Client-Based Adaptive Load Balancing in Service-Oriented Systems

Jan Küting[1] Helmut Dispert[1] Joseph Morgan[2]
Jay Porter[2] George Wright[2]

[1]Kiel University of Applied Sciences, Germany
Faculty of Computer Science and Electrical Engineering
jan.kueting@student.fh-kiel.de, helmut.dispert@fh-kiel.de

[2]Texas A&M University, Texas U.S.A
Department of Engineering Technology and Industrial Distribution
{morganj, porter, wright}@entc.tamu.edu

ABSTRACT
We consider the problem of sender-initiated adaptive load balancing in service-oriented distributed systems. We investigate an approach that is based on local observation of response-times. Since in this system the balancing decision is not based on remote state observation and information exchange, the approach allows to decouple the load balancing mechanism from any concrete technology platform. First tests and comparisons with shortest queue scheduling are promising. The results indicate that response-time-based load balancing performs surprisingly well under circumstances where the variation of the observed response-time is not caused by variation in message size or request type.

Categories and Subject Descriptors
C.2.0 [**Computer-Communication Networks**]: Network Architecture and Design, Distributed Systems

General Terms
Design, Experimentation, Measurement, Performance

Keywords
Client-Based, Adaptive, Load, Balancing, Sharing, Response-Time

1. INTRODUCTION
"One of the most important potential benefits of loosely-coupled distributed systems is in the area of resource sharing." [3]. The sender distributes requests to different network nodes or a receiving node gives requests away to other nodes. In either way, the goal is to spread requests over multiple nodes in the system. An important distributed system is a system that has been designed with service-oriented principals in mind. Middleware-based approaches such as Enterprise Service Bus (ESB) and Business Process Management Suite (BPMS) are playing a growing role in the industry [5].

Under the assumption that the role of load balancing will play a growing role in providing massively scalability of services, this paper uses the concept of a virtual service which is based on today's common standards. A virtual service consists of multiple service instances. Load is then balanced between those instances. Any consumer can access a virtual service transparently without having the knowledge about load balancing is taking place under the cover. The concept of service virtualization has a much broader sense, which includes unified deployment and management capabilities. Those topics are not covered by this work.

Since our goal is to provide a fully transparent approach which does not require any changes to a service's implementation, grounding balancing decisions on information that can only be provided by such an implementation is questionable. If the approach would be based on the queue length for example, any service instance would have to provide monitoring and information exchange capabilities. Or as Othman et al. put it: "To achieve this level of communication, application servers must be programmed to accept load balancing requests (as well as client requests) from the adaptive load balancing service. However, most distributed applications are not designed with this ability, nor should they necessarily be designed with that ability in mind since it complicates the responsibilities of application developers" [10].

We propose to base the load-balancing decision on local observation exclusively. Any service consumer can observe the response-time by its own. Therefore the approach decouples sender and receiver because no balancing-related information has to be exchanged between the communication partners.

The main question we will address in this paper is whether or not an adaptive load balancing approach whose balancing decision is based upon locally observed response-time is capable of delivering similar results as a balancing scheme that is based upon a load-index such as the queue-length as observed by a remote node. Further we investigate the circumstances in which the *Response-Time-Based* approach might perform well or not well.

The investigation is based on simulation and covers differ-

ent scenarios like variation in request type and other setups where clients and service instances are located in different networks. In addition, we present a framework to perform load balancing transparently, which is based on *Web Services*. But we could have chosen any other technology instead, for example Representational State Transfer (REST) [4]. In addition, the approach is grounded on the Domain Name System (DNS) to realize service virtualization and late binding which is associated with it.

2. RELATED WORK

Load balancing is a wide research topic and has been studied intensively over the last decades. Middleware based approaches have been studied for example by Othman et al. [10] and Putrycz [11].

Load balancing for large-scale networks has mainly been applied on web access or on lower protocol levels by using specific hardware to spread internet traffic to multiple hosts.

Decision making which is based on estimating the response-time has been studied, for example by Ferrari and Zhou [3], and Banawan et al. [2], but to the author's knowledge, an experimental study of load balancing which is purely based on local observation of the response-time has never been given.

3. BALANCING APPROACH

We propose a client-based (sender-initiated) load balancing approach that is based on the locally observed response-time.

3.1 Late Binding and Address Translation

Since a client must not know the physical endpoint address of any service instance, we need a procedure to address the virtual service as a whole - without naming any particular instance. We further need to translate this address to a physical endpoint dynamically. We call this address a virtual address and further assume that we can translate any virtual address a_v into a set of physical addresses A at runtime by applying a *one*-to-n table. Such table might be defined as $A_{a_v} = \{a_1, a_2, \ldots, a_n\}$, where a_i is a physical endpoint address of one particular service instance.

It is clear that a client must not have knowledge about address translation. Considering n clients in a system, such information would have to be managed by n components separately and independently.

In addition to address information, there is a clear need to manage meta-information about a service (such as the description of its contract for example). Since all service instances are basically the same (only the physical address differs), the centralized address translation component could also manage meta-information for the virtual service as a whole.

The Domain Name System offers distributed address translation and decouples consumer and provider. The system is in place for decades and has been proven to scale for very large environments. Therefore we propose to use DNS to resolve a virtual address into physical ones by using the TXT-record type [9].

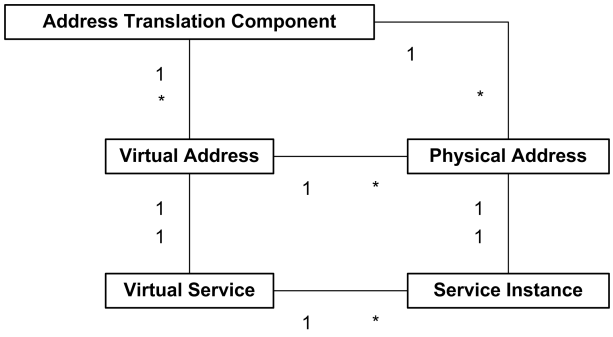

Figure 1: Core entities and their relationships (UML Class Diagram)

Instead of placing ordinary text, we can simply place a full qualified URI. By introducing different URI types, meta-information can also be embedded by publishing an additional *URI*, which points to the location where such meta-information can be accessed. This might be an *LDAP* server or just a *web server*. The following example (listing 1) maps the virtual address *business-service.fh-kiel.de* to *three* different instances (*ISC Bind* zone file [7]).

We introduce the keyword *wsdl::* to define an *URI* that points to the *WSDL* information for the virtual service. WSDL is used to describe a web service contract [12]. Further, such description is applicable for all service instances, since they only differ in their endpoint address. The *port::* keyword defines an *URI* that points to one service instance of the virtual service. Any additional meta-information might be added by introducing additional keywords.

3.2 The Ratio Approach

The approach is based on selecting service instance at random. From a client's point of view, a service instance is called a *channel*. We assume that the resulting probability for selecting channels should reflect the ratio between the observed response-time. For example if we only have *two* channels and the response-time that corresponds to these channels have both the same value (for example *10* milliseconds), we expect to end up with the probability of *50* percent choosing channel A and *50* percent choosing channel B. The response-time for both channels could be *two* seconds in average. This is a quite long time compared to *10* milliseconds of the previous example. In fact the client simply does not know what is long and what is not. The request might be a complex one that just takes some time. We assume that the resulting ratio has to take the differences between the observed response-time values into account. Not the actual values are of interest; the relative differences are.

Therefore the approach is based on calculating the ratio between the best channel's response-time t_b and any given channel's response-time t_i. The normalized weight per channel (*performance-index*) can be defined as $f_i = \frac{t_b}{t_i}$. We assume that $t_b \leq t_i$ since t_b has been selected as the best (lowest) response-time. Figure 2 shows the resulting performance-index for any channel by assuming that the best channel's response-time is *10* milliseconds. The performance-index for the best channel is assumed to be *unity* - which is the best possible performance-index. As the figure demonstrates, the

Listing 1: Forward mapping of a virtual service address to three service instances using the TXT-record (ISC Bind)

```
business-service   IN   TXT   "wsdl::http://www.fh-kiel.de/business.wsdl"
                   IN   TXT   "port::http://web01.fh-kiel.de/service.aspx"
                   IN   TXT   "port::http://web02.fh-kiel.de/service.aspx"
                   IN   TXT   "port::http://web03.fh-kiel.de/service.aspx"
```

resulting performance-index goes down exponentially as the difference between the response-time of the worse channel becomes greater than *10* milliseconds.

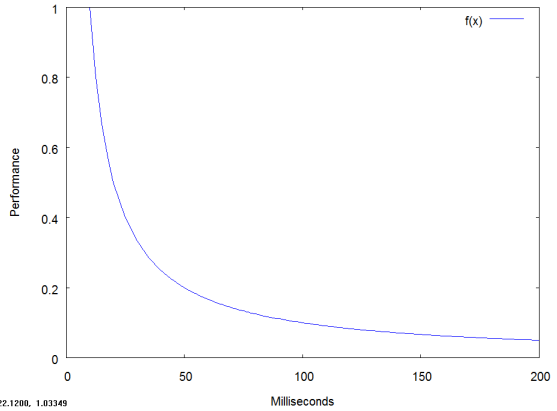

Figure 2: Performance-Index

The probability for selecting channel i is then defined as $p_i = \frac{f_i}{\sum f_i}$. We propose to select channels randomly by applying the principal of two choices [1, 8]. Therefore two channels are selected randomly according to the probabilities (non-uniformly with replacement). Then the channel with the lower response-time is taken. It is important that random channel selection is based on replacement. Therefore there is a certain chance of selecting a worse channel twice, which guaranties that a bad channel is still selected from time to time. If a bad channel would never be chosen again, the system would not detect that such channel became better in the meantime.

3.3 Balancing Strategies

The *Response-Time-Based* strategy uses the Ratio approach for calculating the performance-index. Channel selection is based on the *Two Choices* principal.

The *Queue-Length-Based* approach relies on information exchange between a service instance and a client. The count of request in a service instance's queue is transferred to the client by embedding such information inside a response message. As we have realized, the approach cannot be based on always taking the best channel (e.g. lowest queue length). Therefore the approach uses a modified version of the ratio approach. The performance-index is defined as $f_i = \frac{l_b+1}{l_i+1}$, where l_b is the lowest queue length and l_i the queue length for the channel i. Channel selection is performed by applying the *Two Choices* principal.

The *Shortest Queue* approach assigns new jobs to that service instance that has currently the least count of jobs in its processing queue. Winston [14] has shown that this policy is optimal for homogeneous systems under certain assumptions such as request distribution. The approach has been used in a way where we assume that a client always knows the current queue length of all service instances. No information is exchanged between communication partners. For sure, this is only possible using simulation and is supposed to demonstrate the performance of other approaches compared to the theoretical maximum.

Finally we use the *cyclic splitting* approach [13], where the ith request is forwarded to the *(i mod k)*th service instance (having k service instances). This static approach is also well known as *Round-Robin*.

3.4 Failover

A service instance might fail at any time. We assume that the underlying middleware is able to detect failures. In such cases, the *load balancer* might decide to instantly take another channel which has not become unavailable yet.

However, the system has to detect if a channel - which has failed in the past - became available again. The system could continue to try a channel which has failed. The probability for choosing that channel could be influenced by the number of tries made so far. Therefore the system would choose the channel less often as the number of failures increases. But there is a certain tradeoff to make: The delay for trying a channel might be expensive. The amount of time needed depends on the time-out behavior of the underlying protocol.

To overcome this dependency a system might not use a regular business request for detecting channel recovery. This has the advantage that such detection mechanism could be performed in *parallel* to regular request processing. Therefore the technique would not have a direct impact on the throughput. Sending an *ICMP* ping request for example would not solve the issue because receiving the ping *echo* does not necessarily mean that the *service* is alive. Any other form of communication between a client and a service instance violates the transparency requirement (see section 1).

We propose a solution that *misuses* a regular business request for detecting channel recovery. The technique is based on a *special* request type that is supposed to have no business related functionality. Such operation, for example *IsAlive()*, simply sends a response message back to signal service activity. The system could use this request type to detect channel recovery in *parallel* to regular request processing while the throughput is not affected. The solution has the disadvantage that each service must provide such an operation.

4. SIMULATION

Experimental results have been obtained from simulation. The process is based on sequential discrete-event simulation [6]. Networks can be arranged hierarchically. The smallest amount of time is one microsecond. To simulate the CPU, processing-jobs are queued and served in *FIFO*

order. Message encoding and decoding is not simulated. Data transmission on any communication link is possible at any time for any simulation object even simultaneous. The system behaves as if each client would have its own private full-duplex communication link to all service instances.

5. RESULTS

This chapter presents the results obtained from performing load balancing strategies under certain scenarios. We investigate how those strategies perform by operating *eight* clients and *four* service instances.

5.1 Default Experiment Setup

An experiment takes *410* seconds, where each *10* seconds the system state is changed to produce additional stress on one particular service instance. *Stress* is produced by adding a new client to the system, which sends requests to one particular service instance exclusively without the use of any load balancing mechanism. Such a client is called a *flooder* since it *floods* the target instance with requests. Those requests are not included in the calculation of the throughput. A Request is sent as fast as possible as soon as the response message of the previous request has been received. A request message is *512* bytes in size. A response message is *256* bytes long.

An experiment is split into *two* parts. During the first half, each *10* seconds, the system introduces additional *stress* on one particular service instance. After *21* iteration (after *210* seconds), every *10* seconds *stress* is reduced incrementally. After *400* seconds, the system has reached a state where there is no stress as it were at the beginning of the experiment.

5.2 Comparison of different Strategies

The *Queue-Length-Based* approach (figure 3c) performs slightly better in terms of the total throughput than the *Response-Time-Based* approach does (figure 3a). Both approaches adapt to the experiment scenario almost equally well by favoring the *blue*[1] channel less often.

Surprisingly, the *Shortest Queue* approach (figure 3d) performs not that much better than expected. We have to keep in mind that the approach uses a *magic hand* that transmit the current queue length of all service instances instantaneously to any client on a per-request basis. A client always knows the queue length of all service instances. In terms of total throughput, *Shortest Queue* performs slightly better than the other adaptive approaches do. In addition, *Shortest Queue* omits the *blue* channel completely (after around *40* seconds). It is clear that this has a direct impact on the overall throughput.

Under the given experiment scenario, load balancing which is based only on local observation of the response-time is able to deliver a quite good performance in terms of total throughput as a comparable approach like *Queue-Length-Based* does, which is based on information exchange.

[1]The blue channel is labeled as Channel 1.

5.3 Request Type Variation

The following experiment has been set up in the same way as the previous. Each message is *512* bytes long. But the experiment uses *four* different kinds of request types: The first message type takes *eight* milliseconds to process on a service instance. Each following request type takes a multiple of *eight* milliseconds: *16* milliseconds for the second one, *32* milliseconds for the third one and finally *64* milliseconds for the last request type. We expect a more significant noise-level in the observed response-time than we observed during the previous experiment.

Figure 4 shows the total throughput per service instance. A comparison between the approaches makes clear that *Response-Time-Based* (figure 4a) favors the *blue* channel too often. *Queue-Length-Based* (figure 4b) in contrast, adapts well to the changing condition: It favors the *blue* channel significantly less often. Therefore more requests can be placed on the remaining service instances as the observed throughput indicates. Unsurprisingly, *Shortest Queue* (figure 4c) adapts even better than *Queue-Length-Based* does.

An improvement of the *Response-Time-Based* approach might ground its balancing decision on the average response-time. Instead of considering the last known value, the technique calculates the average value of the previously known values for the response-time for each channel. Then, the performance-index is calculated by following the *Ratio* approach as describes in section 3.2. But in this case, instead of taking the actual response-time into consideration, the calculation is based on the averaged value (the last *20* requests). Figure 4d shows that the approach is able to adapt to the situation, but does not perform well at all. It still favors the blue channel too often as we compare it with *Queue-Length-Based* or *Shortest Queue*.

We conclude that the *Response-Time-Based* approach performs well if we assume that the processing-time for different operations is more or less equal or the distribution of the associated request types does not result into a high variation in the observed response-time.

5.4 Different Networks

The following experiment uses - again - *eight* clients and one virtual service that consist of *four* instances. While previous experiments used one single network, the following test uses *two* different networks with identical properties (bandwidth is *10* megabit/sec, latency is *one* millisecond). Clients and service instances are distributed equally (*four* clients and *two* instances per network). To enable inter-network communication, a third network is introduced that interconnects both local networks. Such network is initially able to transmit *10* megabit/sec. The initial latency is also *one* millisecond.

During the experiment, each *10* seconds, the latency of the interconnecting network is increased by *two* milliseconds. Therefore we expect the observed response-time to change over time. Requests that are served by service instances of the local network are served faster than requests that are serviced by instances that are located in the other network. After *21* increments (after *210* seconds) the latency is decreased by *two* milliseconds each *10* seconds. After

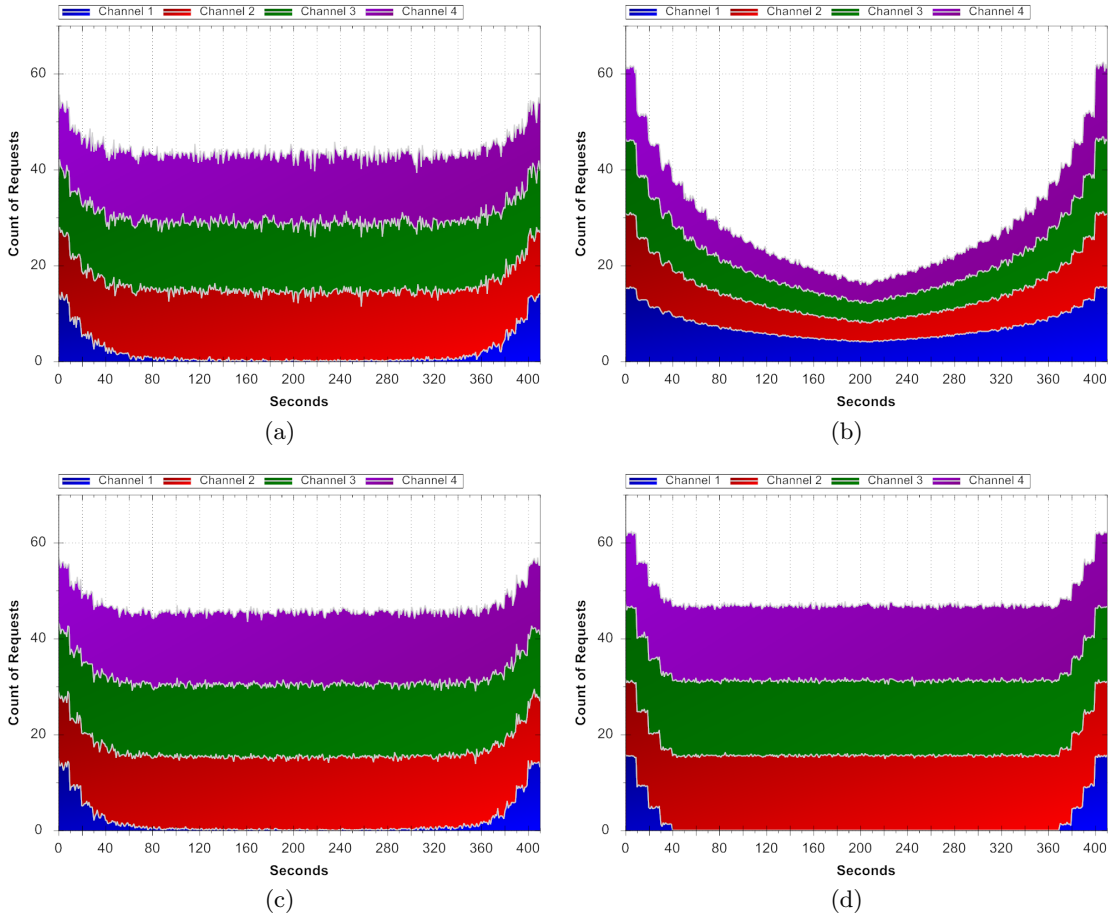

Figure 3: Comparison of the throughput between different balancing strategies: (a) Response-Time-Based; (b) Round-Robin; (c) Queue-Length-Based; (d) Shortest Queue

400 seconds, the latency of the interconnecting network has reached its initial value of *one* millisecond.

It is obvious that the *Queue-Length-Based* approach (figure 5a and 5b) does not adapt well to the changing condition. The resulting throughput performance is quite similar to those obtained from *Round-Robin* during the first experiment (see section 5.2). Because the balancing mechanism is unaware about the increasing latency of the interconnecting network, it cannot adapt to it at all. Any balancing decision that is not based on response-time or is unable to detect the increasing latency of the interconnecting network, will obviously fail to adapt.

In contrast, *Response-Time-Based* (*second row*) is based on the response-time and is therefore able to detect the increasing mismatch. It adapts well to the changing condition. As the latency increases, clients favor there local service instances. All clients can remain a total throughput of about *50* requests per second, independently of the increasing latency of the interconnecting network. This is comparable to the total throughput we observed from the *Queue-Length-Based* approach at the beginning of the experiment, where the latency had not been changed yet.

5.5 Additional Probes

The last experiment focuses on the question, if the performance of the *Response-Time-Based* load balancing approach can be increased by introducing additional probe messages. The approach depends on response-time information that is observed by measuring the time that elapsed on a per-roundtrip basis. If the average load of all instances does not change over time, we expect a client to observe more or less the same response-time. If the load condition changes with the frequency of f_s, we expect a client to detect those changes accurately, if the frequency of sending requests f_r is higher than the frequency of those changes (e.g. $f_r > f_s$). If not, there are simply not enough requests in order to observe those changes accurately enough, since sending requests is the only method for observing the response-time.

The following experiment consists of *one* network. The bandwidth is *10* megabits per second. The latency is *0.4* milliseconds. Both properties do not change over time. The network contains *32* clients and *one* virtual service that consist of *four* instances. Each client sends a request every 1.6 seconds. A client will not send another request, if there is still a request pending. In such a case, a client has to wait for the next period. The processing-time is supposed to take *eight* milliseconds per business request.

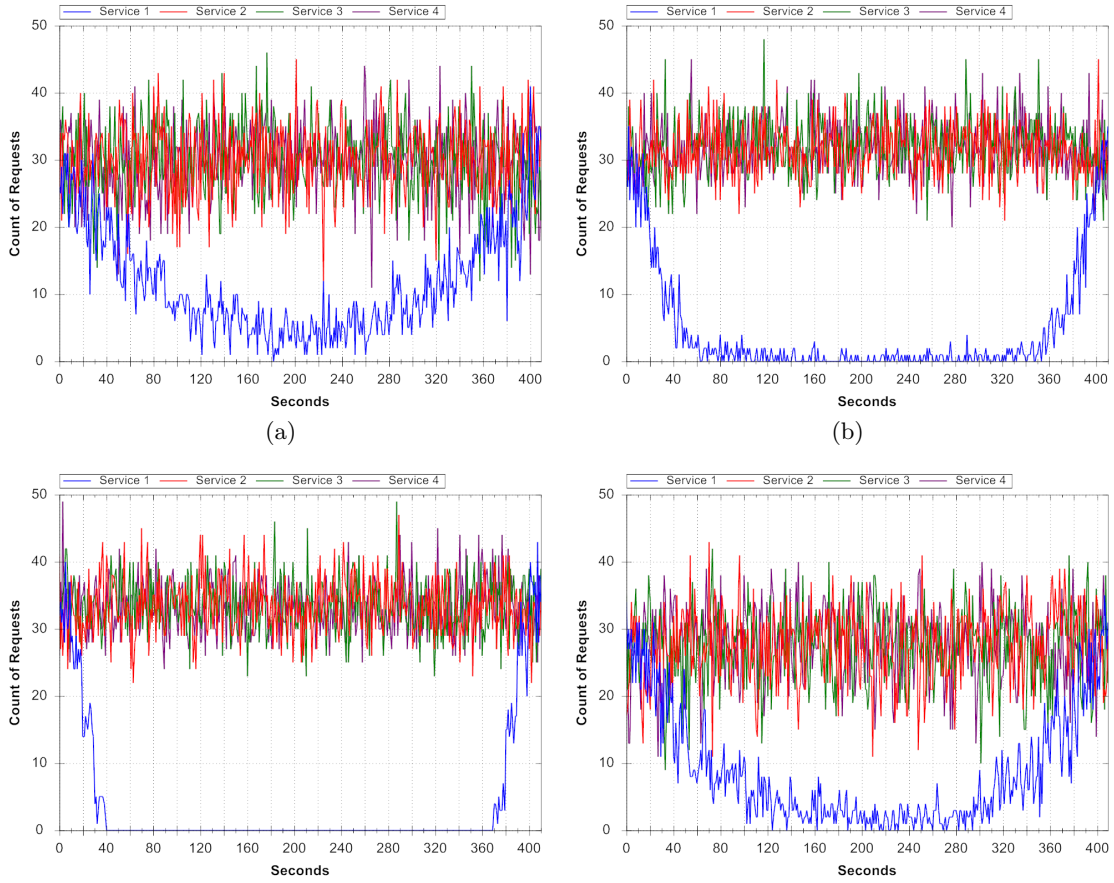

Figure 4: Comparison of the throughput per instance with request type variation: (a) Response-Time-Based; (b) Queue-Length-Based; (c) Shortest queue; (d) Response-Time-Based (Averaged)

Either a client does not send additional probe messages at all, or such probe-messages are sent in a certain interval. A client does not have to wait to receive the corresponding response message of a probe message in order to send another one. A probe message is sent to all service instances at once. This says that a client periodically sends out probe messages to all service instances simultaneously. By having *four* instances, each client sends out *four* probe messages. The processing-time to process one probe message is supposed to take *one* millisecond.

In addition, the system contains *four* clients, which send requests to one particular service instance. Those clients do not make use of any load balancing mechanism. They are supposed to produce additional stress on their particular target instance. The important aspect is that the target of those clients changes every second. The target instance is chosen by iterating over all instances available. During the first second of the experiment, the first instance is chosen. During the following second, those clients will target instance number *two* and so forth. After the last instance has been *stressed*, the first one is chosen again. We notice that the global system state is changing dramatically every second during the experiment.

Figure 6 shows the results for the response-time per message which has been observed during all experiment variations (averaged values over *five* runs). During the first variation (*No Probes*), a roundtrip took - in average - less than *21* milliseconds. By sending additional probe messages, the system is able to improve the average response-time per message. The period for sending a probe message went from initial *800* milliseconds down to *200* milliseconds. We observe a significant improvement during the first *three* variations which do include probe messages. The average response-time per message had been improved from initially around *21* milliseconds (no probes) down to around *17.4* milliseconds (*four* probe messages per client each *400* milliseconds). As we go further by sending even more probe messages, the improvement vanishes.

The results indicate that old information about the system state is useful in the sense that a balancing decision can be based on it. The quality of the resulting performance - in terms of response-time per message - highly depends on how old the information is. More accurate information is more useful than older information. Increasing the period of sending additional probe messages leads to more accurate information about the system state and finally to an increase in the overall performance.

Based on this observation, a system might implement response-

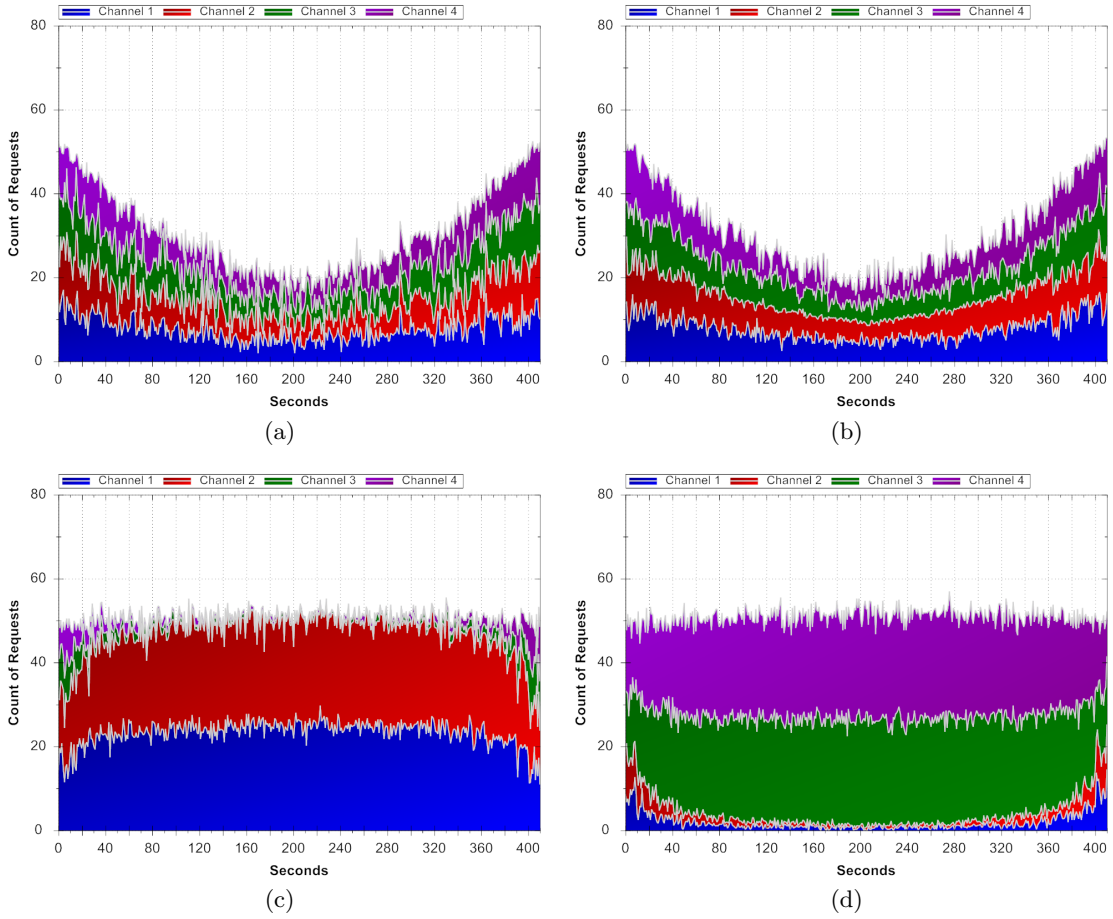

Figure 5: Comparison of the throughput with different networks (left: first network with four clients and two service instances; right: second network with the same amount of clients and service instances): (a) Queue-Length-Based (first network); (b) Queue-Length-Based (second network); (c) Response-Time-Based (first network); (d) Response-Time-Based (second network)

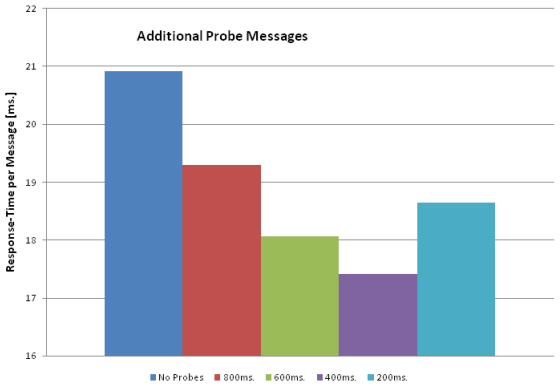

Figure 6: Comparison of the response-time per message with and without additional probe messages (averaged over five runs)

time-based load balancing by following the dispatcher model, where an *in-the-middle* component routes all requests to service instances. We assume that the overall request send rate of such a dispatcher component is much higher than the send rate of each client, since every request of every client will be balanced by the dispatcher. On the other hand, such implementation might lose any benefits from local client observation in terms of changes in the network topology as we have investigated in section 5.4.

6. CONCLUSIONS AND FUTURE WORK

The first section gave some indications for an increasing demand of load balancing in service-oriented distributed systems. Vendors are shifting their traditional infrastructure forward to service-oriented systems. Further indications are given by emerging trends in shifting applications into the cloud, where computational power can be brought to devices, desktop applications and web-sites. Finally ubiquitous computing promises an increasing number of small embedded devices which might rely on the computing-power of the cloud.

On the other hand, a number of technologies exist that enables service-oriented distributed systems. *Web Services* is clearly a common technology in that field, but others like *REST* gain more and more importance. Load balancing is often coupled to a specific technology or application scenario. Receiver-initiated solutions require a specific implementation on the service side. Sender-initiated solutions require remote state monitoring and information exchange

capabilities, which also require specific implementation on the service side.

For that reason we have introduced an adaptive client-based load-balancing approach which is only based on local observation. By observing the response-time the approach allows to balance load independently of any concrete technology or implementation. Further we introduced a mechanism which is based on *DNS* to perform address translation dynamically. Finally we have investigated the approach through simulation which delivered promising results under some circumstances.

For systems without any variation in request type, the approach is able to perform surprisingly well. There might be a certain tradeoff in using the response-time-based approach, but it offers independence and loose-coupling to the implementation and technology used for service implementation.

Investigation based on variation in request type showed that the approach is not able to adapt. The variation in the observed response-time is simply too noisy in order to clearly identify good performing channels from bad ones. Averaging the observed response-time brought a slightly improvement but the results were not promising at all. However, the experiment used an extreme setup to provoke such effect. The approach might be applicable for systems where service operations do not differ significantly in their response-time behavior.

We propose to concentrate further work on improving the *Response-Time-Based* approach to overcome limitations where clients observe variation in the response-time. We propose to measure the minimum response-time per request type $r_m = \{r_1, r_2, \ldots, r_n\}$. If we assume that the processing-time for each request-type is nearly constant, then the ratio between the best response-time of all request types r_b and any given response-time might be used as a filter to *normalize* the variation in the observed response-time. Therefore the ratio factor for request type i would be given as $r_i = \frac{r_b}{r_i}$. Then, the *Ratio* approach could be used to calculate the performance-index for channel i and request type j by applying $f_{ij} = \frac{t_b * r_j}{t_i}$, where t_b is the best response-time for all channels, t_i the response-time for channel i and r_j the ratio factor for the request type j.

Not surprisingly, the approach performs best in situations where the system must adapt to circumstances caused by the network. The approach can remain a certain level of total throughput, even when there is a huge delay for some service instances which has been caused by the network. Furthermore results indicate that any balancing approach which does not include the response-time or is unable to detect network properties is not able to adapt at all and falls back to a behavior similar to *Round-Robin* (with heavy load on some service instances).

Finally, experiments have shows that the system in order to perform well requires at least a request rate which is higher than the rate in which the system state as a whole changes. By periodically sending hypothetical probe messages, the system could increase its performance where such period was below the period in which the global system state changed.

To overcome situations where the request rate of a client is simply too low, we have proposed to introduce a dispatcher. Such arrangement generally results into a higher request rate, since the dispatcher represents all clients or groups of clients in the system. Therefore the request rate at which the dispatcher operates is clearly higher than the request rate of any client. In addition, we proposed the dispatcher-based deployment as a solution to overcome the lack of influence on a client's implementation, where the approach cannot be embedded into clients directly.

7. REFERENCES

[1] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 593–602, New York, NY, USA, 1994. ACM.

[2] S. A. Banawan and J. Zahorjan. Load sharing in heterogeneous queueing systems. In *INFOCOM*, pages 731–739, 1989.

[3] D. Ferrari and S. Zhou. A load index for dynamic load balancing. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 684–690, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[4] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Richard N. Taylor.

[5] Gartner Inc. Gartner says worldwide application infrastructure and middleware market revenue increased 13 percent in 2007. Jun. 2008. `http://www.gartner.com/it/page.jsp?id=689410`.

[6] S. Ghosh. *Distributed systems: An algorithmic approach*, volume 13 of *Chapman & Hall/CRC computer and information science series*. Chapman & Hall/CRC, Boca Raton, 2007.

[7] Internet Systems Consortium, Inc. (ISC). *BIND 9 Administrator Reference Manual*. Internet Systems Consortium, Inc. (ISC), bind 9.5 edition, 2004-2008. `http://www.isc.org/index.pl?/sw/bind/index.php`.

[8] M. D. Mitzenmacher. *The power of two choices in randomized load balancing*. PhD thesis, 1996. Chair-Alistair Sinclair.

[9] P. V. Mockapetris. Domain names - implementation and specification. 1987. RFC1035 `http://www.ietf.org/rfc/rfc1035.txt`.

[10] O. Othman and D. C. Schmidt. Issues in the design of adaptive middleware load balancing. *SIGPLAN Not.*, 36(8):205–213, 2001.

[11] E. Putrycz. Design and implementation of a portable and adaptable load balancing framework. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 238–252. IBM Press, 2003.

[12] W3C. Web services description language (wsdl) 1.1. March 2001. `http://www.w3.org/TR/wsdl`.

[13] Y.-T. Wang and R. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, 34(3):204–217, 1985.

[14] W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14 No. 1:181–189, Mar. 1977.