Optimizing Client Latency in a Distributed System by Using the "Remote Façade" Design Pattern

Cosmin RABLOU
OctaVIA AG, Germany
cosminrablou@hotmail.com

In this paper we investigate the role of the Remote Façade pattern in the optimization of distributed systems. The intent of this pattern is to wrap fine-grained remote objects in a coarse-grained interface and thus greatly reduce the total number of calls executed over the network. The measurement of the performance gain achieved by implementing this pattern is done through testing with a distributed application written in C# and using the latest Microsoft framework for distributed systems (Windows Communication Framework). Furthermore, we will be presenting the scenarios in which the implementation of the Remote Façade pattern brings a significant performance gain. Finally we show further scenarios in which the performance brought by this pattern can be investigated.
Keywords: *Object-Oriented Programming, Distributed Systems, Design Patterns, Remote Façade, Performance, Optimization, Windows Communication Framework*

1 Introduction

The main reason behind a distributed system is the performance gain when replicating components across the network. However, this performance gain can be easily lost when implementing the wrong architecture. The design patterns for distributed architecture establish general rules that must be followed in order to achieve a high-performance distributed system.

2 Description of a pattern

Every pattern can be uniquely defined using four elements:

- *The name* – is a common term used to describe the pattern. By using pattern names we create a metalanguage, which allows us to design applications at a higher level of abstraction.
- *The problem* – indicates us when to use the pattern. The problem describes the context where the pattern can be applied.
- *The solution* – describes in terms of object-oriented programming the components that build up the design, their responsibilities and the relations between them.
- *The consequences* – describe the benefits and liabilities that occur when implementing the solution. Every solution has both advantages and disadvantages. A design pattern might bring a higher flexibility of the application by decreasing its performance. It is critical to understand these trade-offs, as this is maybe the most important aspect when choosing a design.

3 Remote Façade

In this chapter we will present the Remote Façade pattern by describing its essential elements.

3.1 Intent

Remote Façade provides a coarse-grained facade on fine-grained objects to improve efficiency over the network [1].

3.2 Problem

The object-oriented paradigm has become so wide spread, that even distributed systems can use it. With modern frameworks (like .NET, for an instance) it is easy to create objects that are available remotely.

However, due to the easiness of the development, the system architect and the programmer tend to neglect the performance issues that arise through the usage of the remote methods. The fact is that remote methods are slow. The performance loss can vary based on the configuration of the distributed system, but it is generally accepted that a calling a method over the network is usually at least 1000 times slower than the same method called in-process.

So, even if the framework will reduce the complexity of making a remote call, it cannot improve the performance of the call. In order to execute a remote call, a client must fulfil at least the following tasks:

- *Look up the address of the remote object.* The way that this is done depends on whether the distributed system uses a sort of load balancing or not. If load balancing is not used, then the address is saved directly on the client; however, if we have more than one server, the client must usually contact a server location service prior to contacting the server.
- *Connect to the remote service*. The performance loss depends on the binding type used (TCP/IP, message queues, etc).
- *Serialize the data into a byte stream.* The serialization process transforms the data into a format that can be decoded by the remote peer.
- *Encrypt the data.* If the information sent over the network is sensitive, it must be encrypted. The framework usually fulfils this task, but encrypting/decrypting a byte stream decreases the performance.
- *Invoke the method.* The client must submit through the network the information regarding the method that should be called and the values of the parameters for that method.

Furthermore, we are confronted with another issue: the good design principles state that an object should encapsulate the data in attributes and only provide access to this information through methods (the so-called getter and setter). This is how the fine-grained objects are created: objects with methods that are responsible for a small and atomic piece of functionality.

The fine-grained objects are perfect for the usage within the same process, as these are very flexible. The large number of methods can be combined in different ways, which increases the reusability of these objects for different scenarios. The performance is never an issue in an in-process case, because the calling of a method has (almost) no performance loss.

However, this fine-grained interface performs poorly in a distributed scenario, because in this case every method call involves costs that cannot be neglected. The usage of fine-grained objects implies invoking more methods to perform a high-level task. The resulting performance loss in a distributed system can be dramatic.

3.3 Solution

The Remote Façade pattern defines only remote objects with a coarse-grained interface. These remote objects encapsulate the objects on the server and expose only a small number of methods for calls over the network. This greatly reduces the total number of remote calls that are executed over the network and therefore increases the performance of the distributed application.

However, another consequence is that not all the functionality exposed by the server will be available to the client. This means that the definition of the remote interface requires good planning in order to make sure that the entire server functionality needed by the client is exposed and that only a small number of methods can be invoked remotely.

Usually, a coarse-grained interface is achieved by combining the available methods. For instance the getter and setter methods are combined in bulks and only these bulk methods are exposed over the network in the “Remote Façade”-objects.

Furthermore, a coarse-grained object contains quite a small piece of functionality. Its task is mainly to translate the remote calls in server internal calls and to send the results obtained in these internal calls back to the client.

Another issue that might rise is the granularity of the “Remote Façade”-objects. For instance, if we have to send over the network information regarding an invoice, we will have to provide general information about the invoice, about the invoice lines (the sold products) and about the customer (buyer). Do we create only one object, exposing all necessary methods or do we create an object for the invoice and another one for the customer? There is no wrong answer to this question, but we would recommend defining as few “Remote Façade”-objects as possible, be-

cause this way we have a better overview (and control) on the number of exposed methods.

3.4 Structure

For a better understanding of this design pattern, we choose as an example the Customer class, which contains 6 attributes: an unique ID from the database, a name, an address, a value-added tax identification number, an e-mail address and a phone number. This class exposes getter and setter methods for each attribute, which means we have a total of 12 methods.

Such a class would perform poorly if called remotely in a distributed environment. Therefore, we create a "Remote Façade" class, which defines only 2 methods: one getter and one setter for all attributes. Only this class will be available for distributed calls, ensuring that the total number of remote calls will decrease.

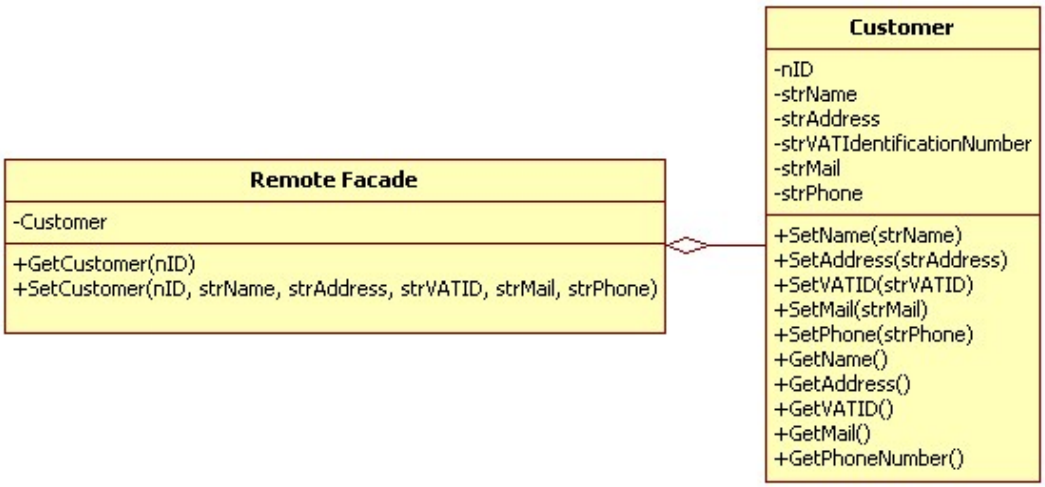

Fig. 1. Remote Façade class

The new class works as a wrapper and contains only functionality that translates the remote calls in server-internal calls. The main business logic is still contained in the old Customer class.

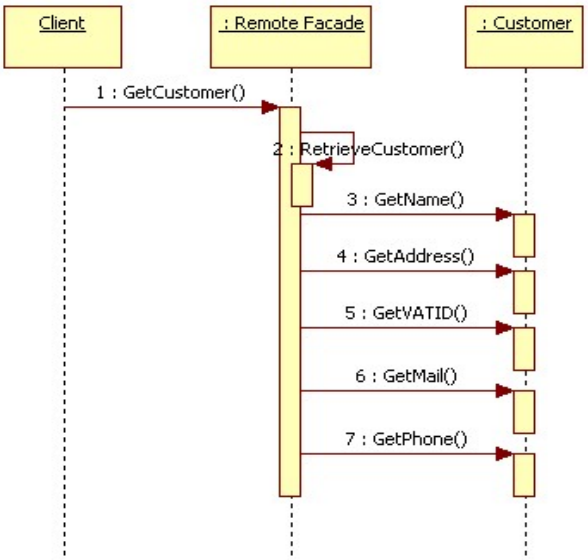

Fig. 2. Remote Façade diagram

3.5 Consequences

Benefits

- *Improved performance* due to the reduced number of remote calls. Most of the times, the bottleneck in the case of a distributed application is the transport over the network. Decreasing the total number of remote calls reduces the client latency and therefore greatly improves performance.
- *Simplified server interface*. Defining only a reduced number of remote calls helps reducing the server complexity. The client will only access high-level functionality that has been tailored to fit its needs. This is also the main motivation behind the Façade pattern [2].
- *Reduced dependencies* between client and server - also due to the reduced number of remote calls. Loose coupling between components (in our case, classes) is one of the basic principles of good object-oriented design as it leads to classes that are more available for reuse.

Liabilities

- *Additional programming effort/additional costs***.** The Remote Façade classes must be considered when designing a distributed architecture. This increases the total cost of the application. The Remote Façade classes might not be difficult to program, due to the fact that they only translate the remote calls in server internal calls, but we can have a large number of classes, depending on the architecture and the complexity of the application.
- *Increased number of classes***.** By implementing the Remote Façade design pattern you add to your application a number of classes that wrap the server functionality. The number of these classes depends on the complexity of your application. It can be difficult to manage this number of classes, especially if the server application is already complex.
- *Performance loss* due to a possibly increased quantity of data sent during a remote call**.** By grouping methods in a Remote Façade we also group the parameters that are sent across the network. It is easy to imagine a scenario where not all the data sent by the server is also needed on the client. However, the performance loss caused by the amount of unnecessary data being serialized / deserialized and sent over the network in one remote call is usually smaller than the performance gain due to the implementation of the Remote Façade.

4 Measuring the performance of a distributed system

When measuring the performance of a distributed application, there are two different parameters that should be considered:

- *Client latency* is the time that passes from the moment when the client initiates the remote call until the response sent by the server reaches the client.
- *Data throughput* is the average amount of data that is sent / received over the network by the server within a certain time period (for instance, 1 second).

Every distributed system should strive to reduce the client latency and to have a high data throughput.

In our sample we will only consider the client latency, as this is sometimes considered more important than the throughput. The reason is that usually an upgrade of the server machine or a faster network can often increase the data throughput, while the client latency tends to remain the same.

5 Client latency with Remote Façade

As we have already discovered during the presentation of the Remote Façade pattern, there are two elements that can influence the client latency when implementing the pattern:

- *The reduced number of remote calls* decreases the client latency and therefore greatly improves the performance of the distributed system
- *The increased amount of data serialized / deserialized over the network* increases the client latency and decreases the performance.

Our task has been the measurement and comparison of an application's performance when using / not using the pattern Remote Façade.

6 Test environment

For the testing environment, we used a network containing two computers connected via Ethernet. A router was responsible for forwarding the data packets between the client and server. This is a common way of setting up a LAN and therefore is a typical scenario for running a distributed system.

The server and the client both run Windows XP SP3 as an operating system. The .NET Framework 3.5 must be installed on both systems in order to be able to run a Windows Communication Framework -based application.

We used TCP/IP as the communication channel between client and server. In order to eliminate as much as possible the influence of the network traffic on the performance of our test application, the traffic through the network was reduced to a minimum.

The firewall was turned off (both on server and client). This ensures that the firewall cannot cause a delay due to some additional checks performed on the IP packets sent over the network.

7 Test application

We used Windows Communication Framework (WCF) and C# to write the application that helped us measure the client latency.

WCF uses the Broker pattern to implement a distributed application [3], separating the details of the remote communication from the application logic.

WCF needs three parameters in order to be able to define a remote communication [4]. These parameters are also known as the ABC:

- *The **A**ddress* indicates the location of the server
- *The **B**inding* indicates the protocol used to communicate between the server and the client.
- *The **C**ontract* specifies the interface that is implemented by the server.

These parameters are defined in the configuration file of both the client and the server. We used sockets as a connection between the server and the client. The sockets represent a very common communication scenario when working with distributed applications.

Our test application covers the following scenarios:

- A distributed application implementing/not implementing the Remote Façade pattern requesting only one parameter. The parameter chosen in this case was an int (as a data type).
- A distributed application implementing/not implementing the Remote Façade pattern requesting two parameters. The second parameter was declared as a string and had a total length of 38.
- A distributed application implementing/not implementing the Remote Façade pattern requesting three parameters. The third parameter was implemented as a string and had a length of 58.
- A distributed application implementing/not implementing the Remote Façade pattern requesting four parameters. The fourth parameter was declared as a string and had a total length of 7.
- A distributed application implementing/not implementing the Remote Façade pattern requesting five parameters. The fifth parameter was declared as a string and had a total length of 17.
- A distributed application implementing/not implementing the Remote Façade pattern requesting six parameters. The second parameter was declared as a string and had a total length of 14.

The list of parameters in each presented scenario contains the same list of parameters from the previous scenario (see above) and introduces an additional parameter.

A distributed application implementing the Remote Façade pattern groups all parameters in a method call. A distributed application not implementing the pattern will issue a separate remote call for each additional parameter.

We tested the client latency for each scenario by running bulks containing 100 identical

remote calls. Each bulk of remote calls was run six different times in order to ensure that the average value obtained is statistically representative. This gives us a total of 6(rounds) * 6(total parameter count) * 2(with/without the pattern) * 100(count of remote calls in a bulk) = 7200 test cases.

The first remote call was executed separately and not included in any of the bulks. The first call fulfills an additional task: it opens the communication channel. This is a time consuming task. In our tests the first call had a latency that fluctuated from 270 milliseconds to 600 milliseconds, while a usual call takes approximately 1 millisecond. But once the communication channel was open, all subsequent remote call from our test reused this resource. Therefore, the obtained results are representative for the scenario where the client and the server already established the communication and the remote calls are executed at a steady rate. If the server is idle and does not receive calls for a longer time it will shut down the communication channel and a new client call will have to re-open it.

Similar bulks implementing/not implementing the pattern and having the same number of parameters are tested subsequently in order to ensure that possible network traffic bursts do not influence the test dramatically.

The data sent over the network was identical for similar scenarios. For instance if we compared the cases implementing/not implementing the Remote Façade pattern with four parameters, the values of these parameters were in both cases identical. This how we ensured that the amount of data sent over the network did not influence the test results.

Moreover, the data sent over the network was cached in the memory during the server initialization. This means that every call accessed the data from the cache instead of getting it from the database. Therefore, we can state that the database latency has no influence on our test results.

We switched off the security for the NetTcp binding in WCF (in the application configuration file). By default, the security is enabled for a distributed application written using WCF. If the security is not disabled for the test application (both on server and client) it can influence the overall performance of each remote call.

Below you can see some of the client code that was used to generate the 7200 remote test calls.

```
class Program
{
static void Main(string[] args)
{
InvoiceServiceClient client = new
InvoiceServiceClient();
//The first call has to be executed
separately
FirstCall(client);
Console.WriteLine("Round 1");
Compare(client);
Console.WriteLine("Round 2");
Compare(client);
Console.WriteLine("Round 3");
Compare(client);
Console.WriteLine("Round 4");
Compare(client);
Console.WriteLine("Round 5");
Compare(client);
Console.WriteLine("Round 6");
Compare(client);
Console.WriteLine("Done");
Console.ReadLine();
}

private static void
FirstCall(InvoiceServiceClient client)
{
Stopwatch s = new Stopwatch();
s.Start();
client.GetCustomer1(1);
s.Stop();
Console.WriteLine("First call,
milliseconds: {0}\r\n",
s.ElapsedMilliseconds);
}

private static void
Compare(InvoiceServiceClient client)
{
Stopwatch s = new Stopwatch();
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer1(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 1
parameter, milliseconds: {0}",
s.ElapsedMilliseconds);

s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
}
s.Stop();
```

```
Console.WriteLine("Without Remote
Facade, 1 parameter, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer2(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 2
parameters, milliseconds: {0}",
s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
client.GetName(1);
}
s.Stop();
Console.WriteLine("Without Remote
Facade, 2 parameters, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer3(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 3
parameters, milliseconds: {0}",
s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
client.GetName(1);
client.GetAddress(1);
}
s.Stop();
Console.WriteLine("Without Remote
Facade, 3 parameters, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer4(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 4
parameters, milliseconds: {0}",
s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
client.GetName(1);
client.GetAddress(1);
client.GetVATIdentificationNumber(1);
}
s.Stop();
Console.WriteLine("Without Remote
Facade, 4 parameters, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer5(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 5
parameters, milliseconds: {0}",
s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
client.GetName(1);
client.GetAddress(1);
client.GetVATIdentificationNumber(1);
client.GetMail(1);
}
s.Stop();
Console.WriteLine("Without Remote
Facade, 5 parameters, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetCustomer(1);
}
s.Stop();
Console.WriteLine("Remote Facade, 6
parameters, milliseconds: {0}",
s.ElapsedMilliseconds);

s.Reset();
s.Start();
for (int i = 0; i < 100; i++)
{
client.GetID(1);
client.GetName(1);
client.GetAddress(1);
client.GetVATIdentificationNumber(1);
client.GetMail(1);
client.GetPhoneNumber(1);
}
s.Stop();
Console.WriteLine("Without Remote
Facade, 6 parameters, milliseconds: {0}
\r\n", s.ElapsedMilliseconds);
Console.WriteLine("");
}
}
```

8 Test results

The obtained client latency results for each bulk of remote calls are presented in table 1. Each value represents the duration in milliseconds of 100 identical remote calls. This is the interval of time from the first request sent to the server until the client receives the last

response. The lower the client latency, the better application performance is.

Table 1. Client latency calculated during 6 rounds of tests with/without implementing the Remote Façade pattern

	1st round	**2nd round**	**3rd round**	**4th round**	**5th round**	**6th round**
Remote Facade 1 parameter	132	106	87	88	94	90
Without Remote Facade 1 parameter	122	96	87	90	86	86
Remote Facade 2 parameters	130	113	89	91	88	94
Without Remote Facade 2 parameters	239	196	176	180	176	181
Remote Facade 3 parameters	132	98	89	102	92	108
Without Remote Facade 3 parameters	365	264	267	267	265	270
Remote Facade 4 parameters	136	95	92	108	109	105
Without Remote Facade 4 parameters	404	356	357	360	376	365
Remote Facade 5 parameters	118	98	102	109	105	113
Without Remote Facade 5 parameters	448	452	444	448	449	455
Remote Facade 6 parameters	122	98	106	117	96	121
Without Remote Facade 6 parameters	549	533	535	546	545	556

Based on the results contained in table 1 we further calculated the average client latency for each test case. Table 2 shows the obtained average values. The client latency values obtained when implementing the pattern are presented in the second column while the values obtained by not implementing the patterns are shown in the third column.

Table 2. Average client latency with/without implementing the Remote Façade pattern

Number of distinct parameters	**Implementing Remote Facade**	**Without Remote Facade**
1	99.5	94.5
2	100.8333333	191.3333333
3	103.5	283
4	107.5	369.6666667
5	107.5	449.3333333
6	110	544

The client latency can be better compared if we analyze the graphical representation of the results (figure 3). Without Remote Façade, the client latency grows in an almost perfect arithmetic progression when the number of parameters sent over network increases due to the fact that the number of remote calls increases too.

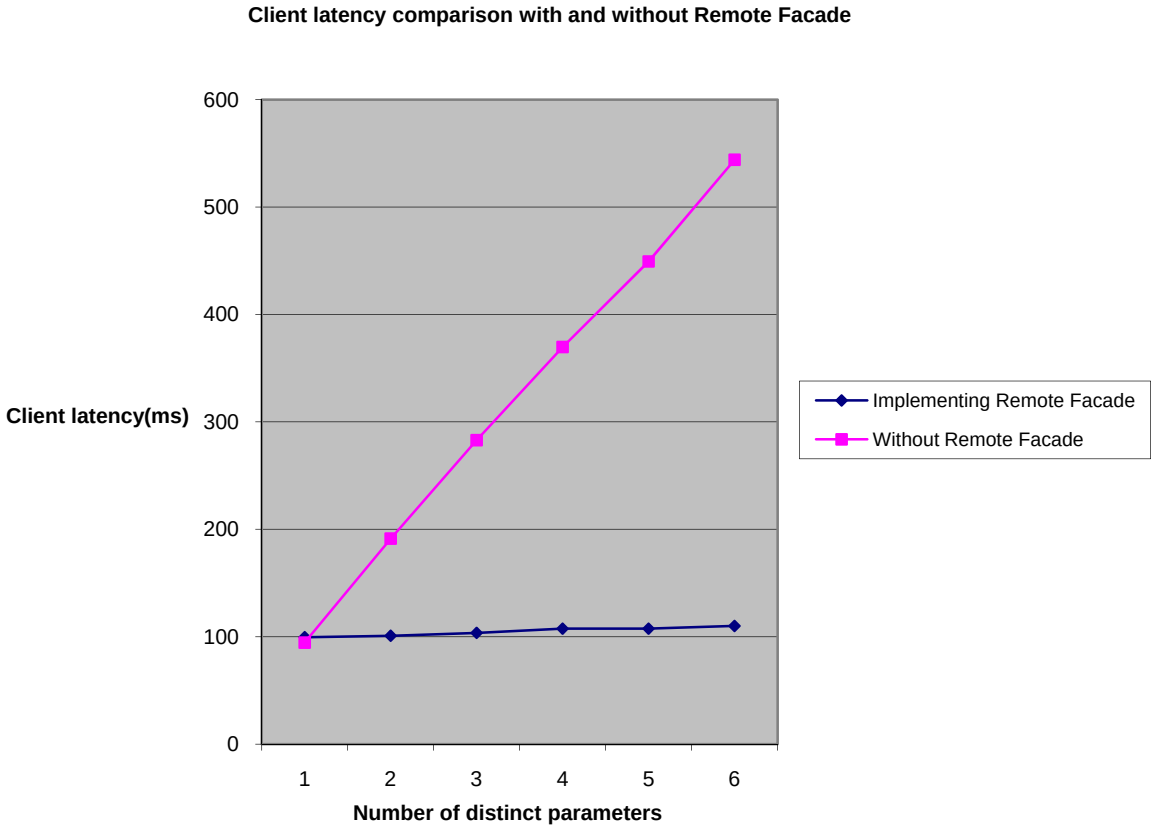

Fig. 3. Comparison of client latency with/without Remote Façade

The only case when the version that is not implementing the Remote Façade pattern performs better is the scenario where we have only one parameter. However, this case can never occur when implementing Remote Facade because, as we already stated, the intent of this pattern is to group methods (and therefore their parameters too) in objects with coarse-grained interfaces.

It is interesting to observe the performance degradation that occurs when the number of parameters grows and the pattern is not used. The client latency multiplies in this case with a factor comparable to the number of remote calls (parameters). Compared to this performance degradation, the performance loss when using the Remote Façade pattern can almost be neglected.

Analyzing the graphical data representation, we can notice the almost perfect linear growth of client latency with the number of parameter. This shows us that there is no data aberration in the samples that we have chosen. Furthermore, this means that it makes sense to calculate for each sample the percentage of performance loss relatively to the first scenario (containing only one parameter). The results are shown in table 3.

Table 3. Client latency performance loss compared to the one parameter-scenario

Number of distinct parameters	Performance loss (%) when implementing Remote Facade	Performance loss (%) without Remote Facade
2	1.340033501	102.4691358
3	4.020100503	199.4708995
4	8.040201005	291.1816578
5	8.040201005	375.4850088
6	10.55276382	475.6613757

When the Remote Façade pattern is implemented, the maximal performance loss due to an increased number of parameters sent over the network is about 10,5%. However, the similar test case implemented without the pattern, has a performance loss of approximately 475%.

9 Conclusions

The results show without a doubt that the implementation of the Remote Façade pattern increases the performance of a distributed application. This comes as no surprise, as reducing the number of consequent remote calls should decrease the client latency.

However, the novelty brought by this study is the huge performance difference between a version implementing the pattern and one that doesn't.

We would expect that processing a larger number of parameters (the serialization / deserialization and especially the sending over the network) has more influence on the performance. The serialization / deserialization is done in-process (on the server and on client, respectively), so that the performance loss due to the converting of a larger amount of data can be neglected.

But we thought that at least by sending a larger amount of data we would negatively influence the performance of our distributed application. However, the tests have shown that this is not the case. The performance loss is only about 10% when the amount of data sent over the network increases significantly. We have a data amount growth of 6700%, as in the first case we sent only 4 bytes while in the last case we sent 272 bytes.

There is only one explanation that can justify this result: all parameters transmitted during a remote call were sent in only one IP packet. Therefore, we can state that as long as all parameters fit in one IP packet the only factor that influences the performance of a distributed application is the number of remote calls.

Furthermore, it would be interesting to determine the point where the usage of the pattern is beneficial for the performance of the application. As we can see, if we have only one value that we pass as a parameter in the remote call, then the best choice is to go without using the Remote Façade pattern. The performance difference is not very significant (less than 5%).

For all other cases (two parameters or more) the implementation of the pattern brings a major performance improvement. The performance gain when choosing the pattern is directly proportional with the number of parameters that are grouped in one remote call. For instance, in the case of 6 parameters, the version implementing the pattern was almost 5 times faster than the version not implementing the pattern.

In conclusion, we would recommend at least considering the implementation of the Remote Façade pattern whenever a distributed system is designed and implemented. The more parameters are grouped in a remote call, the better the overall performance of the distributed application is.

10 Next steps

It would be interesting to analyze in another paper if the results observed in this study are confirmed also when using much larger amount of data (a larger dataset, for instance).

References

[1] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee and R. Stafford, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002.

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] C. Rablou, *Distribution strategies in clustered systems*, Bucharest, 2010.

[4] S. Resnick, R. Crane and C. Bowen, *Essential Windows Communication Foundation (WCF): For .NET Framework 3.5 (Microsoft .Net Development),* Addison-Wesley, 2008

Cosmin RABLOU has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2001. He joined the same year the team at Derdack GmbH, Germany, where his main focus was on the telecommunications and mobile solutions development. In 2007 he joined OctaVIA AG, where he currently develops SAP applications for telecommunications. His main professional interests are object-oriented design and programming (with primary focus on design patterns) and distributed systems (mainly web applications).