Remote Key-Value Database

Operational Concept Document

Version 1.0

CSE 681 Software Modeling and Analysis

Chunxu Tang, Fall 2015

Instructor: Dr. Jim Fawcett

Contents

1 Executive Summary

This document is about the design of a remote key-value NoSQL database. Nowadays, NoSQL databases are becoming more and more popular because it supports flexibility of data structures. And a key-value database is an important genre of databases, which stores key-value pairs of data and supports fast CRUD operations. Additionally, currently, database is usually setup in a remote server, which is not the same machine the user is using. So a database which supports remote access is critical for software development and application usage.

In this document, in Chapter 2, I give an introduction of basic concepts of key-value NoSQL database, Windows Communication Foundation (WCF) and Windows Presentation Foundation (WPF). And in Chapter 3, I would like to present some use cases of a remote key-value database, including the usage of different levels of organizations, the usage in the final project and utilization of people from different positions in software development. Chapter 4 demonstrates the structure of my design from the perspective of top level packages. In Chapter 5, I illustrate some critical issues of my design. I partition the topics into 5 sections, and I would like to discuss the issues such as how to demonstrate requirements, how to implement write client and read client, how to handle invalid XML files, etc. These issues might be common in practice for a remote key-value database design.

2 Introduction

In recent years, we witnessed a significant development of large scale distributed systems. And some of them are utilized to implement databases to store a large amount of flexible data. In Project #2, I designed a NoSQL in-memory key-value database, which is an important category of databases. And in this document, I would like to demonstrate the design of a remote key-value database, which a user could access it remotely through Windows Communication Foundation.

2.1 Concepts of Key-Value NoSQL Database

Database Management Systems (DBMS) are the higher-level software, working with lower-level application programming interfaces (APIs), which take care of these operations[1]. Database Management

Systems are supported by database models, which define the structures to organize the management of data. The traditional SQL (Structured Query Language) databases, such as MySQL and PostgreSQL, are based on relational model. The relations bring into the benefits that data is organized into strict structured groups. SQL databases also provide a very well understood data management model that supports **ACID** properties (Atomicity, Consistency, Isolation, and Durability). These properties ensure that database transactions are processed reliably[2].

In a SQL database, there is usually a fixed schema of the types of the records in the tables. SQL databases emphasize on the integrity, consistency and structure of the data. While, for nowadays, with the large amount of data, it is not feasible to put them into strict tabular relationships. Additionally, queries into SQL databases usually result in joins of data from different tables, which is low efficient and causes a large latency.

While, for NoSQL (mostly translated as "not only SQL", though originally referring to "non SQL") databases, they are trying to provide a more flexible organizations for semi-structured data and un-structured data. A NoSQL database is designed to support one or more of the following:

- Very large collections of data
- High throughput with data from streams
- Support tree or graph models for its data
- Support heterogeneous collections of data

For a NoSQL database, it mainly falls into one of the four categories below:

- Key-Value (Redis, Riak)
- Column (HBase, Cassandra)
- Document (MongoDB, CouchDB)
- Graph (Neo4j, Titan)

Key-value database maps simple keys (for example, strings) to complex types (such as strings, sets, lists) like a huge hash table. It provides great flexibility because of its simplicity. Some designs, such as Redis, even stores most data in memory to gain higher **CRUD** (Create, Read, Update, Delete) speed. In modern design, this kind of NoSQL databases usually work as cache server.

Additionally, it is also important to recognize the weakness of a key-value database. For this kind of databases, they often lack indexes and scanning capabilities. And KV stores could not help you much if you need to perform queries on data, other than basic CRUD operations.

2.2 Windows Communication Foundation

The Windows Communication Foundation (WCF), previously known as "Indigo", is a runtime and a set of APIs in the .NET Framework for building connected, service oriented applications. It is designed using service-oriented architecture principles to support distributed computing where services have remote consumers[3]. It is sophisticated, flexible and configurable.

WCF implements a basic model to allow a client and server application to communicate[4]:

- On the server, you indicate what methods you'd like remote client applications to be able to call.
- On the client, you specify or infer the signatures of the server methods you'd like to call.
- On both the server and the client, you choose a transport and communication protocol.
- The client establishes a connection to the server.
- The client calls a remote method, which executes transparently on the server.

To take advantage of WCF, in my design of remote key-value database, all communications between processes and machines are implemented using Windows Communication Foundations.

2.3 Windows Presentation Foundation

Windows Presentation Foundation (WPF) is a graphical subsystem for rendering user interfaces in Windows-based applications by Microsoft. WPF, previously known as "Avalon", was initially released as part of .NET Framework 3.0[5]. WPF makes it quite easy for a use to design graphical interfaces by utilizing XAML (Extensible Application Markup Language), which is a XML-based language.

Figure 2-1 A WPF Example[1]

The graphical interface above illustrates an example which leverages WPF. When a client clicks "Click Me!" button, there will be a new windows popped, showing that the action of mouse is notices.

WPF has following attributes[4]:

- It supports sophisticated graphics, such as arbitrary transformations and 3D rendering.
- Its primary measurement unit is not pixel-based, so applications display correctly at any DPI (dots per inch) setting.
- It has extensive dynamic layout support, which means you can localize an application without danger of elements overlapping.
- Rendering uses DirectX and is fast, taking good advantage of graphics hardware acceleration.

In the remote key-value database design, to display the messages more clearly, and to construct the requests more straightforwardly, I prefer to use WPF for my design of graphical user interface.

2.4 Applications and Uses

For this design, because it is a course project, the users are first the developer, teaching assistant and the instructor. This project, remote key-value database, combines the design of the key-value NoSQL database in project #2, and this remote database will also be used in final project to build a large scale of system. So the first application is to work as a well-functional remote database which fulfills:

- Support operations on a key-value databases remotely:
    - Add and delete key-value pairs.
    - Edit values including addition/deletion of relationships, editing text metadata, and replacing an existing value's instance with a new instance.
    - Persist database content to an XML file.
    - Restore and augment database content from an XML file.
    - Accept a time interval or number of writes after which database content will be persisted.
    - Query content of the database, including
        - Value of a specified key.
        - Children of a specified key.
        - Set of keys matching a specified pattern.

[1] Dr. Fawcett's WPF Demo, http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/cse681codeL11.htm

        - Keys contain a specified string.
        - Keys contain values written within a specified time-date interval.
    - Create a new immutable database.
- Design write clients to send data to the remote key-value database. The content of different messages are defined in an XML file, and read at the client startup. The write clients also need to have a high-resolution timer to record communication time.
- Write clients accept an option switch to determine whether messages are logged to the console or GUI.
- Design read clients to send query requests to the remote key-value database. It also read an XML file when it starts, which defines the messages to send.
- Read clients provide options to:
    - Display some part of each response as it is processed.
    - Set a specified number of requests to send, start a high-resolution timer to record the time

Furthermore, in practice, a remote key-value database is beneficial for different scales of companies. Such a database could provide remote access of flexible semi-structured and un-structured data. No matter the organization is small, medium or large, it could benefit a lot from the usage of the database.

And from the perspective of people of different positions, the remote database is also helpful. For example, the final project, which is a storage management subsystem for software collaboration federation is extremely crucial for software developers. The remote database could be utilized in collaboration server, repository server, etc.

The details of use cases and applications are demonstrated in Chapter 3.

2.5 Structure

To design the structure of the remote key-value database, it needs to fulfill the requirements, that

- To support operations such as add, delete, persist on a remote key-value database.
- To support write clients to send data to the remote database.
- To support read clients to send query requests to the remote database.
- To support getting messages from XML files.
- To support other additional functionalities for write clients and read clients.

Considering the requirements above, the top level package diagram is show below.

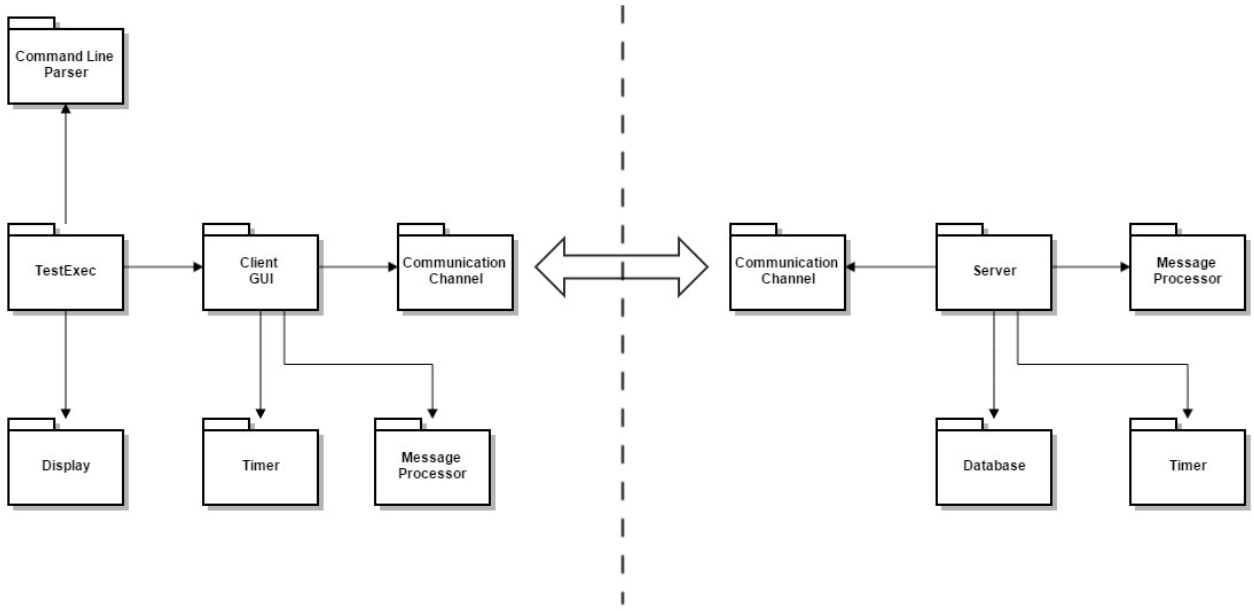

Figure 2-2 Remote Key-Value Database Package Diagram

There are two sides in the top level package diagram: the client side and the server side.

- Client side
  - Test Exec
  - Command Line Parser
  - Display
  - Client GUI
  - Timer
  - Message Processor
  - Communication Channel
- Server side
  - Server
  - Database
  - Message Processor
  - Timer
  - Communication Channel

For the client side, there is a Test Exec package to demonstrate that all the functional requirements have been met. The package invokes Display package, Command Line Parser package and Client GUI package. If there is no such packages, no credit will be obtained even if I have had them correctly implemented.

Command Line Parser is utilized to fulfill requirement #10. In that requirement, the test executive shall accept from the command line the number of writers and readers to start. So I need to design a Command Line parser to read and parse command line options to check the number of writers and readers.

Display package works to display testing information during the running of Test Exec package. It needs to demonstrate all the requirements are fulfilled clearly and straightforwardly.

Client GUI package is the core of client side design. It invokes Timer, Message Processor and Communication Channel. Client GUI starts read clients and write clients to send requests to remote key-value database. It uses Message Processor to obtain requests necessary to send, uses Timer to record end to end time, and uses Communication Channel to send out the requests and get the responses.

Message Processor is an important component of the whole structure. For the read clients and write clients, they both need to obtain messages to send from XML files. So XML Parser's functionality is to read and parse content of XML files, and obtain the information of the messages. Additionally, to obtain the messages need to send, Message Processor package also defines the request structure information. It defines what a request should contain such as name of request, time stamp of start time and so on.

Timer is utilized to record elapsed time. For example, in requirement #8, read client is asked to provide option to start a high-resolution timer when the first request is sent and stop the timer when the last request is received. So a timer is necessary to obtain the elapsed time.

Communication Channel defines the communication protocol between the client and server. It needs to use Windows Communication Foundation (WCF), which I have given an introduction in Section 2.2, to construct the transmission protocol. The requests are serialized into XML format, and are sent to the server. The server de-serializes the XML files, obtains the requests, processes them, serializes the responses into XML format, and sends them back. After these procedures, the Communication Channel in client side de-serializes the responses, and gets the response information.

Server package is the core in server side. It invokes Database package, Message Processor package and Communication Channel package. It acquires requests from Communication Channel, obtain real message information from Message Processor, and process them through Database package.

Message Processor in server side is the same as that in client side in my design. In this side, its work is mainly process the messages and construct them into real messages which could be directly sent to the Database.

Database package is the key-value in-memory NoSQL database designed in project #2. It supports different kinds of requests such as adding/deleting of key-value pairs, persisting database content into XML files, restoring and augmenting database content from XML files, etc.

The details of every package are discussed in Chapter 4.

3 Uses

First of all, the users of the remote key-value database are developer (myself), teaching assistants and instructor. They could check whether the design fulfills the requirements. And the design will be utilized in the final project, which involves managing a large repository's data, recording continuous integration and test activities, managing notifications to a large collection of clients, and building and maintaining templates for test configurations, collaboration sessions, work package descriptions, etc. Additionally, to extend the use cases of my design, I also would like to demonstrate other use cases. For these use cases, I present them in two perspectives: scale of organizations and different positions of people.

3.1 Uses of different scales of organizations

3.1.1 Uses of small organizations and individuals

For small organizations (tens of developers) and individuals, the amount of data is not very large. An in-memory key-value database might have already been enough for their product data storage. Suppose there is a mobile app or web app developed, if the data is stored in an in-memory database, the speed of CRUD operations will be very fast, and customers could have an excellent user experience of the product. Additionally, because the database provides remote access, the database could be deployed in a different server from the server which holds the web page information. The learning curve of a key-value database is rather gentle. This advantage dramatically reduces the cost of the small organizations and individuals, and helps them to put more concentration on product development.

3.1.2 Uses of medium organizations

For this kinds of organizations (hundreds of developers), not only the products are more complicated, but the developing teams are also larger. Only one key-value in-memory database may not be enough for an

organization of this scale. Several such databases and even more are necessary for their products and developing teams. Additionally, sometimes, the databases are must be deployed in different servers for load balancing. Under this circumstance, the databases must provide the capability of remote access. For a developing team, because of the speed of CRUD operations of a key-value database, some hot data (popular source code) could be stored in the database. And for the products they developed, key-value databases could work as cache servers for other on-disk databases. Only the latest data is stored in the key-value database, and it could make the service more fluent.

3.1.3 Uses of large organizations

For large organizations (thousands of developers or even more), they usually need to provide consistent and safe services for a large amount of customers. To handle performance issues, sometimes, a cluster of key-value in-memory databases might work together as a cache cluster.

Here, I would like to take Amazon as an example. Around Christmas or Thanks Giving Day, it is obvious that there are spikes of shopping for short periods. Thousands of requests may sent to the server in just one second. If just on-disk databases are used, the clients may have to wait for a long time for the web page to response. While, actually, some data is hot data, which is visited frequently, such as the home page, and some commodities with large discounts. This kind of data could be stored in key-value databases, because they could provide much higher speed than on-disk databases. Additionally, a NoSQL database also provides flexibilities for un-structured and semi-structured data, which is very common of web data. Furthermore, the remote key-value database not only permits users to access the database remotely, but also provides capability of massive requests processing. There are send queue and receive queue in a remote key-value database, and it could handle many requests simultaneously.

Additionally, for a large organization, it may also need remote key-value databases for storage management subsystem for software collaboration, which I will talked about later.

3.2 Uses of people of different positions

3.2.1 Developers

For developers, their data might not be very large, which could be stored in one or more machines' memory. The fast access of data stored in in-memory key-value databases could boost their development. For example, software testing is a crucial part in software development, and the tests usually need to be run for

many times, especially when there are new commits of original source code. If every time, the tests are loaded from disks, this will definitely creep the whole development. To solve this problem, the most frequent used tests could be stored in the key-value in-memory databases. Because it is relatively fast to load data from this kind of database, the development will benefit a lot. Additionally, the server which stores the tests should be a remote server which is only public to a specific group of people. A remote key-value database could be deployed on the server to provide access to the tests.

3.2.2 Technical Leaders

In general, a technical leader is responsible for underlying architecture for the software team, as well as for overseeing the work being done by other software developers. In most cases, the repository of data is not larger than the memory of a machine. So, actually a key-value in-memory database could be leveraged to store a collection of source code of a project. It could also record the continuous integration and test activities. It will be convenient for the technical leaders to manage the whole projects.

Moreover, it is natural that the technical leaders do not need to setup the database on his/her own machine. There should be another server to hold the remote database to provide corresponding services.

3.3 Uses in Final Project

In the final project, I need to design a storage management subsystem for software collaboration federation. The structure of the system is shown in Figure 3-1.

Software Collaboration Federation is a collection of clients and servers and associated software designed to support activities of a software development team, e.g.[6]:

- Creating and publishing plans for development.
- Writing and publishing specification and design documents.
- Preparing new source code packages.
- Acquiring existing source code packages for reuse.
- Building execution images and loadable libraries.
- Executing tests: unit tests, integration tests, regression tests, performance tests, stress tests, and acceptance tests.
- Deploying software executable and documentation.
- Reporting progress, key events and failures.

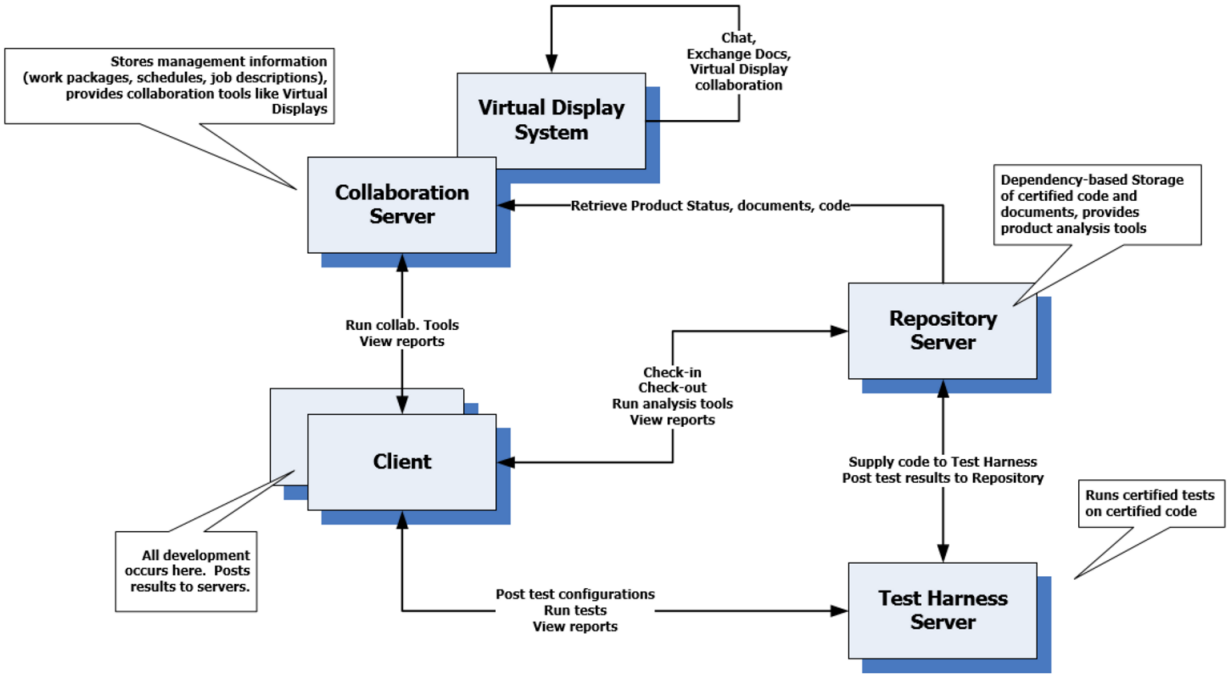

Figure 3-1 Software Collaboration Federation (SCF)[1]

A remote key-value database could be utilized in these components:

- Collaboration Server: the database could store package details.
- Repository Server: the database could store source code, and it may also provide metadata associated with each stored item (the metadata could just be the key).
- Build Server: the database could cache the execution images and libraries.

4 Structure

In this section, I will demonstrate details of the packages in the remote key-value database design, including Command Line Parser, Test Exec, Display, Client GUI, Timer, Message Processor, Communication Channel, Server and Database.

[1] Project #5, Dr. Fawcett, http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project5-F2015.htm

4.1 Overview

As what I have put in the introduction section, the tasks of this design could be summarized as:

- To support operations such as add, delete, persist on a remote key-value database.
- To support write clients to send data to the remote database.
- To support read clients to send query requests to the remote database.
- To support getting messages from XML files.
- To support other additional functionalities for write clients and read clients.

The top level package diagram is shown below.

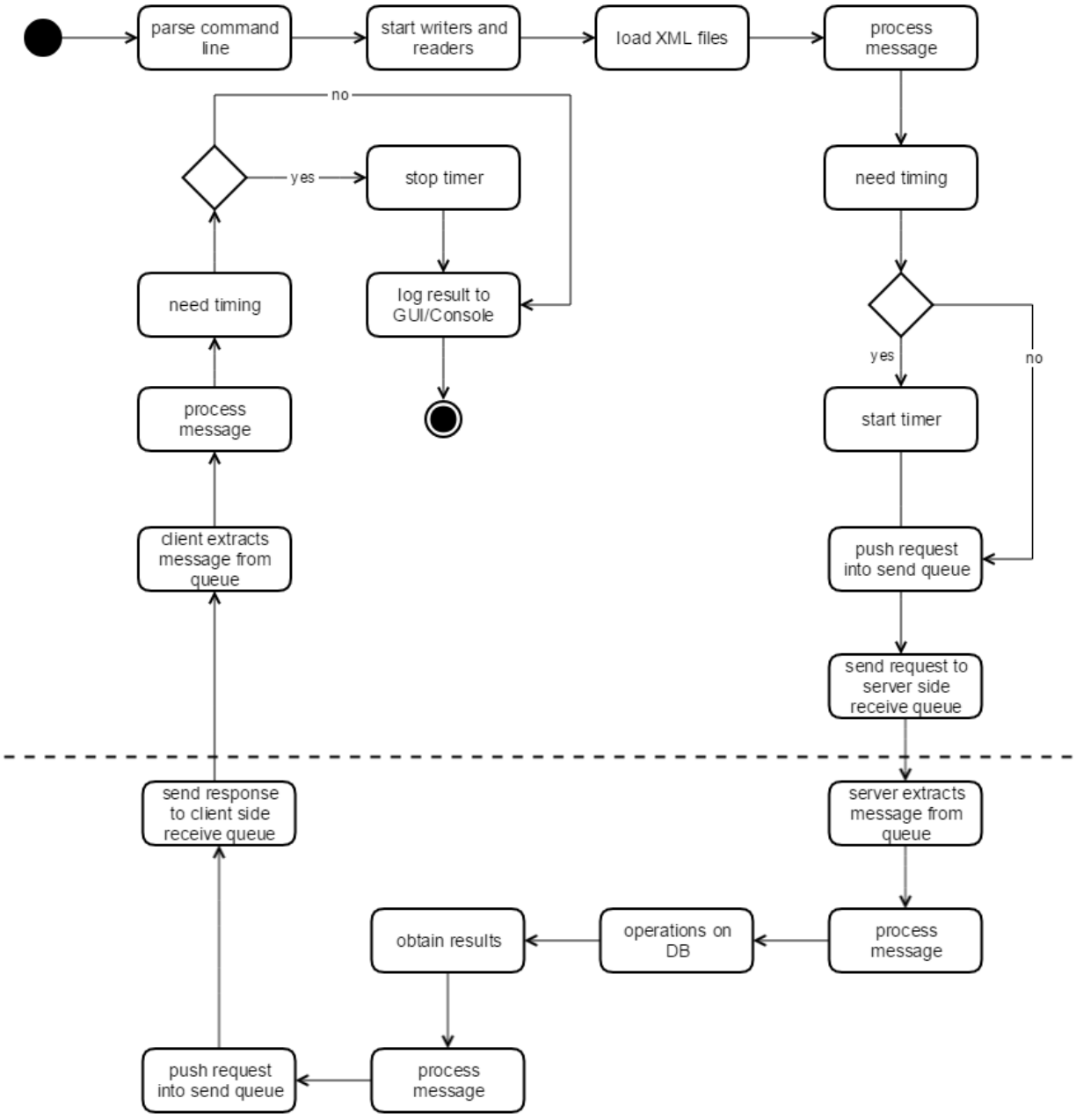

Figure 4-1 Top Level Activity Diagram

To fulfill the requirements above, I draw the top level activity diagram of the remote key-value database, which is illustrated in Figure 4-1. The whole procedure is described below:

- Test Exec starts, and according to requirement #10, it needs to accept from the command line the number of writers and readers to start. So in this step, it needs to start the Command Line Parser to parse the command line to get necessary information. Then, corresponding numbers of writers and readers are started.

- According to requirement #5 and requirement #7 in project #4, there should be XML files defining messages to send, and these XML files are loaded at clients start up. So in this step, corresponding XML files are read and parsed.
- After XML files have been loaded, Message Processor begins to process the contents, and constructs corresponding requests.
- Here, whether timing is necessary or not needs to be determined. For write clients and read clients, they both accept options to record elapsed time. So in this step, the program needs to check whether timing option is provided. If it is provided, then timer will start, otherwise, the program will skip the timer step.
- Now, the request is pushed into client side send queue. Please notice that because there are multiple write clients and read clients, so actually, multiple instances will push their messages into the send queue.
- WCF takes care of the communication part, and requests are sent to server side receive queue.
- The server extracts the requests from the receive queue one by one, and utilizes Message Processor to reconstruct the messages.
- The messages will be processed into real requests which could be sent directly to the key-value database.
- The key-value database processes the requests, and sends the responses back. Here, even though results have been obtained, the program still need to process the results into appropriate message format.
- The message is pushed into server side send queue, and waits for sending.
- Now, message has been sent to the client side. The client extracts the response from receive queue and reconstruct it into appropriate message format.
- Whether timing is necessary or not, is checked again. If it is necessary, the timer must be stopped to obtain elapsed time.
- Then the responses are logged onto GUI or console, according to different kinds of clients (write clients/read clients) and different logging options.

4.2 Test Exec, Command Line Parser and Display Packages

Figure 4-2 Packages Related to Test Executive

Figure 4-2 illustrates the packages in the client side, which are Text Exec, Display and Command Line Parser. To fulfill the requirements of testing the whole remote key-value database design, I also draw the UML class diagrams of Display, Test Exec and Command Line Parser, which is shown in Figure 4-3. In Test Exec, it contains methods to test all the necessary requirements and invokes Command Line Parser to parse the command line options, because it needs to accept from the command line the number of writers and readers to start. Display class is also leveraged by Test Exec to display the contents of requests and responses.

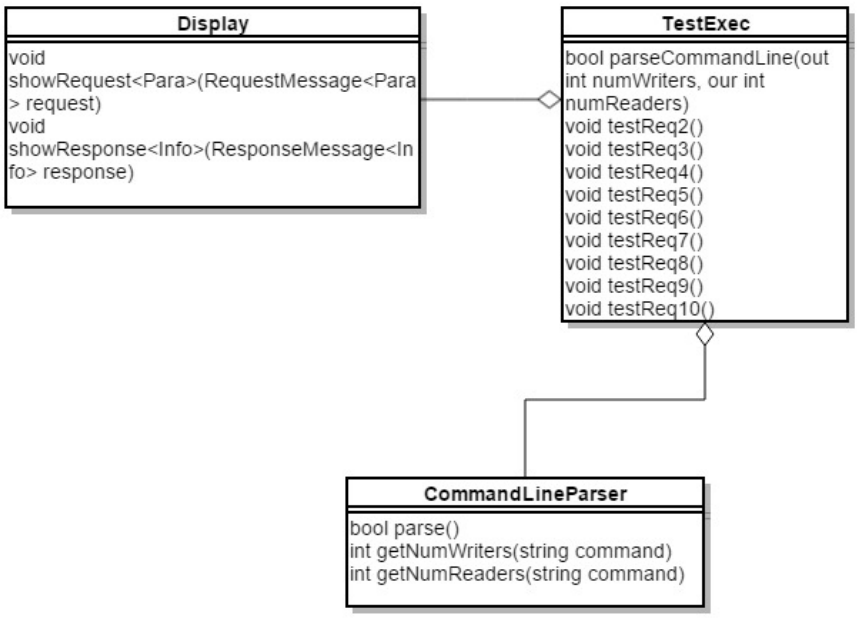

Figure 4-3 Test Exec, Display and Command Line Parser Class Diagram

4.2 Client GUI Package

Client GUI package works as a core in the client side. It invokes write clients and read clients to send operations to the remote key-value database. The activity diagram is illustrated in the figure below. The activities of write client are highly similar to those of read client. So here, I'll just demonstrate the whole procedure for the write client.

- Test Exec package invokes a definite number of write clients.
- The clients load XML files and utilize Message Processor to get the requests which need to be sent to the remote key-value database.
- Check whether timing is necessary. If yes, the timer will start.
- Send the requests to Communication Channel's send queue, and Communication Channel will send the requests to server side receive queue.
- Server extracts the requests and processes them. After that, it will send the responses back to client side receive queue.
- Client receives the responses from the receive queue, and obtain the contents of the messages.
- Check whether timing is necessary. If yes, the timer will stop.
- Check whether displaying results on GUI is necessary. If yes, the results will be shown on WPF GUI. Otherwise, they will be printed on the console interface.

For the write client and read client, their GUIs may also provide functionality to construct new requests directly from the GUIs, which may be similar to what I show in Figure 4-5 and Figure 4-6. While, actually, according to the requirements of project #4, because loading messages from XML files may be more convenient, GUI is mainly used for display. So, a client GUI such as the GUI in Figure 4-7 may be enough for display.

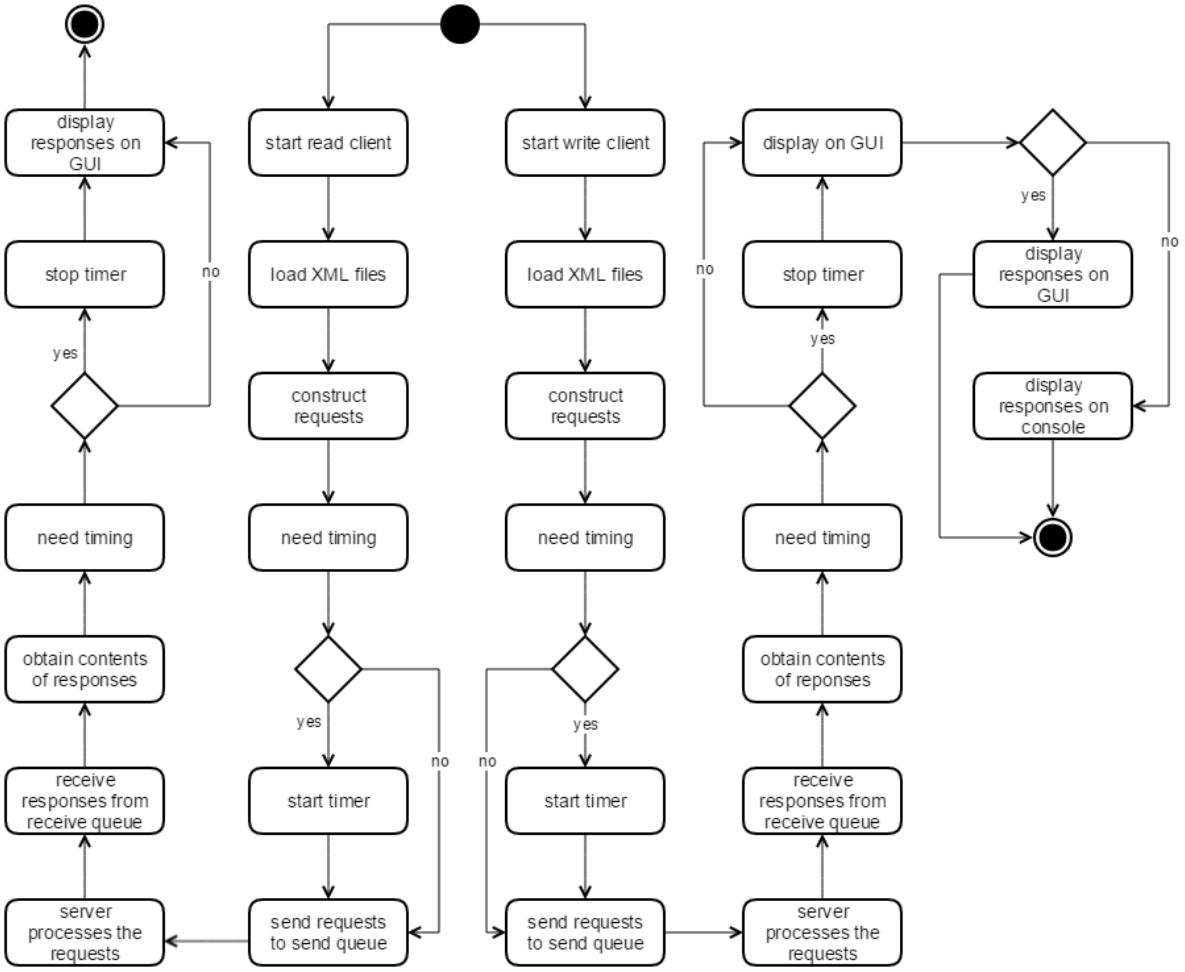

Figure 4-4 Client and GUI Activity Diagram

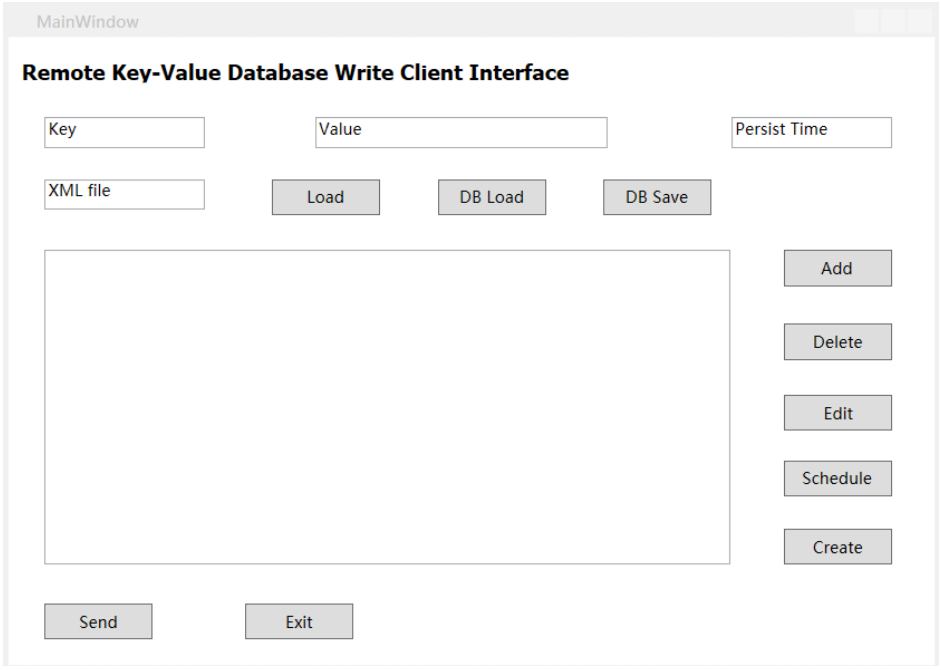

Figure 4-5 Write Client Interface

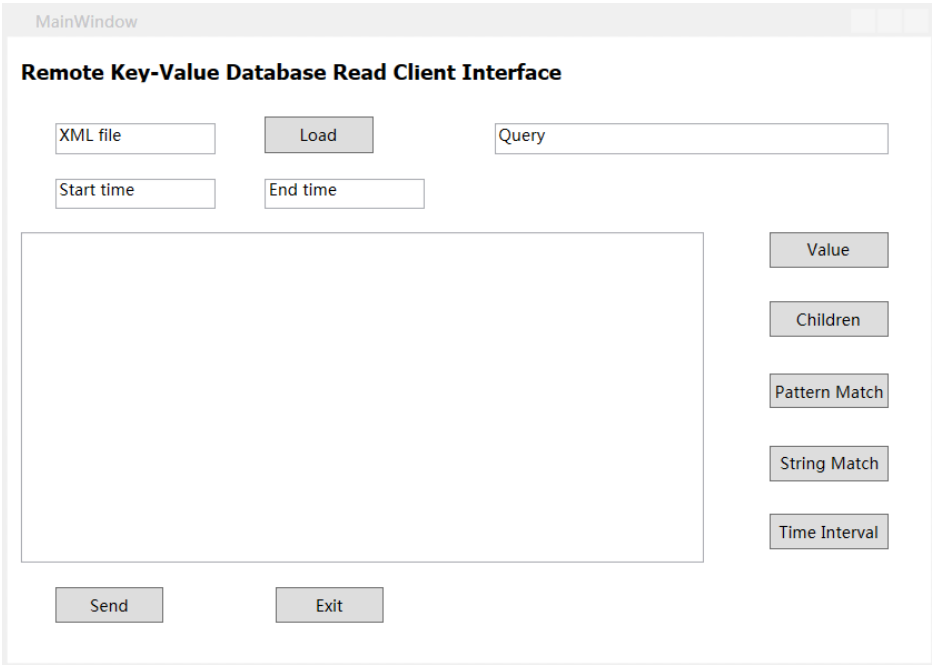

Figure 4-6 Read Client Interface

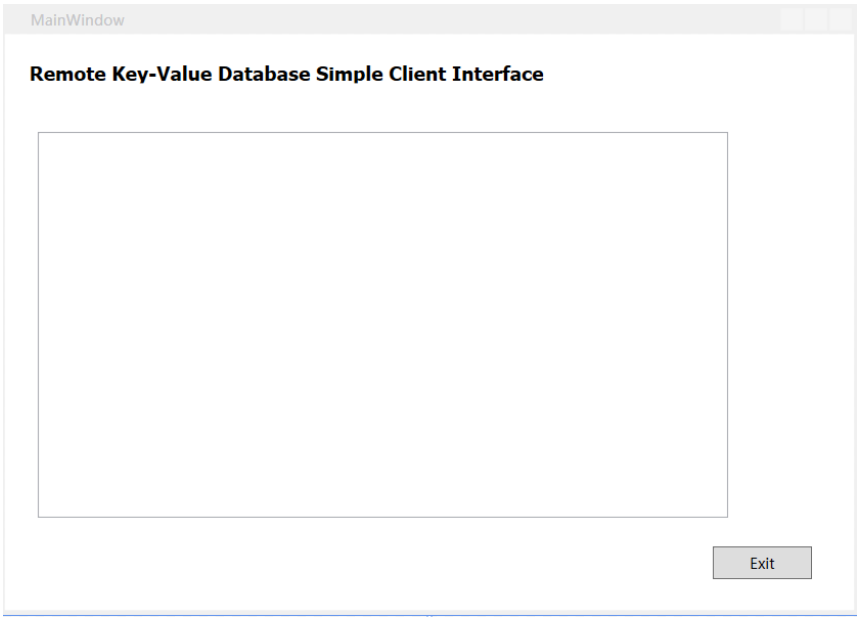

Figure 4-7 Simple Client Interface

4.3 Timer Package

Timer package is mainly leveraged to fulfill timing requirements in write clients and read clients. It invokes a high-resolution timer to record start time before the requests are sent out and record stop time after all the responses are received. The class diagram of Timer is shown in Figure 4-8.

In my design, I also put a Timer in server side. Even though now, there is no requirement to record timing information in server side, it is still crucial for the server to record information related to latency.

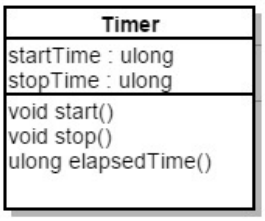

Figure 4-8 Timer Class Diagram

4.4 Message Processor

The core functionality of Message Processor is to encapsulate and de-encapsulate messages. For the client, Message Processor needs to load XML files first to obtain the messages of write clients and read clients. The contents of the XML files are organized into Request Message, which a class contains the type of the message, a dictionary to store parameters, start time, stop time, source URL and target URL. The class diagram which is related to Request Message and Response Message is shown in Figure 4-9. Here, I put the message types into an enumeration which is consisted of eight different kinds of types:

- Save: ask the remote key-value database to persist contents to XML files.
- Load: ask the remote key-value database to load an XML file.
- Add: add a key-value pair to the remote key-value database.
- Delete: delete a key-value pair from the remote database.
- Edit: edit the value of a key (including addition/deletion of relationships, editing text metadata, and replacing an existing value's instance with a new instance).
- Query: implement a query on the remote database.
- Schedule: schedule a time interval for the remote database to persist contents.
- Create: ask the remote database to create a new immutable database instance constructed from the result of any query that returns a collection of keys.

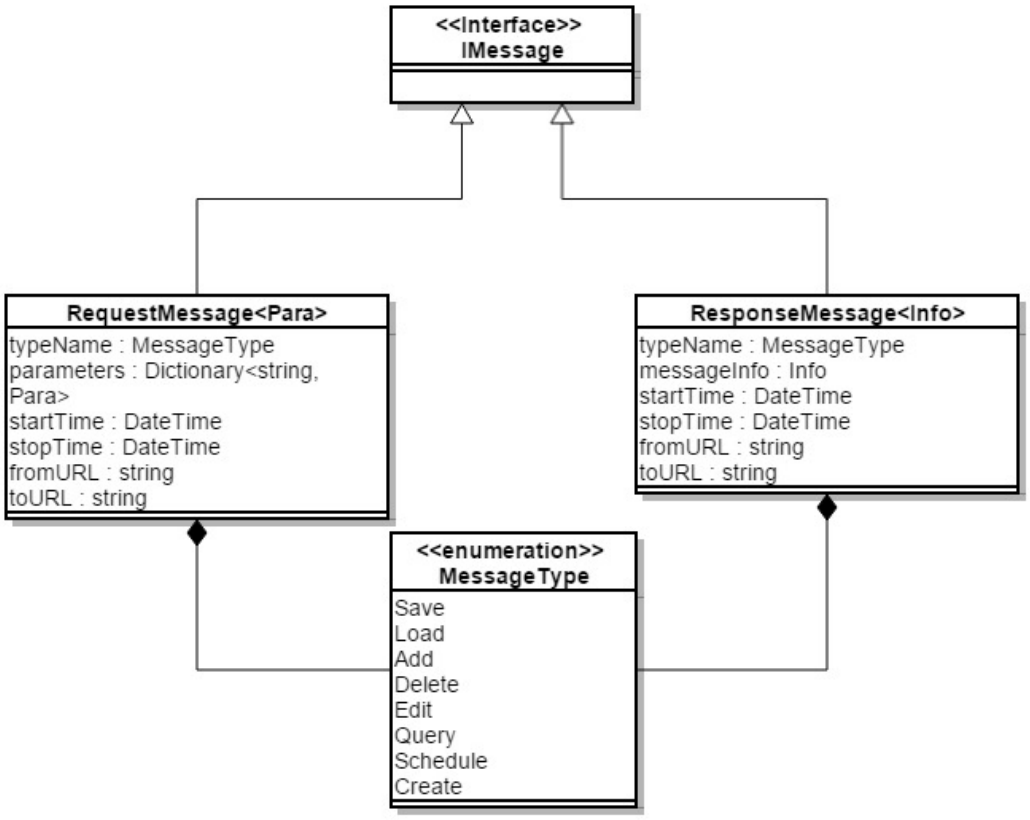

Figure 4-9 Request Message and Response Message Class Diagram

For a request, there is a dictionary to store parameters of the request. According to different message types, there will be different parameters, and the details are shown in Table 4-1.

- For “Add” operation, it needs to provide the key and the value, so in the dictionary, the key type is string, and the value type is Tuple<Key, DBElement<Key, Data>>.
- For “Delete” operation, it is similar to “Add” operation.
- For “Save” operation, it just need to send the request to the server.
- For “Load” operation, the key type is string, and the value type is also string, which represents the file name.
- For “Query-Value”, the key is “Query-Value”, which is a string, and the value is the target key.
- For “Query-Children”, “Query-Pattern Match”, “Query-String Match” and “Query-Time Interval Match”, they are very similar to “Query-Value”, so I will not describe them here.
- For “Create”, the value type is List<Key>, which are the keys to create new immutable database instance.

- For "Edit-Name", "Edit-Description", "Edit-Children", "Edit-Add Child", "Edit-Remove Child", and "Edit-Payload", they are all very similar to each other. Their value types are just corresponding to their target editing values.
- For "Schedule" operation, the parameter value type is an integer, which represents the time interval for persistency.

Table 4-1 Parameter Key Types and Value Types of Operations

Operation	Parameter Key Type	Parameter Value Type
Add	string	Tuple<Key, DBElement<Key, Data>>
Delete	string	Tuple<Key, DBElement<Key, Data>>
Save	string	NULL
Load	string	string
Query-Value	string	Key
Query-Children	string	Key
Query-Pattern Match	string	string
Query-String Match	string	string
Query-Time Interval Match	string	DateTime/List<DateTime>
Create	string	List<Key>
Edit-Name	string	string
Edit-Description	string	string
Edit-Timestamp	string	DateTime
Edit-Children	string	List<Key>
Edit-Add Child	string	Key
Edit-Remove Child	string	Key
Edit-Payload	string	Payload
Schedule	string	Int

For a response message, it contains a generic type, Info to represent the return value of corresponding operations. The operations and their corresponding Info types are shown in Table 4-2.

Table 4-2 Info Types of Operations

Operation	Info Type
Add	bool
Delete	bool
Save	bool
Load	bool
Query-Value	DBElement<Key, Data>
Query-Children	List<Key>
Query-Pattern Match	List<Key>

Query-String Match	List<Key>
Query-Time Interval Match	List<Key>
Create	VirtualDB<Key, Value>
Edit	bool
Schedule	bool

4.5 Communication Channel

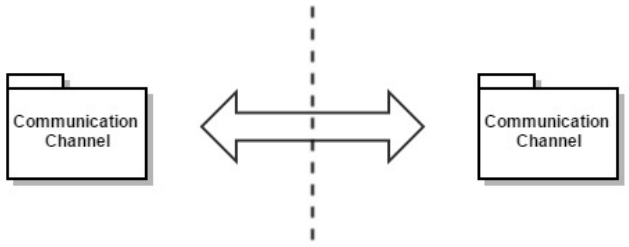

Figure 4-10 Communication Channel Package Diagram

There are two communication channels in the design, one at the client side and the other one is at the server side. The send and receive of messages is taken care by WCF. For the messages transferred, I need to define DataContract attribute of the classes and DataMemeber attributes of the data members. Then, WCF will implement the serialization and de-serialization during the communications. For every side of the design, there are two blocking queues respectively, send queue and receive queue. This design also means that for every side, there will be a sender and a receiver. The activity diagram of the communications between two channels is shown in Figure 4-11. The procedure is:

- Client adds a request into the send queue in the client side.
- The sending thread extracts the request from the send queue and send it to server side's receive queue via WCF.
- At the server side, the receive thread extracts the request from receive queue, and then server executes corresponding work according to the request, and finally generates the response. The response is pushed into server side send queue.
- The sending thread reads the response from the blocking queue, and sends the message back to client side receive queue.
- At the client side, the receive thread obtains the response from the receive queue, processes it and display the response.

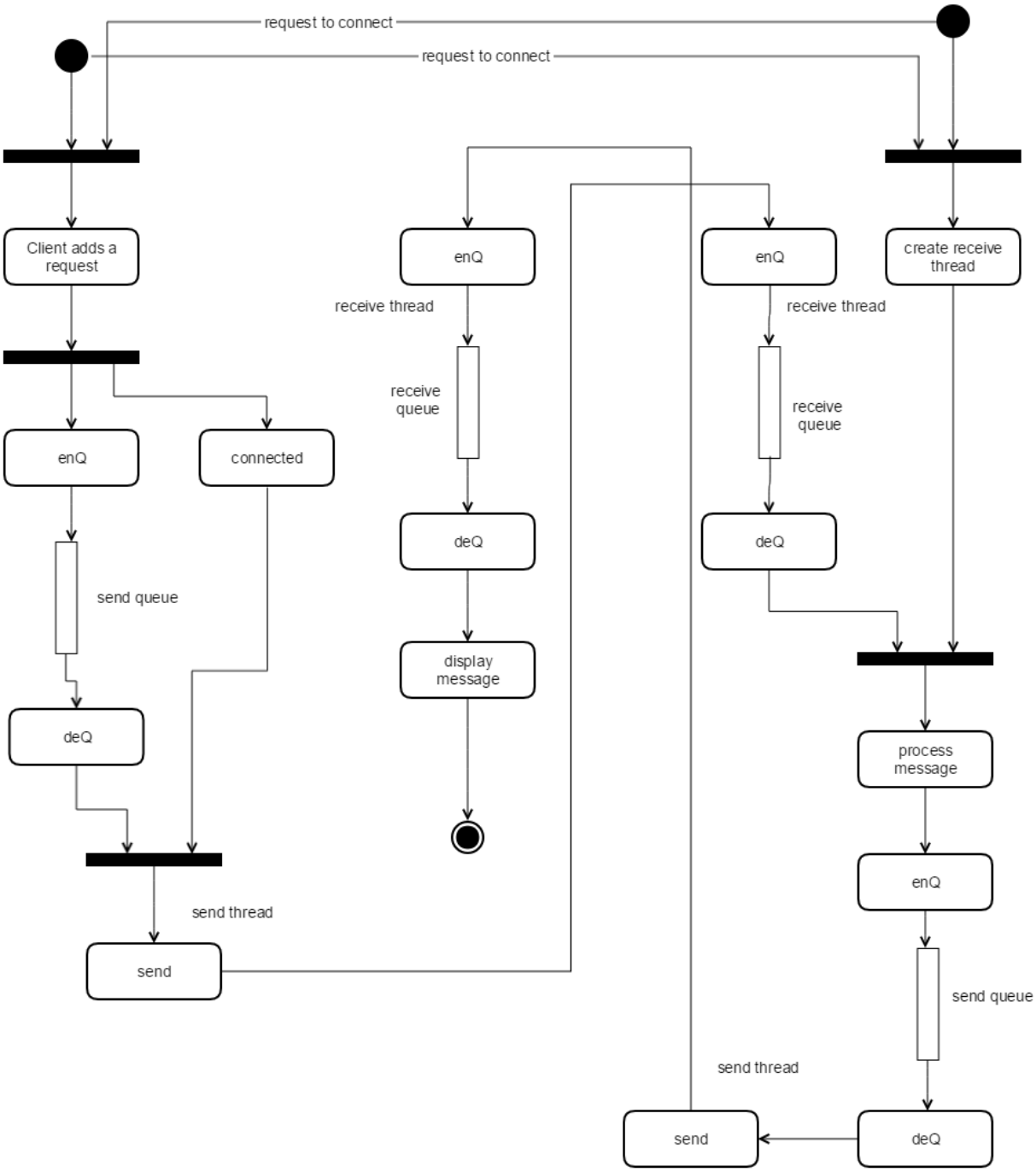

Figure 4-11 Communication Channel Activity Diagram

4.6 Server

The Server package works as the pivot at the server side. It invokes Communication Channel, Database, Message Processor, and Timer to construct the whole server side.

This package obtains the request through Communication Channel, reads the messages by using Message Processor, and sends the operations to the database. The database processes the requests and sends the results back to the Server. Then the Server utilizes Message Processor to construct responses, and sends them back to client side via Communication Channel.

4.7 Database

For the key-value database, I just reuse the design of project #2. The top level package diagram of the database is demonstrated in Figure 4-12. And the class diagram of the database is shown in Figure 4-13.

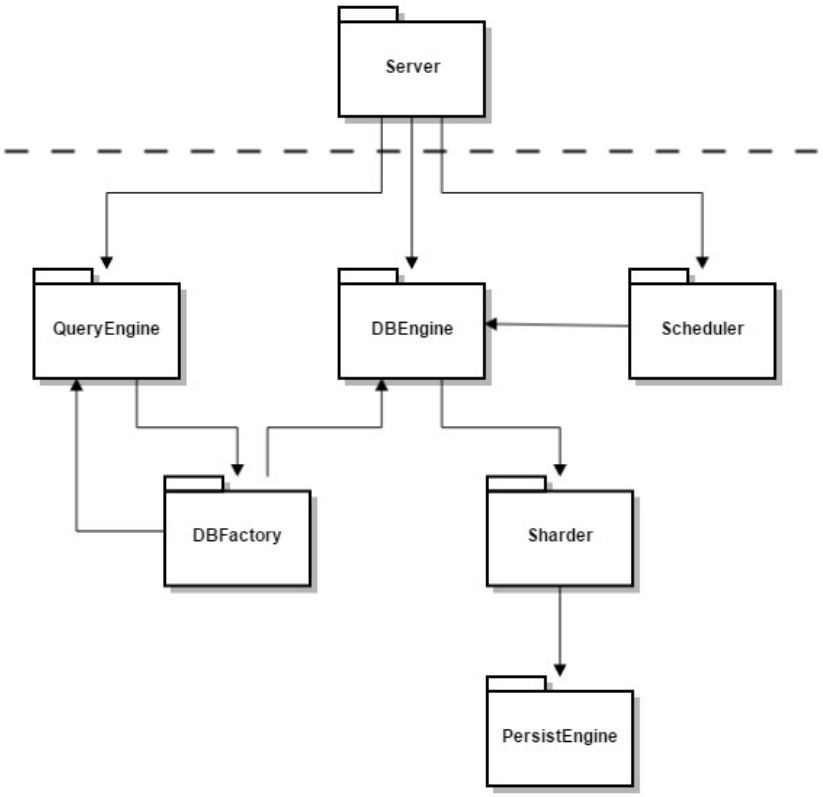

Figure 4-12 Key-Value Database Package Diagram

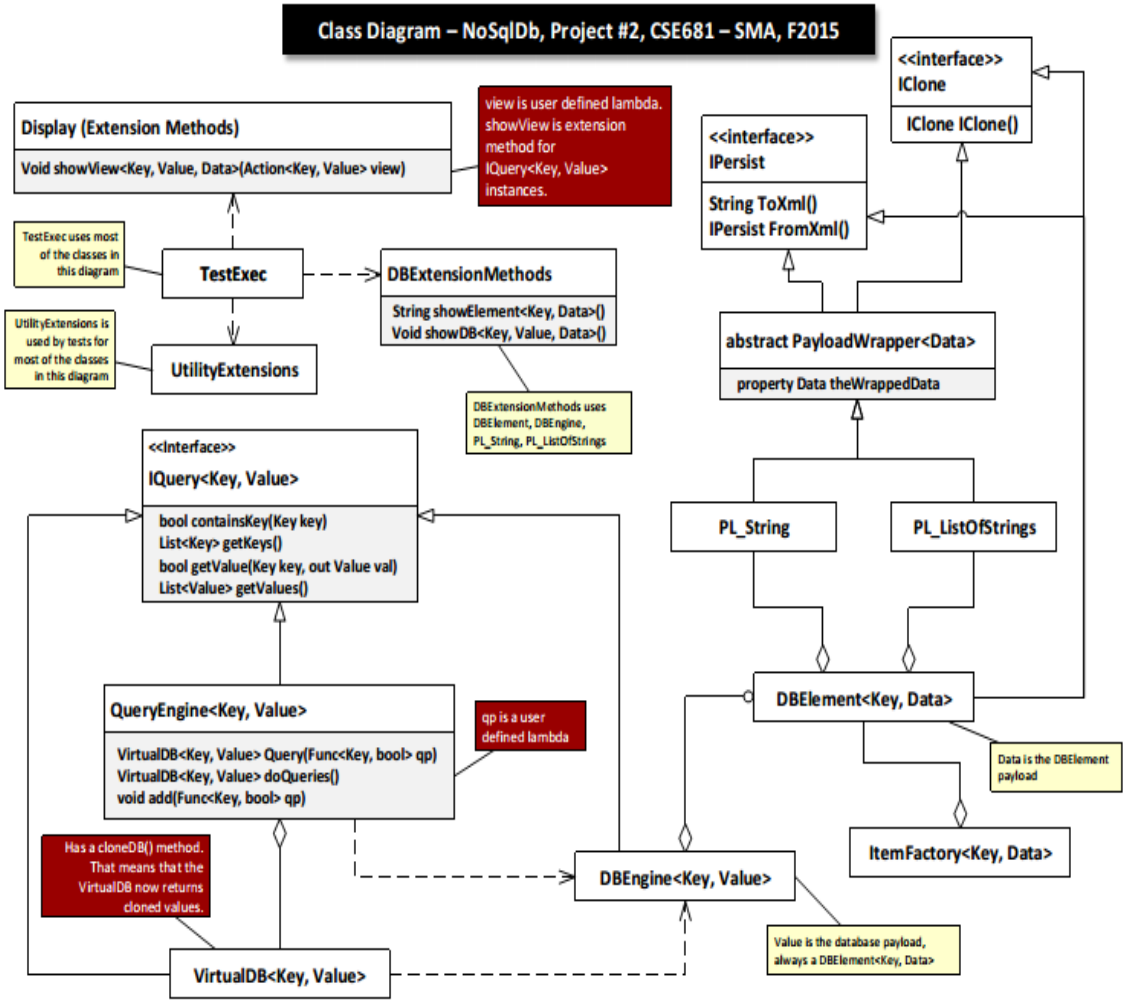

Figure 4-13 Key-Value Database Class Diagram[1]

In original database design in project #2, there are Item Factory and Item Editor packages. While, in this project, because the requests are provided by the client, which generated through loading XML files, it is not necessary to keep the two packages any more. So Server package directly interacts with the database, and sends corresponding requests to the database. After the procession of the database, it will receive the responses, and send them back to the client via Communication Channel.

[1] http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project2HelpF15/ClassDiagram.pdf

5 Critical Issues

5.1 Demonstrating Requirements

Issue #1: How to demonstrate requirements fulfilled effectively?

Solution:

The students could get points only if they have demonstrated clearly that the requirements are fulfilled. So I need to design the display package and GUI carefully to illustrate that effectively.

Impact on Design:

It will be effective to provide a method for each test that announces the requirement number and displays requests and responses respectively.

5.2 Client GUI

Issue #1: Should write client and read client implemented in one class or in two separated classes?

Solution:

For the write client and read client, according to the requirements, it is true that they have similar functionalities. While, the two kinds of clients still have some differences. For example, write client shall accept an option switch to determine whether messages are logged to the console or GUI, which read client is not required to implement. Furthermore, for the two clients with different purposes (one to send data to the database, and the other one send queries to the database), to put them in one class violates **SOP** (Single Responsibility Principle). And what if afterwards, I need to add some other kinds of clients? In that case, I may have to add new methods in the one large client class, which goes against **OCP** (Open Closed Principle). So, a better approach is to partition the write client and read client into two different classes, and they both inherits from an interface.

Impact on Design:

Design the write client and read client into two different classes, which both implements a common interface, IClient.

Issue #2: Should design two different GUIs for write client and read client?

Solution:

For the GUIs for write client and read client, according to the requirements, the first purpose is just to display messages contents. From this perspective, it seems that one GUI is enough for write client and read client. However, considering the extensibility of the design, GUI may also provide functionalities to construct requests to send to the remote key-value database, it's better to have different GUIs for write client and read client.

Impact on Design:

Design different GUIs for write client and read client respectively.

Issue #3: For multiple write/read clients, should there be multiple GUIs?

Solution:

It is normal that during the test, I need to start multiple write clients or read clients to send requests and receive responses. I have demonstrated that there should be different GUIs for write client and read client in the previous issue. But, what if there are multiple write clients? Do I need to create multiple GUIs for them? It may not be necessary. During the test, I may start lots of write clients and read clients, there will be too many GUIs if I create multiple GUIs for them respectively. One possible solution to partition the messages from different write/read clients is to use multi-threads.

Impact on Design:

Utilize multi-threads, including locking mechanism to partition messages from different write/read clients, and just use one GUI for one kind of clients.

5.3 Message Processor

Issue #1: How to handle invalid XML files?

Solution:

It should be very common, that a user may provide invalid XML files for the read client and write client. So, when XML parser is trying to parse the files and get the requests necessary to send, it must be very careful, to check whether the input is valid. For example, a user may provide a request with addition

operation of a key-value pair, however, he/she may just give the key, without the corresponding value. Under this circumstance, the parser should stop and ask for valid inputs.

Impact on Design:

For XML parser, I may design some input checks to avoid program crash because of invalid inputs. Additionally, it may also be beneficial for me to design the error check to provide more information. For example, if the user forgot to provide the value, as what I have described above, it will automatically warns that a value is expected.

Issue #2: Is it necessary to design different XML parsing mechanisms for read clients and write clients?

Solution:

Considering the fact that read clients and write clients are both leveraged to send requests to the remote database, and the only difference is the type of the requests, it may not be necessary to design different kinds of XML parsers. Specifically, I could add an additional XML element to differentiate the messages are for read clients or write clients. And an example is shown below.

<Client type>

Read

</Client type>

Impact on Design:

Design only one kind of XML parsing mechanism for both of write clients and read clients, but add an additional XML element in the XML files to determine the messages are for write clients or read clients.

Issue #3: In Request Message, why a dictionary is used, not just a string?

Solution:

In my design of Request Message class, I use a dictionary with string type as the key, and a generic type (Para) as the value. In this way, all the requests information related to the operations on the database could be saved in the dictionary. What about a string to store the data? As what I have analyzed in Chapter 4, for different requests, there are different type of parameters. For example, for Edit command, it may edit the

children, which is of List<Key> type, while, it may edit the payload, which is of Data type. To use a string to represent these complicated data structures is not easy. Additionally, actually, the serialization and de-serialization of the Request Message is taken care by WCF, so it is not necessary to put them in a string. WCF will automatically serialize the messages into string (XML) to send.

Impact on Design:

Use a dictionary in Request Message class to store operation information. Leave the serialization and de-serialization to WCF.

5.4 Communication Channel

Issue #1: How many blocking queues should be implemented for each side of Communication Channel?

Solution:

Because there are multiple write clients and read clients in the client side, and they may send out their requests simultaneously. So for the client side, and the server side, they need to implement blocking queues to process the requests. Furthermore, a thread should be created to process each thread. In my design, there are two blocking queues in each side of Communication Channel, which means that in each side, there are a receive queue and a send queue. And a receive thread and a send thread are also created for the queues. This design is also necessary for massive requests processing, which may cause a performance issue.

Impact on Design:

Design two blocking queues (send queue and receive queue) for each side of Communication Channel.

Issue #2: How to differentiate responses for read clients from those for write clients?

Solution:

When the client side Communication Channel's receive queue receives the responses, it may not know whether the response is for a read client or a write client. But, because in my design, there are different GUIs for read clients and write clients, finally, the responses need to be classified. While, for my design of Response Message class, there is a Message Type to achieve that. There are 8 types of messages, and only

"Query" is for the read client. So when Message Processor processes the responses, it should not be difficult to read the message type to determine destination of the responses.

Impact on Design:

Leave the classification of the responses to Message Processor.

5.5 Timer

Issue #1: Should timing information contained in the messages?

Solution:

If my target is just to fulfill the requirements, it seems that write clients and read clients just need to check the current time before the sending of the requests and after the receiving of the responses. In this case, including timing information in the messages transferred is not necessary, and may cause performance slow down. However, considering the extensibility of the whole design, timing information such as the start time of the requests, is not only useful for the clients, but also beneficial for the servers. The server could also utilize the time to calculate the latency between clients and server. So, in my design, I put timing information in the messages.

Impact on Design:

Messages transferred between clients and server needs to contain timing information.

Issue #2: What about the elapsed time recorded is lower than calculation unit?

Solution:

It's common that sometimes the requests are processed very fast, whose time is even lower than the calculation unit. For example, if the timer calculates timing in seconds. For some requests, the end-to-end time may be smaller than 1 second, causing the final elapsed time is just 0. To solve this kind of problem, I need to use a high-resolution timer whose calculation precision is 1 microsecond/nanosecond to calculate elapsed time. After the high-resolution timer is used, if there still be a 0 elapsed time, it may reveal a bug in the design, because the whole procession should never smaller than 1 microsecond/nanosecond.

Impact on Design:

Use a high-resolution timer to calculate elapsed time.

Issue #3: Is it necessary for the server to have a Timer package?

Solution:

In my top level package diagram, the server side also has a Timer package. While, according to the requirements, it seems that there's no need for the server to record elapsed time. However, considering that in practice, it is also crucial for the server to obtain the timing information, such as communication latency and processing time, it still might be beneficial for the server to have a Timer package.

Impact on Design:

Add a Timer package in the server side.

5.6 Others

Issue #1: How to ensure security of the remote database?

Solution:

Security is a crucial topic in a remote database design. There are two parts in the security, one is related to the database itself, and the other is related to the communication channel. If the database is not secure enough, a client who does not have enough permission could still modify other data, which will cause information leakage. So, a sophisticated database should consider implementing robust access control mechanism. Additionally, for the communication channel, some security mechanisms may also necessary in modern remote database design to avoid messages be decoded. While, there is a trade-off between cost (time) and security. Because there may not enough time to design security mechanisms and the main uses for my project are developers, teaching assistants and instructor, I do not need to care too much about it.

Impact on Design:

Design access control and secure channels only when time is enough.

Issue #2: How to ensure the performance of the remote database?

Solution:

To ensure the performance of the remote database, I need to implement a performance test on the remote database. Multiple write clients and read clients should be started simultaneously, and they all need to send requests to the server. Timers in the client side should be utilized to record elapsed time. Because in my design, there are two blocking queues in each side of client and server, normally, the server should be able to handle massive requests received at the same time. However, when there are too many requests, the remote database may still experience a slowdown, which needs to be detected carefully.

Impact on Design:

Implement a stress test on the remote database.

6 Conclusion

In practice, a remote key-value database is widely used because of its fast CRUD operations, support of semi-structured and un-structured data, and capability of remote access. In this document, I demonstrated a design of a remote key-value database, including the concepts of key-value NoSQL database, Windows Communication Foundation, Windows Presentation Foundation, use cases of the application and structures of the packages. Finally, I discuss some critical issues such as security, complexity and performance.

Appendix

```
namespace Project4Demo
{
  ///////////////////////////////////////////////////////////////////
  // MessageType Enumeration
  // - The enumeration defines the 8 different types of requests, which
  //   the client could send to the remote database.
  public enum MessageType
  {
    Save,      // Ask the database to persist its content to XML file
    Load,      // Ask the database to load/augment content from an XML file
    Add,       // Ask the database to add key-value pair
    Delete,    // Ask the database to delete key-value pair
    Edit,      // Ask the database to edit value of a key
    Query,     // Query the database
    Schedule,  // Ask the database to schedule a time for persistency
    Create     // Ask the database to create an immutable database
  }

  ///////////////////////////////////////////////////////////////////
  // IMessage Interface
  // - An interface which is actually empty. It just represents the interface
  //   of messages transferred.
  public interface IMessage { }

  ///////////////////////////////////////////////////////////////////
  // RequestMessage Class
  // - The instance of RequestMessage represents the requests which will
  //   be sent to the remote database.
  [DataContract]
  public class RequestMessage<Para> : IMessage
  {
    [DataMember]
    public MessageType typeName { get; set; } // the type of the request

    [DataMember]
    public Dictionary<string, Para> parameters { get; set; } // the parameters for very
request

    [DataMember]
    DateTime startTime { get; set; } // start time of the request

    [DataMember]
    DateTime stopTime { get; set; } // stop time of the request

    [DataMember]
    string fromURL { get; set; } // source URL

    [DataMember]
    string toURL { get; set; } // target URL

    //----< constructor >-------------------------------------------
    RequestMessage()
    {
      typeName = MessageType.Add;
```

```
      parameters = new Dictionary<string, Para>();
      startTime = new DateTime();
      stopTime = new DateTime();
    }
  }

  ///////////////////////////////////////////////////////////////////
  // ResponseMessage Class
  // - The instance of ResponseMessage represents the requests which will
  //   be sent to the remote database.
  [DataContract]
  public class ResponseMessage<Info> : IMessage
  {
    [DataMember]
    public MessageType typeName { get; set; } // the type of the response

    [DataMember]
    public Info messageInfo { get; set; } // the information of the message in the
response

    [DataMember]
    DateTime startTime { get; set; } // start time of the response

    [DataMember]
    DateTime stopTime { get; set; } // stop time of the response

    [DataMember]
    string fromURL { get; set; } // source URL

    [DataMember]
    string toURL { get; set; } // target URL

    //----< constructor >-------------------------------------------
    ResponseMessage()
    {
      typeName = MessageType.Add;
      startTime = new DateTime();
      stopTime = new DateTime();
    }
  }
}
```

References

[1] SQL vs NoSQL, https://www.digitalocean.com/community/tutorials/understanding-sql-and-nosql-databases-and-different-database-models.
[2] ACID Wikipedia, https://en.wikipedia.org/wiki/ACID
[3] WCF Wikipedia, https://en.wikipedia.org/wiki/Windows_Communication_Foundation.
[4] Joseph Albahari, Ben Albahari, C# 5.0 in a Nutshell: The Definitive Reference, 5th edition, 2012.
[5] WPF Wikipedia, https://en.wikipedia.org/wiki/Windows_Presentation_Foundation
[6] Project #5, Dr. Fawcett, http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project5-F2015.htm.