Service Load Balancing with Autonomic Servers: Reversing the Decision Making Process

Remi Badonnel[1,2] and Mark Burgess[1]

[1]Oslo University College
St Olavs plass, 0130 Oslo, Norway
[2]LORIA - INRIA Grand Est, Nancy University
Campus Scientifique, 54500 Vandoeuvre, France
badonnel@loria.fr, burgess@iu.hio.no

Abstract. Load balancing faces new challenges in the framework of autonomic servers deployed in data centers. With traditional push-based strategies, the authoritative decision is made by the load balancer, which decides to which server the requests are forwarded. However, the autonomy of servers is often incompatible with these strategies, as they may accept or refuse to process a request on a voluntary basis. We present in this paper the benefits and limits of a pull-based load balancing strategy for transferring the authority from the load balancer to the autonomic servers. We describe the underlying functional architecture with two different schemes and quantify the performances through an extensive set of experiments.

1 Introduction

Autonomic computing has become a major paradigm for dealing with the growing complexity of systems and networks and simplifying their maintenance [1]. In particular, we can consider autonomic servers that are capable of managing themselves based on closed control loops in order to: configure their components, detect and correct their failures, monitor and control their own resources in an optimal manner, and diagnose and protect themselves against attacks. These servers can typically be deployed in data centers. They may provide support for multi-tier applications and services, and share the load of client requests.

A variety of algorithms [2] has been proposed in order to balance the workload among servers in an optimal manner. However, autonomic computing favors autonomous components that are weakly coupled rather than traditional hierarchical systems with strong couplings (based on an obligation model). Autonomic servers seem therefore to pose new challenges with regard to this load balancing.

In traditional approaches, the load balancing strategy is performed in a push-based (obligation) manner (see Sub-figure 1(a)), which means the decision of whether a server should receive a request or not made exclusively by the load balancer. Autonomics skeptics often imagine that this kind of approach is fundamental to the idea of "control" and the idea of component autonomy stands in the way of proper resource sharing if servers will not do as they are told by an authoritative controller. We have already argued against this viewpoint [3] and show this belief to be erroneous below.

One might argue that a push-based load balancer can easily estimate server resources by observing network parameters such as on-going connections. Monitoring agents can also be deployed on the servers to evaluate their performance. While server-state information can improve the load balancing mechanism with traditional servers, this adds an unnecessary overhead with autonomic servers. The authoritative decision is kept by the load balancer, which decides to which server the request is forwarded. Moreover, this authority is in conflict with an autonomic server's preference to make decisions by itself and to interacting on a purely voluntary basis. So there are two reasons why push based sharing is undesirable in autonomic networks.

Why then should we preserve a push-based strategy for balancing load among autonomic servers? In this paper we explore the arguments in favor of *pull-based* load balancing strategy (see Sub-figure 1(b)) where autonomic servers can manage load at their own convenience. Each server knows its own capabilities and state more quickly and accurately than any external monitor, so it seems reasonable to explore the idea that it is the best judge of its own performance.

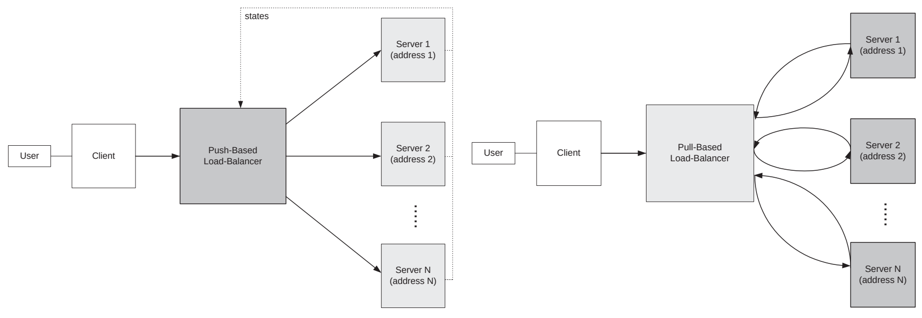
By pulling service requests from a load balancer servers can decide to take on work depending not only on their available resources but also on many other internal parameters including their willingness to interact, their internal scheduled management operations, protection procedures when attacks are detected, and any policy. The authoritative decision is therefore transferred from the load balancer to the autonomic servers.

The paper is structured as follows: we present in Section 2 our pull-based load balancing strategy for autonomic servers. We describe the underlying architecture and show how this scheme can be instantiated with two different return path strategies. We evaluate the performance and scalability of the solution through an extensive set of experimentations in Section 3. A survey of related work is given in Section 4. Finally, Section 5 concludes the paper and presents future research efforts.

2 Pull-Based Load Balancer

In this work, we explore the benefits of a pull-based load balancer for distributing workload among autonomic servers. The key motivation is to make the load balancing strategy more flexible and adaptive with respect to the servers. This is compatible with the goals of autonomic computing. Autonomic servers interact only on a voluntary basis. Their autonomy contradicts the basic tenets of push-based load balancing.

With the push-based strategy, the decision of whether a host should receive a request or not, is taken by the load balancer. Some solutions permit to a load balancer to exploit state information transmitted (voluntarily and hopefully on time) by the servers, but the push-based load balancer takes the final decision. In [4] with traditional servers, it was shown in actual hardware that this decision making could be a limitation on the dispatch rate. For instance, the least-connections algorithm [5] keeps track of the number of active connections each server currently has. The dispatcher then forwards requests (i.e., new connections) to the server with the fewest active connections. Another example consists of sending server-state information to the balancer using a dedicated protocol such as the Dynamic Feedback Protocol [6]. The balancer then determines based on that information which server will handle the request. This adds overhead to the process. By considering a pull-based strategy, the authoritative



(a) Push-Based Load Balancing (b) Pull-Based Load Balancing

Fig. 1. Push-Based vs Pull-Based Load Balancing with Autonomic Servers

decision is transferred from the balancer to the servers themselves since this is where the important information is located. Indeed, we can view the interaction between balancer and server as a competitive game. If the balancer acts first, it has imperfect information about its opponent's condition and the coalition of dispatcher and servers can lose productivity through poor forwarding decisions, but if the server acts first it does not need information about the dispatcher's state to make the most economically motivated decision since there is no penalty if there is "no work to be done". Hence, the servers have effectively perfect information as far as the overall productivity is concerned. The load balancer role can typically be played by the client itself, by a DNS server or by a dedicated dispatcher (interacting as a proxy).

We consider the pull-based load balancer to be a dispatcher with a central queue. All of the requests from clients are kept by the dispatcher so that the autonomic servers can pull requests from this queue at their own convenience i.e., whenever the server makes a voluntary decision to process a request. This includes whenever the server has free resources, but is not limited to that particular condition. A server may refuse to process requests for other reasons: when failures have been detected, when new components have to be installed and configured, when attacks have been detected and protection procedures have to be executed, or when the server simply does not want to (unable or unwilling to comply). In any event, the other servers adapt automatically to the best of their own ability and policy.

2.1 Underlying Architecture

We will detail in this section how this pull-based strategy can be instantiated using two different return path strategies. A first possible architecture consists in using a dispatcher with NAT (Network Address Translation). Let consider the application example of web servers. The dispatcher receives multiple HTTP requests from clients via its public address. When a client sends such a request to the load balancer, the request is first stored into the central queue of the load balancer. The requests are usually processed according to a FCFS (First Come First Serve) queue scheduling discipline [7]. The key difference with the push-based scheme is that the requests have to stay at the load balancer stage until autonomic web servers start to pull requests.

Each of the servers can pull requests from the dispatcher at any time. If client requests have been issued and are waiting in the balancer central queue, the web server is dynamically served by the dispatcher. We can observe that the load balancing process has been reversed and that server pulls have to be managed by the dispatcher. The dispatcher uses the NAT mechanism to transfer a client request to a given web server. The dispatcher translates the target IP address to the one of the client using Reverse NAT and sends the reply to the client.

The NAT mechanism requires that the load balancer keeps state of each ongoing connection. This may contribute to a bottleneck effect [8] at the load balancer when the traffic rate is high.

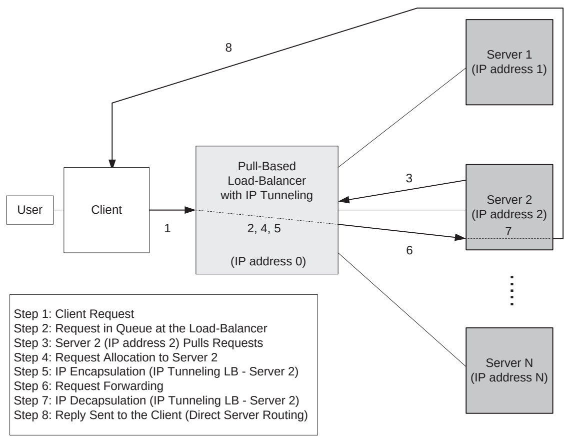

8
Server 1
(IP address 1)
Pull-Based
Load-Balancer
with IP Tunneling
2, 4, 5
(IP address 0)
User
Client
1
3
Server 2
(IP address 2)
7
6
Server N
(IP address N)
Step 1: Client Request
Step 2: Request in Queue at the Load-Balancer
Step 3: Server 2 (IP address 2) Pulls Requests
Step 4: Request Allocation to Server 2
Step 5: IP Encapsulation (IP Tunneling LB - Server 2)
Step 6: Request Forwarding
Step 7: IP Decapsulation (IP Tunneling LB - Server 2)
Step 8: Reply Sent to the Client (Direct Server Routing)

Fig. 2. Pull-Based Load Balancer with Direct Server Return

With the NAT mechanism, the web server has to send the resulting reply back to the pull-based load balancer. An alternative solution consists in modifying the return path. Three different strategies [9] can usually be considered to send the reply to the client: (1) the bridge path strategy when the dispatcher is implemented at level 2 and interacts as bridge, (2) the route path strategy when the dispatcher is implemented at level 3 and interacts as an intermediate router, and (3) the direct server routing strategy when the reply is directly sent back by the server to the client. The pull-based load balancer with NAT corresponds to the second strategy. We propose to deploy the pull-based scheme using a direct server return, in order to decrease the bottleneck effect i.e., so that the reply is not forwarded by the load balancer.

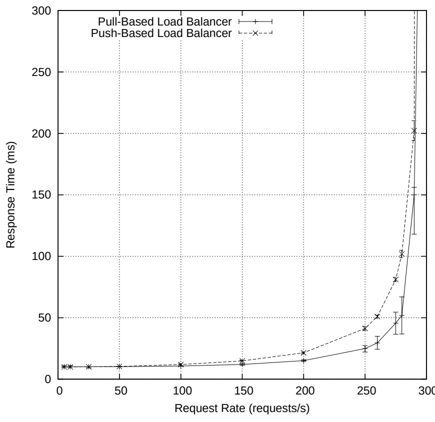
This second architecture is depicted in Figure 2. It requires to establish IP tunneling between the pull-based load balancer and the web servers. The load balancing process essentially differs from the previous one by the manner how requests are forwarded among components. The pull-based load balancer is also implemented as a dispatcher, but uses IP tunneling instead of NAT. When receiving client requests (Step 1), the dispatcher first stores them into the central queue (Step 2) according to a FCFS scheduling. A server can then pull requests from the dispatcher (Step 3). It may be served by the dispatcher if client requests are queuing (Step 4). In that case, the dispatcher establishes IP tunneling with the web server: the request is encapsulated into another IP packet (Step 5) and is forwarded to the web server (Step 6). The server then decapsulates the request (Step 7). When it has successfully processed the request, the server can directly send the reply back to the client using a direct server return. This return path increases the load balancer performance as it does not require to keep state of the connections at the load balancer.

3 Experimental Results

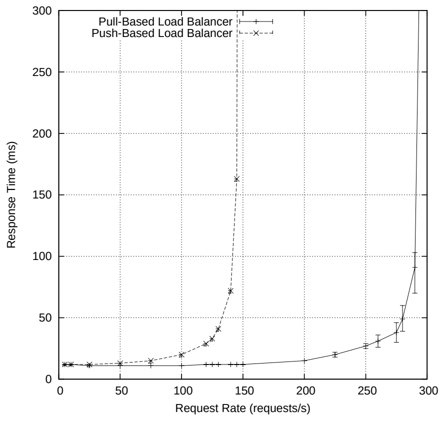
We evaluated the performance of our pull-based load balancing scheme through an extensive set of experiments, which can be compared directly with earlier physical experiments performed in [4] (these previous experiments were however limited to push-based load balancing with regular servers). The simulations were performed with the JMT open source suite for queuing network modeling and workload analysis [10] developed by the Performance Evaluation Lab at the Polytechnic University of Milan. We extended it so that we can model the behavior of a pull-based load balancer with direct server return. We considered during the experiments a system composed of a load balancer L at the front office and a set of n servers $S = \{S_1, S_2, ..., S_n\}$ at the back office. The load balancer L implements either a push-based scheme or a pull-based scheme depending on the scenario. We assume that the arrival and completion process distributions follow a typical Poisson distribution in discrete time. Despite the controversy regarding the inter-arrival times, we use Poisson distributed arrivals, this allows a direct comparison with [4] and we would not expect the choice to affect the broad conclusions of our results.

3.1 Performance with Homogeneous Servers

In a first series of experiments, we were interested in analyzing the performance of our solution with homogeneous servers. We modeled a system with three servers $\{S_1, S_2, S_3\}$ of same capabilities. Each server is capable of processing an average of 100 requests per second. We measured the average response time obtained with the pull-based load balancer while varying the request rate from 5 to 300 requests per second. We assume here the natural notion of response time perceived by users, that is, the time interval between the instant of the submission of a request and the instant the corresponding reply arrives completely at the user. We compared these values with the response time provided by a load balancer implementing a classic push-based scheme where servers are taken without consideration of their current queue length or latency. Indeed, autonomic servers may refuse to provide any server-state information, and we consider that they interact in a voluntary basis making least-connection algorithms inefficient. The experimental results are summarized in Figure 3(a). The horizontal axis corresponds to the different request rate values. We plotted for each of them the average response time obtained with the pull-based load balancer (plain line) compared to the one obtained with the push-based load balancer (dotted line). The two graphs show the same tendency when we increase the request rate: the



(a) Homogeneous Servers

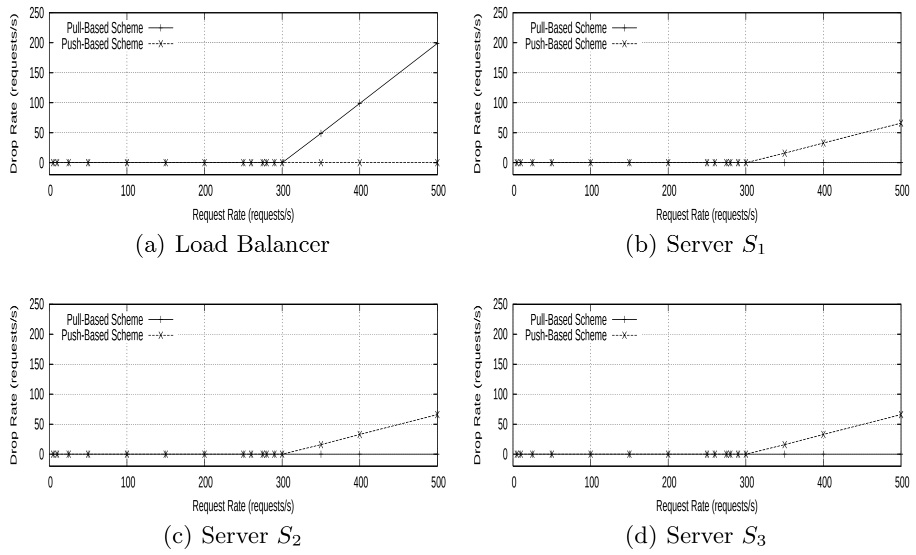


(b) Heterogeneous Servers

Fig. 3. Response Time with Autonomic Servers

response time first stays relative low and stable (on the left of the figure), then grows exponentially when we tends to 300 requests per second (on the right of the figure). This value corresponds to the theoretical maximal processing rate of the system (the sum of three servers capable of processing 100 requests per second each). When comparing the two load balancing schemes, we can observe that the performance is quite similar at a low response rate (from 5 to 100 requests per second). The push-based load balancer is even better than the pull-based strategy at the lowest rates. However, at high response rates (from 100 requests per second), we clearly see that the pull-based load balancer produces faster response times than the push-based load balancer.

Additional parameters need to be taken into account to complete and refine this comparison. In particular, as load balancer and servers have finite queue sizes, we measured the request drop rate while varying the request rate values. The drop rate is the number requests per second never processed at all by an element of the system. We plotted these rates in Figure 4 for respectively the load balancer and each of the three servers. When looking at these results, we first can observe that the system starts to drop requests at around 300 requests per second (compare to [4]), when the request rate exceeds the maximal processing rate. However, the dropped requests are not distributed in the same manner in the system. In the push-based scenario (dotted line), the dropped requests are equally distributed among the three servers. The load balancer does not drop any requests during the experiments, even with high request rates. The reason for these results is that all the requests are dispatched by the load balancer to one of the servers whatever the server is overloaded or not (i.e., it simply pushes the problem downstream).



(a) Load Balancer (b) Server S_1

(c) Server S_2 (d) Server S_3

Fig. 4. Drop Rates with Homogeneous Servers by using the Pull-Based Strategy (plain line) and the Push-Based Strategy (dotted line)

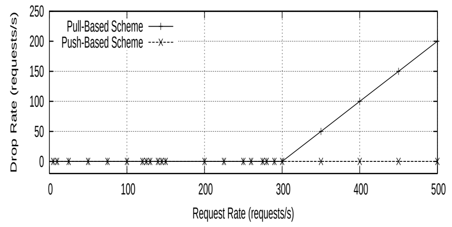
Intuitively, the pull based mechanisms needs a longer queue at the dispatcher since this is where waiting builds up at the system bottleneck, but this does not imply that greater resources are needed. The pull strategy is like an airport check-in queue: servers pull passengers from a single line that fills the same space as would multiple lines. The same resources are simply organized differently. Moreover, the reliance on a single dispatcher should not be a problem if redundancy is factored into the calculation [11].

As the servers have homogeneous capabilities and the load balancer takes servers each in turn, the dropped rate is almost identical for the three servers and corresponds to the third of the total drop rate in the system. In the pull-based scenario (plain line), requests are only dropped by the load balancer. The servers process requests on demand by pulling the load balancer. As a consequence, the servers are not overloaded and do not need to drop requests during the experiments. The requests are waiting at the load balancer and are directly dropped by it when the request rate is too high for the system.

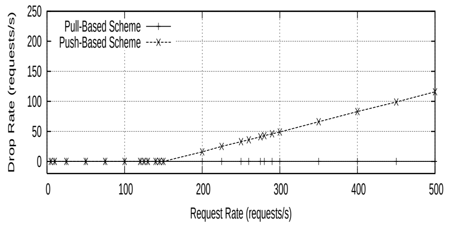
3.2 Performance with Heterogeneous Servers

In a second series of experiments, we analyzed the same scenario with heterogeneous servers. The average service rate for servers S_1, S_2, S_3 is respectively of 50, 100 and 150 requests per second. The graphs in Figure 3(b) represent, as previously, the comparison of response times with the pull-based strategy and the push-based strategy. This scenario produces worst performances than the scenario with homogeneous servers. However, the impact has not the same scale on the two strategies: the difference is minor with the pull-based strategy while the difference is major with the push-based scheme. In the push-based scenario, the response time grows faster when the arrival rate tends to 150 requests per second. The server S_1 with the smallest capacity becomes overloaded earlier than the two other servers. The push-based load balancer is not capable of transferring the workload on servers S_2 and S_3. The workload is equally distributed on the three servers (a third per server). The response time therefore increases significantly at around 150 requests per second when server S_1 is overloaded (a third of 150 requests per second). In the pull-based scenario, server S_1 can reduce its workload by pulling less requests from the load balancer. In that manner, the load balancer can maintain the requests on its queue and dispatches them to the higher capacity servers. As the total processing capacity $(50 + 100 + 150)$ is the same than in the homogeneous scenario (3×100), the response time grows exponentially when the arrival rate tends to the same saturation value of 300 requests per second.

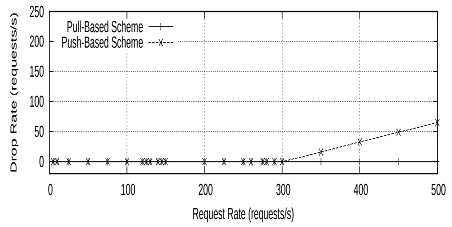
We observe the same phenomenon by measuring the drop rates of servers (see Figure 5). In the push-based strategy, the load balancer does not drop requests. The drop rate is unequally distributed among the three servers, as they start to drop requests when they becomes overloaded e.g., when the total workload is three times equal to their processing capacity. Server S_1 drops requests earlier (at 150 requests per second) than servers S_2 and S_3 (at around 300 and 450 requests per second). In the pull-based strategy, the load balancer is the only element which drops requests in the system (at 300 requests per second), as the servers can reduce the workload by their own in order not to be overloaded.



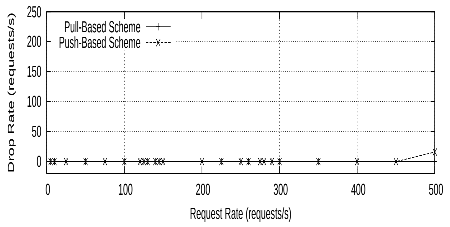
(a) Load Balancer



(b) Server S_1



(c) Server S_2



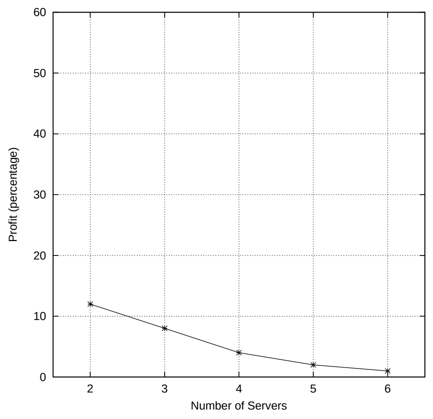
(d) Server S_3

Fig. 5. Drop Rates with Heterogeneous Servers by using the Pull-Based Strategy (plain line) and the Push-Based Strategy (dotted line)

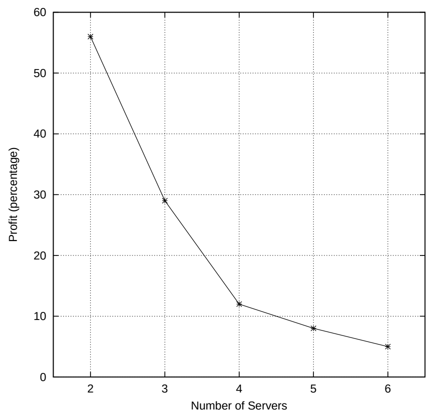
3.3 Scalability and Bottleneck Effect

The third series of experiments evaluates the scalability of our pull-based load balancer when adding extra servers. We measured the average response time while varying the number of servers from 2 to 6 in the system and calculated the profit in percentage of the pull-based strategy compared to the push-based one. We considered during the experiments that the servers are homogeneous with a same processing capacity of 100 requests par second on average. We plotted the experimental results for four different request rates $\{100, 200, 400, 600\}$ in Figure 6.

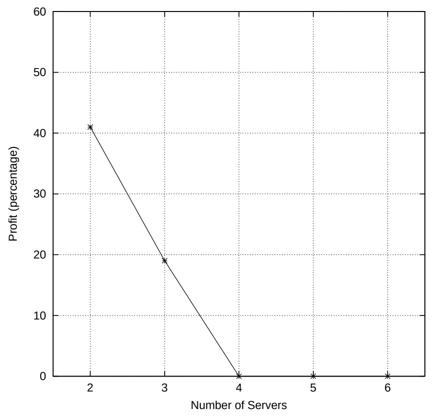
We can first observe that the profit generated by the pull-based load balancer for a given request rate decreases when we add extra servers. For instance, Sub-figure 6(a) shows that the profit at a rate of 100 requests per second goes from around 12 percents with 2 homogeneous servers to around 2 percents with 6 homogeneous servers. The pull-based strategy is therefore much more valuable if the server resources are scarce. Indeed, the push-based strategy with homogeneous servers stays relatively competitive as long as the resources are sufficiently dimensioned. Heterogeneous servers may produce a better profit with the same scenario, in particular when one of the servers has a significantly smaller capacity. In that case, the smaller server can regulate the request rate so that the load balancer forwards the requests to the other servers. Another natural question is to what extent the request rate impacts on the pull-based strategy. When comparing the four sub-figures, we can distinguish two major phases. In a first phase (Sub-figure 6(a) and Sub-figure 6(b)), increasing the request rates



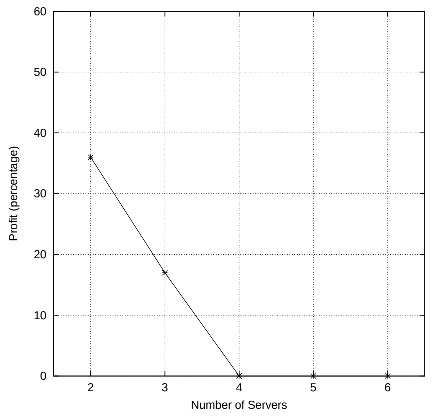
(a) 100 requests/s



(b) 200 requests/s



(c) 400 requests/s



(d) 600 requests/s

Fig. 6. Profit of the Pull-Based Strategy Compared to the Push-Based Strategy on Adding Extra Servers. Results can be compared to [4]

improves the profit generated by the pull-base strategy. For instance, with three homogeneous servers, the profit grows from around 7 percents with a rate of 100 requests per second to around 28 percents with a rate of 200 requests per second. However, in a second phase (Sub-figure 6(c) and Sub-figure 6(d)), the profit starts to decrease when we continue to increase the request rate. The profit with three servers decreases to 19 percents at 400 requests per second and to 16 percents at 600 requests per second. The pull-based load balancer becomes even less interesting than the push-based load balancer in the worst cases. We mean, by worst cases, scenarios when the request rate is high and the server resources are sufficiently scaled. Both the pull-based and push-based load balancers suffer from the bottleneck effect at a high load. However, this phenomenon is significantly more important with the pull-load strategy. These experimental results are confirmed by the drop rates presented in figures 4 and 5, where we observed the requests are maintained and then dropped by the load balancer.

We also analyzed the behavior of our pull-based strategy, by comparing experimental results to queuing theoretical models [12]. In our experiments, the authoritative decision of servers is limited to a simple condition on a given resource (maximal processing rate). As a consequence, the system converges to a behavior very similar to a set of servers processing the same queue ($M/M/k$ queuing model). This convergence may be less evident if we consider the full autonomy of servers. Moreover, our research group attempted in [5] a direct implementation of a pull-based load balancer using a Java application in an attempt to demonstrate that pull-based methods would be superior to push methods. The prototype showed performance can often be hostage to implementation issues.

4 Related Approaches

A large variety of algorithms and techniques were proposed for balancing workload to traditional back end servers [5]. Dynamic algorithms can exploit client-state and server-state information. The server-state information can be obtained in an implicit manner: the load balancer estimates the load on the servers by passively monitoring network parameters such as ongoing connections or by activating probes. For instance, we previously mentioned that the least-connections scheme selects the server with the lowest number of ongoing connections. In all of these approaches, the load balancing is performed in a push-based manner. We have already described in [4] (but with traditional servers) a comparative study of these push-based load balancing algorithms used to distribute packets among a set of web servers. Server-state information can also be obtained in an explicit manner. For instance, the Dynamic Feedback Protocol (DFP) [6] permits a dispatcher to deploy agents directly on the back end servers. These agents periodically report relative weights to the load balancer in the form of load vectors. The load balancer then exploits the reported weights to select more efficient servers. This requires a substantial overhead and infrastructure. In distributed systems and grid computing, management platforms [13] and service middlewares [14, 15] can dynamically align the allocation of resources to infrastructure and business requirements. A significant amount of effort has been expended to design resource management methods based on analytical queuing models [11], decision theory [16], and planning and scheduling techniques [17]. These optimizations are made possible thanks to server-state information collected by agents. In our approach, we show that there is no need for these feedback loops, completely autonomous behavior suffices to solve the problem effectively by voluntary cooperation [3].

5 Conclusions and Future Work

We have explored the motivation for the use of pull-based load balancing between autonomic servers and have quantified the benefits and limits of such a scheme, as one approaches the theoretical limit of a perfect implementation. Autonomic computing advocates greater decentralization of autonomy and only weak coupling of components through cooperative communication. It makes traditional server-state and least-connection inapplicable or inefficient. Our study shows that relaxing the desire for mandatory control of servers using a central controller, and instead allowing them to cooperate voluntarily through only weak coupling, is not the disadvantage that skeptics imagine; quite the opposite, it has the potential to exceed the performance of a push approach, while maintaining better security for each component.

Comparing experimental results to push-based results with autonomic servers we find, as expected, that the pull-based balancer is more sensitive to the bottleneck's resources than the push-based balancer. It surpasses the push-based balancer when the workload is high compared to the expected server acceptance rate (i.e., at high "voluntary utilization"), and when it is sufficiently low to avoid the bottleneck limitation. When we restrict the decision of autonomic servers to a simple condition on a given local resource, the pull-based load balancer converges as expected to an $M/M/k$ model. This convergence may be less evident with servers having a full autonomy. In future work, we are interested in measuring the performance of voluntary dispatch in a large scale virtualization environment [18] where these considerations are especially relevant. We are also planning to model the behavior of autonomic servers using high order internal chain architectures, and in evaluating how these new parameters impact on our pull-based load balancer.

Acknowledgment

This work was supported by the EC IST-EMANICS Network of Excellence.

References

1. Murch, R.: Autonomic Computing. IBM Press (2004)
2. Yu, P.S., Cardellini, V., Colajanni, M.: Dynamic Load Balancing on Web-Server Systems. IEEE Internet Computing **3**(3) (June 1999)
3. Burgess, M.: An Approach to Understanding Policy Based on Autonomy and Voluntary Cooperation. In: Proc. of 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'2005), Barcelona, Spain, October 24-26. (2005) 97–108
4. Burgess, M., Undheim, G.: Predictable Scaling Behaviour in the Data Centre with Multiple Application Servers. In: Proc. of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'2006), Dublin, Ireland. (2006) 49–60
5. Berggren, A.: Presenting a Prototype for Pull Based Load Balancing of Web Servers. Oslo University College, Norway (May 2007)
6. Kersey, C.: Dynamic Feedback Protocol (DFP). http://dfp.berlios.de/draft-eck-dfp-00.txt (August 2005) IETF Internet Draft.

7. Gross, D., Gross, D., Harris, C.M.: Fundamentals of Queueing Theory. Wiley Interscience, Third Edition (February 1998)
8. Jung, G., Swint, G.S., Parekh, J., Pu, C., Sahai, A.: Detecting Bottleneck in n-Tier IT Applications Through Analysis. In: Proc. of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'2006), Dublin, Ireland. (2006) 149–160
9. Bourke, T.: Server Load Balancing. O'Reilly and Associates (August 2001)
10. Bertoli, M., Casale, G., Serazzi, G.: An Overview of the JMT Queueing Network Simulator. Technical Report TR 2007.2, Politecnico di Milano - DEI (2007)
11. Cunha, I.S., Almeida, J.M., Almeida, V., Santos, M.: Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments. In: Proc. of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'2007), Munich, Germany. (May 2007) 129–138
12. Badonnel, R., Burgess, M.: Dynamic Pull-Based Load Balancing for Autonomic Servers. In: Proc. of the 11th IEEE/IFIP Network Operations and Management Symposium (NOMS'08), Salvador, Brazil, Short Paper, IEEE Press (April 2008)
13. Singhal, S., Arlitt, M.F., Beyer, D., Graupner, S., Machiraju, V., Pruyne, J., Rolia, J., Sahai, A., Santos, C.A., Ward, J., Zhu, X.: Quartermaster - a Resource Utility System. In: Proc. of the 9th IFIP/IEEE International Symposium on Integrated Network Management (IM'2005), Nice, France. (2005) 265–278
14. Magaña, E., Lefèvre, L., Serrat, J.: Autonomic Management Architecture for Flexible Grid Services Deployment Based on Policies. In: Proc. of the 20th International Conference on Architecture of Computing Systems (ARCS'2007), Zurich, Switzerland. (March 2007) 157–170
15. Adam, C., Stadler, R., Tang, C., Steinder, M., Spreitzer, M.: A Service Middleware that Scales in System Size and Applications. In: Proc. of the 10th IFIP/IEEE International Symposium on Integrated Management (IM'2007), Munich, Germany. (May 2007)
16. Nassif, L.N., Nogueira, J.M.S., de Andrade, F.V.: Distributed Resource Selection in Grid Using Decision Theory. In: Proc. of the 7th IEEE Symposium on Cluster Computing and Grid (CCGrid 2007), Rio de Janeiro, Brazil. (May 2007) 327–334
17. Diao, Y., Keller, A., Parekh, S.S., Marinov, V.V.: Predicting Labor Cost through IT Management Complexity Metrics. In: Proc. of the 10th IEEE International Symposium on Integrated Management (IM'2007), Germany. (May 2007) 274–283
18. Begnum, K.: Xen Virtualization and Multi-Host Management using MLN. Tutorial of the ACM Conference on Autonomous Infrastructure, Management and Security (AIMS'2007) (June 2007) Oslo University College, Norway.