Document Version 1.01

June 10, 2008

CORAL Server Project

Software Architecture Design Proposal for Client and Server

Andrea Valassi (CERN IT-DM)

1. Introduction

The purpose of this document is to propose a comprehensive architecture design for both the server-side and client-side software deliverables of the CORAL server project [1]. The main guideline behind the proposed strategy is that the system is best described in terms of pairs of client-side and server-side components which should be designed together, as each of the tasks performed by the client is mirrored by a closely related task performed by the server. Using an object-oriented approach, it is suggested that abstract interfaces should be used to provide loose coupling between the different components and minimize software dependencies. This design has been successfully used in the past to produce prototypes of the full system architecture, first using python [2], and more recently in C++ [3, 4].

By presenting a more general overview of the client/server interaction that is necessary to provide the required relational functionalities of CORAL, this note is meant to complement previous design documents that focused on lower-level aspects of the system, such as the server-side component responsible for the management of execution threads [5] or the TCP packet structure for the client/server communication protocol [6]. Some aspects of the architecture of the present code [7] are also briefly reviewed and suggestions are formulated about possible ways to improve it according to the proposed design.

2. System overview

The CORAL server project must produce two software deliverables: a server executable and a client plugin. The task of the client plugin (named CoralAccess for consistency with the other CORAL plugins) is to implement the user-level abstract interfaces defined in the RelationalAccess package, by forwarding to the server all requests that imply data exchange with remote database backends. The task of the CORAL server is to listen for incoming requests from clients, execute them locally using the relevant plugin (OracleAccess, MySQLAccess, SQLiteAccess or FrontierAccess) and send the results back to the appropriate client.

Communication between client and server is required to take place by the exchange of TCP packets. Packets should conform to a protocol [6] that has been agreed with the developers of a third software deliverable, an intermediate “proxy” server that may optionally sit between client and server, in order to cache client requests if so requested: in particular, the packet headers should contain a flag indicating whether each packet is cacheable. While the client and server components proposed here could also be reused for the development of the proxy cache [2, 3], this document will only focus on the design of the client plugin and of the server.

Execution flow for a user request

Users issue requests to CORAL by interacting with the abstract interfaces defined in the RelationalAccess package. As an example, let’s consider a user that wishes to bulk-insert 10 rows into an Oracle table, via the CORAL server. Through steps which will not be described here, let’s assume that the user has eventually retrieved an instance of the CoralAccess class implementing the IBulkOperation abstract interface. Let’s concentrate on what happens from this moment onwards: the user first calls 10 times “processNextIteration” to schedule each of 10 rows for insertion, then he calls “flush” to trigger the actual insertion into the database. The typical execution flow is the following.

1. Some of the user calls *do not result in any action against the CORAL server* (in the example considered, “processNextIteration” only triggers the client plugin to cache locally the 10 rows scheduled for insertion). Eventually, a user request (in this case, the “flush” command) triggers the client plugin to select the (request) data that should be sent to the server for remote processing (e.g. the 10 rows, and either the table name or a token for an existing remote bulk operation instance).

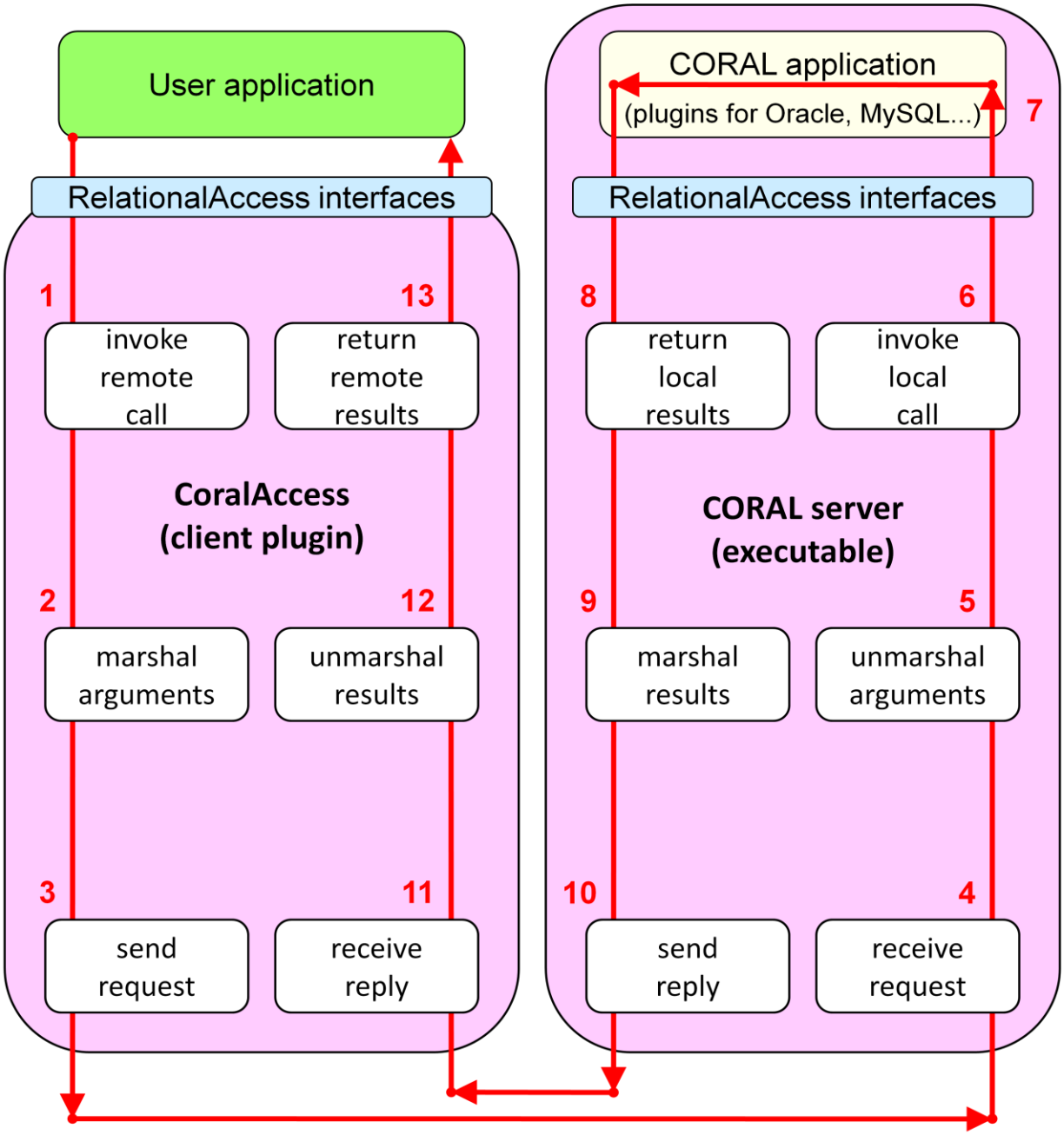

Figure 1 – Activity diagram for the execution of a client request on the CORAL server

2. The client plugin encodes (marshals) command and arguments into a message string (or byte stream).
3. The client plugin wraps the message into a TCP packet with the agreed header and sends it to the server over a socket.
4. The server executable (or one of its socket threads) receives the TCP packet, unwraps the message string and delegates its handling to a server execution thread.
5. The server execution thread decodes (unmarshals) command and arguments from the message string.
6. The server execution thread invokes the appropriate procedure(s) of a local CORAL application.
7. The CORAL application executes the appropriate action(s) by delegating them to the backend plugins (the OracleAccess plugin loaded on the CORAL server, in this case).

8. The server execution thread selects the (reply) data that should be sent back to the client (e.g. “OK” and the number of rows inserted, or “NOT OK” and an exception type and message).
9. The server execution thread encodes (marshals) results into a message string.
10. The server executable (or one of its socket threads) wraps the message into a TCP packet and sends it to the client.
11. The client plugin receives the TCP packet from the server and unwraps the message string.
12. The client plugin decodes (unmarshals) results from the message string (e.g. “OK” and the number of rows inserted, or “NOT OK” and an exception type and message).
13. The client plugin returns results (or throws an exception), thus giving back control to the user.

This execution flow described above is shown in Figure 1, where each of the 13 steps of processing is represented by a small box within one of the two large purple boxes on the left (the client plugin) and on the right (the server executable).

3. Design considerations

The analysis of the behavior of the system shown in Figure 1 suggests two considerations:

1. Each of the tasks that need to be performed by the client when delegating a request to the server is mirrored by a closely related task performed by the server. For instance: the client defines an operation against the remote database (e.g. insert 10 rows into a table with a given name), the server executes it; the client encodes the description of this operation into a message string, the server decodes it; the client sends a message wrapped as a TCP packet, the server receives the packet and dispatches the handling of the message. The same parallelism applies to the reception of the replies from the server.

2. The various tasks performed by the client are largely independent. For instance: when the client defines an operation that should be executed by the server against the remote database (e.g. insert 10 rows into a table with a given name), this is independent of how the client encodes the description of this operation into a message string (e.g. in binary or text format); also, the exchange of TCP packets between the client and the server is completely independent of the chosen encoding (e.g. binary or text) and of the fact that they contain the description of a relational operation (e.g. the same infrastructure could be used by the client to ask the server to evaluate “1+1”, expecting “2” as a reply). The same applies to the tasks performed by the server.

Using an object-oriented approach, and keeping in mind also the long-term maintainability of the software, these considerations lead to two design guidelines for the software architecture of the system:

1. Decompose the client and the server, each into three components with well defined responsibilities. For the client: defining a remote operation, encoding the description of the request into a message, sending the request packet. For the server: receiving the request packet, decoding its message into the description of an operation, executing the request. Each of the three components is also responsible for the corresponding task implied by the processing of the reply from the server. Use abstract interfaces to provide loose coupling between the three components on each side.

2. Adopt a 'horizontal' development model, where for each component a single team (and where appropriate a single package) is responsible for developing both the server and the client halves of it (packet sending and receiving; request decoding and encoding; query definition and execution). Test together the two halves of each component, in tests not involving the other two components.

4. System architecture and class design

The above design considerations and guidelines lead to a system architecture design that involves the two new abstract interfaces (blue) and six new sets of classes (yellow) represented in Figure 2. The diagram also includes existing CORAL interfaces (light blue) and classes (light yellow).

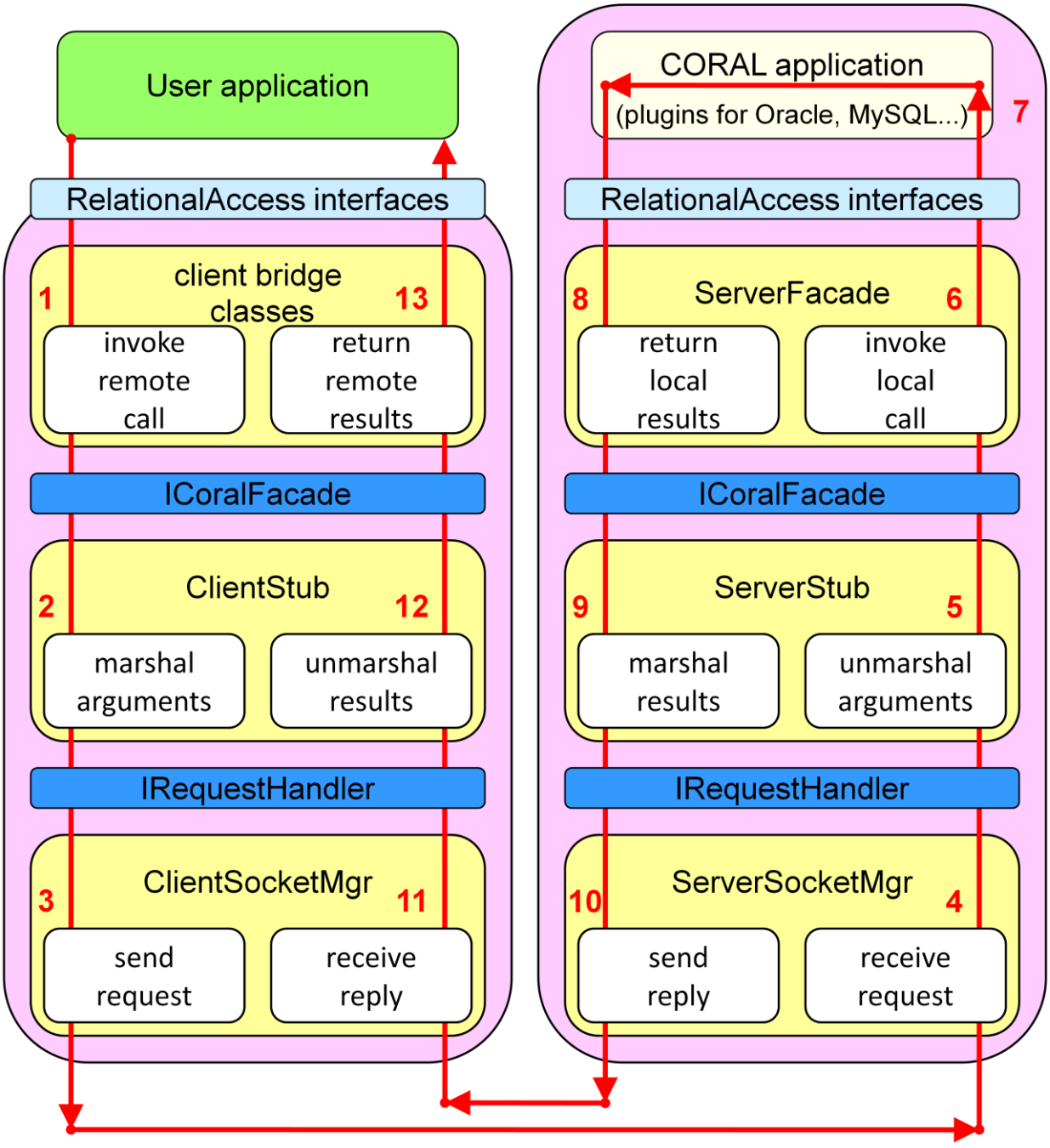

Figure 2 – Proposed component decomposition for the CORAL server and the client plugin

A more detailed description of the role and content of the two interfaces, and of the responsibilities of the six sets of classes follows below.

```
/// ObjectID type used for sessionID, cursorID, bulkOperationID (valid Token's are always > 0).
typedef int Token;

class ICoralFacade
{
public:
  /// Destructor.
  virtual ~ICoralFacade(){}
  […]

  /// Create a new session and return its sessionID.
  virtual const Token connect( const std::string& dbUrl,
                               const coral::AccessMode mode = coral::Update ) const = 0;

  /// Release a session
  virtual void releaseSession( const Token sessionID ) const = 0;
  […]

  /// Create a new cursor for the given query and return its cursor ID.
  virtual const Token executeQuery( const Token sessionID,
                                    const IPortableQueryDefinition& qd ) const = 0;

  /// Release a cursor
  virtual void releaseCursor( const Token cursorID ) const = 0;

  /// Fetch a new bulk of rows from the given cursor.
  virtual const std::vector<coral::AttributeList> fetchRows( const Token sessionID,
                                                             const Token cursorID,
                                                             const unsigned int maxSize ) const = 0;

  /// Fetch all rows from the cursor.
  virtual const std::vector<coral::AttributeList> fetchAllRows( const Token sessionID,
                                                                const IPortableQueryDefinition& qd ) const = 0;
  […]
};
```

Figure 3 – Definition of the ICoralFacade abstract interface

ICoralFacade

The architecture shown in Figure 1 is essentially that of an RPC (Remote Procedure Call [8]). Remote procedure execution, however, does not concern each and every call to a method of a RelationalAccess abstract interface. As pointed out above, some of the user calls do not result in immediate execution against the server. RPC rather applies to operations that are logically defined by the client plugin in step 1 (e.g., “bulk-insert these 10 rows into the table with this name”, or “bulk insert these 10 rows using the remote bulk operation identified by this token”) and are then executed by the server in step 6: these operations do not have

an immediate translation into a single method call of a RelationalAccess abstract interface, but they involve instead the execution of several such methods of the appropriate plugin (OracleAccess in this case) on the server.

Defining upfront the list of all such operations is extremely important as this corresponds to defining *which user-level requests require a communication over the network with the CORAL server*. This task must be accomplished at the very beginning of development, as this is the driving factor in the implementation of the client-side and server-side processing steps described in bullets 1-2 and 5-6 respectively. It is therefore suggested that an abstract interface ICoralFacade should be introduced to list all these operations. The name proposed indicates that a server-side class implementing this interface acts as a façade [9] to the CORAL RelationalAccess interfaces, as discussed more in detail below.

An incomplete prototype of the definition of the ICoralFacade interface is show in Figure 3. The list of methods presented here is far from complete: only the most basic methods for managing queries and connections are included, while many other methods should be included for managing transactions, DML operations (single-row and bulk insert, update, delete), DDL operations (create, drop or alter table, including options to add, delete or modify constraints such as primary, unique and foreign keys). This partial definition, however, is already useful to point out some aspects of the interface and of the system functionalities more in general.

1. *The CORAL server is normally stateful*. Several user calls result in acquiring remote database resources (e.g. an Oracle session or an Oracle cursor) which should be kept open until another user call releases them. In the CORAL RelationalAccess API, this generally happens when a method call returns a C++ pointer to a new instance that should be explicitly deleted by the user. In a three-tier architecture involving the CORAL server, an open cursor on a remote Oracle database is mirrored by a coral::ICursor instance in the client application, but also by a coral::ICursor instance in the CORAL server. It is the responsibility of the CORAL server to register the latter instance in an object store, to be able to locate it and either iterate over it or release it when so requested by the client. The present design assumes that an integer *Token* is used by the client plugin to manage operations associated to the remote cursor on the CORAL server.

2. While queries are one of the most important functionalities that the system should provide, from the software point of view they are only one of the several use cases involving a network round-trip between the client and the server, i.e. one of the many methods listed in the ICoralFacade interface. Two methods associated to queries could actually be used, one keeping an open cursor as described above, and one for the special case (requested by the developers of the proxy cache) where all rows should be retrieved at the same time. Both methods should use an abstract interface (here a new interface named IPortableQueryDefinition) to encapsulate all parameters of the query.

IRequestHandler

This abstract interface is meant to provide the client-side and server-side loose coupling to the components responsible for exchanging messages (over TCP) between the client and the server. From the client point of view, every call to a method of the ICoralFacade interface should result in one logical message being sent over TCP to the server, and it is expected that *every request message is answered by one and only one reply message*. As pointed out in bullets 8 and 12 of the Introduction, the reply message will typically contain a status indicating success or error, followed by the encoded return values of the ICoralFacade method, or an exception type and message. In both cases, anyway, one and only one reply should be delivered. The definition of this interface can thus be very simple, as shown in Figure 4. Classes implementing this interface

only need to contain a single method: when they receive a request message, they need to answer with a single reply message.

```
class IMessage
{
public:
  /// Destructor.
  virtual ~IMessage(){}

  /// Is this request cacheable?
  virtual const bool cacheable() const = 0;

  /// The message payload.
  virtual const std::string& payload() const = 0;
};

class IRequestHandler
{
public:
  /// Destructor.
  virtual ~IRequestHandler(){}

  /// Handle a request message and return a reply message.
  /// The ownership of the reply message is delegated to the caller.
  virtual std::auto_ptr<const IMessage> replyToRequest( const IMessage& request ) = 0;
};
```

Figure 4 – Definition of the IMessage and IRequestHandler abstract interface

The IMessage interface, also shown in the same figure, must contain a method to retrieve the message payload as a string or byte stream that can be sent over the network. In addition, this interface should also make it possible to retrieve any metadata associated to the message that should be written in the TCP packet header, such as the cacheable flag [6]. To make it clearer, the IMessage interface does not assume that messages implementing it will one day be wrapped according to the agreed packet header protocol [6]. Implementing the packet header protocol is the responsibility of a specific implementation and a specific client of the IRequestHandler interface, the ClientSocketMgr and ServerSocketMgr classes described below.

ClientSocketMgr and ServerSocketMgr

This is a pair of classes that should be developed together within the same package, as they provide mirroring functionalities. Fragments of their definitions are shown in Figure 5.

A ClientSocketMgr implements the IRequestHandler interface. Amongst its properties are the name and the port number of the server it should connect to, which are specified in the constructor. At creation time, this object opens a socket to the specified server, which remains until the object is deleted or until the server closes

the communication. Users of the ClientSocketMgr (like the ClientStub described below) give it a request message to be sent to the server and expect to receive back a single reply message.

Given the external requirement that client/server communication should take place by the exchange of TCP packets whose headers should conform to an agreed protocol [6], it is the responsibility of the ClientSocketMgr to ensure that this constraint is fulfilled: before sending a message to the server, this class should wrap it inside a TCP packet with the agreed header format.

Several client threads may use the same ClientSocketMgr, sending and receiving back messages asynchronously over the same socket: to this end, it is task of the ClientSocketMgr to write a new requestID into each new TCP packet before sending it to the server. The agreed packet header protocol [6] does offer the possibility to specify a requestID. After sending a request packet with a given requestID over the socket, it is the task of the ClientSocketMgr to wait for reply packets sent by the server, cache them in a local queue and distribute them to the correct requester, i.e. make sure that a request with a given requestID gets back the reply for that same requestID.

The size of messages sent over the ClientSocketMgr may exceed the TCP buffer size of the client socket or of the server socket: for this reason, it is also the responsibility of this class to write into the packet header the total size of the TCP packet being sent, as this allows the server to merge together fragmented packets. The agreed packet header protocol [6] does offer the possibility to specify the packet size.

```
class ClientSocketMgr : virtual public IRequestHandler
{
public:
  /// Constructor.
  ClientSocketMgr( const std::string& serverHost,
                   const int serverPort );
  […]
};

class ServerSocketMgr
{
public:
  /// Constructor from an IRequestHandler reference.
  ServerSocketMgr( IRequestHandler& handler,
                   const int port,
                   const int timeout=60,
                   const int nHandlerThreadsPerSocket=5 );
  […]
};
```

Figure 5 – Definitions of the ClientSocketMgr and ServerSocketMgr classes

A ServerSocketMgr does not implement any of the interfaces described above. It holds a reference to an IRequestHandler, which must be specified in the constructor. Amongst its properties are also a port number and the size of the pool of handler threads for each client socket, which may also be specified in the

constructor. The server-side socket manager listens on the specified port for client connections: when a new connection is requested, it opens a socket to the given client and spawns a socket thread to handle requests coming from that client. Each socket thread can have several handler threads associated to it (all sharing the same IRequestHandler, or each having their own IRequestHandler, in which case a reference to an IRequestHandler factory should rather be specified in the constructor). When a TCP packet arrives through the socket, the socket thread unwraps the message and delegates its handling to the IRequestHandler instance in one of the handler threads: this will then return a reply message, which the ServerSocketMgr has the responsibility to wrap into a TCP packet (with the correct requestID in its header) and send this back to the client.

Internally, the ServerSocketMgr could be implemented using the design proposed in a previous document [5]. To minimise software dependencies, it is however suggested that this functionality should be decoupled from request message handling by the use of the proposed IRequestHandler abstract interfaces.

ClientStub and ServerStub

This is another pair of classes that should be developed together within the same package, as they provide mirroring functionalities. Fragments of their definitions are shown in Figure 6. These classes are adapters [9] between the high-level ICoralFacade and the low-level IRequestHandler abstract interfaces. Their names, chosen from RPC [8] terminology, indicate that these classes have the task of translating a method call of the procedure being remotely executed (ICoralFacade) into data exchange between client and server using the relevant transport layer (IRequestHandler).

```
class ClientStub : virtual public ICoralFacade
{
public:
  /// Constructor from an IRequestHandler reference.
  ClientStub( IRequestHandler& handler );
  […]
};

class ServerStub : virtual public IRequestHandler
{
public:
  /// Constructor from an ICoralFacade reference.
  ServerStub( ICoralFacade& façade );
  […]
};
```

Figure 6 – Definitions of the ClientStub and ServerStub classes

The ClientStub class implements the ICoralFacade interface. An instance of this class holds a reference to an IRequestHandler interface, which must be specified in the constructor. The clients of this class are the CoralAccess “bridge” classes described below. When a client invokes an ICoralFacade procedure, this class is expected to: encode the opcode and arguments of this call into the payload of a request message; delegate

the handling of the message to the IRequestHandler and receive a reply message; decode the payload of the reply message into a return value for the ICoralFacade procedure (in case of success, or throw an exception of the appropriate type and with the appropriate message in terms of failure).

As mentioned above, messages exchanged by the IRequestHandler interface must have a flag indicating whether they are cacheable or not [6]. It is the responsibility of the ClientStub to decide which request packets are cacheable, depending on the ICoralFacade procedure called (and, more unlikely, on its arguments).

The ServerStub class implements the IRequestHandler interface. An instance of this class holds a reference to an ICoralFacade interface, which must be specified in the constructor. The client of this class is the ServerSocketMgr class described above. When a client asks the ServerStub to handle a request message, this class is expected to: decode the payload of the request message into the opcode and arguments of an ICoralFacade procedure; invoke the procedure of the referenced ICoralFacade; encode the value returned (or exception thrown) by the ICoralFacade into the payload of a reply message and give it back to the client.

Reply messages exchanged by the IRequestHandler interface are also expected to hold a flag indicating whether they are cacheable or not [6]. It is the responsibility of the ServerStub to decide which reply packets are cacheable. In the simplest implementations, a reply is cacheable if the corresponding request is cacheable.

It is the responsibility of the ClientStub and ServerStub components to choose a message encoding format, for instance using binary streams or readable text code. Internally, these two components and the messages they manipulate could be implemented in several ways, including by the use of templates as done in the present CoralServer code [7]. It is however suggested that the present code, which does largely follow the general architecture described in Figure 1, should be decomposed into components using abstract interfaces as suggested by Figure 2 in this document, to reduce software dependencies. In the proposed design, the specific choice of encoding cannot have any impact on the other components of the system, thanks to the decoupling provided by the ICoralFacade and IRequestHandler interfaces.

```
class Cursor : virtual public coral::ICursor
{
public:
  /// Constructor from an ICoralFacade reference.
  Cursor( ICoralFacade& façade,
          const Token sessionID,
          const Token cursorID,
          const IPortableQueryDefinition* qd,
          const int rowCacheSize );

  /// Positions the cursor to the next available row in the result set.
  /// If there are no more rows in the result set false is returned.
  bool next();
  […]
};
```

Figure 7 – Definition of the Cursor “bridge” class in the CoralAccess client plugin

Client “bridge” classes (CoralAccess) and ServerFacade

This is another pair of sets of classes that should be developed and tested together, as they provide mirroring functionalities. It is not suggested however that they should be part of the same package. Client “bridge” classes, on one side, and the ServerFacade, on the other side, provide the core relational functionalities of the client plugin and of the server executable: hence, they would naturally be released in two separate packages CoralAccess and CoralServer.

The term “bridge” classes is used here to indicate the many concrete classes in the CoralAccess client plugin, each of which implements one of the many user-level abstract interfaces defined in RelationalAccess. A possible incomplete definition of one such class, Cursor, is shown in Figure 7: this class holds a reference to an ICoralFacade, which must be specified in its constructor. The core relational functionality of the Cursor object is not implemented inside Cursor itself, but is forwarded to the ICoralFacade. For instance, iteration over the rows retrieved by a query cursor, as required by the coral::ICursor::next method, is delegated to the ICoralFacade::fetchRows method.

The term “bridge” was chosen to indicate that the use of the ICoralFacade on the client side decouples RelationalAccess abstract interfaces from their implementation according to the “bridge” pattern [9]. Indeed, two different implementations of ICoralFacade are proposed in this design document: the implementation for local processing provided by the ServerFacade class described below, and the RPC implementation provided by the ClientStub described in a previous paragraph. The ClientStub class has the same abstract interface as the ServerFacade and it represents a proxy [9] to the latter.

While operation of the system in the full client/server mode requires CoralAccess classes to hold a reference to a ClientStub for remote processing on the CORAL server, the interest of the modular architecture proposed here is that it is also possible to test the same CoralAccess classes against the ServerFacade directly, bypassing the encoding of procedure calls into messages and their exchange as TCP packets over the network. *The complete functionality of the CoralAccess bridge classes and of the CoralServer ServerFacade class can be developed and tested even if the implementation of the client and server stubs or socket managers is missing or incomplete.*

```
class ServerFacade : virtual public ICoralFacade
{
public:
  /// Constructor from a CORAL connection service reference.
  ServerFacade( coral::ConnectionService& connSvc );
  […]

private:
  /// The object store manager (owned by this instance).
  IObjectStoreMgr* m_storeMgr;
  […]
};
```

Figure 8 – Definition of the ServerFacade class

The ServerFacade class implements the ICoralFacade interface. Fragments of its definition are shown in Figure 8. This is the core of the CORAL server: it is a CORAL application running on the server, which holds a reference to a CORAL connection service (specified in its constructor) and uses it to process all relational requests coming from remote clients, using locally loaded plugins for the appropriate database technology. This class is a façade [9] to CORAL RelationalAccess: it manipulates various instances of concrete classes that implement one of the RelationalAccess interfaces for one of the database backend plugins, while these objects hold no reference or pointer to the façade.

As pointed out in a previous section, database resources which are open on the remote database backends (such as Oracle sessions or cursors) must be mirrored by object instances (ISessionProxy or ICursor in the example) in the CORAL server. It is the responsibility of the ServerFacade class to maintain a list of all object instances which have been created by the client plugin and not yet released, and to be able to access them using their object ID to perform relational operations. It is suggested here that this task could be delegated internally to an IObjectStoreMgr, owned by the ServerFacade.

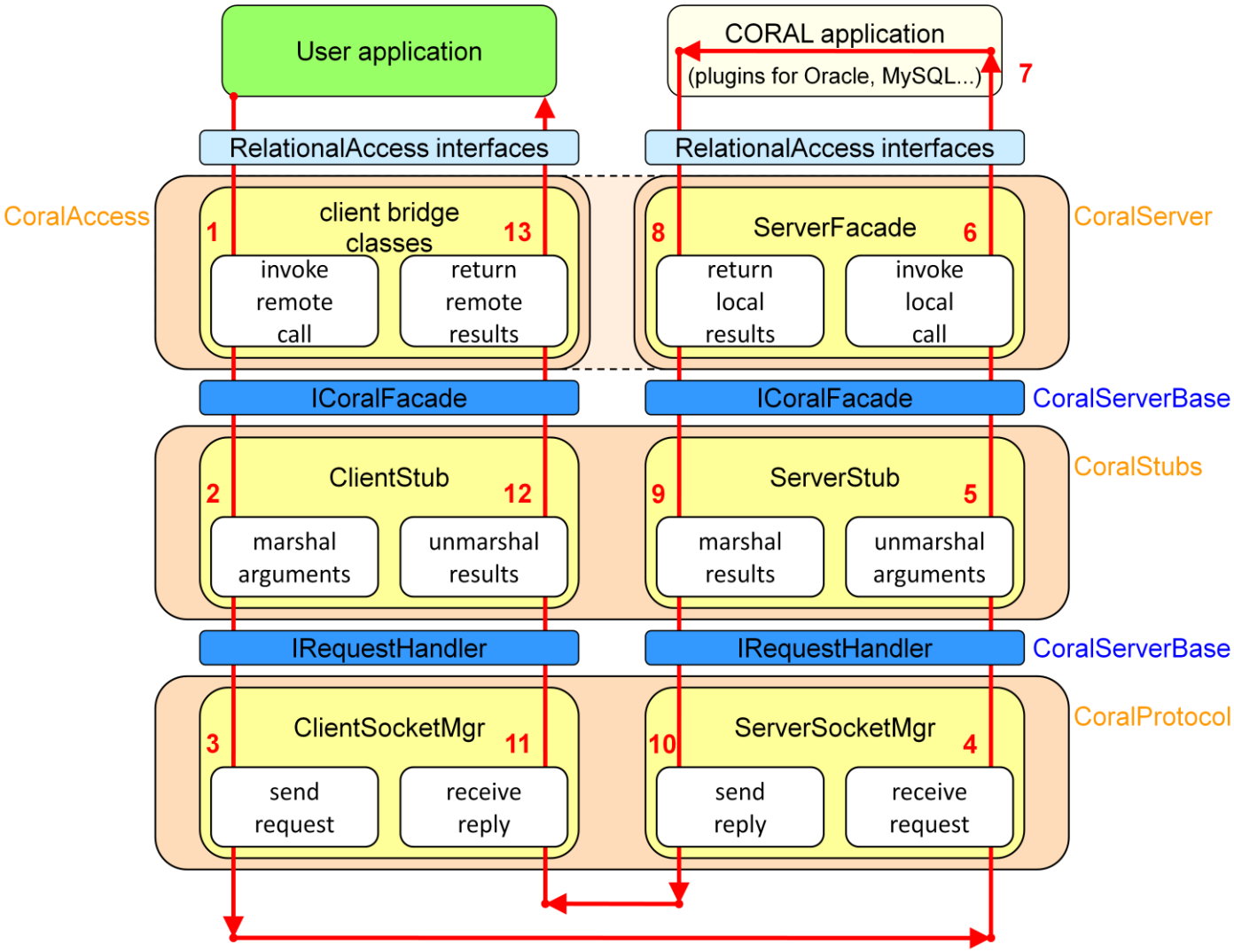

Figure 9 – Proposed package decomposition for the CORAL server and the client plugin

5. Package decomposition and test design

Keeping into account the general design considerations presented in section 3, as well as the various comments about the specific classes proposed in section 4, it is suggested to split the CORAL server project software into five packages as shown in Figure 9. The list of proposed packages is the following:

1. **CoralServerBase**. This base package contains all the abstract interfaces and type definitions that are shared by concrete components in two or more other packages: ICoralFacade (including Token and IPortableQueryDefinition) and IRequestHandler (including IMessage). It is mainly meant to contain public headers, but may optionally contain the simplest concrete implementations of some interfaces (such as a SimpleMessage or a DummyRequestHandler for tests).

   All of the other four packages below depend on CoralServerBase, which in most cases provides the only (loose) coupling between them. *Agreeing on the definition of the abstract interfaces in CoralServerBase should therefore be the very first step of development: this would allow the developers of the different implementation packages described below to work in parallel without being affected by the status of development in the other packages.*

2. **CoralProtocol** (or CoralSockets). This package provides a library that contains both the ClientSocketMgr and the ServerSocketMgr: only their headers are public, while any collaborating classes are defined in the private source area. This package should also contain tests of the ClientSocketMgr against the ServerSocketMgr. The tests may involve a DummySocketServer executable using a DummyRequestHandler to return a reply message that is essentially the same as the request message. It would be useful to prepare the following tests: test that all 256 characters can be exchanged correctly; test that very long messages are fragmented and merged back correctly; test that the server is able to reply to several client threads in parallel, and that the client is able to distribute replies correctly according to the incoming requestID.

   In addition to the above, this package may also expose in its public header directory the definition of the C++ class describing the TCP packet header conforming to the agreed protocol, if this is required by the developers of the proxy cache (as it is the case for the present code). The development of the proxy cache could actually benefit by the reuse of the full ClientSocketMgr and ServerSocketMgr classes: this has been briefly reviewed elsewhere [2, 3], but is beyond the scope of this document.

3. **CoralStubs** (or CoralStreamers, or CoralMarshal). This package provides a library that contains both the ClientStub and the ServerStub: only their headers are public, while any collaborating classes are defined in the private source area. It is suggested that the implementation of these two components should be based on a lower level streaming library for streaming vectors of simple types and coral::AttributeList's to byte stream, and that encoding and decoding using this library should be extensively cross-tested.

4. **CoralServer**. This package provides a library that contains the ServerFacade class: only its header is public, while any collaborating classes (such as the IObjectStoreMgr and its concrete implementation) are defined in the private source area. The library only depends on CoralServerBase and on RelationalAccess.

   In addition, this package also contains the CORAL server executable, requiring a dependency on CoralStubs and CoralProtocol. The implementation of the executable only needs to instantiate a ConnectionService, a ServerFacade using the service, a ServerStub using the façade and finally a SocketMgr using the stub.

5. **CoralAccess**. This package provides a library that contains all of the client bridge classes. Like the other CORAL plugins, it has no public headers, as all the implementation code is private. Most of the library only depends on CoralServerBase and on RelationalAccess, as relational functionalities are implemented in terms of an abstract ICoralFacade interface (using the bridge pattern).

   One class however needs to choose a specific ICoralFacade implementation that should be used by all bridge classes: this typically happens in the Connection class, as this is where a socket is opened to the server. The standard client plugin must therefore depend also on the CoralStubs and CoralProtocol packages. If the plugin receives a request to connect to a database via the CORAL server, the Connection class instantiates a ClientSocketMgr towards the server and a ClientStub using this socket manager, then passes the ICoralFacade stub to all bridge classes.

   It is also recommended, however, that an option should be set in the plugin for tests (for instance, using a URL suffix, "coral_LOCAL://"), making it possible to connect to a local ServerFacade directly rather than through the network. A dependency of CoralAccess on the CoralServer library is also required in that case.The full CORAL test suite, or any selected subsample of it, could be used to test extensively the functionalities of the CoralAccess bridge classes against the ServerFacade class.

Acknowledgements

The design proposed in this document has greatly benefitted from useful discussions with the other members of the CORAL team in IT-DM (Radovan Chytracek, Dirk Duellmann, Giacomo Govi, Alexander Kalkhof, Zsolt Molnar, Ioannis Papadopoulos, Witold Pokorski) and with the members of the Atlas-online SLAC team (Rainer Bartoldus, Sarah Demers, Andy Salnikov, Dong Su). All these discussions are gratefully acknowledged.

References

1. Z. Molnar, "Coral Proxy Server and Connection Pool" (January 2008)
2. A. Valassi, "Coral Server python Prototype" (November 2007)
3. A. Valassi, "Coral Server C++ Design Proposal" (April 2008)
4. A. Valassi, CoralServerPrototype source code in COOL CVS
5. Z. Molnar, "Coral Server Software Design Description" (January 2008)
6. Z. Molnar, "Coral Server Protocol" (January – May 2008)
7. Z. Molnar et al., CoralServer, CoralProtocol and CoralAccess source code in CORAL CVS
8. A. Birrell and B. J. Nelson, "Implem+enting Remote Procedure Calls" (1984)
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns", Addison-Wesley (1995)