A comparative performance evaluation of different implementations of the SOAP protocol

José A. García, Roi Blanco, Antonio Blanco and Javier París
Department of Computer Science
University A Coruña
A Coruña, Spain
Email: {josegarcia, rblanco, blanco, javierparis}@udc.es

***Abstract*—This paper presents a study evaluation of the SOAP [1] protocol performance between two different implementations: Java (Axis2) [2] and Erlang. This comparison has been carried out using several testbeds with input and output data of different sizes. More concretely, we developed three different web services representing typical scenarios likely to be found in real environments. The evaluation is two-fold: we measured both the number of requests per second answered (throughput) by each server and the response to a common server workload, mixing stress and stand-by phases. The Erlang [3] functional programming language claims to be especifically designed and suited for *distributed, reliable and soft real-time concurrent systems*. Morever, its built-in lightweight processes management and easeness of replication within distributed environments stand out Erlang as an appealing choice for service oriented architectures (SOAs) [4]. On the other hand, we compared this new approximation with the well-known Apache Axis2 project, as it is widely employed on the Web Services field by the Java community. This work allows us to conclude that the Erlang server is more suitable when the computational cost of the web service is low, whereas the Axis2 server is more efficient as the service workload increases.**

I. Introduction

Many companies need to offer interoperable services to their customers. The use of SOA architectures provides a solution for this problem by means of a standard protocol called SOAP. The main goal of this work is to assess the viability of using a SOAP server developed in Erlang to communicate heterogeneus applications. Achieving an adequate performance at this layer of the SOA architecture would allow the transparent use of Erlang for higher layers. The reason is that many of the requirements of SOA, like process replication or fault tolerance are built-in into Erlang.

SOA provides a high number of specifications dealing with several problems like the orchestration of registered services, security issues and the quality of the deployed services. The quality of these services and their coordination are two of the main points in this work. Moreover, this work tries to provide a new approach to the construction of SOA architectures by supporting the use of the functional programming language Erlang.

To perform the evaluations, we selected several web services that comprise inputs and outputs of a service in a real environment. Additionally, we chose several different requests for each service to add a degree of variability in the requests launched by the cluster of clients.

Partly supported by MEC TIN2005-08986

A. The Simple Object Access Protocol

SOAP can be coarsely described as a messaging protocol for web services. It provides a communication protocol used to access different web services trough a loosely coupled infrastructure that provides scalability and flexibility using different implementation technologies and network transports.

The SOAP protocol allows the interoperability among different systems by providing a standard communication channel. Many of the new desktop applications, embedded systems or PDA applications need this protocol to comunicate in an homogeneus way.

B. Erlang

Erlang is a concurrent programming language and runtime system that provides a virtual machine and several libraries. It was designed by Ericsson to develop distributed, fault tolerant, soft real time, non stop applications. It supports hot swapping so code can be changed without stopping the system.

Concurrency is implicit in Erlang. The use of processess as a basic abstraction is due to the design of Erlang as a language for the development of fault tolerant systems.

Erlang has not constructors inducing side effects to an implicit store with the exception of communications among threads (processes, in Erlang terminology). With Erlang's primitives for concurrency, it resembles formal calculi such as Milner's CCS[5] or Hoare's CSP[6].

In Erlang, new threads can be created with the primitive *spawn*. Once evaluated, it returns the *process identifier* (PID), of the newly created lightweight process. In other to allow interaction among processes, a couple of asynchronous message passing primitives are available:

- Asynchronous send:

  ```
  Pid ! Msg
  ```

  Msg is sent to process Pid without blocking the sending process. If Pid exists, the message is stored in Pid's *mailbox*. Any valid Erlang value can be sent to other processes.

- Mailbox pattern matching:
  ```
  receive
                Pat1 -> Expr1;
                  ...
                PatM -> ExprM
  end.
  ```
  It searches the process mailbox looking for a message that matches one of the patterns $Pat_1, ..., Pat_M$ sequentially. If no such message exists, the process blocks until it arrives. The result is the evaluation of $Expr_i$ with the bindings carried out in Pat_i.

The use of Erlang in this work is based on observations and results of previous works from different researchers [7] and in our own experience using the Java platform to develop applications using the SOA architecture [8].

II. Previous Works

Support for the SOAP Protocol in Erlang has been developing in recent years. Much of this work has been done by the Erlang community without official support. This interest has been fueled by the need for a simple and interoperable way to communicate different commercial applications, which SOAP provides.

The first approach studied in this work is the Xmerl project[9]. Xmerl is a library included within the Erlang/OTP package, with a complex lexical analyzer which can be used to work with XML documents. One of the drawbacks of Xmerl is that it lacks support for XML schemas. The SOAP protocol uses a standard XML schema to define all the possible components of a request. For that reason, providing SOAP support using Xmerl is difficult because all the request must be processed just to know if its structure conforms to the SOAP standard. This makes Xmerl inadequate to use with SOAP.

Another interesting approach is the *erlsoap 0.3.x*[10] project. Erlsoap is a library developed by Erik Reitsma in 2002. It works by dividing each incoming request into more fine grained components using Xmerl. Its main drawbacks are the use of the Xmerl library, which as was explained before, lacks support for XML schemas. Furthermore, some of the data types in the SOAP specification are not supported by this library. Despite all these drawbacks, the erlsoap project is among the first approaches that trie to provide SOAP support in Erlang from a global point of view. Therefore, erlsoap is not an ad-hoc solution to a specific situation but a full aproximation to the developement of a SOAP server.

Nowadays the *Erlsom* project[11] developed by Wilem de Jong allows for the use of SOAP and other specifications thanks to its support for XML schemas. This library includes a lexical analyzer which provides a representation of the XML schema which can be easily used in Erlang.

On the other hand, most previous works on SOAP performance deal with the comparison of different available implementations. For example, the work of Dan Davis and Manish Parashar [12] makes a interesting comparison of the latency of many different implementations. In this works, the authors measure the latency of each different implementation trying to detect the most inefficient scenarios of the SOAP protocol. In our work we do not try to determine which the most innefficient scenarios are but rather to empirically check if our Erlang server is able to provide comparable performance to a commercial implementation.

In [13], Florian Rosenberg defines a set of web services to compare without knowing their implementation. This approach is not used in our work, because the knowledge about the implementation of the web service can be used to know its behaviour beforehand.

III. SOAP Server Architecture

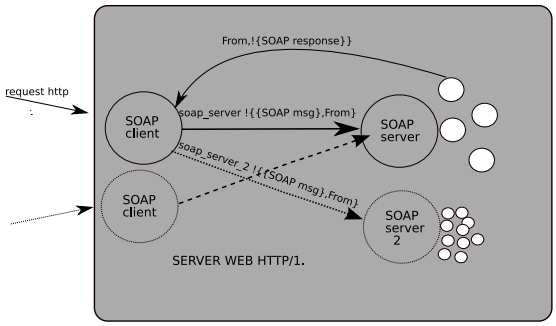

Fig. 1. SOAP server in Erlang

Figure 1 shows the structure of our Erlang server. For every HTTP request, the server creates a process (*soap_customer*) responsible of the communication with the Erlang server process (*soap_server*). Furthermore, the figure also presents the different kind of interactions occurring inside the Erlang server whenever a SOAP request arrives.

Once the erlang client process ¡oalgo¿ the server stands by waiting for the answer to come. Every request made by a client process is assigned a new lightweight process in the server, in charge of processing the incoming SOAP request. This way, that new process is in charge of answering to the client process, thus the server remains available longer.

Another advantageous consequence of the Erlang server design would be that it leaves open the possibility of creating the new processes on a different machine. Hence, the memory and CPU load of the server can be freed at any moment. The reply to clients will be made by the new lightweight process created by the Erlang server following the behaviour shown in figure 1.

In [14] Armstrong presents a comparative study showing the goodness of Erlang inter-process communication with respect to Java or C#. That study also reveals the fact that when several Erlang processes have a high workload, the performance of the server falls-out quickly. We also confirmed empirically this fact, (figure 10). In our Erlang server there is a performance drop because the system is not able to process all the incoming requests and it starts to accumulate them in the server mailbox.

SOAP server replication is almost inmediate, as it only involves the creation of another soap_server process (be it in

the same or in a different machine), and notify the clients so they redirect their messages to this new process instead to the old one (figure 1).

Fault tolerance is built in the language, and defined based on a supervision tree. This is a process structuring model based on the idea of workers and supervisors. Workers are processes which perform computations, that is, they do the actual work. Supervisors are processes which monitor the behaviour of workers. A supervisor tree is a hierarchical arrangement of code into supervisors and workers, making it possible to design and program fault-tolerant software. In this case, the supervisor is in charge of starting the SOAP server up again in case of a system failure. It is possible to define and enable different acting policies in case of a general failure. As it can be seen in figure 2, the different Erlang processes, in case of being more than one, are observed as well by a root proccess that assesses the monitors the behaviour of the system and manages any fails thay may occur.

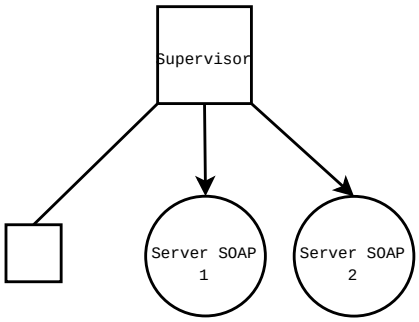

Fig. 2. Supervisor

IV. Experiments and results

A. Data Set

The data set we employed in this paper comes from the implementation of three different use cases described in [15]. They can be used to assess and conform a significative set of the possible interactions supported by a web services framework.

The first web service designed and deployed is a credit card service, where the input and output data are small. In this particular case, the output is restricted to a boolean value indicating whether the operation was successful or not. The input values are a reduced simple dataset representing the typical values in a bank-account operation, like the account number, user id., etc.

The second web service proposed falls into the small data input - big data output class. Concretely, the service modelled is a web news server. It is fed some input parameters, like day range and information sources, and retrieves the headlines from each one of the information sources.

Finally, the last test web service proposed is an online shop. It allows for the lookup of a reference for several providers by introducing a product characteristics, like product key, property list, expiration date, etc. This kind of service has a medium input and output data. The three examples presented encompass most of the possible variants one can expect from a web service. On one hand, there is a small web service (in terms of its input) with a boolean ouput value. The second service is a classical example of an RSS input [16] that allows for the retrieval of a big amount of information based on a small query like a string of text. Lastly, the service with moderate input and output is considered as the most common web service in a real environment.

B. Load test

This study established two different workload sets that will allow for evaluating easily the performance of each one of the servers. The sets embody different client request ratio over time, presented in the figures below.

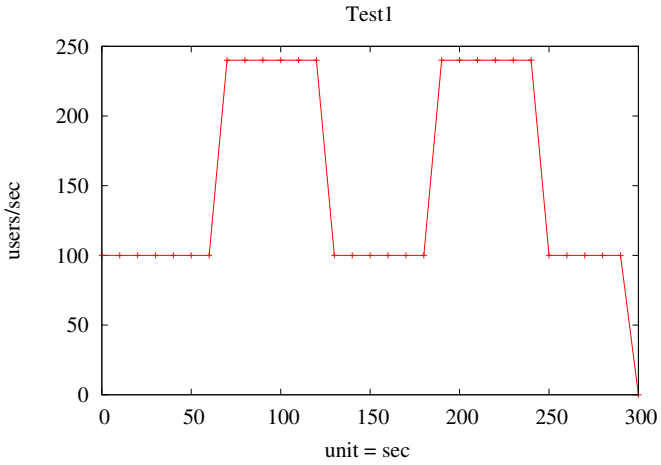

Fig. 3. Load test(test1)

Figure 3 shows the first request scheme over time. We have established a five minutes workload test where new petitions are requested following the pattern presented in the figure. For example, during the first phase of the test, the client cluster will perform a request every 0.01 seconds to the server during a lapse of one minute. It follows an stress phase, where the server will have to handle 240 users per second during one minute as well. This process will take place once again and the test will end with a non-stress phase so it may finish all the client cluster requests open.

This test aims at collating stress and repose phases, in order to find out what the performance of both servers in a *real environment* would be, where the ratio request variance is quite high.

The second test (figure 4) establishes ten phases, each one of them running for 30 seconds. This test increases progressively the number of requests, starting in 100 request/second until the final phase is reached, with 1000 requests/second. The main goal of this test is to try to detect the saturation point and the effective throughput of both servers.

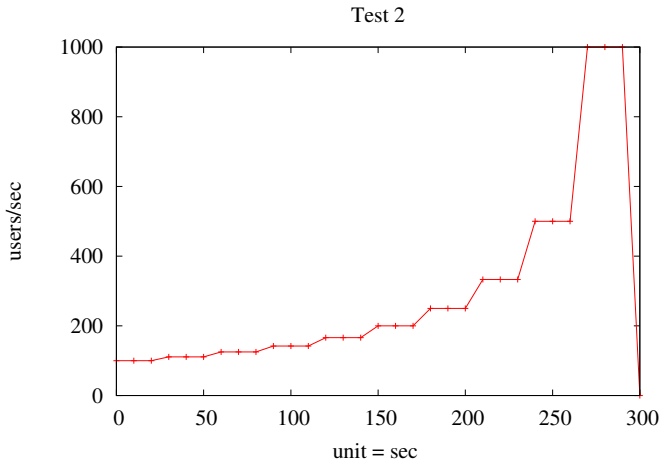

Fig. 4. Load test (test2)

C. Execution environment

We present now the configuration of the environment in which the experiments were run (network and machines).
Using an Internet connection undoubtely introduces a great number of parameters (like network latency, routing time), that would possibly distort the obtained results. This is the reason we opted to employ three machines connected through a LAN, with a 100MB commuted Ethernet. This approach is more likely to yield accurrate results than a global-based (Internet) communication within the client cluster and the server.
Over this network topology we will set up two machines playing the client role; this will be the client cluster in this study. The third of the machines will act as the SOAP server (both Axis2 and Erlang).

	client 1	client 2	server
CPU (MHz)	1700	2800	1800
Cache size(KB)	256	512	128
RAM Memory(MB)	512	1024	1024

Fig. 5. Characteristics of the three machines

As well as topological specifications, we introduced some parameters in order to optimize the performance of both servers. For instance, in the Axis2 server we increased the thread pool size up to 150 threads, and the thread alive time (*threadKeepAliveTime*) for every process in the pool. The number of processes the Erlang server is able to create has been increased up to 500000 and in both servers we established a HTTP connection timeout of 20 seconds.

D. Results: load test (test1)

Table 6 shows the results obtained after executing the first workload test described, over the three web sevices scenarios.

	OsService		CardService		NewsService	
	Erlang	Axis2	Erlang	Axis2	Erlang	Axis2
http-200(Highest Rate)	165.2	230	229.2	234.4	32.4	234.1
http-200 (Total)	39987	37479	37818	38061	11910	38973
Error (timeout)	70	0	0	0	28410	0
Size_sent (MB)	273.62	256.85	26.76	27.30	18.76	57.24
Size_rcv (MB)	27.11	22.77	13.74	11.36	327.46	1078.15

Fig. 6. Test 1 results

The first row of the table shows the highest number of successfully answered request per second. The second row stands for the total number of answers the server was able to answer sucessfully within the execution time established for the test. Next row shows the number of errors in each server. Finally, the last two rows of the table indicate the total size in megabytes of the input and output messages, respectively, in each server.
Differences obtained among the servers in the first test are not significative and therefore none of the technologies shall be completely ruled out. In the online shop web service *OsService*, the one that has a moderate input and output, differences are not very significative as well. However, the number of successful requests answered is higher for the Erlang case (> 2000). This scenario represents the prototype of a standard web service, web input and ouput are medium-grained and thus, conclusions may be directly transferable to a generic web service.

The bottleneck observed in the Erlang server (fig.7) is mostly due to the increase of the computational workload of the process, and the associated serialization and deseralization cost of the SOAP message. As it is explained in section III, each time a client asks for a request, the Erlang server creates a process in onder to answer it. If the wait time of this process is high, every eventual clients that might arrive to the server next stay idle in a wait state, forcing the server to handle every queued waiting process and those under execution. Also, the WSDL document associated to news service (and that may be found at [17]) shows that the message structure is more complex than in the other services. This is the reason why the serialization and deserialization times are higher, as the parser in the Erlang server is not as optimized as the Axis2 parser.

The Axis2 parser, called AXIs Object Model (AXIOM) [18], is *an XML object model designed to improve both memory use and performance during XML processing and is based on pull parsing. By using the Streaming API for XML (StAX) pull parser, AXIOM (also referred to as OM) can control the parsing process to provide deferred building support. Deferred building is the ability of AXIOM to partially build the object model while the rest of the model is built based the user's needs.*

However, for the Erlang case, the client request is parsed on its entirety therefore the longer client answer times.

Figure 7 presents a comparison of the results obtained after

executing the first test on the news web service scenario. The upper-left graph show the number of answers obtained in the client cluster. It may be noticed that during the first phase of test 1 (figure 3), the Axis2 server is able to anwer correctly every request sent. During the second phase of the test, the number of clients accessing the web service increases over time.

On the other hand, the Erlang server is only able to answer 30 requests/second. The lower-left graph states the fact that as the test phases advance, the clients are queued in the server in order for their requests to be attended. This is the reason for the Erlang server to overload at the beginning of the test. The bottom-right graph shows that the server discards requests from the client cluster. It is worth pointing out that the timeout graph (the lower one) agrees with the different phases established in the workload test (figure 3). As a consequence of the client enqueing situation, some requests time-out and consecuently errors start taking place (figure to the right)

Results obtained after executing the first test over the credit card web service show slight non-signiticative differences between servers. This behaviour is plotted in figure 8. In this case, a noteworthy point of the process is that the system is flawless, mostly due to the low computational execution costs of the service, and the small boolean output. This is the reason why the system is not overloaded though the whole test, and the behaviour of both servers is very similar.

Figure 9 presents the results for the online store web service. The upper left graph, just like the other two services, shows the number of successfully answered requests measured in the client cluster. As in the news web service, the Erlang server is overloaded during the first stress phase, and thus in the first non-stress phase (120-180 seconds) the server is still answering requests from the previous one. Something else to consider is, like in the news case, the number of clients connected simultaneoulsy to the server. The users connect to the Erlang server during the first stress phase (phase two of the test) and are queued in order for their requests to be processed by the server. Once the server handles its request, it creates a new process in charge of managing the domain-logic offered by the web service and answering the client, leaving the server idle to keep on answering requests. The number of errors (timeout) shown in the botom-rigth figure are not significative for the global test.

E. Results: load test (test2)

As it is commented in section IV-B this test tries to measure the behaviour of both servers in the case of increasing the client workload over time.

Results presented in table 11 allow us to conclude that the news web service presents a bad behaviour under this stress workload schema execution. It is worth pointing out that the number of errors is up to 62.000 requests; this value is higher than the one obtained in the first test (around 28.000) and

	OsService		CardService		NewsService	
	Erlang	Axis2	Erlang	Axis2	Erlang	Axis2
http-200(Highest Rate)	166,5	315,9	689,5	486,3	30,1	236
http-200 (Total)	63371	78699	72518	79378	17135	76970
Error (timeout)	18000	800	0	0	62000	3375
Size_sent (MB)	437,07	539,33	51,22	56,93	27,28	113,04
Size_rcv (MB)	42,97	53,59	26,35	27,98	471,62	2130,30

Fig. 11. Test 2 Results

goes accordingly with those results (the number of clients per second is higher). The perfomance of the Erlang server in the credit card service is remarkable. In this case, the computational workload and the serialization and deserialization processes are low. Therefore, it is possible to conclude that the Erlang server is able to create and destroy efficiently a higher number of processes than the Axis2 server.

Finally, the online store web service has a good performance in both servers. The Axis2 server has a number of errors (timeout) not significative with respect to those observed in the Erlang server.

In figure 12, it can be seen that the Erlang server handles over 600 succesful replies per second in the final stage of the test. This performance is much higher than in Axis2, which serves 400 per second. The main reason for this numbers can be seen in the lower graph, which shows the number of simultaneously connected users to each server. In this case, the Erlang server has around 600 simultaneously connected users, while the Axis2 server has 3500. This good performance show that the Erlang virtual machine is very good at creating and destroying large numbers of processes. The top right graph in figure 12, shows that all arriving requests to the Erlang server are processed inmediately. This does not happen with the Axis2 server, whose behaviour is irregular in the final part of the test. This web service has a much lower computational load than the previous ones, and so it behaves like a ping service that can be used to measure the latency between the cluster of clients and the server.

In figure 13 it can be seen that the trend observed in *test1* can also be seen in *test2*. The lower graph shows that the Erlang server reaches peak performance in the 100th second of the test. From that moment the Erlang server will process 150 requests per second until the end of the test.

On the other hand, the Axis2 server peaks at the 200th second, when the number of users waiting increases. In that moment, the load of the server is 100%, which means that the server has no spare capacity for new requests, and the new arriving ones have to wait.

V. Optimization issues in Erlang server

This experiment showed that the Axis2 server has better performance due to having a very efficient processing model called AXIOM (Axis Object Model).

To try to reduce the problem in the Erlang server, we have tried a simple approach. The Erlang server does not create new processes when the request is the same as a previous one, that is, the server creates a response cache to reduce the number

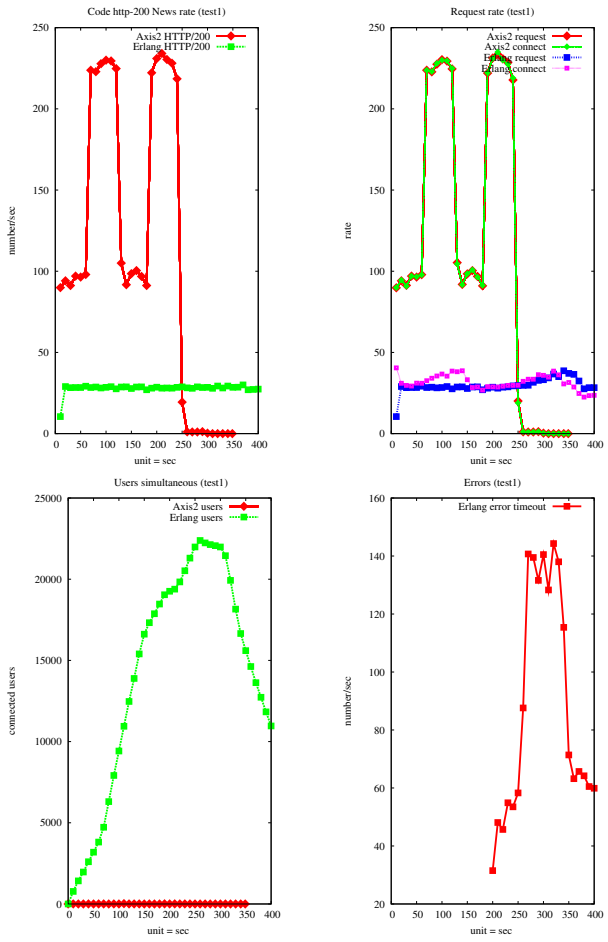

Fig. 7. News Service test1

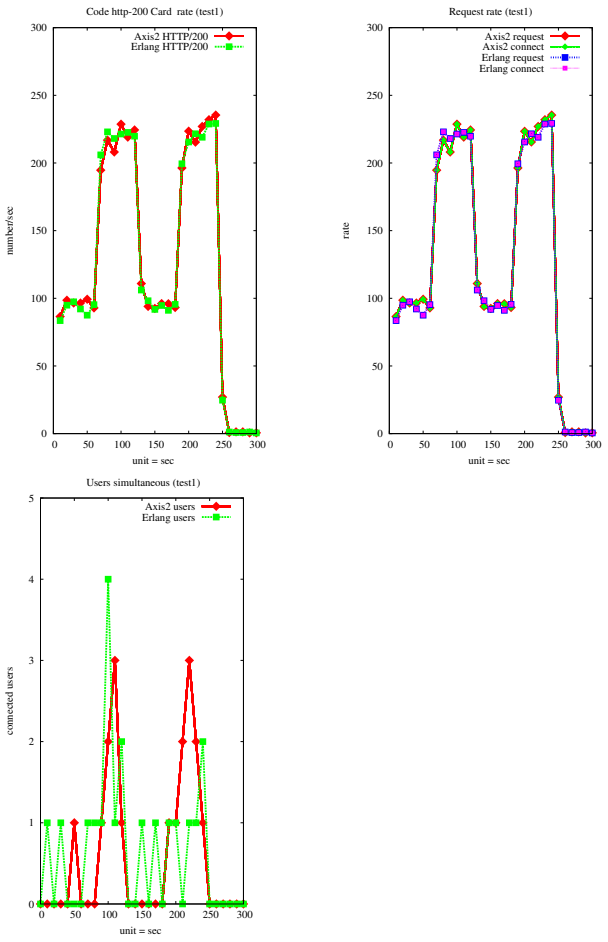

Fig. 8. Card Service test1

of processes created and prevent an explosive growth. This optimization also shows an efficiency problem in the parser that processes the XML requests in Erlang.

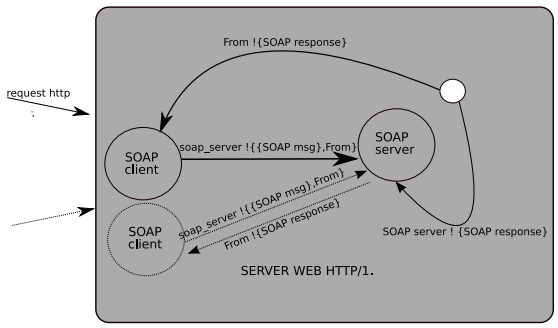

Fig. 14. Erlang server (Cache)

As it can be seen in figure 14, in the first request of a set of identical ones a process is created to reply to the client. This server adds the reply to the cache so that new incoming requests which are identical do not create new processes, and thus prevent a performance decrease due to the saturation of the server as in the news web server.

When a new client of the web service sends a request the server first checks if there is an indentical one in the caché. If there is none, the server creates a new process following the approach shown in figure 1. On the other hand, if the request has already been made before, the server itself will retrieve the reply from the cache and send it to the client. As a result the cache reduces the number of concurrent processes which helps the server achieve better performance.

In order to see if this change delivers better performance than the previous one, we test both the medium load case (web service for an online store) and the high load case (news web service). The test used is *test2* (figure 4) which increases the number of new clients as the tests goes on.

As it can be seen in the results, the new approach (figures 15 and 16) provides an increase in performance(both reducing the number of errors and increasing the number of correct answers) in the news web service, which was one of the worst cases for the Erlang server. However, the server peaks at around 150 clients per second. The main reason for this limit is not in the processing to generate the replies as most of them are already in the cache, but rather in the use of an inefficient serialization-deserialization process. The performance hit is bigger for WSDL documents with a complex structure, as

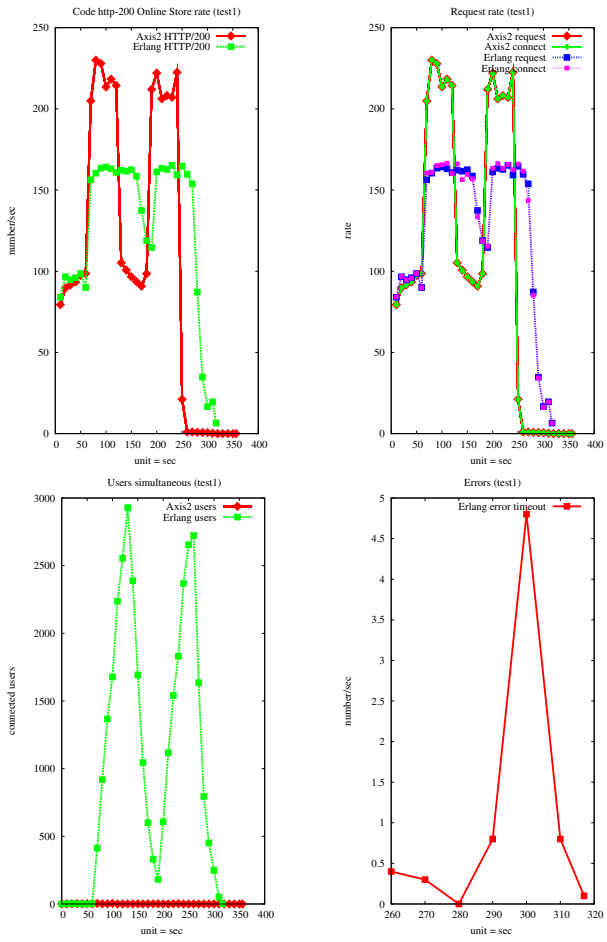

Fig. 9. Online Store service test1

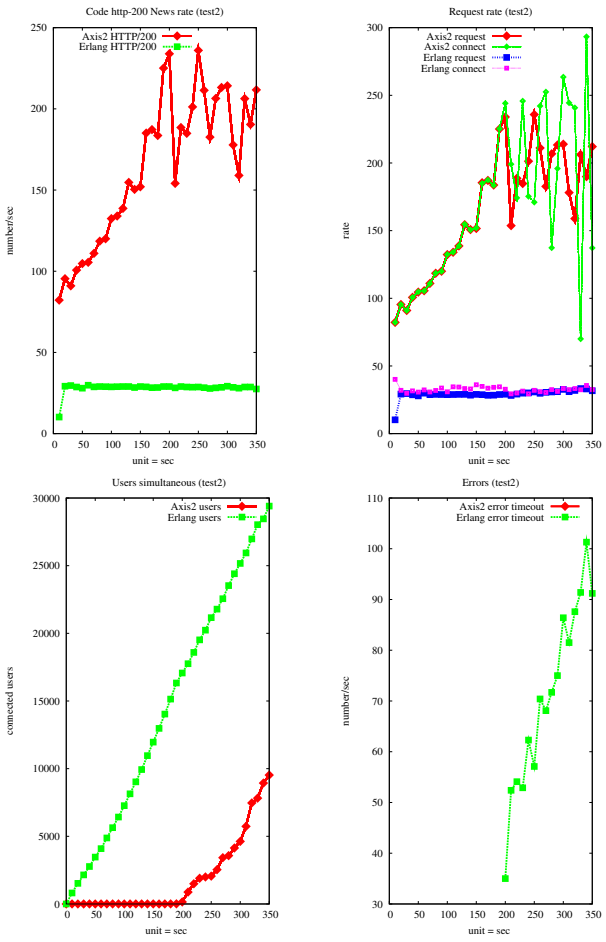

Fig. 10. News service test2

in the news web service. In this case, the mean serialization time for the reply to the client is 25ms, which is too high to prevent an accumulation of clients when the client arrival rate is high (as in *test2*). Due to this, in figure 7 the Erlang server peaks at 30 replies per second.

For the online store service the tests shows that the number of served requests in the last stage of the test by the Erlang server is much larger than the Axis2 server. the Erlang server peaks at 700 requests per second, while the Axis2 server only servers 200 requests per second. The higher performance of the Erlang server is due to the low computational load of this example, and that Erlang is better suited for managing a large number of concurrent processes. In addition, the Axis2 server had several timeouts which are not important for the result of the tests.

VI. Conclusions

The work presented in this paper shows that the performance of the Erlang server is good when the computational load of the services provided is low. The reason is that the running time of the process that creates the reply is low, and the parser inefficiency is not so important because the structure of the WSDL document is simple.
Likewise, the Erlang server has an adequate performance when the load of the service is moderate. Again, this is due to a not very complex WSDL document and a moderate running time for generating the reply. The figure 9 shows that the performance for this kind of service is similar in both servers.

However, when the computacional load of the service is high and the structure defined by the WSDL document is complex, the performance of the Erlang server is low because our server does not have an efficient parser. The parser needs an average of 25ms to build the SOAP reply for a complex request. This time limits the maximum number of requests that can be served in a second to 40, which is the reason for the performance problems shown in figure7.

VII. Future work

In the near future, we plan to test the Erlsom partes with different WSDL document structures looking for inefficiencies in the project. This will lead to performance improvements in the worst part of our server: the parser.
Another research line is developing a communication layer for processes using the BPEL [19] specification included in

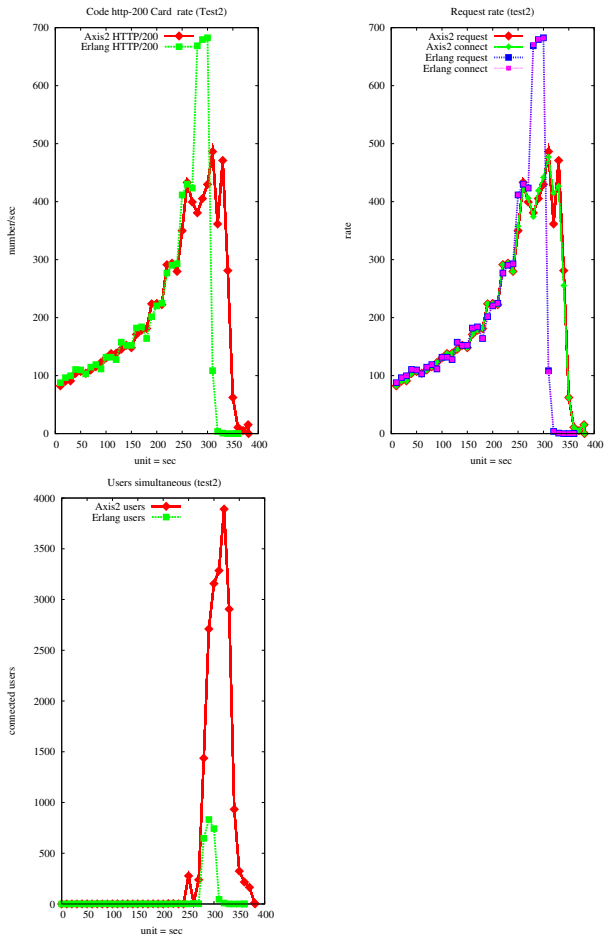

Fig. 12. Card Service test2

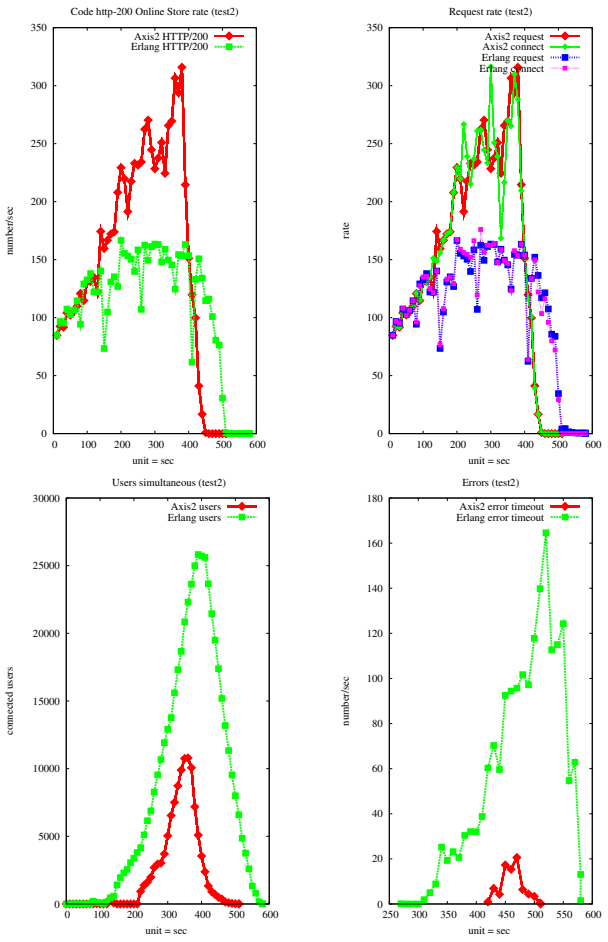

Fig. 13. OnlineStoreService test2

the SOA architecture, moving the implementation layer into Erlang.

REFERENCES

[1] N. Mitra, "SOAP version 1.2 part 0: Primer," W3C, W3C Recommendation, June 2003.

[2] Axis2 homepage. [Online]. Available: http://ws.apache.org/axis2/

[3] Erlang homepage. [Online]. Available: http://www.erlang.org

[4] M. E. J. Ang and A. Arsanjani., "Patterns: Service-oriented architecture and web services." Tech. Rep., 2004.

[5] R.Milner, *A Calculus for communication processes*. Stinger Verlag, 1980.

[6] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[7] M. N. J Halén, R Karlsson, "Performance measurements of threads in java and processes in erlang," Tech. Rep., November 1998. [Online]. Available: http://www.sics.se/ joe/ericsson/du98024.html

[8] J. A. García, A. Blanco, and R. Blanco, "Incorporating applications to a service oriented architecture," in *Proceedings of 5th WSEAS International Conference on SYSTEM SCIENCE and SIMULATION in ENGINEERING (ICOSSE06) December 16 - 18, 2006*, December 2006, pp. 401–407.

[9] U. Wiger, "Xmerl - interfacing xml and erlang," in *Proceedings of Sixth International Erlang/OTP User Conference*, 2000.

[10] Erlsoap homepage. [Online]. Available: http://forum.trapexit.org/viewtopic.php?t=6331

[11] Erlsom homepage. [Online]. Available: http://sourceforge.net/projects/erlsom

[12] D. Davis and M. P. Parashar, "Latency performance of soap implementations," *ccgrid*, vol. 0, p. 407, 2002.

[13] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping performance and dependability attributes ofweb services," in *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 205–212.

[14] J. Armstrong. Concurrency oriented programming in erlang. [Online]. Available: http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf

[15] N. Wickramage and S. Weerawarana, "A benchmark for web service frameworks." in *IEEE SCC*, 2005, pp. 233–242.

[16] "Rss 2.0 specification," 2006. [Online]. Available: http://www.rssboard.org/rss-specification

[17] N. Wickramage and S. Weerawarana. A benchmark for web service frameworks. [Online]. Available: http://www.cse.mrt.ac.lk/ narada/

[18] Apache axiom. [Online]. Available: http://ws.apache.org/commons/axiom/index.html

[19] P. Wohed, W. M. van der Aalst, M. Dumas, and A. H. ter Hofstede, "Pattern based analysis of bpel4ws," 2002. [Online]. Available: citeseer.ist.psu.edu/556822.html

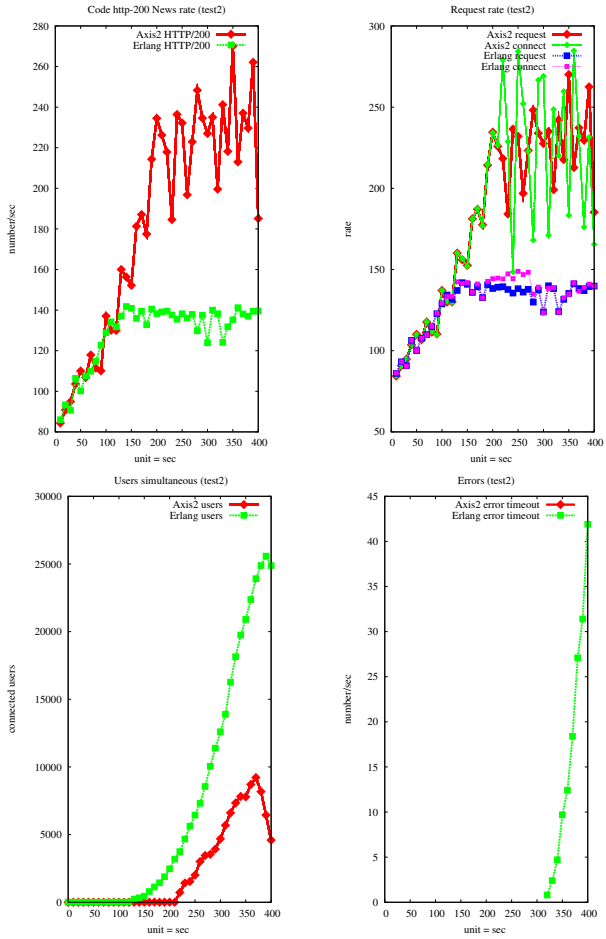

Fig. 15. News Service test2

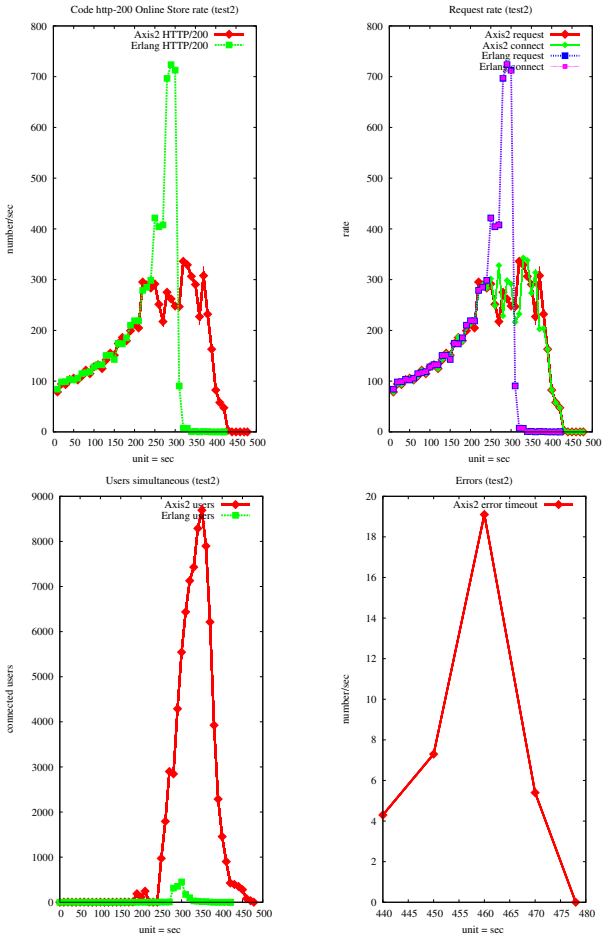

Fig. 16. Online StoreService test2