Optimizing service availability in VoIP signaling networks, by decoupling query handling in an asynchronous RPC manner

Voichiţa Almăşan and Iosif Ignat
Technical University of Cluj-Napoca
Computer Science Department
28 G. Baritiu Street, 400027 Cluj-Napoca, Romania

Abstract

This report presents some aspects of the communications in distributed systems, with a special accent on the needs of VoIP systems. It describes some interactions between servers, using different types of queries, that belong to various protocols. It also presents the problem of the server waiting synchronously for each individual query to be executed and thus the need for decoupling query execution from the rest of the processing. The problem is similar to the one of slow peripheral devices. This would suggest that the solution to the query execution decoupling problem resembles the DMA functionality from the hardware, that is, an asynchronous interaction is needed. Performance matters and future development of the proposed query execution decoupling layer are also evaluated and discussed.

1. Introduction

The domain of distributed systems has become very popular nowadays and it has come to a point where it needs to take into consideration many efficiency aspects, as well as many security threats. The weak point in most distributed systems is the fact that their nodes are connected via unreliable links. On top of these links, reliable communication should be performed, as message passing and asking for services from other nodes is a key-idea in the functioning of any flavour of distributed system. This is why, we can say that the network links between the nodes of these systems are the parts where they are most likely to fail. A type of failure that it is very difficult to act against is the lack of availability, which could have been generated by a denial of service (DoS) attack or by an inappropriate design of the system. This type of failure could be due to a network link bottleneck, or to the overcharge of a node from whom you are requesting a service. The network link's bottleneck could be prevented through traffic shaping and QoS classes, while the node's overcharging can be solved by using load balancing more subtle techniques.

The lack of availability is especially dangerous when one or more services, such as SIP servers, are trying to contact another, very loaded, distant server, such as a database or DNS server, by querying it for some particular piece of data or information. The heavy load of the distant server, and thus large response time for its clients, is inducing large response times for the SIP servers' clients, even if the SIP servers might not be very crowded. This is why, a means of decoupling the interaction with one server's clients from the interactions with various types of servers would be needed, similar to the need of decoupling slow I/O from the processor, in hardware.

Section 2 will give the reader an overview of some distributed systems that are faced with the need to execute a lot of queries, possibly of various types, throughout their existence. Section 3 will focus on explaining the solution we have come up with, to improve the performance of systems described in section 2, namely web-based systems, peer-to-peer systems, and VoIP systems, especially. Section 4 will present and interpret some results obtained using our prototype. Section 5 will conclude the article and will show some future development directions in improving performance of VoIP systems and other distributed systems, by finding programming models for decoupling the launching of a query from its actual time-consuming execution.

2. Background and related work

Among distributed systems that suffer most from the lack of ability to communicate with one another are the World Wide Web, peer-to-peer systems, and also VoIP distributed systems, namely VoIP signaling networks.

Within the *WWW*, the look-ups are both for finding nodes, such as DNS look-ups, and for retrieving data, for example when content-based searches are performed. In the WEB data look-ups, database queries might also inter-

vene [10].

Within the *peer-to-peer systems*, look-ups are essential, being the core-idea of this network. These look-ups refer both to nodes belonging to the same peer-to-peer overlay, for routing purposes [9] based on the overlay's structure and rules, and also to data, which can be found by looking-up in a Distributed Hash Table (DHT).

Within the *VoIP systems*, look-ups are performed by the signaling network, which is the distributed system component of the VoIP network, for routing purposes [7]. For example to find out the address of the next hop or the destination IP address of a call when it is being initiated, at least one DNS look-up should be performed. On the other hand, for retrieving some data that the system needs at a certain moment, such as billing information, database queries are usually used. We can see that the needs of the VoIP signaling networks somewhat combine the ones of the WWW and the ones of the peer-to-peer systems. The look-ups, independent on their nature, are essential in SIP or P2P SIP, the signaling protocols widely used within VoIP nowadays, because clients must find the registrars that are responsible for them, either by using SIP techniques [4], for simple SIP, or by using peer-to-peer techniques, for P2P SIP [6]. SIP servers must perform a lot of queries because, throughout their lifetime, they must oftenly perform the UAC (User Agent Client) role, besides the UAS (User Agent Server) role [5]. In what follows, when using the term *VoIP system*, we shall especially refer to its signaling network, which represents the distributed system whose problems we are aiming to solve.

It results that the look-up/query mechanism is crucial to distributed systems, and methods should be found to improve it. Such improvement methods have been studied by other research groups, too. They can be at server-level, at network-level and at client level. Improvements at server-level are described in [2], where some algorithms are presented for optimizing query scheduling algorithms for better response times from the server side. Also, [1] does a description of means for decoupling DNS server selection from the actual request routing, at server-level. Improvements at network-level are done using QoS techniques. Studies for client level optimizations have been performed mainly for database queries on the grid [11]. Our scope for query types is larger than just database queries. We will try to be as generic as possible in the approach, since we aim to apply our research to as many query types as possible, even if we are mainly trying to solve the needs of VoIP systems. Improving the response of a distant service, using QoS techniques or load balancing techniques, is beyond our scope. We will focus on improving the behaviour of a given service provided by a node of a distributed system, regardless of the load of the other members of the system.

In what follows, we will focus mainly on the needs of VoIP systems. The problem we are actually trying to solve is that if a server entity running on a single node has enough memory and bandwidth to carry out N calls at a time, but is waiting for a very busy group of DNS servers to resolve m, much smaller than N, queries, it will seem to not be responding very easily or sometimes not at all to a new client that is trying to contact it. This usually happens because the server is overloaded, but it can also be due to a security threat, which we wish to act against, in case it appears. The situation that is created sugests an asynchrony between the requester and the provider of the answer, and also the fact that the requester should not block its activity while waiting for the answer. This problem has already been addressed on the grid, when it comes to database queries [11]. What we are trying to do is to generalize the approach for databases to any type of look-up or query, and make it protocol independent, since we have seen that VoIP systems need both many DNS look-ups and many database queries. In what follows, the word *query* will be used for both address look-ups and data queries.

3. Architecture

The main idea for decoupling the network I/O on the client-side is to pass the responsibility of waiting for the answer of the query or look-up to another entity, dedicated solely to this purpose. This way, the server entity we were talking about in section 2 will no longer be busy waiting for answers from the distant services it has contacted, but will be able to respond to and treat new clients. If the distant services that have been contacted are delaying their answers on purpose, the service we provide, as a whole, will not be affected by them, which is our main purpose, since it gives higher availability to our service.

The mechanism that has been imagined is similar to the DMA (direct memory access) mechanism from the hardware, where you request something from a slower peripheral device, you program the DMA controller to give that device all the information it needs to carry out your request, and then you may do other things in parallel with the transfer, until the DMA controller notifies you that the transfer has finished, through an interrupt signal. In the case of the distributed system, *the processor* is represented by the foreground useful service running on the given node, *the slow external device* is the remote server from whom we are requesting something, while *the DMA controller* will be the auxiliary layer that is represented as a background entity and finally acts as a client for the protocol that the distant server uses. We will furtherly call this entity *the background query executor*.

In the case of a multi-processor or multi-core node, which is very frequent for server machines nowadays, this model will be far more efficient than having just a single

server entity running on top of the node, since the new programming model we are introducing will take advantage of the parallel underlying physical architecture. Of course, we assume that the operating system is aware of the parallel underlying architecture and is able to optimize the resource scheduling accordingly. On single-processor single-core systems, the approach will also be more efficient than just having a same single thread for receiving incoming calls and for querying distant servers, but the impact will not be as visible as in multi-processor or multi-core systems, where the hardware itself enables true processing parallelism. The gain in efficiency even on single-processor single-core machines is due to the fact that in many cases there are things that we may wish to do in parallel with the query, that do not depend on the query's result, that, for example, help an incoming call to be initiated faster. On the other hand, from the point of view of the security threat described earlier, if an interrogated server refuses to answer, or delays its answer very much, on purpose, this model will prevent other calls, that don't have to deal with this malicious distant server, from being affected at all by its DoS attack. If we view the waiting performed for a query as a sleep of indeterminate duration, we are able to do other things on the SIP server while this sleep is performed, which leads to the idea of rendering each query as an asynchronous RPC (Remote Procedure Call), which is described in [8].

An approach we could take, for the design of the background query executor, would be to launch a new thread or process for each new query. For many queries, this will charge the system very much. A better design would be to have a fixed number of workers for the background query executor, which is enough for the system's needs and, at the same time, is within reasonable limits with respect to what the hardware can offer. This approach is also better than launching a new thread for each newly initiated call, since each OS kernel has a limited number of threads that it can handle well, and we wish the number of calls per second that are processed by our SIP server to be much greater than the thread limit imposed by the OS. Making the SIP server's limits more independent of the underlying OS's and hardware's limits is one the purposes of this proposed architecture and also the reason why we have tried to find a better solution than just simple thread-launching.

The elements that compose one individual *background query executor worker* can be seen in figure 1. They are: the requesting component, the listening component and the executing component. Their interaction follows many of the rules of the Observer design pattern [3].

The requesting component is responsible for the formatting of the query such that the background query executor would understand it. To do this, it should implement a special interface, of the decoupling layer. The requesting component interacts with the listening component by passing it the responsibility to schedule the execution of the query and to execute a callback specific to the query after having the query response. In the RPC analogy that we have started, this is the client that will not wait synchronously for the result produced by calling the procedure on the remote server. We can see each query as a procedure that is being called on the interrogated server, since, at this point, we do not care either about the protocol used for interrogating the server, or about the algorithm that the server uses to compute the answer.

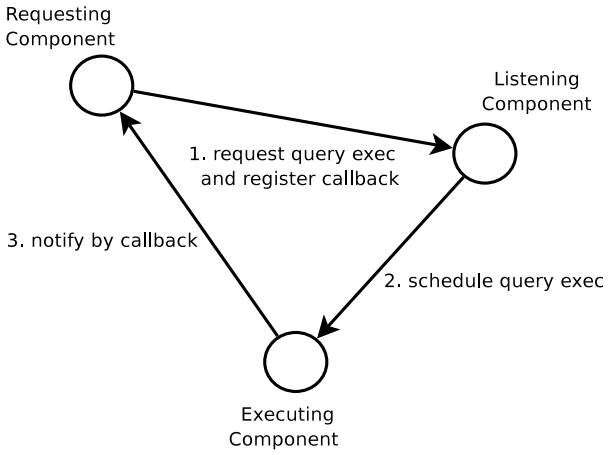


Figure 1. Interaction between the different components of a worker.

The listening component is the core component of the background query executor and is responsible for managing the queries, for scheduling query execution, for keeping some special information related to the queries, if needed, and, last but not least, for merging together queries that are addressed to the same server, for efficiently using the network bandwidth for the message passing. In the RPC analogy that we have started, this is the layer that converts the RPC request in a PDU (Protocol Data Unit) that is specific to the type of server that is being interrogated. All these are optimized for the least number of queries for one server, that is for the least number of queries launched by the background query executor. The listening component interacts both with the requesting component, by receiving new queries from it, and with the executing component, by giving it queries to be executed and by passing it the responsibility of executing the specific callback.

The executing component is responsible for actually con-

tacting the distant server, for executing the query and also for executing the query-related callback afterward. It interacts both with the listening component, which provides the queries to be executed, and with the requesting component, via the registered callback. This component is transparent to the RPC client represented, in our case, by the requesting component.

After seeing the functional description of an individual background query executor worker, you can find a layered illustration of the system architecture that we propose, as a whole, in figure 2. This includes both the pre-existing foreground useful service and the newly introduced background query executor, where both of them are composed by fixed, pre-established, numbers of individual workers, which are not necessarily equal. In the classical approach, on the left of the figure, the only entity that is figured is *the foreground useful service* running on the node. In the newly introduced decoupled network I/O version, on the right of the figure, the entity figured above is *the foreground useful service*, while the entity figured below is *the background query executor*. As stated above, the background query executor could have more than one worker, even if it is a fixed number, and the layered structure of the workers, combined to achieve their common goal, can be seen here. The same is true for the foreground useful service, represented in the figure. It should be mentioned here that each foreground useful service worker can only interact, in our current model, with one background query executor worker. The interaction function defined on the foreground useful service's workers is, thus, surjective even if it is not bijective. One can notice that the main part of the *requesting functional component* is contained by the foreground query executor, and some parts of the *executing functional component* are also contained in the foreground query executor, namely the parts responsible for the callback execution.

The *decoupling layer*, that is represented mainly by the requesting component in the foreground useful service, allows the separation between the server main functionality of the node, which should receive incoming clients, and the client secondary functionality, which should interact with other servers using various communication protocols, via queries. This is achieved by introducing a protocol-independent interface between the two. We believe that this separation is needed because it induces far less interaction and less negative influence between the server functionality, which should promptly treat all of its clients, and the client functionality, which should wait for another server's answer. The SIP protocol, the most widely used signaling protocol in VoIP systems, is a protocol that could benefit very much from this decoupling because, by its nature, SIP allows an entity to act both as a user agent server and as a user agent client, throughout its lifetime [5]. These 2 SIP roles do not imply mainly queries, but it is obvious that the

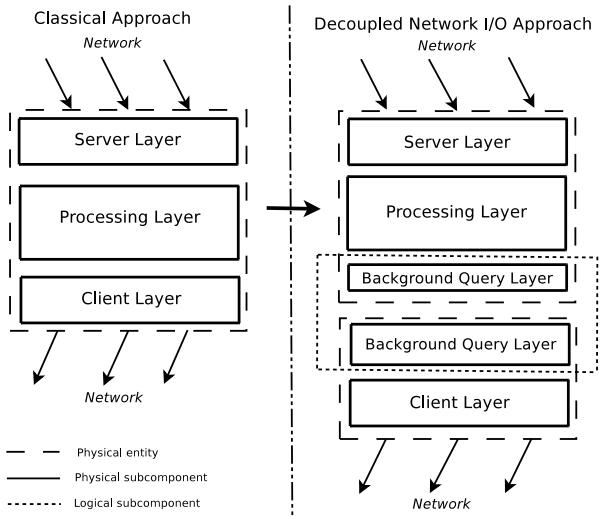


Figure 2. Comparison between system architecture approaches.

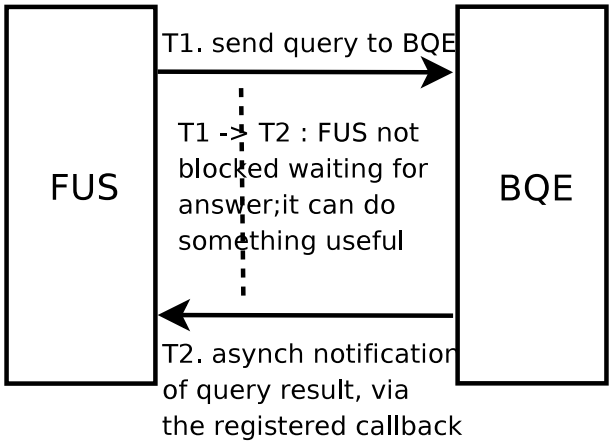
SIP server role will perform fewer interrogations to other servers than the SIP client role.

To continue our analogy with the low-level hardware architecture, the callback notification means, described within the executing component, will be similar to and correspond to the interrupt system of the processor. This is why the background query executor allows the registration of a callback for every query that it needs to execute. You can see how the API function of the background query executor looks like, below. From the RPC point of view, this API is the particular way in which each remote procedure will be called, on the servers that the RPC client is able to contact.

```
register_query(query_text, server_type,
       query_callback);
```

You can also see a schematic description of how the foreground useful service and the background query executor interact and, thus, of how the decoupling mechanism works, in figure 3. One can see that there is no busy waiting between the moment of launching the query and the moment when the foreground useful service is notified about the result of the query, via the registered callback. Instead of the busy waiting, which we have elliminated by the proposed architecture, the SIP server actively continues to prepare the call to be established, or treats another call, if the answer to the query takes very long.

The efficiency of this API, and of the decoupling layer that lies behind it, will be evaluated below, in section 4, by running some tests on the prototype of the background



FUS = Foreground Useful Service
BQE = Backgound Query Executor

Figure 3. Schematic description of the decoupling mechanism.

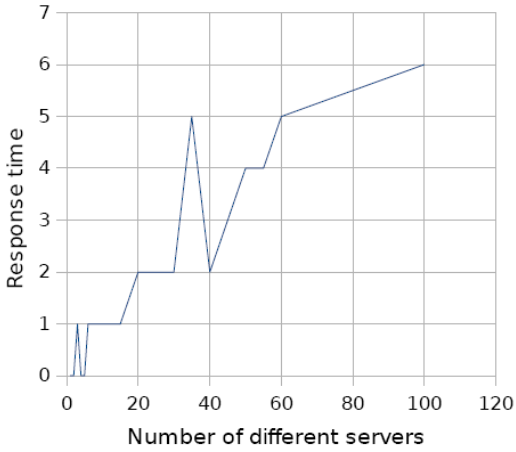


Figure 4. Influence of an increasing number of different servers on the performance of the background query executor.

query executor. The tests that have been conceived and run have the main purpose of validating the idea of decoupling the query execution from the server that needs the query results.

4. Performance evaluation

When it comes to evaluating the performance of this prototype, we have to first identify some correlations between external factors and the prototype, throughout its existence. We have identified two such correlations, namely: *the number of different servers* that should be contacted at a given moment and *the number of identical queries* that should be executed at a given moment. For simplicity, we will consider only DNS look-ups in what follows, since they are easier to launch and it is easier to estimate their duration. For database queries, the duration depends on the degree of difficulty and on the internal scheduling of query execution, besides the network delay and the load of the server, which influences the DNS look-ups too.

The testing setup that we are considering is of having more clients that only contain the requesting layer, described in section 3, and one background query executor worker. For proving the fact that even on a single-processor, single-core machine the described prototype will perform well, we have used such a machine for the tests that have been performed. In the first test, the clients' requests are addressed to different servers and in the second test, the clients' requests are all identical and addressed to the same server. The reader should recall that in section 3 we have described how the background query executor is optimized for the least number of message passing through the physical network.

Increasing number of different servers at a given moment. In figure 4 we can observe that the response time of the prototype varies very little, but linearly, with respect to the number of different servers. One can notice from this figure that the response time for 100 different servers is close to 5ms, while the duration of a DNS query for names such as roedu.net or google.com is between 80-200ms. This means that the degradation introduced by the decoupling layer is about 5%. For less than 10 servers, the response time of the decoupling layer alone is neglectable, being less than 1ms. If we consider that this layer enables a new, parallel programing model, which has been described into more detail in section 3, that allows the foreground server to do other things while its query is being processed, the extra 5% of time spent processing a query is a neglectable price to pay.

Increasing number of identical queries at a given moment . In figure 5 we can observe that the response time of the prototype varies very little and is almost constant with respect to the number of identical queries per one server. This response time never increases over 1ms, which is neglectable, with respect to the time it takes for a query to get executed, as shown earlier. We can conclude, from this experiment, that the number of identical queries per distant server does not influence

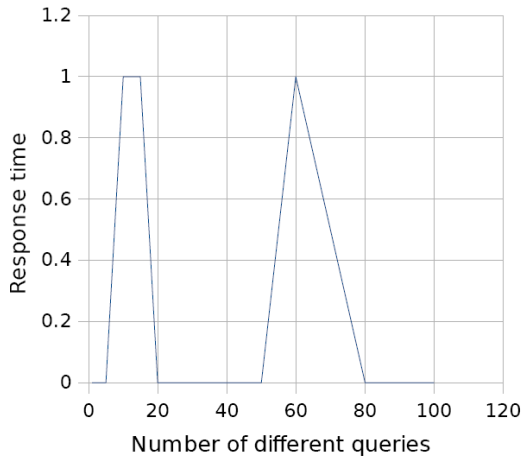


Figure 5. Influence of an increasing number of identical queries on the performance of the background query executor.

the response time of the background query executor, which proves the optimization we have performed, of minimizing as much as possible the number of messages passed through the physical network to the same server.

By combining the results obtained for a variable number of different servers with the results for a variable number of identical queries, we obtain the behaviour of the system in a real environment, where, at a certain moment, we could have a variable number of servers with a variable number of queries for each. It follows from the measurements described here that only the number of servers influences the performance of the decoupling layer, when running on a single-processor, single-core machine. We can imagine that the degradation introduced on a multi-processor or multi-core machine would be much less, if even visible. It could also be benefic for the load of the distant server, if more of its clients would contain this feature.

5. Conclusion

In the previous sections we have seen the description of a decoupling layer within servers, that separates them into a foreground service and a background query executor, in an asynchronous RPC manner. The two interact via a simple interface, which gives functionalities similar to the DMA in hardware systems. We have evaluated the performances and the overhead introduced by this decoupling layer and we have seen that it performs well, the introduced overhead being insignificant when compared to the duration of the actual query. Also, we have seen that the architecture that we propose minimizes and sometimes even prevents the risks of some DoS security threats.

In addition to previous work, which focuses mostly on server-level or network-level optimizations, or on particular distributed systems, we have tried to be generic in our studies, even if we were mainly focusing on the needs of VoIP and especially SIP distributed systems. Because in VoIP systems nodes need to interact a lot, using various protocols, the generality of the query was taken into account and we addressed different types of queries, both DNS queries and database queries.

In the next version of our prototype, to improve performance results, some aspects could be optimized following some directions: actual query execution, even more low-level architecture awareness, and specific VoIP behaviour optimizations. We should also measure how the prototype performs on a multi-processor or multi-core architecture in a real-life environment. Moreover, since we are aiming to address multiple types of queries, a generic query description language and query response description language would be very useful to improve the asynchronous RPC-like part, that would help the background query executor to be even more query-type independent.

References

[1] H. A. Alzoubi, M. Rabinovich, and O. Spatscheck. MyXDNS: A request routing DNS server with decoupled server selection. In *Proceedings of the 16th international conference on World Wide Web*, pages 351–360, 2007.

[2] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *Proceedings of the 18th International Conference on Data Engineering*, pages 716–732, 2002.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2002.

[4] J. Rosenberg and H. Schulzrinne. Session initiation protocol (SIP): locating SIP servers. RFC 3263, Internet Engineering Task Force, June 2002.

[5] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[6] K. Singh and H. Schulzrinne. Peer-to-peer internet telephony using SIP. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 63–68, 2005.

[7] H. Sinnreich and A. B. Johnson. *Internet Communications using SIP. Delivering VoIP and Multimedia Services with Session Initiation Protocol*. Wiley, 2006.

[8] R. Srinivasan. RPC: remote procedure call protocol specification version 2. RFC 1831, Internet Engineering Task Force, Aug. 1995.

[9] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.
[10] E. Wilde. *Wilde's WWW*. Springer-Verlag, 1998.
[11] GRelC data access service. http://grelc.unile.it/.