see Chapter 2.1), the DB DataTypes (see Chapter 2.2) and the DB Events handled by the DBEventHandler (see Chapter 2.3)

3. **The JDBC Driver Part** in which the PAID JDBC driver classes are located. This driver has to be implemented, because our client requested an easy integration of PAID into StarNetwork without handing out his own sourcecode. The benefits of this way of integration are descibed in section 9 „Design Rationale“. The JDBC driver's layout is described in detail in Chapter 2.4

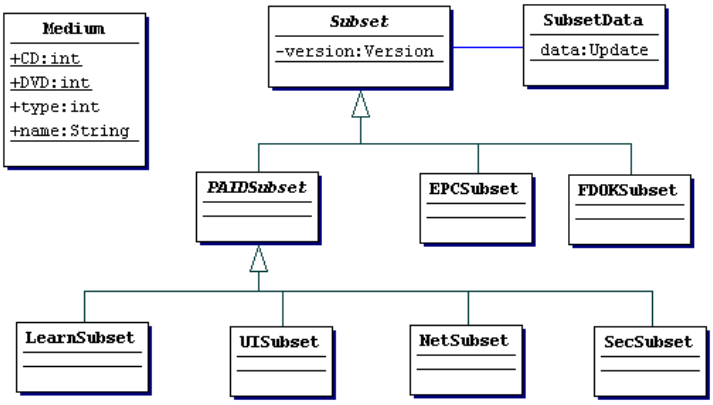
2.1 The Main Package of the Database Subsystem



This Package contains all major parts of the Database Subsystem (except the Events hierarchy itself and the DataTypes). It contains the EventHandler, which handles all incoming events, the Database Controller, which handles all attached database systems, the Cache, which contains all outgoing SQL queries and their results and the UpdateStore, which stores all Updates for Database Subsets.

DBEventHandler	Listener to be registered at the PAID event bus, which registers to all incoming events the Database subsystem is interested in.
DatabaseController	Manages the bulk of all local databases and returns the right database (LocalDB) when given a specific Subset.
LocalDB	The handler for a specific local database, which performs all actions on this database.
LocalDataSource	Interface for any media (description see chapter Data Types) that allows reading a specific subset and then insert it into a database.
MediaDataSource	Interface for any kind of locally available media providing pre-manufactured data updates.
Cache	The package which handles the cached queries (all queries which couldn't be answered locally). This package can store, read and expire SQL queries and their ResultSets.
UpdateStore	This package handles the Database Updates of a PAID server. All incoming updates are stored in this UpdateStore, until they could be sent to all subscribed clients. The UpdateStore also expires old updates that couldn't be sent to all clients, when the server runs out of free disk space.

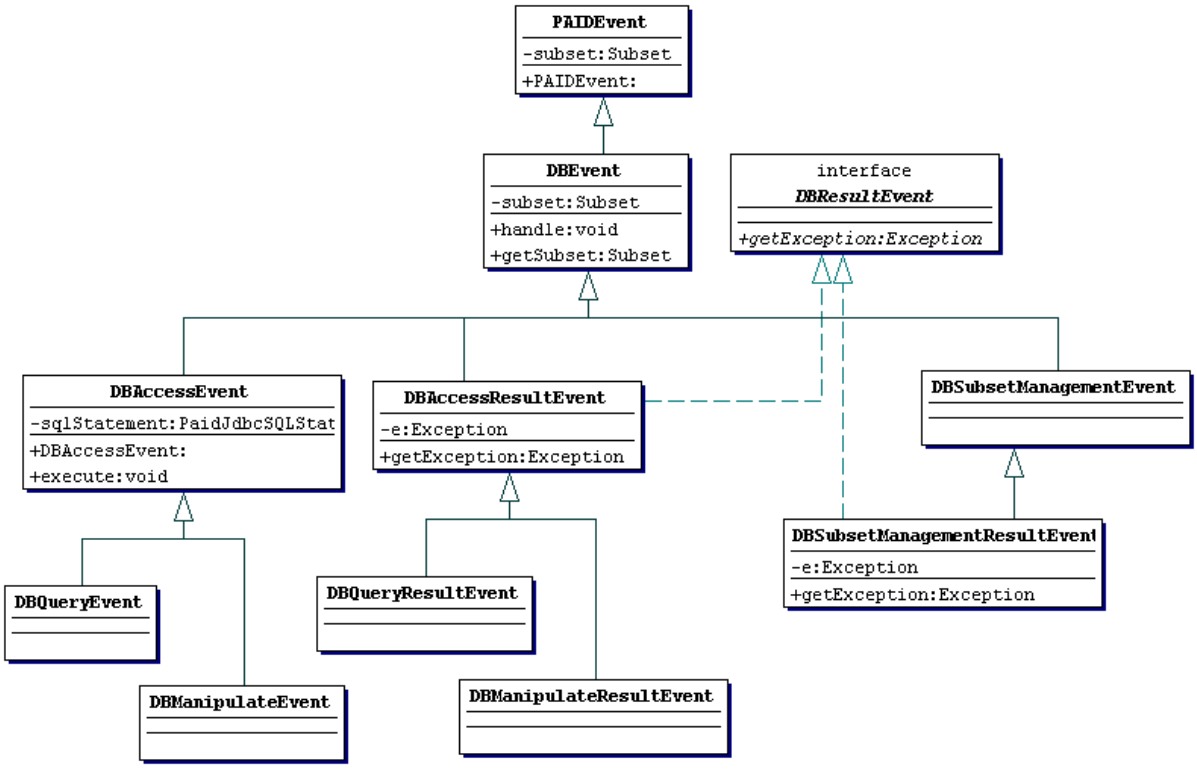
2.2 Database specific Data Types



These are the main external data types, used by the events of the Database Subsystem.

Subset	Root (abstract) class for all Database Subsets. The application specific subsets are all derived from this class. All Subsets have a version. The version of all currently installed subsets are tracked by the Database Subsystem, which also ensures that the incoming updates fit to the installed Subset version.
Medium	Specifies a media which contains Database Subsets that can be inserted inserted in the local database. This makes it very simple to add new medias in future. Currently supported medias are CD and DVD.
SubsetData	Container for specific Database Subsets and their updates.

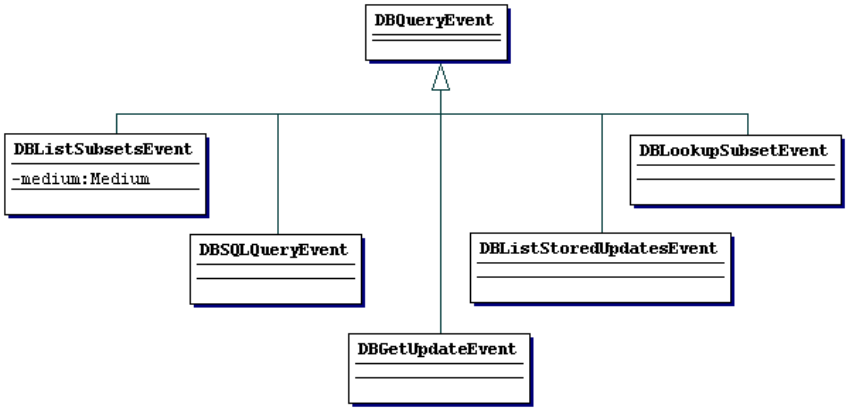
2.3 Event hierarchy



This is the main DB Event hierarchy containing all the abstract event classes. Each of the events shown here has several derived events, which actually are non-abstract and are described in sub-chapters of this chapter.

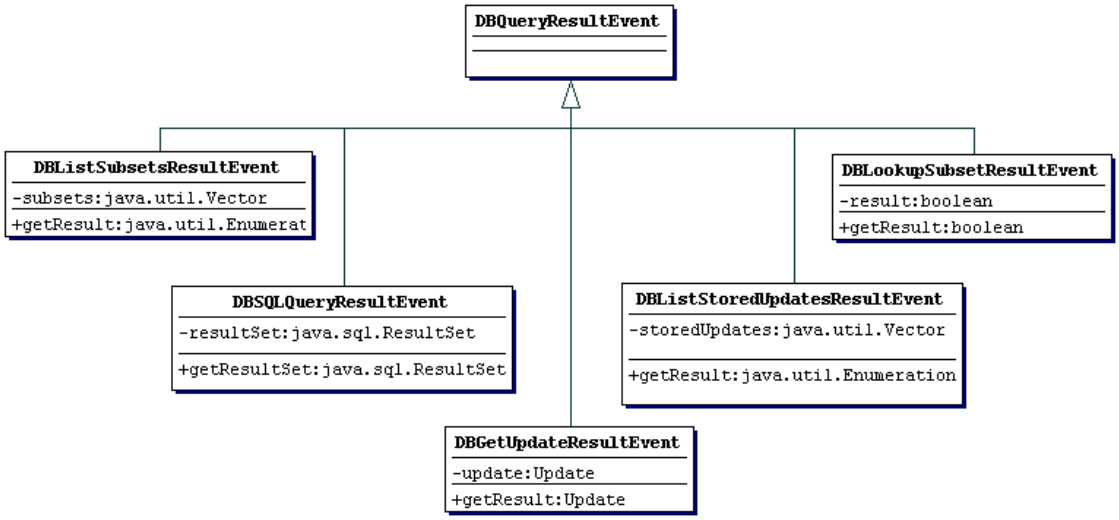
DBEvent	Main Database Subsystem event
DBAccessEvent	Root for any event requesting or manipulating data by SQL queries. Its sub-event types, DBQueryEvent and DBManipulateEvent, are described later on in this chapter. Note: the PAIDEvent's subset field must be filled in correctly.
DBAccessResultEvent	Root for any event returning a result for a DBAccessEvent query. Implementing the interface DBResultEvent, it supports the getException Method returning the exception, which occured during the update/query excecution on the remote side, if any occured. Its sub-event types, DBQueryResultEvent and DBManipulateResultEvent, are described later on in this chapter.
DBSubsetManagementEvent	Root for any events concerning subset management. All ist sub-event types are described later on in this chapter. Note: the PAIDEvent's subset field must be filled in correctly.
DBResultEvent	Interface which should be implemented by any event which may be the answer on another DB-event.

2.3.1 DBQueryEvent



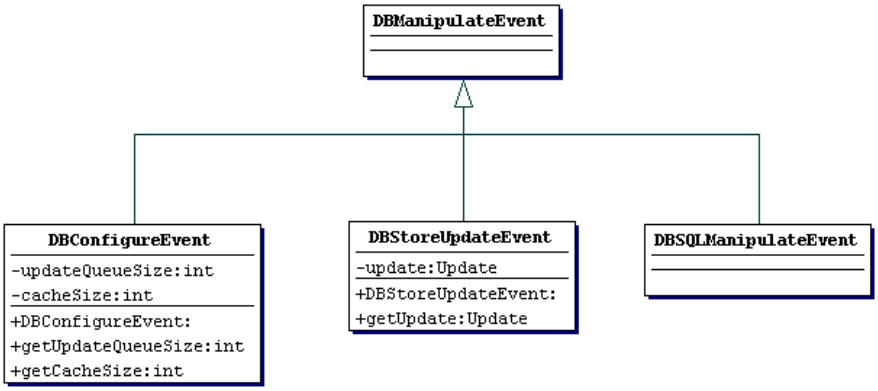
DBQueryEvent	Root event for any event requesting data from the database.
DBListSubsetsEvent	Request to list all Subsets stored in the local database. The answer is provided as a DBListSubsetsResultEvent.
DBSQLQueryEvent	Request to run a specific SQL query on the local database. The answer is provided as a DBSQLQueryResultEvent.
DBGetUpdateEvent	Request to get a specific Subset Update from the UpdateStore. The answer is provided as a DBGetUpdateResultEvent.
DBListStoredUpdatesEvent	Request to list all Subset Updates available in the UpdateStore. The answer is provided as a DBListStoredUpdatesEvent
DBLookupSubsetEvent	Request to find out whether a subset is available in the local database or not. The answer is provided as a DBLookupSubsetResultEvent.

2.3.2 DBQueryResultEvent



All events shown here are ResultEvents for DBQueryEvents.

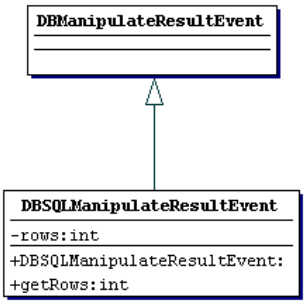
2.3.3 DBManipulateEvent



DBManipulateEvent	Root event for any event requesting data manipulation on the database.
DBConfigureEvent	Request to change Database Subsystem configuration data (this is mainly done by the Learning Subsystem). Currently available configuration values are: updateQueueSize and cacheSize. If a value is set to '-1', the value won't be changed.
DBStoreUpdateEvent	Request to store a given Update in the UpdateStore (this event is triggered by the Learning Subsystem). If the maximum UpdateStore size is reached, the oldest Update in the UpdateStore is removed.

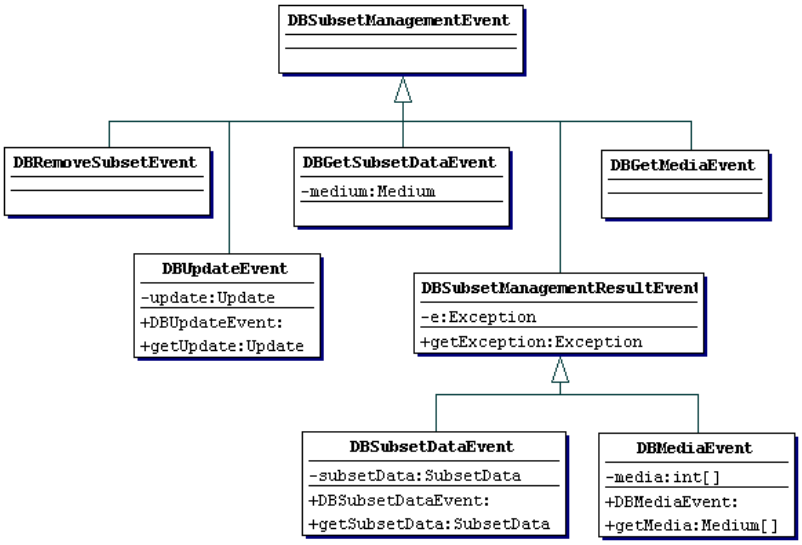
DBSQLManipulateEvent Request to manipulate any data in the local database using a SQL statement. If the specified Subset is a non-PAIDSubset, the manipulation request will automatically be replicated up to the next PAID server. The answer is a DBSQLManipulateResultEvent.

2.3.4 DBManipulateResultEvent



All events shown here are ResultEvents for DBManipulateEvents.

2.3.5 DBSubsetManagementEvent

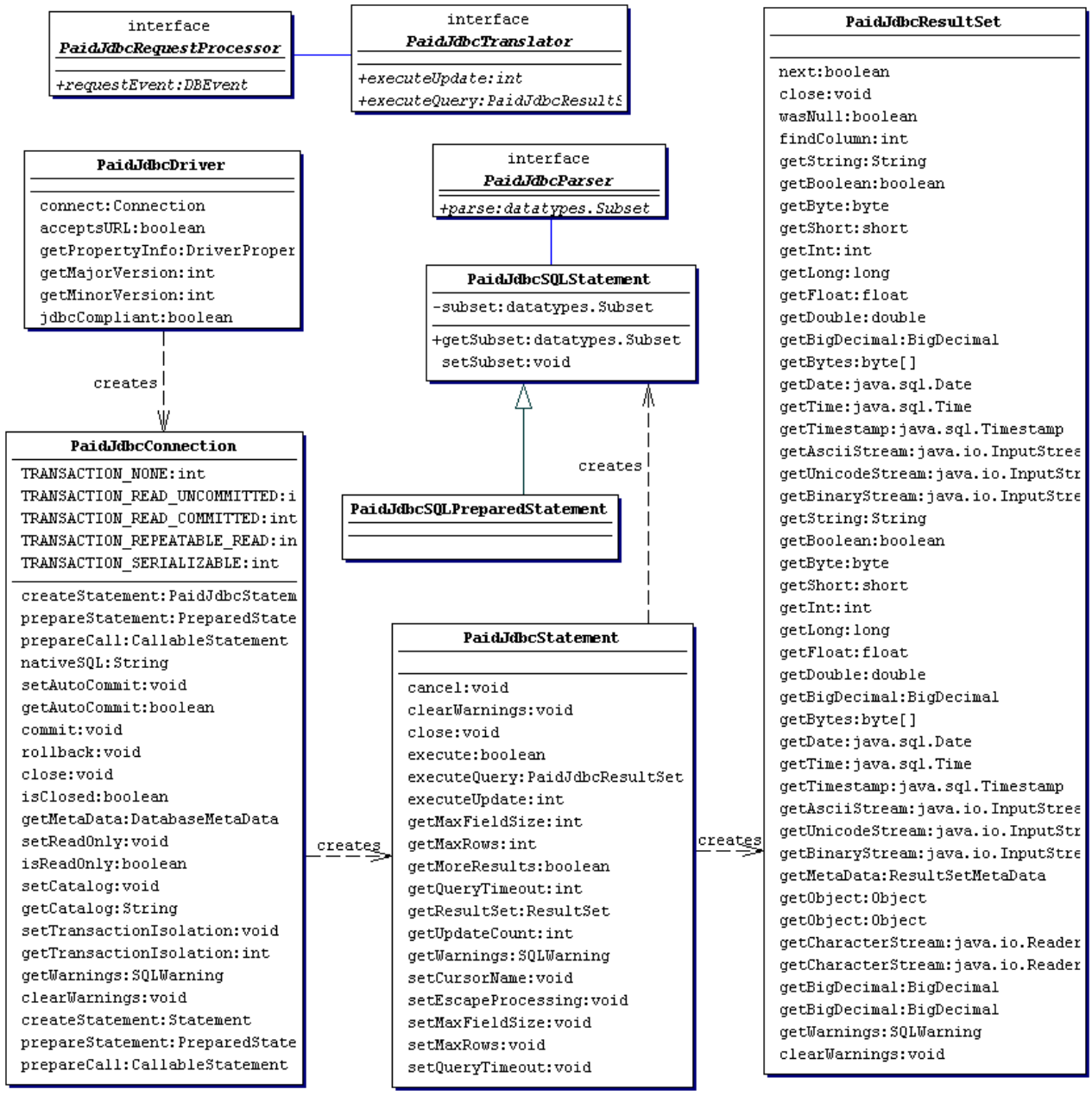


DBRemoveSubsetEvent Request to remove a specified Subset from the local Database.

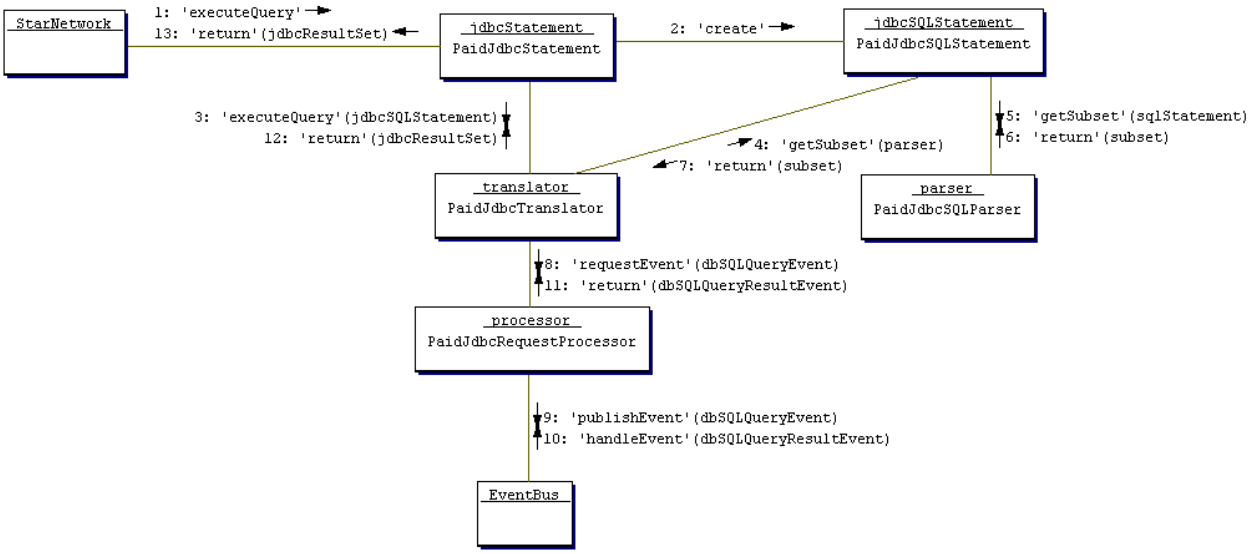
DBUpdateEvent This is an incoming Update for a Subset Data. The Database Subsystem will verifiy the Update's version information to ensure that it matches the currentliy stored Subset Data and then will execute the Update automatically.

DBGetSubsetDataEvent	Request to read a specific Subset Data from a given media and send a DBSubsetDataEvent as answer. If the given media is null, the Subset Data is extracted from the local database. If the Subset Data is not available in the local database, the event is handed on to the next parent server.
DBSubsetManagementResultEvent	This class implements DBResultEvent and contains a field for a maybe occuring exception. It is the parent for DBSubsetDataEvent and DBMediasEvent.
DBSubsetDataEvent	This is an incoming Subset Data. The Database Subsystem will verifiy the SubsetData's version to ensure that it's version is later than the version of the Subset Data currently stored in the local database and then will remove the currently stored Subset Data (if available) and add the new SubsetData in the local database.
DBGetMediasEvent	Request to list all available medias for Subset Data.
DBMediasEvent	The DBGetMediasEvent answer, which lists all available medias available on the local system.

2.4 JDBC driver



This implements a JDK 1.1 compliant JDBC driver, which parses the JDBC queries executed by StarNetwork, checks for the concerned Database Subset and translates them to events on the Event Bus. The ResultSet for these queries can be serialized when sent over the network and then represents a complete JDBC result set on destination side.



The PAID JDBC Driver / StarNetwork Collaboration is shown by the above diagram.

Incoming JDBC queries from StarNetwork are translated to PAID Events by the PaidJdbcTranslator using a PaidJdbcSQLParser. The PaidJdbcRequestProcessor is function/event translator which publishes the events on the Event Bus, waits for the answer event and returns it back to the translator. The translator extracts the Result Set and returns it to StarNetwork.

The 'DBSQLQueryEvent' published by the PaidJdbcRequestProcessor is answered by the Database Subsystem either from local database / local cache or via Network and is answered by a 'DBSQLQueryResultEvent' on the Event Bus.

3 Concurrency Identification

Since the database subsystem has to run in multiple threads there may occur concurrencies between them. This has to be resolved within this subsystem when shared objects like the localDBs are accessed from multiple threads simultaneously. Concurrent access on the local database should be resolved by the database management system itself. The PAID database subsystem must be thread safe.

4 Hardware/Software mapping

4.1 Database Subsystem Performance

4.1.1 General system performance

Because of the usage of Java as implementation language the general performance will be lower than a native implemented system. Though there exist just in time compilers for a variety of platforms and the processor speed duplicates every 18 months it might not be too interesting.

4.1.2 Input/Output performance

Since the sourcecode of StarNetwork will not be available and because of that reason it is not able to integrate directly into StarNetwork, a JDBC driver for transparent access of local or remote data has to be implemented. This means that every received SQL-Statement has to be parsed in order to extract the subset information. Furthermore every call to this driver has to be sent over the event bus, which may result in lower performance depending of the implementation of the event system. Of course, when a query cannot be answered from the local database, the performance depends on several other subsystems and the network connection. At this time no qualified statement can be made on the performance for local or remote requests. This has to be recovered by the testbed team using several scenarios.

4.1.3 Processor allocation

Since the PAID database subsystem is multithreaded and many requests may come in simultaneously, multiprocessor systems can be useful where performance problems can occur, for example the servers at the big dealerships can be equipped with more than one processors to handle the large numbers of possible requests.

4.1.4 Memory allocation

Because normally several requests to a PAID server may occur it is surely useful to have lots of memory on servers. However, it depends on the number of users accessing a server and the number of concurrent requests, how much memory the server needs, because for example every Resulset that must be sent over the network has to be hold in memory with its complete data. Of course, if the received SQL-statements result in too big Resultsets, the best solution may be to look at the sourcecode of the calling process or think about the database model.

4.2 Connectivity

Connectivity is handled by the Network Subsystem, which provides an event-based architecture to all other subsystems, which enables them to communicate with each other and over the network using events.

Events going up the hierarchy need no destination (besides 'up'), but events going down the hierarchy must have a destination. As the Database Subsystem does not send any events down the hierarchy without any initiating request, the Database Subsystem does not have to know, which client nodes are connected to it. All answers to incoming requests are routed down the way they came up.

4.3 Network architecture

The database subsystem needs the possibility to send request to parent servers and results to child servers. This should be provided by Network.

5 Data Management

The PAID system provides an easy access to distributed data in a transparent way. However, the data has to be divided into coherent subsets. How this is done is explained in Ingo Schneider's diploma thesis.

At this time it was decided that every PAID server needs a local database, but it would be possible to have servers without databases, too. In that case every request would have to be answered by the parent server and a permanent network connection would be needed.

In a final stage of PAID, StarNetwork's file access for images (and other data) has to be modified to use database queries. This can either be done by writing some kind of wrapper which translates these file accesses to database query events by the StarNetwork Integration Team or by modifying the application's source code to directly access the database instead of using other methods like direct file access or http-like protocols. The latter way is certainly the more reasonable one, but can only be done with the application's source code.

5.1 Data Types

The various types of data that will be used in the PAID system are:

Parts catalog (EPC)
Data on individual vehicles (FDOK)
PAID user information (user profiles)
PAID server information
Miscellaneous persistent data for all PAID subsystems.

These data will all be stored in an SQL databases. Any database management system that supports a JDBC driver is suitable.

In order to access any JDBC-capable database, the SQL statements have to follow the ANSI standard. Otherwise there might be difficulties in parsing them or executing them on a different database. If the ANSI standard does not fit on a specific database management system, the tool JConnect from Sybase may be helpful to translate between ANSI and DBMS specific SQL.

By the way, an object-oriented database could be helpful, but StarNetwork uses relational databases at this time. Therefore PAID has to work with it, too.

It is not supposed to provide a file system via PAID, instead all needed files like images will be stored in the database, too. This is done for reduction of the complexity of distributing changes. If files are in the database, for example in a BLOB field, the same way of updating, subscribing and so on can be used.

As PAID is implemented

5.2 Location Transparency

PAID provides a transparent access to data through the PAID JDBC driver. Calls to the driver are translated into events and answered by either the local database or any parent PAID server. In that way the data, the client application accesses, normally is distributed and accessible in a transparent way: The application does not have to know about the location of the data.

5.3 Data Organization

As mentioned above, data is divided into subsets, and the user has the possibility to install some of them locally and access the other ones remotely. Therefore it must be assured that at least the top PAID server has access to all available subsets.
When local data is corrupted in any way, it has to be reinstalled and all the updates that ocurred in the meanwhile have to be executed another time. The advantage of this behavior is that there is no need to archive the local databases. It might be nice to have local achiving, because recovery would be a little bit faster then, but it is not needed.

5.4 Server Loads

The only available data about average or worst case requests and data changes is extracted from Ingo Schneider's diploma thesis.

5.4.1 FDOK data traffic

The following table shows representative figures for FDOK data traffic. The updates shown are a total over 5 weeks.

Vehicle	Area	Update Size	Total Size	# of Vehicles
Utility	Europe w/o	9.8 MB	43.0 MB	23000
Utility	Germany	9.4 MB	31.0 MB	15000
Utility	Others	0.8 MB	2.1 MB	2000
Passenger	Europe w/o	2.2 MB	14.5 MB	35000
Passenger	Germany	3.8 MB	20.0 MB	48000
Passenger	America	1.2 MB	10.0 MB	21000
Passenger	Others	0.2 MB	0.9 MB	2000

5.4.2 EPC data traffic

Typical traffic for the Electronic Parts Catalog would be 3 MB/week for database updates, and 7 MB/week for image updates.

6 Global Resource Handling

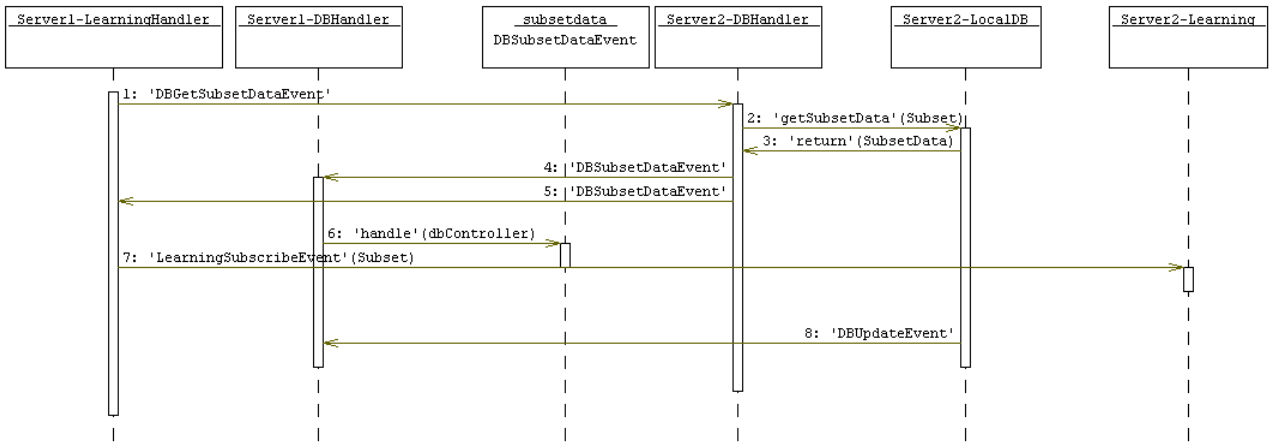
The main purpose of the database subsystem is to provide an efficient data management for the Daimler-Chrysler dealerships. Doing this the integrity of the data is of highest importance. This means that the data from its point of transmission to the point of reception is guaranteed to be safe and error free. Further the updates and manipulation of the data are also done in an efficient manner.

The data source is divided into two main classes i.e. media source (local data source CD, DVD, floppy and local cache etc.) and remote data source which is the set of all data located remotely with respect to the machine. The location of the data has been abstracted from the requester/manipulator of the data, this means that the database subsystem will try to locate and get data on its own without any input from the dealer.

7 Software Control Implementation

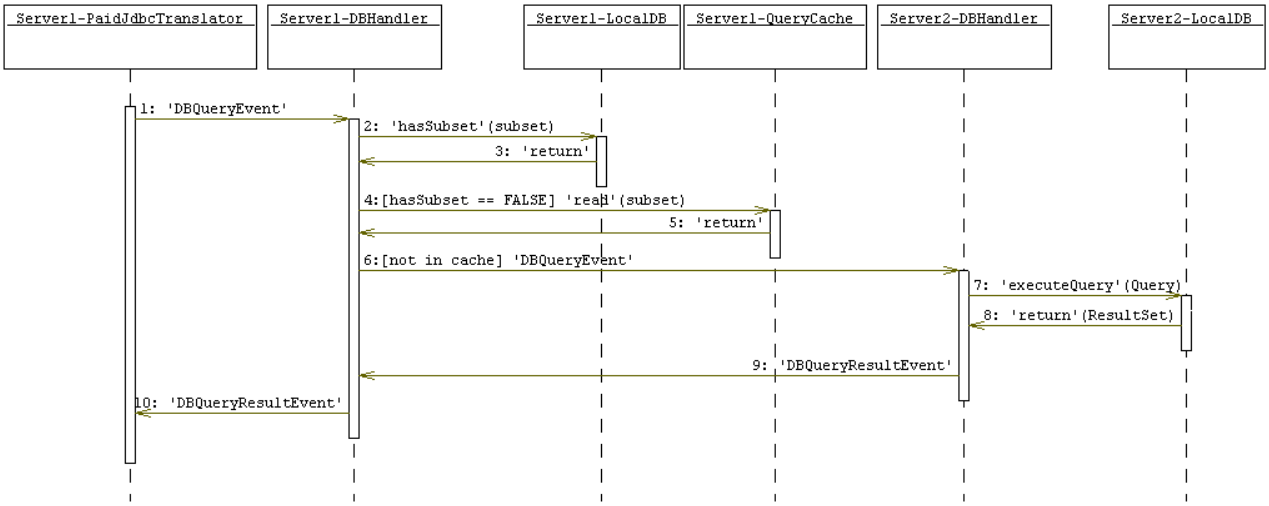
7.1 External control flow (between subsystems)

7.1.1 Adding a new Subset to local database



1. Learning Subsystem on server 1 sends a 'DBGetSubsetData' event to the (remote) server 2.
2. DBHandler on server 2 calls the 'getSubsetData'-method of the local DBController.
3. DBHandler receives the subset data and sends a 'DBSubsetDataEvent' to server 1, recieved by the Learning subsystem on server 1 and by the Database subsystem (DBHandler) on server 1.
4. DBHandler on server 1 calls the event's 'handle'-method to apply the data to the local database.
5. Learning subsystem on server 1 sends event to Learning subsystem on server 2 to subscribe for updates on this particular subset.
6. The first update is automatically initiated by Learing subsystem on server 2.
7. DBHandler on server 2 sends an 'DBUpdateEvent to server 1 (along with the update data).

7.1.2 Database query sequence diagram



1. StarNetwork subsystem on server 1 (precisely: the PaidJdbcTranslator-component) sends 'DBQueryEvent' to DBHandler on server 1.
2. DBHandler on server 1 calls method 'hasSubset' of the local DBController to determine if the subset is locally available.
3. DBHandler receives either 'TRUE' or 'FALSE'. If the answer is 'TRUE' it queries the local database for the specified data.
4. If the data is not locally available, the (local) cache is queried for it.
5. Cache returns the reqested data or a null pointer (marking the data not available)
6. If the Cache cannot provide the data, too, DBHandler sends a 'DBQueryEvent' to the (remote) server 2.
7. DBHandler on server 2 receives the data.
8. DBHandler on server 2 sends a 'DBQueryResultEvent' to DBHandler on server 1
9. DBHandler on server 1 sends a 'DBQueryResultEvent' to PaidJdbcTranslator (which returns the given Resultset to starnetwork).

7.2 Concurrent control

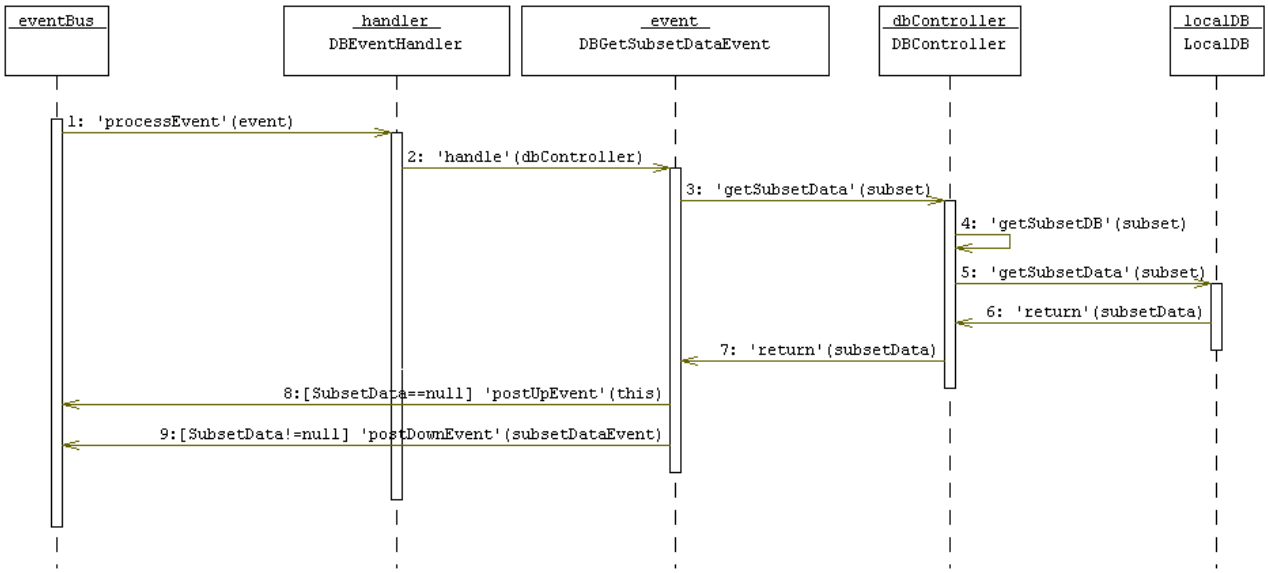
No concurrent control diagrams are needed, as this is mainly done by the database management system and the event-handlers (see main Chapter 'Concurrency Identification').

A later implementation of the PAID Database Subsystem should include transaction handling on DBSQLManipulateEvents to enable all aplications to manipulate data using multiple queries during one transactions.

The incoming updates, however, are handled using transactions.

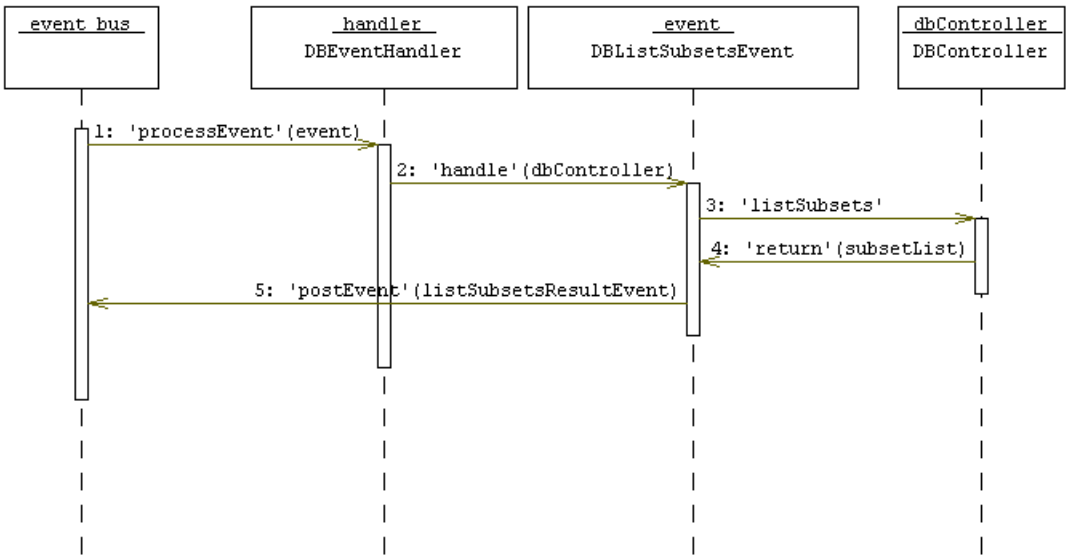
7.3 Internal control (within a single process)

7.3.1 DBGetSubsetDataEvent sequence diagram



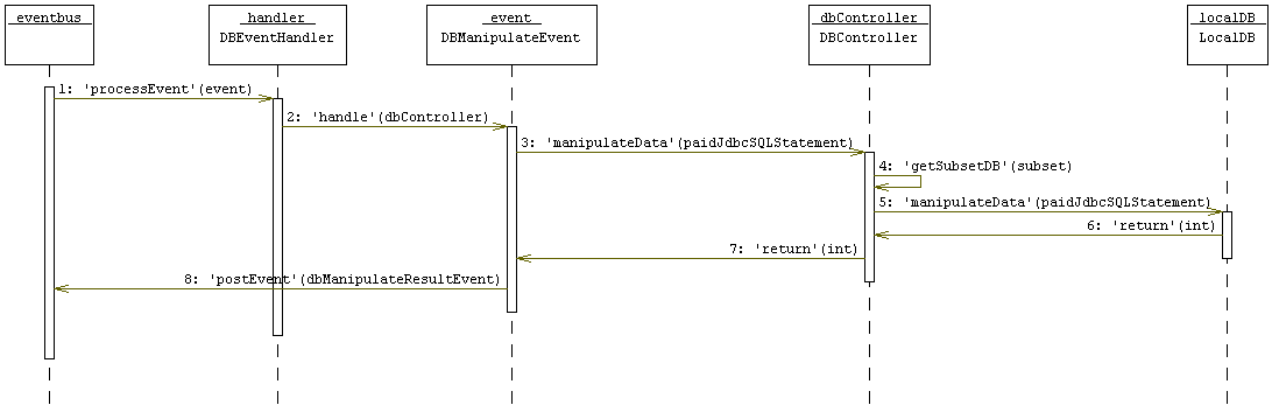
1. dbHandler receives a 'DBGetSubsetDataEvent' from the event bus.
2. dbHandler calls the 'handle'-method of the event and provides the local dbController as Parameter.
3. the event calls dbController.getSubsetData along with the requested subset's identifier.
4. dbController calls 'getSubsetDB' to find the local DB holding the specified subset's data
5. If the subset was found in one of the databases, the method 'getSubsetData' of this database is called.
6. The local DB returns the subset data
7. The dbController returns the subset data
8. If subsetData is a null pointer (means the subset is not locally available), the event posts itself ('DBGetSubsetEvent') to the remote servers event bus
9. In any other case, the event's handle method posts a 'DBSubsetData'-event (containig the subset data) to the event bus .

7.3.2 DBListSubsetsEvent sequence diagram



1. dbEventHandler receives a 'DBListSubsetsEvent' from the event bus.
2. dbEventHandler calls the event's 'handle'-method and gives the local dbController as parameter.
3. The event calls dbController.listSubsets.
4. dbController returns ist list of availaible subsets.
5. the event's 'handle'-method posts a 'DBListSubsetsResult'-event on the event bus.

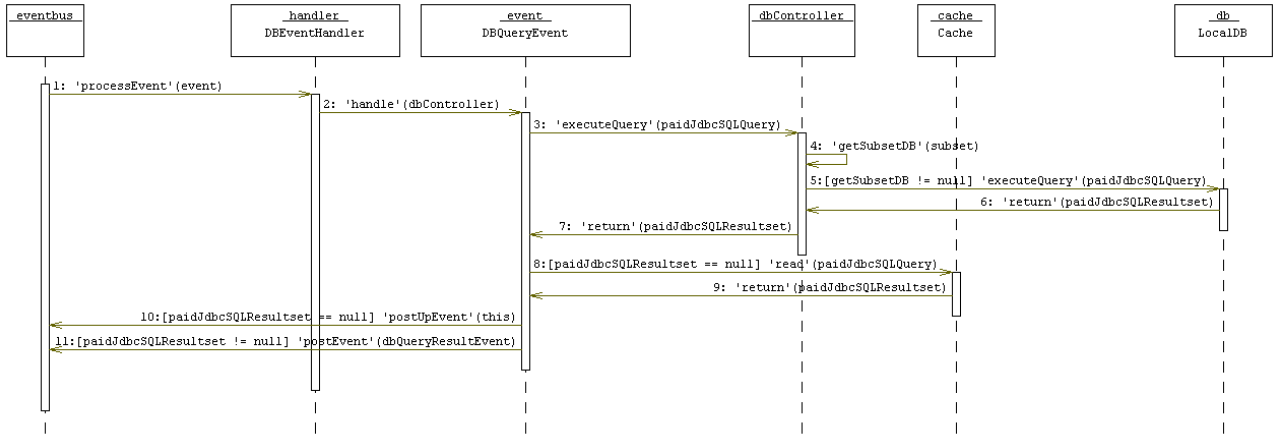
7.3.3 DBManipulateEvent sequence diagram



1. dbHandler receives a 'DBManipulateEvent' from the event bus.
2. dbHandler calls the event's 'handle'-method and gives the local dbController as parameter.
3. The event calls dbController.manipulateData along with the contained SQL statement.
4. dbController uses its 'getSubsetDB'-method to determine the local database holding the subset on which to apply the SQL statement.
5. dbController calls the 'manipulateData'-method of the local DB returned by 'getSubsetDB'.
6. The local database returns the number of affected rows.
7. dbController returns the number of affected rows to the event's 'handle'-method.

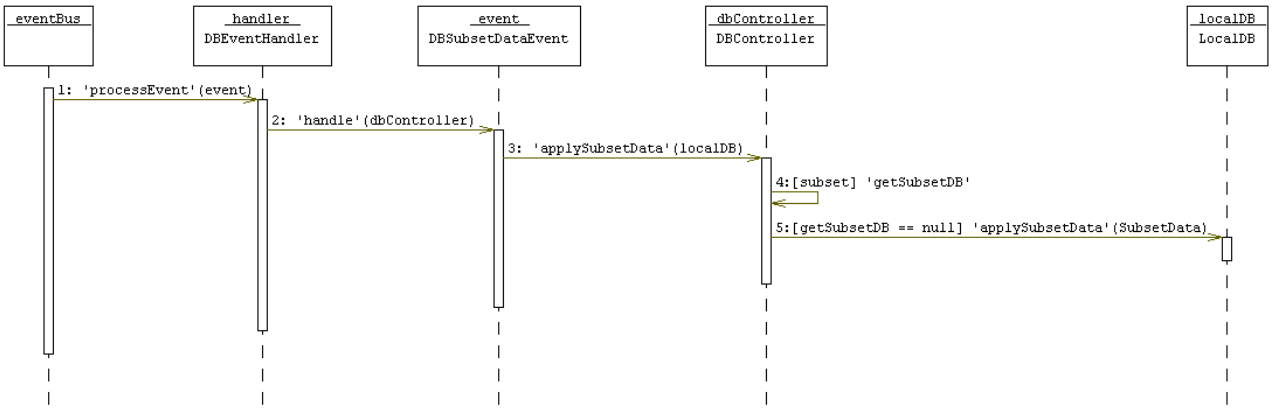
8. A 'DBManipulationResultEvent' containing the number of affected rows is posted on the event bus.

7.3.4 DBQueryEvent sequence diagram



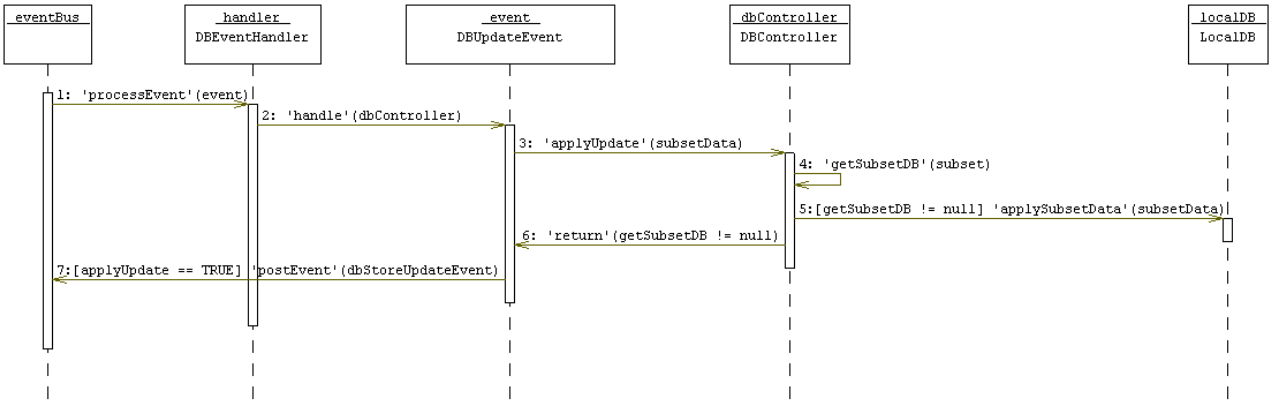
1. The event handler receives a 'DBQueryEvent' from the event bus.
2. The event handler calls the event's ' handle'-method along with the local dbController as parameter.
3. The event's 'handle'-method calls dbController.executeQuery along with the SQL query to be executed
4. dbController uses ist 'getSubsetDB'-method to determine the local database holding the subset on which the query should be executed.
5. If the result is not a null pointer, the 'executeQuery'-method of the local DB is executed
6. The local DB returns a resultset containing the requested data.
7. dbController returns either a resultset or a null pointer.
8. If the subset is not availaible in the local databases (a null pointer has been returned), local cache is read to find out if this query has been executed before.
9. The local cache returns either a resultset containing the data or a null pointer.
10. If the requested data is now available, a 'DBQueryResultEvent' is posted on the event bus.
11. If the cache returned a null pointer (the query has not been executed before), the event's 'handle'-method posts itself ('DBQueryEvent') on the parent server's event bus.

7.3.5 DBSubsetDataEvent sequence diagram



1. The event handler receives a 'DBSubsetDataEvent' from the event bus.
2. The event handler calls the event's 'handle'-method along with the local dbController as parameter.
3. The event's 'handle'-method calls the 'applySubsetData'-method of the dbController along with the subset data contained in the event.
4. dbController calls its 'getSubsetDB'-method to ensure that the subset is not already in the database.
5. If 'getSubsetDB' returned a null pointer, the most applicable local database is selected and ist 'applySubsetData'-method is called.

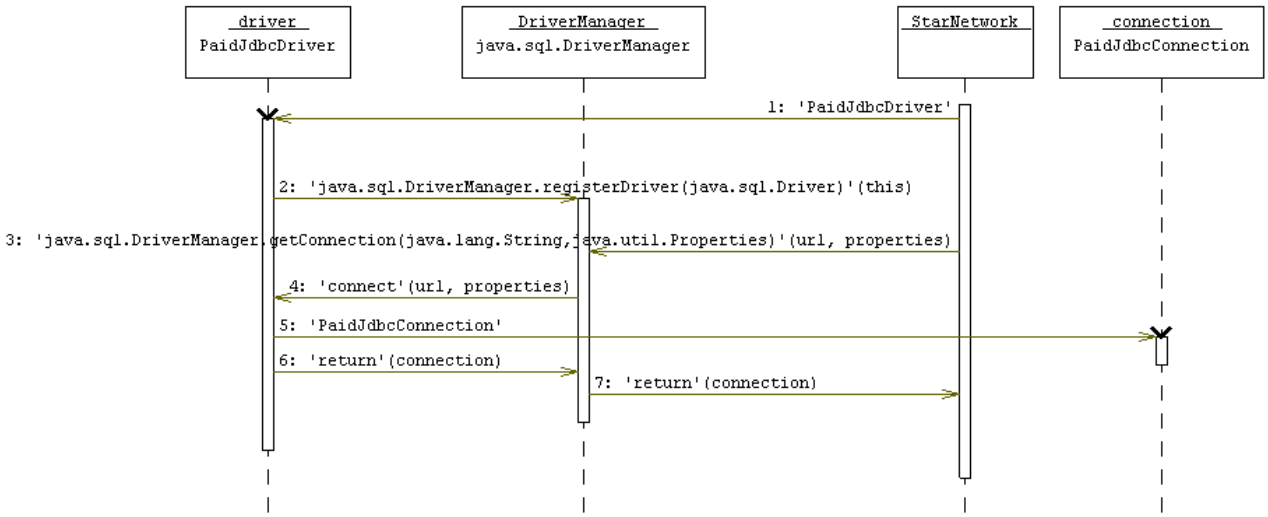
7.3.6 DBUpdateEvent sequence diagram



1. The event handler receives a 'DBUpdateEvent' from the event bus.
2. The event handler calls the 'handle'-method of the event and provides the local dbController as parameter.
3. 'dbController.applyUpdate' is called along with the subset data containing the update.

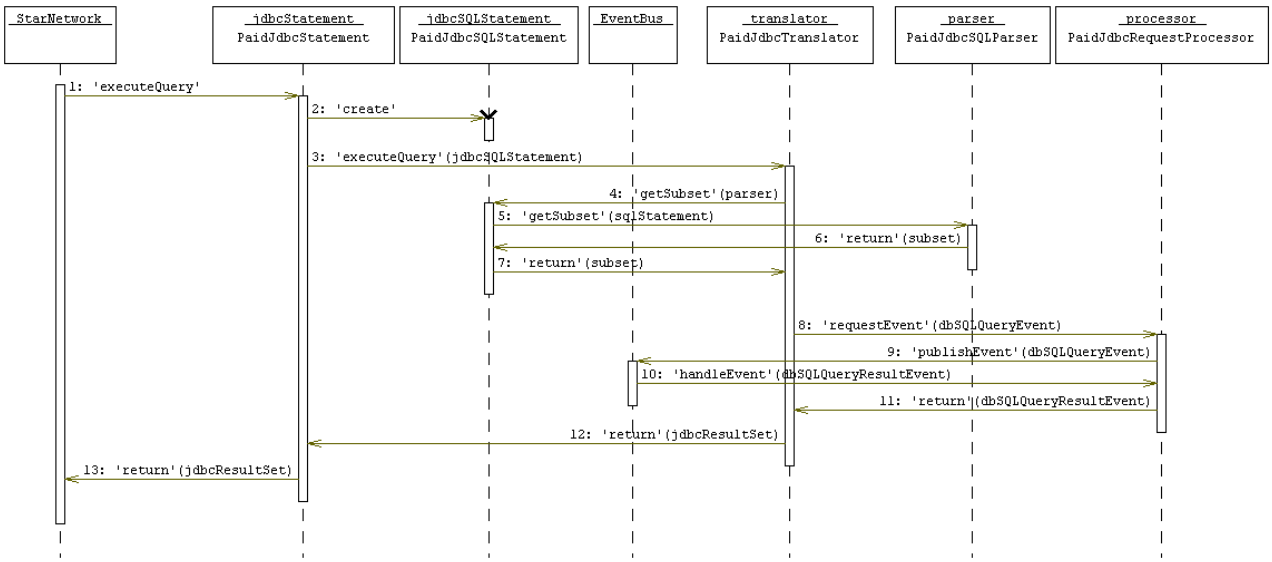
4. DhController uses its 'getSubsetDB' to determine the database holding the original (outdated) subset.
5. If the method didn't return a null pointer (which means that the subset is not in any of the local databases and therfore cannot be updated) the 'applySubsetData'-method of the local DB is called.
6. dbController returns if the operation was successfull.
7. If the operation was a success, a 'DBStoreUpdateEvent' is posted on the event bus to store the update data in the local update queue.

7.3.7 JDBC driver initialization sequence diagram



1. StarNetwork subsystem initiates creation of the paidJdbcDriver.
2. paidJdbcDriver registers at the DriverManager
3. StarNetwork requests a Jdbc connection from the driverManager.
4. driverManager calls 'paidJdbcDriver.connect' to connect to the given URL
5. paidJdbcDriver creates a 'PaidJdbcConnection'.
6. The driver returns the connection to the driverManager.
7. The driverManager returns the connection to StarNetwork

7.3.8 JDBC executeQuery sequence diagram



1. StarNetwork calls 'connection.createStatement'
2. connection creates a new instance of 'PaidJdbcStatement'
3. connection returns paidJdbcStatement to Starnetwork
4. StarNetwork calls 'paidJdbcStement.executeQuery' along with the query string
5. paidJdbStatement creates a new instance of 'PaidJdbcSQLStatement'
6. paidJdbStatement calls 'translator.executeQuery' along with the query string
7. translator calls 'paidJdbcSQLStement.getSubset' along with the parser (wich is an instance of PaidJdbcParser)
8. paidJdbcStatement extract the subset from the query string (using the parser) and returns it
9. translator uses 'generateEvent' to create a new 'DBQueryEvent'
10. translator calls 'processor.requestEvent' along with the newly created event and the result event to be returned
11. processor posts the event on the event bus
12. processor receives the incoming "DBQueryResultEvent" from the event bus
13. processor returns the containes paidJdbcResultset to translator
14. translator returns the resultset to StarNetwork

7.4 User Interface

The Database Subsystem has no user interface. All user interaction is handled by the StarNetwork Integration / User Interface Team, which provides some Events to show messages and dialogues to the User.

8 Boundary Conditions

8.1 Initialization

The JDBC driver is loaded by StarNetwork. This is done by either creating an instance of the driver class (it then normally registeres with the DriverManager) or by changing the Java system property jdbc.drivers and starting a new JVM.

The PAID Database Controller must be instanciated during system startup. The Database Controller then creates a JDBC connection to each database locally available. It also has to register the Database Event Handler at the Event Bus.

The Database Subsystem must be instanciated after the Event Bus is set up, so all other Subsystems can access their persistent storage during initialization. The Database Subsystem itself only depends on the Event Bus.

8.2 Termination

All threads are designed to terminate correctly without any loss of data or data inconsistency. That way, the system can be terminated at any state.

However, if an incoming event is already handled by the DBEventHandler, the termination procedure must wait until the handling of this event is finished. So, during system shutdown it is just necessary to stop all incoming events and the Database Subsystem will be idle within a minute.

8.3 Failure

The system is capable of handling all possible failures without any loss of data or data inconsistency.

9 Design Rationale

9.1 Data access

9.1.1 Non-intrusive implementation

Pro: non-intrusive implementation (as JDBC driver)	Pro: intrusive implementation (changing StarNetwork Code)
Can be done without source code	Can be implemented as at much lower cost (no parser necessary)
Client wants that	Efficient and less error-prone
	Requests can be tailored to use the complete database functionality
	Avoids any security problems that might occur when the SQL query and the affected subset are transmitted seperately, because this implementions allows it to have the SQL statement built at server side so that no server has to trust ist clients.
	Applications, that are not PAID-aware and use JDBC methods that can not be redirected, have very high integration costs

Resolution: we will implement PAID as a JDBC driver as no StarNetwork source code is available.

9.1.2 Unified Data Access Model (Location Transparency)

Pro: unified Data Access Model	Pro: a own class for each Data Access Type
Transparent access	Requester can specify the data source manually if he wants to
Requester does not have to know where the data comes from	
Much less code	

Resolution: Unified Data Access Model. However, the implemented algorithm which decides where the data is requested from, must be efficient and intelligent.

9.1.3 Persistent storage via SQL in Subsets

Pro: persistent storage via SQL in Subsets	**Pro: simple Data Storage**
More complex data structures can be stored.	Simpler and less code
Storage mechanism is unified to all other subsystems	Every other subsystem has to deal with SQL queries
Storage can be replicated if needed	

Resolution: storage via subsets, because this complex storage is needed by other subsystems

9.1.4 Stored data managed via object oriented Subsets

Pro: object oriented Subsets	**Pro: subset implementation as class model**
Independence of data model (no code changes when new applications are added)	Simpler to implement
Code kept simple and clear cut.	All existing queries must be changed to be conform with the subsets.

Resolution: storage via object oriented subsets, because independence of data model is required by the client. As we do not want to change all existing StarNetwork code, we will send the existing SQL query together with the affected subset. As we do not check the query whether it belongs to the specified subset, this is a security hole. This issue should be addressed before delivery.

9.1.5 Allowing all possible SQL queries

Pro: allowing all possible SQL queries	**Pro: reducing complexity (no predefined queries)**
Some existing queries have to be changed.	A lot easier to implement
Decreases the programmer's freedom	Quicker code
Runs without any code changes	

Resolution: we will allow any SQL query, because we cannot change the StarNetwork Code not to use predefined SQL queries.

9.2 Data storage

9.2.1 Simple query cache for query responses

Pro: simple query cache	Pro: complex storage mechanism
A clear separation between cache and aftersales database	If a new query just slightly differs to the last one, the old result can't be used to answer the new query
Cache can be expired easily	
A lot easier to implement. Storing query responses in the aftersales database is very difficult, as the atomic amount of manageable data is a subset and a query doesn't a complete new subset to the database	

Resolution: we will implement a simple query cache, as the performance that might be gained by implementing a complex storage mechanism would most certainly be lost by the much more complex caching algorithm.

9.2.2 No local data encryption

Pro: no local data encryption	Pro: only encrypted data is stored in DB
System will be less efficient	We don't depend on other encryption software (all-in-a-box)
Local data encryption doesn't make much sense. The local system itself can be seen as secure.	
This job can be done by someone else	

Resolution: we won't provide local data encryption, as this job can also be done by the local database system and/or local file system, if needed.

9.3 Replication

9.3.1 Simple Backward Replication algorithm

Pro: client-side database changes are simply inserted on parent server	**Pro: complex Backward Replication algorithm**
Client database updates are so small that no complex algorithm is needed: these changes don't depend on each other. Easy and efficient to implement	Avoiding any possible conflicts

Resolution: we will simply store the Client database updates, as there probably won't be any possibility of conflicts; besides the possibility is very low that two dealers will report any changes on the same car at the same time.

9.3.2 AddSubset packages are extracted when needed

Pro: packages for AddSubset are extracted form parent server's DB when needed	**Pro: packages are stored on the parent server and are extracted after each update**
This reduces the needed storage space by 50%, because otherwise all local data must be stored in the database and as a update package. An extracted update is guaranteed to be 100% 'fresh'	Response time for 'addSubset' would be much shorter

Resolution: we will extract the needed updates from local database. As it is does not happen very often that a dealer adds a subset via network, this can take some time.

9.3.3 Client can only add a subset when parent is subscribed

Pro: AddSubset can only be requested when parent server is subscribed to this subset	Pro: a subset can always be added
Easier to handle If the updates travel through the parent server anyway, then why not subscribe to that subset? This will simplify the dealer's decision which subsets he should subscribe to, because only reasonable subscribable subsets will be displayed.	The dealer has total freedom and can do whatever he wants to do, no matter if it makes sense or not. If a Server runs out of disk space, addSubset will still be possible.

Resolution: we will only allow an addSubset if the parent server is subscribed to this subset, because the other possibility does not make much sense.

9.3.4 Only notify when new update arrives

Pro: only send notification when a new update arrives	Pro: uncoming updates are pushed immediately
A busy dealer's connection won't flooded, if a new update arrives	Easier to implement; no complex learning algorithm is needed Multicast technology can be used more efficiently

Resolution: efficient update transport and code simplicity are not as important as a good connectivity when the dealer is busy.