deployments at best a heuristic tuning or auto-tuning method will be used, we use the Ziegler-Nichols heuristic method [8, 29] to obtain P, I, and D gains with a quality that can be compared to such real-world deployments. First, the P gain is increased until the system starts to oscillate. Then, the Ziegler-Nichols charts are used to calculate the respective gains for a PID controller.

4.3 Distributed feedback channel

The task of the distributed feedback channel is to transmit control information from the controller to the clients. The typical implementation approach is to let the server communicate with each individual client. However, previous work has shown that for large amounts of clients such a system cannot be easily implemented without a specialized broadcasting infrastructure [9].

Consequently, we contribute with the integration of a broadcasting infrastructure based on an overlay network that can be used to *flood* information to the set of clients. We use application-level broadcast [28] based on a tree-based network structure as depicted in Figure 6. Therefore the height of the tree—and thereby the propagation delay—grows only logarithmically with the amount of clients. To prevent failures caused by single clients, we use multiple independent trees, so that each client receives broadcasts from multiple sources. To detect partitions we use a keep-alive mechanism. The server regularly broadcasts cryptographically secured sequence numbers to the tree. A client can detect problems, if it has not received sequence numbers for some time or if individual sequence numbers are missing. However, due to the closed user group in our application scenario we expect a low node churn.

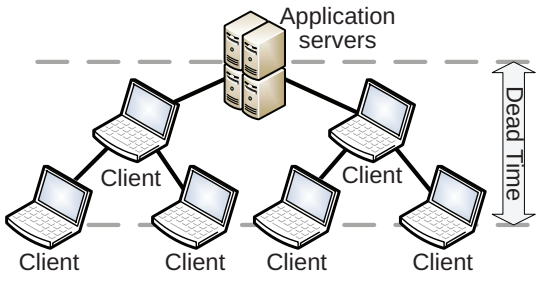

Figure 6: Distributed feedback channel. The servers send information to only a few clients with a tree-based network structure used to propagate data between clients. Parameters of the tree are the arity and the height.

For our distributed broadcast channel there is a trade-off between the total number of clients and the stability of our system:

- With a growing number of clients the height and/or arity of the tree have to increase to accommodate more clients. However, with a growing height or arity the dead time of our system increases as well, which in turn leads to instabilities such as oscillation.
- With increasing height of the tree, the clients' average distance to the server increases. Thus, it takes longer until a server broadcast is received by a majority of the clients. Similarly, with increasing arity each client has to transmit information to a larger amount of children, lowering the bandwidth available for each individual transmission.
- When the dead time of the system increases, this decreases the stability, making the system susceptible to overshoots or oscillation. It is possible to mitigate this issue by decreasing the gains of the controller. However, this slows down the system, potentially preventing it from responding to load changes in time.

By adapting the arity and the height of the N-ary tree, we can control the trade-off between the maximum amount of clients and the dead time. For example, consider an average propagation delay between two clients of 100 ms and a maximum tolerable dead time of 5 seconds. This would allow a total tree height of 50. When using a binary tree, this equals a theoretic total of $22.5 \cdot 10^{14}$ clients. In practice, the number of clients is therefore not limited by our broadcast channel, but by the lower bound of the request rate acceptable for each individual client. Under most configurations, the feedback channel itself is capable of supporting an amount of clients several dimensions higher than the maximum amount reasonably supported by the servers of the system. As a consequence, we can use the additional capacity to increase the redundancy of our tree, allowing for correct propagation of broadcasts in cases where individual clients fail.

As a majority of bidders generally places bids shortly before the auction deadline and as the set of bidders is predefined for each auction, node churn during this critical period is typically smaller than in comparable application scenarios—if not non-existent at all. Therefore, our distributed feedback channel is not specifically adapted to high node-churn scenarios. However—depending on the concrete scenario—alternative application-level broadcast protocols can be used for such cases.

5. PERFORMANCE MEASUREMENTS

In this section we evaluate the effectiveness of our rate control approach. The goal of the measurements is to evaluate different system configurations in regard to stability and performance. While there are several common benchmarks for evaluating Web applications, they are not applicable to our temporally decoupled system. RUBiS (Rice University Bidding System) [3] models an auction site similar to eBay. While in RUBiS the typical user interactions such as browsing of auctions and consulting of bid histories are modeled, in our system the focus lies on the request rate control of temporally decoupled bid submissions. Similarly to RUBiS, TPC-W [25] models a Web shop to test the performance of Web server and database systems. As a consequence, neither RUBiS nor TPC-W can be used in their current specification to evaluate our rate control system, as they are not applicable to temporally decoupled systems. Instead, we evaluate our system in the auction scenario specified in Section 5.4.

In the following evaluations we first examine the open-loop behavior of our system and the system's response to changes in the input. Then, we examine the closed-loop behavior of our controller in a simulated auction scenario using group-based and interval-based control. In addition, we also compare our approach to alternative solution ap-

proaches, such as an approach piggybacking control information in HTTP requests and another approach rejecting requests on the server-side [1].

5.1 Configuration

In the evaluation we focus on single-threaded clients using *data-queue* for transmission and the processing and queuing delay as metric. In addition, we also conducted measurements where we limited the number of concurrently processed requests at the server to 1, and subsequently used the size of an unbounded queue of the waiting requests as a metric for the controller. Such a setup better reflects application servers with a fixed number of worker threads. However, as we did not observe significant differences in comparison to the processing and queuing delay metric, we only show the results for the processing and queuing delay metric in this section, which also reflect our results for the queuing size metric.

5.2 Test setup

For the evaluation, we measure how our controller works for a CPU bound service, as this has been indicated by our industrial partner to be the limiting factor. Our service performs a simple calculation—prime factorization—that requires about 100 ms of processing time. An increasing concurrency rate also increases the processing delay for each individual request. The hardware setup consists of six standard PCs (Personal Computers) running on Ubuntu 8.04 server edition. The first PC hosts the server, while the remaining PCs act as load generators. The server is implemented as Java application using Jetty to provide HTTP access. The clients are also implemented in Java with one thread per simulated client. Each load generator is responsible for the simulation of multiple clients. Due to the fact that we test a CPU bound service, local network performance is not an issue.

As all clients run on the same local network and as multiple clients run on the same host, the propagation delay between individual clients is low. To account for the fact that dead time has a considerable influence on the performance of a controlled system, we implemented *dead time simulation* into the distributed broadcast channel used to propagate control information to clients. This simulation works by enabling a client to hold feedback information for a particular amount of time before forwarding this information to the next client. This allows to simulate different network conditions, such as latency on wide area networks (WANs).

5.3 Response to request change

First, we evaluate how fast our system reacts to sudden changes of the request rate. The evaluation was done in two distinct steps depicted in Figure 7.

In the "No Control" curve the behavior of the uncontrolled system is shown and in the "PID Controller" curve our system is extended with a PID controller that provides closed-loop control. In both cases we double the amount of active clients and verify how the system reacts to this input change. In the "No Control" case, doubling the amount of clients also doubles the processing time at the server. In the "PID Controller" case, the controller tries to keep the response time stable at 1 000 ms. We can see a peak at the beginning, when the controller needs to find an optimal range at the start of the simulation, and during the change, when the controller needs to adapt group sizes to the new client sizes accordingly (see circles).

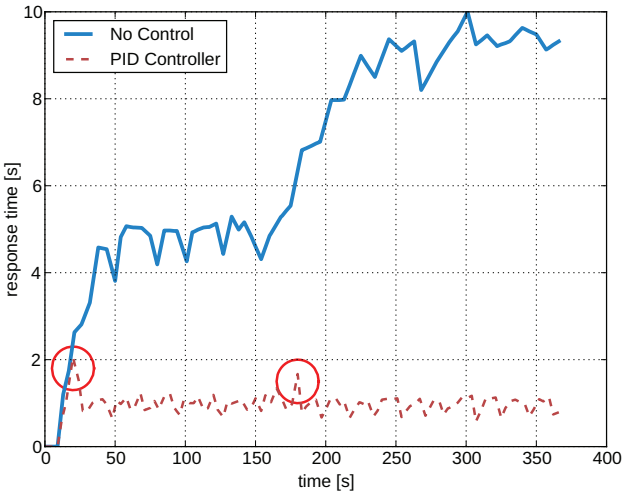

Figure 7: Response to system change: Without controller and with PID controller in place

Stability of the controller is a considerable factor, as it does not only affect the processing time at the server, but also the total throughput of the system. With a low stability caused by an unstable PID controller, there are time periods where no clients send requests, although the server is not fully loaded. In addition, also the case where too many requests are sent by clients can cause decreased throughput due to overload at the server. Generally, the goal of a controller is to reach the best performance possible with a given limit for the stability.

5.4 Auction scenario

For the second test we examine the behavior of our controller when used for the characteristic load of our auction scenario. In particular, we examine the performance during the peak load that resembles the bids of a first-price sealed-bid auction [15].

To model the peak load we distribute each client's request rate according to a Gaussian distribution with $\sigma = 50$ seconds and $\mu = 130$ seconds. There is a total of 1 000 clients and each client transmits a total of 30 requests. For each request at the server, the server executes a calculation that takes 100 ms. The set-point of the controller that reflects the acceptable processing delay is set to 1 000 ms.

The results of this evaluation for group-based rate control are presented in Figure 8. In the first measurement we examine the response time when no rate control is used, i.e., not even open-loop. The maximum response time peaks at around 10 000 ms. The reason for the plateau at 10 000 ms is given by the fact that the concurrency of each client is limited to 1 and that thus for 100 concurrent requests for an operation that takes 100 ms the total average will not exceed 10 000 ms. In the next three measurements we examine the behavior of our PID controller for different amounts of dead time.

In the first case, the simulated dead time is near zero. The actual dead time is non-zero due to inevitable delays in our

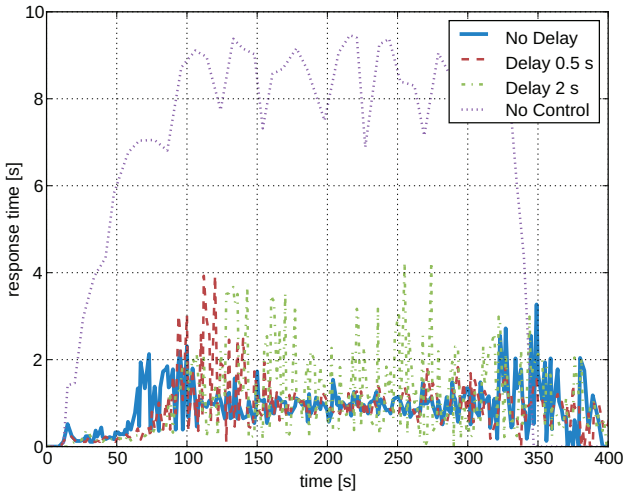

Figure 8: Auction scenario with: PID controller, group-based control, average process delay metric

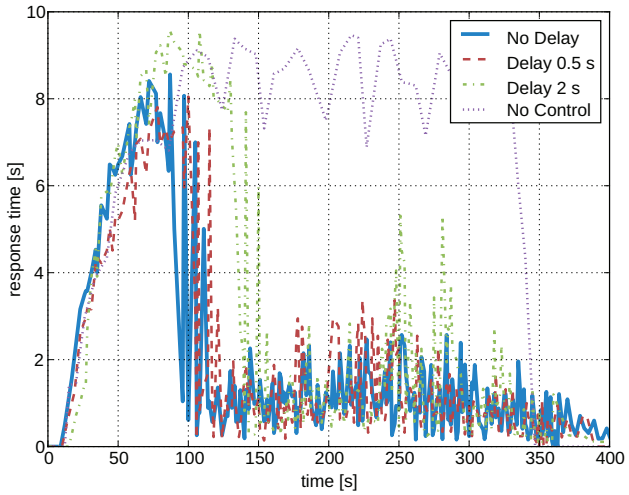

Figure 9: Auction scenario with: PID controller, interval-based control, average process delay metric

broadcast channel. In the second case we simulate a dead time of 500 ms per level of the broadcast tree and in the third case a dead time of 2 000 ms. In each of the cases there is a total of 3 levels in a quinary (5-ary) tree. While the controller is able to constrain the processing delay in the first case, in the second case it takes some time until the system is stable. In the third case we see a relatively large oscillation during the whole test, suggesting that the dead time is too high for our controller. In a real-world application scenario such propagation delays between nodes are considerably lower. If we assume an propagation delay of 50 ms, a total of 30 levels would yield a maximum propagation delay of 1500 ms, comparable to the maximum propagation delay of the simulation with a dead time of 500 ms. A quinary tree with 30 levels would support a theoretical maximum of $1.16 \cdot 10^{21}$ nodes, more than anyone would need in practice.

During the evaluation of group-based rate control we observed that the limited granularity of possible group sizes can have negative impact on the performance of the system. Especially when large groups are used, the possible group sizes that can be declared using a single divisor and a single modulo value often considerably differ from the manipulated variable calculated by the controller. As mitigation strategy, we enhanced group-based control to use lists of divisors and modulo values, instead of two single values. Clients matching a divisor/modulo combination in the list are allowed to transmit information. This allows for a more fine-grained specification of group-sizes, and thereby for a better stability of the controller.

In Figure 9 we show the results for interval based rate control. While the output of interval-based control is stable after the initial peak, variations are slightly larger than with group-based rate control. The stable behavior in Figure 9 is achieved by limiting the maximum difference between two consecutive rate control parameters calculated and broadcast by the controller. The respective limit is automatically derived based on the estimated amount of clients and the measured average time per operation. Due to this limitation the controller requires some time until it can reduce the load during steeply increasing client request rates.

5.5 Comparison

In this section we compare the effectiveness of our approach with two alternative approaches found in related work (Section 7) that work without distributed feedback channels.

In the first approach in Figure 10 we use a *piggyback* strategy where we embed rate control feedback in HTTP responses. When an HTTP client sends an HTTP request to a server, the server includes rate control information in the HTTP response. The results show that the piggyback approach is unstable and leads to oscillation. A major cause for this problem are variable dead times and the fact that control information at clients cannot be updated between individual requests of a client. Thus, the system is only able to provide request rate control, but not admission control as our distributed feedback channel.

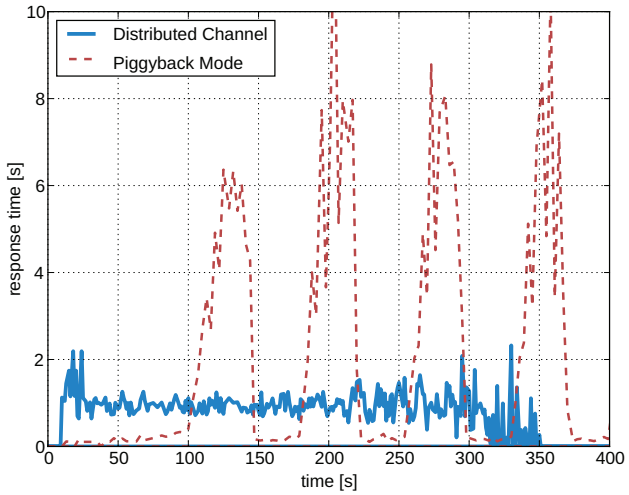

Figure 10: Comparison between distributed feedback channel and piggyback approach

In the second approach in Figure 11 we use a *reject* strategy similar to the control-theoretic approach by Abdelzaher et al. [1] where we reject requests at the server instead of the client. When a client is not able to establish a connection to the server, we use an exponential backoff strategy similar to TCP's retransmission timer [19] without the adjustments using the Smoothed Round Trip Time (SRTT) [21]. The original approach is not directly applicable to our application scenario, as implementation of a *reject/request-ratio* would not allow us to reject requests on a per-client basis. Instead, we reject requests at HTTP level instead of TCP level, as our existing decoupling mechanism depends on the client IDs. As a consequence, the performance of the *reject* approach in our evaluation is restricted by our required adaptations.

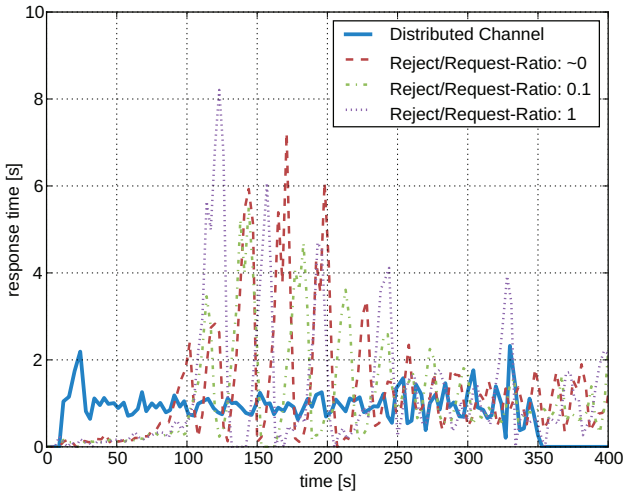

Figure 11: Evaluation of reject/request-ratios

With the *reject* strategy the results depend on the ratio between the cost of accepting a request and the cost of rejecting a request. Figure 11 depicts the results for different ratios between accepting requests and rejecting requests. In the first case processing a request takes 100 ms, while we do not artificially delay the time for rejecting a request. In the additional two cases we increase the time for rejecting a request to 10 ms and 100 ms to simulate a reject/request-ratio of 0.1 and 1 respectively. While the results show that the stability does not significantly differ between different reject/request-ratios, there is a considerable difference in the throughput. While our distributed feedback channel is able to achieve a throughput of 8.6 requests per second, the throughput in the best *reject* approach is only 7.0 requests per second. When increasing the reject/request-ratio the throughput declines further to 5.0 and 2.9 requests per second accordingly. Therefore, the benefits of our distributed feedback channel are primarily relevant in scenarios with a high reject/request-ratio, where the cost of rejecting a request is not negligible. Abdelzaher et al. have shown that this is the case in Web-server end systems, where they measured a reject overhead of 1.1 ms per request, while processing a request for a zero-sized URL took 1.604 ms [1]. A more detailed examination of the reject/request-ratio's influence is given in the next section.

6. LIMITATIONS OF OUR APPROACH

The applicability of our approach for certain application scenarios can be determined by comparing the effort required at the server for processing requests with the effort required for rejecting requests and with the typical ratio between average load and peak load. For comparison, we assume a simple system that does not use distributed load control: If a request cannot be processed, it is rejected by the system and the client does not try to retransmit the request.

First, we examine the theoretical limitations due to the cost difference between accepting and rejecting requests. We then proceed by discussing the implications of these trade-offs on real-world application scenarios.

6.1 Theoretical limitations

We denote the maximum request rate that the system can sustain without rejecting requests as $r_{process}$, the factor between the system load under normal operation and the system load during a peak as $factor_{peak}$, and the factor between the cost of accepting a request and rejecting a request as $factor_{request}$ ($= \frac{cost\ for\ request}{cost\ for\ reject}$). We then calculate an upper bound for the server's effort of rejecting requests when not all requests can be processed by calculating $r_{process} \cdot \frac{factor_{peak}}{factor_{request}}$: We have $factor_{peak}$ as many requests, but rejecting each request only takes $\frac{1}{factor_{request}}$ of the time it would take to process these requests.

If $factor_{peak}$ is equal to $factor_{request}$ the system is not able to handle any business requests during peak load, as rejecting requests takes up all available resources. Thus, in this case our distributed broadcast channel would be able to double the effective capacity of the system, as the resources previously used for load control can now be used for request processing. With a higher $factor_{request}$ the advantage of our distributed broadcast channel increases, while with a lower $factor_{request}$ the advantage decreases. For example, if $factor_{request}$ is 100, but $factor_{peak}$ is only 10, the upper bound of rejection costs during the peak load is about $\frac{1}{10}$ of the system's overall performance. In this case, the advantage of *outsourcing* the broadcast channel is rather limited.

6.2 Real-world applications

In a real world system the examination of our system's advantages are more complex than the calculations for the simplified model given in Section 6.1: (i) Our model does not deal with retransmission of requests. If a request is rejected, the client would need to try a retransmission at some point. This increases the advantage of our approach. (ii) We assume that each request is transmitted independent of other requests. In reality, a client can use information from earlier requests to optimize the rate for later requests. This in turn decreases the advantage of our approach, but may lead to unstable behavior as shown in Figure 10.

While our approach cannot prevent clients from *trying* to cheat by ignoring control information, the server can easily verify if individual clients act according to the broadcast control information. If clients continuously violate those rules, the server can ban those clients from further participation in the auction by rejecting requests without processing them. In addition, to prevent clients from injecting bogus messages into the network, clients will only forward messages digitally signed by the server.

7. RELATED WORK

Our system is not the first effort to dynamically control request rates of clients on higher level application layers. For example, several papers [1, 13, 20] describe approaches that use controllers to react to system load and thereby manage to improve the system's overall performance. The main difference to our work is that we do not take actions on the server side (such as rejecting requests or content adaption), but rather try to solve the problem directly at the source—at the client side. This section gives a brief overview of relevant and related work. First, we discuss load control mechanisms in standard Internet protocols. Afterwards, we proceed to specialized solutions for particular types of Web applications.

Transmission Control Protocol.

TCP based servers [21] are able to implicitly control the transmission rate of clients by only allowing a particular amount of parallel connections and by using a *window* to restrict the amount of data waiting to be processed. If SYN packets of clients are dropped, clients use an exponential backoff approach [18, 19] to time retransmissions, which can further be improved with a PI controller [14]. In our application scenario, TCP based rate control is not fully applicable as our goal is to prevent overload by filtering requests directly at the clients.

Control-theoretic approaches.

The goal of RacingSnail [7] is to monitor and optimize the performance of existing systems. The authors state that each system exhibits an optimal performance at a specific request rate. With a lower request rate the system is underloaded, and with a higher request rate the system is overloaded. RacingSnail is implemented using a blackbox approach where one module is responsible for measuring and analyzing the server's performance, and another module is responsible for slowing down client request rates. Compared to our adaptive rate control approach the application scenario and the solution approach are different. While our system deals with scalability aspects in case of large amounts of clients distributed over the Internet, the application scenario of RacingSnail is request rate control within more self-contained systems. As a consequence, our system mostly contributes with techniques required to facilitate request rate control for large amounts of clients, while RacingSnail deals with issues such as how existing blackbox services can be slowed down.

Abdelzaher et al. [1] describe performance control of a Web server using classical feedback control theory. Another control-theoretic approach guaranteeing bounded and predictive response times by Web servers been based published by Lim et al. [13]. Chan et al. [4] propose a fuzzy PI controller to guarantee proportional delay differentiation on Web servers by providing a better service to a premium class of users. The main difference to the first two works is that we directly control the request rates at clients. The work of Chan et al. could complement our solution.

Self-adapting service level.

Philippe et al. describe an approach for an self-adapting service level [20] that allows some components in an application server to dynamically degrade or upgrade their level of service, thereby trading a lower service level for a better overall performance of the server. This could complement our solution in general, but is not applicable for our specific application scenario.

In addition to the discussed publications there is a number of publications [2, 5, 11, 16, 23, 26, 27] that deal with admission control for Web server systems. Most existing systems adapt parameters at the server to influence the load produced by clients, e.g., by deciding which requests should be rejected. In contrast, our technique frees the server from rejecting individual requests, as we directly filter requests at the client.

8. CONCLUSION

This paper presents a new approach for adaptive load control and performance for first-price sealed-bid auctions. Our main contribution is the integration of (i) a distributed feedback channel to transmit control information from the server to the clients with (ii) decoupling strategies that allow to constrain client requests directly at the client side and (iii) a PID controller that adaptively controls the input parameters of those decoupling strategies to facilitate an optimal server utilization.

In comparison to established techniques, our system allows to control the quality of higher layer protocols without relying only on the load control mechanisms provided by lower layer protocols. Unlike related work, the servers in our system do not need to communicate with each single client individually. In particular, we can control request rates of clients even before a connection to a server is established, leading to a significant reduction in the required server-side resources. Our system does not only prevent clients from overwhelming the capacity of the servers, but allows to reduce the capacity required at the server-side infrastructure for applications that show temporary peaks in transmission rate or that can queue and send client requests in a single batch request, whereby data obsoleted by newer information can already be filtered at the client side.

Concluding, our approach provides viable means to manage high loads in first-price sealed-bid auctions without requiring large clusters able to cope with brief peak loads. Controlling transmission rates at clients decreases the number of required servers and makes more efficient use of available resources. A potential drawback is the necessity of a distributed feedback channel, which, however, could be approximated through similar infrastructures. As future work we identify the use of a Smith predictor [22, 30] to compensate for dead time due to delays and thereby stabilize the controller as well as an evaluation of fuzzy controllers [4] to verify if they are able to yield acceptable results in our system.

Acknowledgments

The authors would like to thank Markus Jung for the implementation of the work described in this paper. This work has been partially funded by the Austrian Federal Ministry of Transport, Innovation and Technology under the FIT-IT project TRADE (Trustworthy Adaptive Quality Balancing through Temporal Decoupling, contract 816143, `http://www.dedisys.org/trade/`).

9. REFERENCES

[1] T. F. Abdelzaher, K. G. Shin, and N. T. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.

[2] M. Andersson, J. Cao, M. Kihl, and C. Nyberg. Admission control with service level agreements for a web server. In M. H. Hamza, editor, *EuroIMSA*, pages 275–280. IASTED/ACTA Press, 2005.

[3] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA*, pages 246–261, 2002.

[4] K. H. Chan and X. Chu. Design of a fuzzy PI controller to guarantee proportional delay differentiation on web servers. In M.-Y. Kao and X.-Y. Li, editors, *AAIM*, volume 4508 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 2007.

[5] D. Dyachuk and R. Deters. Transparent admission control and scheduling of e-commerce web services. In J. Filipe and J. A. M. Cordeiro, editors, *WEBIST (Selected Papers)*, volume 8 of *Lecture Notes in Business Information Processing*, pages 124–136. Springer, 2007.

[6] L. Froihofer and K. M. Goeschka. Balancing of dependability and security in online auctions. In *38th Int. Conference on Dependable Systems and Networks (DSN'08) (Supplementary volume)*, 2008.

[7] M. Goldstein, O. Shehory, R. Tzoref-Brill, and S. Ur. Improving throughput via slowdowns. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 11–20, New York, NY, USA, 2010. ACM.

[8] T. Hägglund and K. J. Åström. Revisiting the Ziegler-Nichols tuning rules for PI control. *Asian Journal of Control*, 4(4):364–380, Dec. 2002.

[9] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In O. Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 1999.

[10] J. O. Henriksen, R. M. O'Keefe, C. D. Pegden, R. G. Sargent, and B. W. Unger. Implementations of time (panel). In D. W. Jones, editor, *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 409–416, New York, NY, USA, 1986. ACM.

[11] M. Kihl, A. Robertsson, A. Andersson, and B. Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web*, 11(1):93–116, 2008.

[12] M. M. Kokar, K. Baclawski, and Y. A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[13] S. S. Lim, C. Lee, C. W. Ahn, C. G. Lee, and K. H. Park. An adaptive admission control mechanism for a cluster-based web server system. In *IPDPS*. IEEE Computer Society, 2002.

[14] C.-H. Lung and O. W. W. Yang. Evaluation of an adaptive PI rate controller for congestion control in wireless ad-hoc/sensor networks. In *CSE (2)*, pages 597–602. IEEE Computer Society, 2009.

[15] R. P. McAfee and J. McMillan. Auctions and bidding. *Journal of Economic Literature*, 25(2):699–738, June 1987.

[16] P. J. Meulenhoff, D. R. Ostendorf, M. Zivkovic, H. B. Meeuwissen, and B. M. M. Gijsen. Intelligent overload control for composite web services. In L. Baresi, C.-H. Chi, and J. Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 34–49, 2009.

[17] S. Mogaki, M. Kamada, T. Yonekura, S. Okamoto, Y. Ohtaki, and M. B. I. Reaz. Time-stamp service makes real-time gaming cheat-free. In G. J. Armitage, editor, *NETGAMES*, pages 135–138. ACM, 2007.

[18] A. Mondal and A. Kuzmanovic. Removing exponential backoff from TCP. *Computer Communication Review*, 38(5):17–28, 2008.

[19] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), Nov. 2000.

[20] J. Philippe, N. D. Palma, F. Boyer, and O. Gruber. Self-adapting service level in java enterprise edition. In *Middleware 2009*, volume 5896 of *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin / Heidelberg, 2009.

[21] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[22] P. Sourdille and A. O'Dwyer. A new modified smith predictor design. In *ISICT*, volume 49 of *ACM International Conference Proceeding Series*, pages 385–390. Trinity College Dublin, 2003.

[23] P. B. Srinivas, S. Ramanathan, and S. Singhal. Web2K: Bringing QoS to web servers. (HPL-2000-61), 2000. `http://www.hpl.hp.com/techreports/2000/HPL-2000-61.pdf`.

[24] G. Starnberger, L. Froihofer, and K. M. Goeschka. Using smart cards for tamper-proof timestamps on untrusted clients. In *Availability, Reliability and Security, 2010. ARES '10. International Conference on*, Kraków, Feb. 2010.

[25] Transaction Processing Performance Council. TPC Benchmark^{TM}W (Web Commerce), 2002. `http://www.tpc.org/tpcw/`.

[26] T. Voigt and P. Gunningberg. Adaptive resource-based web server admission control. In *ISCC*, pages 219–224. IEEE Computer Society, 2002.

[27] T. Voigt and P. Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In G. Carle and M. Zitterbart, editors, *Protocols for High-Speed Networks*, volume 2334 of *Lecture Notes in Computer Science*, pages 50–68. Springer, 2002.

[28] C. K. Yeo, B. S. Lee, and M. H. Er. A survey of application level multicast techniques. *Computer Communications*, 27(15):1547–1568, 2004.

[29] J. G. Ziegler and N. B. Nichols. Optimum settings for automatic controllers. *Transactions of the ASME*, 64:759–768, 1942.

[30] H. Ziyuan, Z. Lanlan, and F. Minrui. Robust auto tune smith predictor controller design for plant with large delay. In K. Li, M. Fei, G. W. Irwin, and S. Ma, editors, *LSMS (1)*, volume 4688 of *Lecture Notes in Computer Science*, pages 666–678. Springer, 2007.