Network and database connection

- Establish network connection between the CORAL client and CORAL server
- Establish database connection between the CORAL client and the CORAL server

Solution
Acceptor-Connector pattern for both problems

Info
http://www.cs.wustl.edu/~schmidt/PDF/Acc-Con.pdf

UML
Acceptor-Connector.jpg

Explanation

- The server must handle multiple TCP connections: TCP *event demultiplexing* is necessary.
- The server must listen to multiple passive ports: *data channel, control channel*
  - Only data channel is implemented for the moment
- The server must establish database connection
  - One per TCP connection for the moment
- The server must handle multiple client connections over an established TCP/database connection: *client demultiplexing*. The TCP operations and the relational operations follow the same pattern (establish connection, then handle multiplexed events), therefore the *Acceptor-Connector* pattern is an ideal candidate for both. The UML diagram summarizes the two realizations of the pattern, and their interaction.

Event Demultiplexing

- Need to demultiplex different TCP events (connection request, connection dropped, data arrived)
- Need to demultiplex messages coming from th shared TCP channel

Solution: Reactor pattern for both problems
Info: http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf
UML: Acceptor-Connector.jpg

Explanation
See the event demultiplexing aspect in Problem 1.

Pattern usage

- The classes implementing the *Reactor* are:
  - **CoralServer::Dispatcher**
    - Event handlers are registered in runtime
    - The server may be parametrized to use or not the control channel, need to define ports in runtime, etc.
    - The event loop is in the *_start_a* active method
    - Handle TCP connection requests
    - Dispatch to *CoralDataAcceptor* or *CoralControlAcceptor*
    - *CoralControlAcceptor* is not implemented yet!
  - **CoralServer::CoralDataAcceptor**
    - Read and reconstruct messages in the event loop
    - The class is an *Acceptor* in TCP context, but *Dispatcher* in CORAL messaging context
    - *Acceptor* role in *handleEvents* method (confusing, rename to *accept*!)
    - *Dispatcher* role in *_startMessageHandling_a* active method (confusing, rename to *dispatch!*): the event loop is here.
    - Event handlers are registered in compile time
    - All the events (messages) are known in compile time
  - Common properties:
    - Event loop implemented in active methods (both are *Active Objects*)

Concurrency

The server must serve multiple concurrent requests.

Solution: Through *Reactive Synchronous Event Dispatching* (See Problem 2). The event handlers are *Active Objects*-s. The concurrency model behind the active objects is using of multiple synchronous threads. The threading resources follow the *Thread Pool* pattern. The *Active Object*-s encapsulate the concrete tasks by using the *Command* pattern.
Info: http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf : *Active Object* pattern
http://en.wikipedia.org/wiki/Command_pattern : *Command* pattern
http://en.wikipedia.org/wiki/Thread_pool : *Thread Pool* pattern

UML: *Thread Pool Structure.jpg*: class diagram for the *Thread Pool* and *Active Object* aspects. The reference documents summarizes the dynamics: the Coral Server objects can be mapped to them easily.

Implementation
The static and dynamic structure follow the above documents. Some remarks:

- The threading resources are encapsulated in the *Worker* objects
- All the *Workers* execute an event loop (see the doc) and race for *Tasks*. They wait when the *SynchronizedQueue* is empty. When *Tasks* are queued, all the *Workers* are waken up, and one of them picks up a *Task*.
- The *Tasks* are created by the active methods: an active method creates a *Task* object, and encapsulates an implementation (servant) method of the active object into it. The *Task* is scheduled by using the *ThreadPool::enqueue()* method, then the active method returns immediately. The *Task* will in fact include a pointer to the object, a pointer to the method to be called, and a copy of (or reference to) the method parameters.
- The implementation (*Servant*) method is executed later, when a *Worker* pops the corresponding *Task*.
- The above mechanism follows the *Command* pattern, see the dynamics in the references..
- When the *SynchronizedQueue* is full, the *ThreadPool::enqueue* (so the active method) blocks.
- All the classes providing active methods inherit from ActiveObject class. This class provides some basic services to the active objects like:
  - Template to create *Command*-s that
  - Encapsulating *Command*-s to *Tasks*, and enqueue them
  - Lifetime management: the object is not deleted until all the running active method returned
  - Exception management: an exception does not propagate out of the active object. In case of exception, the active method is terminated, and the *Worker* picks up another task.
  - The threading model is a template *Policy*. It means that an active object can be turned into an ordinary object (synchronous execution) by simply changing the policy. For the Policy based design see *Andrei Alexandrescu: Modern C++ Design*.

Communication between the client and the server

Extends the TCP socket concept: *CoralSocket*. The *CoralSocket* encapsulates a request/reply message exchange context containing IP address, TCP port, client ID, request ID.

UML: CoralSocket.jpg

Info: *Dependency Injection* pattern (see for instance Dr. Dobb's Journal, #384, page 10)

	TCP socket	**CORAL socket**

Example	`struct sockaddr_in addr;`
	`int sock = socket(PF_INET, SOCK_STREAM, 0);`
	`send(fd, buffer, length, 0);`
		`Connect msg;`
		`TcpSocket tcpSock(created somehow);`
		`CoralSocket sock(tcpSock, clientID, requestID, opcode, cacheable);`
		`msg.write(sock);`

What	Raw data (buffer, length)	Structured, limited set of message objects
Destination	A network host	A CORAL client
How	SOCK_STREAM: select TCP/IP as transport layer	TcpSocket: select TCP/IP as transport layer
Lifetime	During the whole connection	Limited to sending one message only. The *CoralSocket* lifetime is independent from the underlaying (connected) *TcpSocket*.

Design for testability: The *CoralSocket* is isolated from the transport by the *ICoralIOHandler* interface. So, the messaging system may be tested without the networking subsystem, using the *Dependency Injection* pattern and mock message objects (*DataStringStream* implementation of *ICoralIOHandler*, instead of *TcpSocket*). Example: in the *CoralMessaging* unit tests, see the *CoralExceptionTest.cpp*, for example (*_checkAnException* method).

Message exchange example between the client and the server:

1. client connects to the server
    1. create a *TcpSocket*, by using *TcpSocketFactory::CreateConnectingSocket*
    2. Transport layer connection established
2. client creates a concrete message object
    1. the message knows about its opcode
    2. client sets the caching policy
3. client creates a *CoralSocket* object
    1. gives the *TcpSocket* as parameter
    2. specifies its *client ID*

    3. specifies a *request ID*
4. client writes the message into the *CoralSocket*
    1. The message encodes itself into a raw string representation
    2. The string is sent over the network
5. The server receives the raw string from the networking
6. The server reconstructs the message
7. The server creates a *CoralInputSocket*, that encapsulates the client TCP/IP properties, the client ID, the request ID
8. The *CoralInputSocket* and the message object is passed to the message handler
9. The server executes the encoded relational operation
10. The server creates a reply message
11. The server creates a *CoralSocket*, by using the passed *CoralInputSocket* so that the reply contains the proper client and request ID-s
12. The server writes the reply message into the *CoralSocket*
13. The client receives the reply message.

Server control channel, statistics

It is a named pipe. The user can write the following commands to the pipe:

- logon: switch logging on
- logoff: switch logging off
- stat: display the statistics

The debug channel handler is implemented in the DebugController class that is an active object (the controller is executed inside a Worker, in a different thread). The pipe can be accessed by */tmp/CoralServer.<pid>*

The messaging statistics data is collected by *CoralMessaging::Utils::StatisticsCollector*, that is a synchronized *Singleton*.

Synchronizing, locking

Info:

Monitor Object pattern: http://www.cs.wustl.edu/~schmidt/PDF/monitor.pdf

Strategized locking, scoped locking, thread-safe interface:
http://www.cs.wustl.edu/~schmidt/PDF/locking-patterns.pdf

To serialize access on concurrently used objects, we use the *Monitor Object* pattern. An object that requires synchronized access must derive from *CoralBricks::MonitorObject*.

The critical sections of the methods must be protected by using the following macros:

```
void A_Synchronized_Object::method()
{
MONITOR_START_CRITICAL
  // ordinary code
MONITOR_END_CRITIAL
}
```

The macros implement the synchronization by the *Strategized Scoped Locking* pattern. The implementation is provided by the *Boost::thread* library.

All the classes requiring synchronized access are designed and implemented using the *Thread-safe Interface* pattern: it avoids self-deadlocks and provides minimal locking overhead.

CoralMessaging

Implements the message exchange system, according to the *CORAL Server Protocol Description*. The main functionalities:

- The *CoralSocket* implementation
  - To send a message
  - Establish a message exchange context
- *ICoralIOHandler* interface
  - encapsulate an isolate the transport layer
  - defines a *Dependency Injection* point, to be able to unit-test the messaging and the relational layer without networking and the server
- Implementations for the *ICoralIOHandler*
  - *TcpSocket* subsystem
    - factory to create different TCP sockets (active, connecting, listening)
    - TCP event demultiplexer
    - *TcpSocket*: the POSIX socket is encapsulated in here
  - *DataStringStream* subsystem
    - Mainly for tests
    - network-less message exchange between software components
- The message set
  - All the messages encapsulate a relational operation, follow a well-defined protocol
  - Messages implements the *IMessage* interface
  - Messages encodes themselves into raw binary data
  - Messages decodes themselves from raw binary data
  - Implement the *Coral Application Layer* protocol
- Some common utilities
  - Error handling
  - Logging
  - Statistics collection
- *MessageReceiver*
  - Messages are events – the subsystem provides the event demultiplexing
  - Sends/receives messages, according to the *Coral Transport Layer* protocol
  - Segments/reconstructs messages
  - Handles transport problems, it is hidden from the user
  - The message receiver returns full, reconstructed, valid messages only, so only those messages represent events
  - Invalid messages discarded transparently.

MessageReceiver

The main CORAL message / event demultiplexer. Implemented in *CoralMessaging*.
The CoralServer and CoralAccess modules use it differently:

CoralServer

- In the event loop of *CoralDataAcceptor*
  - The only point in the server code that uses the *MessageReceiver*
- Wait for and receive messages continuously
- Any received message represents an event
- The associated relational operation is executed in an active or passive method of the *MessageHandler*.

CoralAccess

- Sends a request message
- No event loop, wait only for one particular reply message
- All the classes exchanging messages instantiate their private *MessageReceiver*

All the messaging protocol is behind the single *MessageReceiver::receive* method.

- Wait until a valid, full message arrives, or timeout occurs
- If message arrives, query for the
  - received message (encapsulated in *IMessage*)
    - server: start the associated event handler
    - client: check if the waited reply message arrived
  - origin (encapsulated in *CoralInputSocket*)
    - server: sends a reply message to the client encoded in the socket
    - client: checks if the message target is really itself (client ID, request ID must match)

The basic mechanism in the *MessageReceiver::receive* method:

1. Receive a packet, by using the *CTL::PacketReader*.
   1. Return valid packet only
   2. Exception if
      1. the line is dropped
      2. garbage arrived
      3. version mismatch in the packet header
2. If valid packet arrived, add it to a *MessageBuffer*
   1. The *MessageBuffer* collects the message segments (the packet payload-s)
   2. All incoming messages have a *MessageBuffer* instance
      1. The line is multiplexed, more packets from more clients may arrive, mixed up
      2. The client ID – request ID pair determines the *MessageBuffer* instance
   3. Checks the message segmentation and opcode match, packet-by-packet
   4. In case of error
      1. send the proper error message to the client
      2. delete the MessageBuffer, associated to the message
         1. It means discarding the whole message
3. Check if with the actually received packet, one of the partial messages got full. If so, then:

    1. The message is in its raw binary format
    2. Reconstruct the message object by using the *MessageFormatter*
        1. Fetch the opcode
        2. Instantiate a message object and call its decode method on the raw data stored in the *MessageBuffer*
    3. If the reconstruction fails, it means that the message is malformed
        1. Send back a *Syntax Error* message
        2. Discard the message
4. Ok, we have a full, successfully encoded message object: we got an event. Return *true*.
5. If timeout occurred during the process, return *false*.
    1. The timeout does not invalidate the message buffers storing partial messages. The next *receive* call continues from the actual state.

Implementation comments:

1. Static structure: *MessageReceiver.jpg*
2. The *PacketReader* state machine: *PacketReader_State_Machine.jpg*
3. The *MessageBuffer* state machine: *MessageBuffer_State_Machine.jpg*
4. The state machines are implemented by using the *Boost::StateChart* library
    1. http://www.boost.org/doc/libs/1_37_0/libs/statechart/doc/index.html
    2. The benefits:
        1. The UML state machine diagram can be mapped directly to the code
        2. Uses state-local variables and explicit state transitions
        3. Enables extremely modular and robust state machine implementations
        4. Yeah, heavily templated...

Event/Message handling in the server

The event dispatching method of *CoralDataAcceptor* runs a *MessageReceiver* in an event loop. If a message arrived, it is passed to the *MessageHandler::handle* method of a *MessageHandler* instance.

The *MessageHandler* represents the relational state of the CoralServer. There is one *MessageHandler* per physical line. Its state machine is in *MessageHandler_StateMachine.jpg*.

The *MessageHandler* may have in two states: *Connected* or *Unconnected*. It must be understood in database connection context. A *Connect* message may establish a database connection and a session. If it is successful, the *MessageHandler* transits to the *Connected* state. The *CloseSession* may terminate the database session (and the connection), and transits the state machine to *Unconnected* state.

In *Unconnected* state, all the messages are discarded, except for the *Connect* message. In *Connected* state, all the relational messages are handled.

The *Connect* and *CloseSession* messages are executed synchronously, while all the other messages are handled asynchronously.

The relational state is represented by an *ISessionProxy* object. It is stored in an instance of the *CoralMessageHandler* class, that is created by the *MessageHandler*, when the state machine transits to *Connected*. The (active) methods of this class are the message handlers themselves, executed in the associated *ISessionProxy* context. All the CORAL relational objects and operations are centralized here, so this single class is the interface between CORAL and the CORAL server!!!!!!!!!!! (modularization issue...).

Static structure: *MessageHandler.jpg*

Message representation, encoding, decoding

Rationale

The rationale behind the message serialization design:

- We know the structure and type of messages in compile time (no type discovery, etc.), both on client and server time.
- The messages must implement the CORAL server protocol
- The protocol is a binary one, taking the different system architectures into account.
- The implementation must support easy implementation and simultaneous usage of different protocol versions.
- As the protocol is subject to rapid changes and prototyping, the implementation must be able to follow those changes quickly.

Design

All the messages have two parts: the CTL header and the message body (payload). The CTL header is implemented in the *CoralProtocol* module, and it is independent from the message body.
The message body implements both the CTL and CAL messages. It is constructed from some well-defined basic types and *message objects*. The message objects may constitute basic types and other message objects as well. So, all the messages can be represented by a tree, where the leaf nodes are always the basic types. The basic types have well-defined serialization rules, and the serialization of the message objects are based on the serialization of their basic types. The list of the simple types are in the protocol description.

From the simple types and message objects, we can construct arrays. The array structure is:

- a word giving the number of array elements
- the array elements

For example, an array of bytes may encode strings, blobs, etc.

All the messages are implemented as individual classes, in the *CoralMessaging* module, collected under the namespace *Message::CAL*. All the message classes share the following semantic:

- Inherited from the abstract IMessage class:
- The messages can be written into a *CoralSocket*
- There are getter methods for the individual message parts. All the getters give constant reference to the message part.
- There are no setters for the message parts. The message structure (relations to the constructing objects) is fully defined in construction time.
- The constructors accept the basic types by value, the message objects and arrays by reference. The message content can be modified only by modifying the message objects.
- The system decides when the serialization/deserialization happens. The user has no control over the process.
- The message is serialized when the user writes it into a *CoralSocket*. During the process, all the message parts are serialized, according to their individual serialization rules.
- The message is deserialized when the message is fully arrived and reconstructed. It means

that all the message parts are deserialized in order, according to their individual deserialization rules. If the deserialization fails, the message is dropped, and the sender is notified (by sending a *SyntaxError* CTL message)
- It means, that the user code always receives syntactically correct messages, it must check only the content (meaningful or not).

Apart from the basic types and arrays, all the message parts must inherit from the *IMessageObject* abstract interface.

- Encode converts the message object into a byte string. This string is written directly into the communication stream (serialization).
- Decode constructs the message object from a byte string (deserialization). The byte string is read directly from the communication stream.

The basic types and arrays have built-in serialization and deserialization rules. All the message serialization and deserialization ends up in serializing and deserializing basic types and arrays.
The above general rules enables that the implementation be governed by a simple embedded language, and code generation behind. The coder must define the structure of the message objects and messages, and the code generator creates the message serialization and desarialization code, and embeds them into the system. The implementation of the code generator is based on the *Boost* metaprogramming library (both template and preprocessor metaprogramming is involved). The embedded language is a simple, declarative macro language.

Info:
http://www.boost.org/doc/libs/1_37_0/libs/mpl/doc/index.html
http://www.boost.org/doc/libs/1_37_0/libs/preprocessor/doc/index.html

Message object defining language

We describe it by giving an example where all the language elements are represented. The example defines the structure of the client monitoring information for the proxy that is piggy-backed on the *Connect* message. The monitoring information contains the PID and the command line parameters of the client process.

The code is generated by the following:

```
#define MSGOBJ_CLASSNAME MonitoringInformation
#define MSGOBJ_CLASSNAME_STR "MsgObj::MonitoringInformation"
#define MSGOBJ_PAR_NUM 2
#define MSGOBJ_PAR_NAME0 pid
#define MSGOBJ_PAR_NAME1 cmdline

#include "ObjectGenerator.h"
MSGOBJ_MESSAGE_CLASSGENERATOR_2(MonitoringInformation, uint32_t, std::string);
```

The code generator will create the *CAL::MsgObj::MonitoringInformation* class, with implementation of all its methods.

The code generator implements the "*encode*" and "*decode*" methods as well. The string is serialized as

an array of bytes; the uint32_t is serialized as a word. These serialization rules are built-in, implemented as templates.

Let's see the language elements:

MSGOBJ_CLASSNAME: Defines the class name of the message. The following type will appear in the system:

```
Coral::CoralMessaging::CAL::MsgObj::MonitoringInformation
```

MSGOBJ_PAR_NUM: number of the message object parameters. This is 2 in our case (the pid and the description).

MSGOBJ_PAR_NAMEX: defines the method names of the getter methods. The numbers after the MSGOBJ_PAR_NAMEX defines the order of the message object parameters (the object elements are serialized/deserialized in this order). It must start from 0 and incremented by 1 at each parameter name.

```
#include "ObjectGenerator.h"
```

Triggers the code generation. Using the values of the macros defined earlier, a class template will be generated. The macros are undefined at the end, so we can use them multiple times in the same file.

```
MSGOBJ_MESSAGE_CLASSGENERATOR_2(MonitoringInformation, uint32_t, std::string);
```

Pulls the type into the code by instantiating the generated template with the message part types. The number prefix (_2) must be equal to the number of message object parameters. Supported numbers are 0-8. The first macro argument must repeat the class name of the generated object, the rest define the C++ types of the message object parameters, in order. The parameter type must be one of the following:

- a basic type for which a serialization/deserialization template is generated
- a class name for a message object. That class must inherit from *IMessageObject*.
- an array of the above types.

Message defining language

The message defining language is similar to the message object defining language, but it generates code for the message classes.
As an example, here is the definition of the *Connect* message:

```
#define CAL_CLASSNAME Connect
#define CAL_CLASSNAME_STR "CAL::Connect"
#define CAL_PAR_NUM 2
#define CAL_PAR_NAME0 connection
#define CAL_PAR_NAME1 monitoring
#define CAL_CACHEABLE 1
#include "CoralMessaging/CAL/CALMessageGenerator.h"
CAL_MESSAGE_CLASSGENERATOR_2(Connect, MsgObj::Connection, MsgObj::MonitoringInformation);
```

Apart from the different prefix (CAL instead of MSGOBJ), the syntax and the semantics are the same like in the message object case. One additional element is the CACHEABLE macro. If it is set, the message will be cacheable meaning that the user can control the cache-ability of the message. If it is

not set, the user has no access to this feature.
As we can see, the *CAL_MESSAGE_CLASSGENERATOR_2* contains message object classes as type definitions for their parameters. They may be basic types and arrays as well.

Testing

A class have an associated unit test class in the *TestSuite* sub-module. The unit tests are implemented by using the *CppUnit* package, and bundled into an unit test suite application. It means that one single application may execute all the unit tests of a module.

The integration tests are in the *Coral::Tests* module.

The classes are designed for testability. The unit tests are whitebox tests, and the use extensively the *dependency injection* pattern, *mock objects*, *fixtures*.

Pattern: an unit test associated to a class has <*classname*>*Test*. Look them in the appropriate *TestSuite* or *UnitTestSuite* modules.