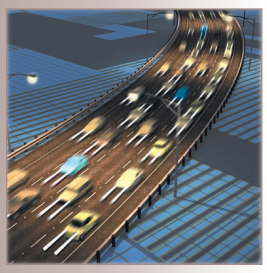

Analysis of Caching and Replication Strategies for Web Applications

Developers often use replication and caching mechanisms to enhance Web application performance. The authors present a qualitative and quantitative analysis of state-of-the art replication and caching techniques used to host Web applications. Their analysis shows that selecting the best mechanism depends heavily on data workload and requires a careful review of the application's characteristics. They also propose a technique for Web practitioners to compare different mechanisms' performance on their own.

Swaminathan Sivasubramanian, Guillaume Pierre, and Maarten van Steen
Vrije Universiteit

Gustavo Alonso
ETH Zurich

Web sites can be slow for many reasons, but the most prevalent one is the dynamic generation of Web documents. Modern Web sites such as Amazon.com and Slashdot.org don't deliver static pages — they generate content on the fly each time they receive a request, customizing their pages for each user. Clearly, generating a Web page in response to every request takes more time than simply fetching static HTML pages from a server. Dynamic generation of a Web page typically requires issuing one or more queries to a database, so access times to the database can easily get out of hand when the request load is high.

Industry and academia have developed several techniques to overcome this problem. The most straightforward one is *Web page caching*, in which (fragments of) the HTML pages the application generates are cached to serve future requests.[1] Content-delivery networks (CDNs) such as Akamai do this by deploying edge servers around the Internet to locally cache Web pages and then deliver them to clients. By delivering pages from edge servers located close to the clients, CDNs reduce each request's network latency. Page-caching techniques work well if the same cached HTML page can answer many requests to a particular Web site. These techniques have proven to be effective,[1,2] but with the growing drive toward personalized Web content, generated pages tend to be unique for each user, thereby reducing the benefits of page-caching techniques.

Page caching's limitations have triggered the CDN and database research community to investigate new approaches for

 1089-7801/07/$25.00 © 2007 IEEE

scalable Web application hosting. We can classify these approaches broadly into four techniques: application code replication,[3] cache database records,[4,5] cache query results,[6] and entire database replication.[7,8] Although numerous research efforts have focused on these approaches, very few works have analyzed their pros and cons and examined their performance. This article's objective is to present an overview of various scalability techniques and compare and analyze their features and performance. To do so, let's first consider some well-known scaling techniques for Web applications.

Techniques to Scale Web Applications

Instead of caching the dynamic pages generated by a central Web server, various techniques aim to replicate the *means* of generating pages over multiple edge servers. Despite their differences, these techniques often rely on the assumption that applications don't require strict transactional semantics for their data accesses (as, for example, banking applications do). They typically provide "read-your-writes" consistency, which guarantees that when an application at an edge server performs an update, any subsequent reads from the same edge server will return that update's effects (and possibly others). Scalable techniques that provide transactional semantics are beyond this article's scope.

Edge Computing

The simplest way to generate user-specific pages is to replicate the application code at multiple edge servers and keep the data centralized (see Figure 1a). This technique is the heart of the edge computing (EC) products at Akamai and ACDN.[3] EC lets each edge server generate user-specific pages according to context, session, and information stored in the database, thereby spreading the computational load across multiple servers. However, this data centralization can also pose several problems. First, if the edge servers are located worldwide, each data access incurs wide-area network (WAN) latency; second, the central database quickly becomes a performance bottleneck because it needs to serve the entire system's database requests. These properties restrict EC's use to Web applications that require relatively few database accesses to generate content.

Data Replication

The solution to EC's database bottleneck problem is to place the data at each edge server so that generating a page requires only local computation and data access. Database replication (REPL) techniques can help here by maintaining identical copies of the database at multiple locations.[7–9] However, in Web environments, database replicas are typically located across a WAN, whereas most REPL techniques assume the presence of a local-area network (LAN) between replicas. This can be a problem if a Web application generates many database updates. If this happens, each update must be propagated to all the other replicas to maintain the replicated data's consistency, potentially introducing a huge network traffic and performance overhead. In our study, we designed a simple replication middleware solution that serializes all updates at an origin server and propagates them to the edges in a lazy fashion. The edge servers answer each read query locally.

Content-Aware Data Caching

Instead of maintaining full copies of the database at each edge server, content-aware caching (CAC) systems cache database query results as the application code issues them. In this case, each edge server maintains a partial copy of the database, and each time the application running at the edge issues a query, the edge-server database checks if it contains enough data locally to answer the query correctly. This process is called a *query containment check*. If the containment check result is positive, the edge server can execute the query locally; otherwise, it must be sent to the central database. In the latter case, the edge-server database inserts the result in its local database so that it can serve future identical requests locally. To insert cached tuples into the edge database, the caches create insert/update queries on the fly. Examples of CAC systems include DBCache[5] and DBProxy.[4]

CAC stores query results in a storage-efficient way — for example, the queries "`select * from items where price < 50`" (Q1) and "`select * from items where price < 20`" (Q2) have overlapping results. By inserting both results into the same database, the overlapping records are stored only once. Another interesting feature of CAC is that once Q1's results are stored, Q2 can execute locally, even though that particular query hasn't been issued before. In this case, the query containment procedure recognizes that Q2's results are contained in Q1's results. CAC systems are beneficial when the application's query workload has range queries or queries with multiple predicates (for example, to find items that satisfy <*clause*1> OR <*clause*2>).

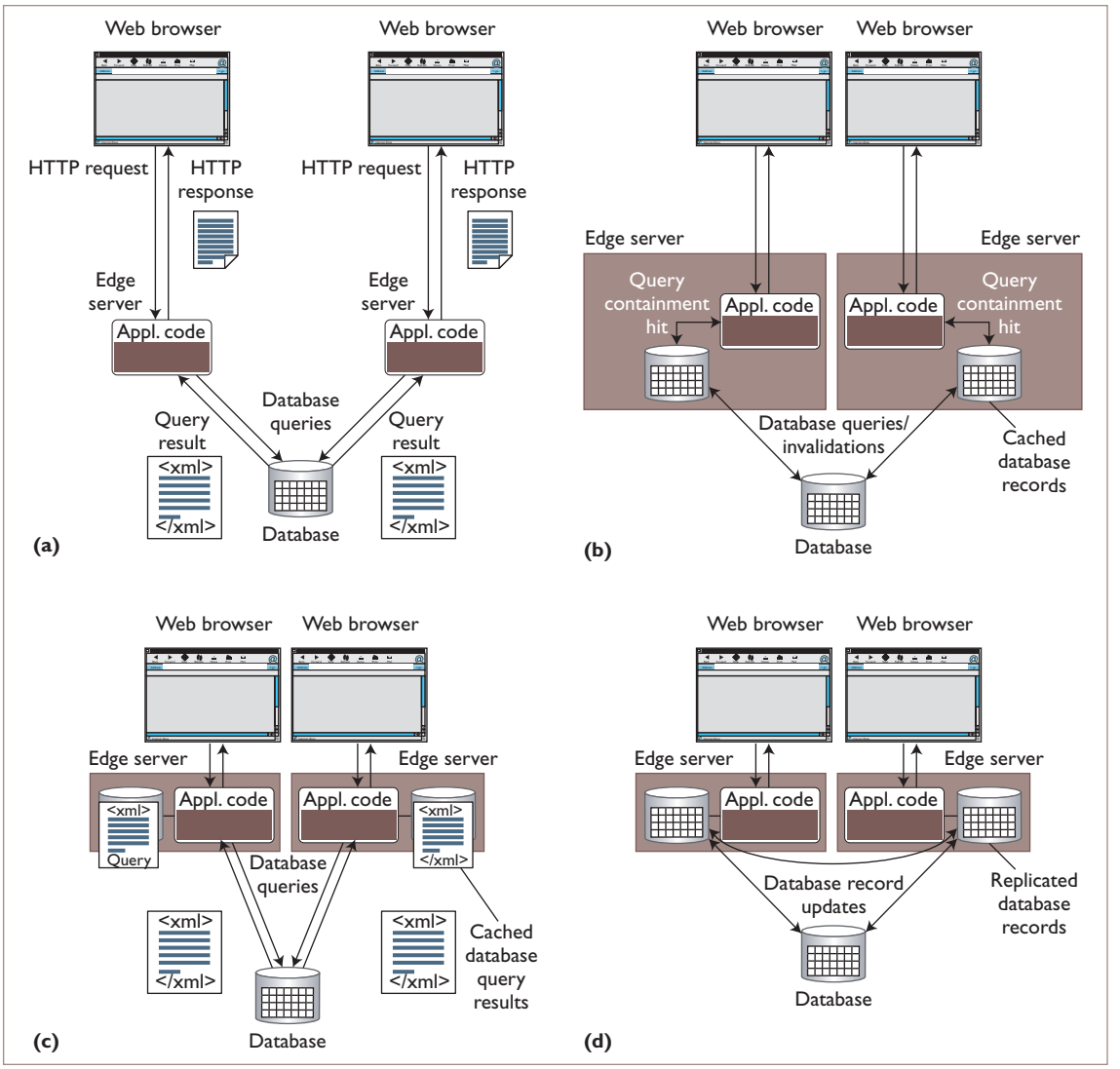

Figure 1. Solutions for scalable Web hosting. We compare (a) edge computing, (b) content-aware caching, (c) content-blind caching, and (d) data replication.

Typically, a query containment check is highly computationally expensive because it must check the new query with all previously cached queries. To reduce this cost, CAC systems exploit the fact that Web applications often consist of a fixed set of read and write query templates. A query template is a parameterized SQL query whose parameter values pass to the system at runtime. Use of query templates can vastly reduce the query containment's search space because the edge-server database must check each incoming query only with a relatively small set of query templates. Using a template-based checking technique in our previous example, we might check Q1 and Q2 only with other cached instances of the template QT1, "`select * from items where price<?`", and not with instances of, for example, QT3, "`select * from items where subject=?`". However, this method can also reduce the cache hit rate. These systems often also use template-based mechanisms to ensure cache consistency. In CAC systems, the update queries always execute at the origin server (the central database server). When an edge server caches a query, it subscribes to receive invalidations of conflicting query templates. In our example, an update to change an item table's price will conflict with QT1.

Content-Blind Data Caching

An alternative to CAC is content-blind query caching (CBC). In this case, edge servers don't need to run a database at all. Instead, they store the

results of remote database queries independently.[6,10] CBC uses a method akin to template-based invalidation to maintain its cached results' consistency.[10] Because the query results aren't merged together in CBC, caching the answers to the queries Q1 and Q2 defined earlier would lead to storing redundant information. Also, the cache will have a hit only if the application (running at the edge) issues the same exact query again at the same edge server. This can potentially lead to suboptimal usage of storage resources and lower cache hit rates, but it has some advantages. First, the process of checking if a query result is cached or not is trivial in CBC and incurs very little computational load. Second, by caching query results as result sets instead of database records, the CBC system can return results immediately. In contrast, CAC pays the price of database query execution, which can increase the load on edge servers. Finally, inserting a new element into the cache doesn't require a query rewrite — rather, it merely stores objects.

Cache replacement is an important issue in any caching system because it determines which query results to cache and which ones to evict from the cache. An ideal cache replacement policy must take into account several metrics such as temporal locality, query cost, and the database's update patterns. Note that cache replacement in CBC is simple because each result is stored independently, and we can apply many popular replacement algorithms.[10] However, because CAC merges multiple query results, its replacement policy should ensure that a query result's removal doesn't affect other cached queries' results.

Performance Analysis

To quantitatively compare these four techniques, we evaluated their performance for two different applications: RUBBoS, a bulletin-board benchmark application that models Slashdot.org, and TPC-W, an industry-standard e-commerce benchmark that models an online bookstore such as Amazon.com.

The RUBBoS application's database consists of five tables containing information about users, stories, comments, submissions, and moderator activities. We filled the database with information for 500,000 users and 200,000 comments. The TPC-W benchmark consists of seven database tables filled with information on 100,000 items and 288,000 customers. For our experiments, we chose the open source PHP implementation of these benchmarks (http://jmob.objectweb.org/rubbos.html and http://pgfoundry.org/projects/tpc-w-php/). Both applications have very different data access characteristics: in a typical bulletin board, users are usually interested in the latest news, so the workload can exhibit high locality, but customer shopping interests can vary in a bookstore application, thereby leading to lower query locality. This can help us study different systems' behaviors for different data access patterns, but these benchmarks are by no means truly representative of actual workload.

In our experiments, we used emulated browsers (EBs) to generate the client workload for both benchmarks and conformed to the TPC-W specification for clients (www.tpc.org/tpcw/tpcw_ex.asp). We set the average think time (the amount of time an EB waits between receiving a response and issuing the next request) to 5 seconds. The user workload for RUBBoS had more than 15 percent of its interactions leading to updates. For TPC-W, we studied the performance for two kinds of workloads: browsing (95 percent browsing and 5 percent shopping interactions) and ordering (equal fraction of browsing and shopping interactions). We modified the client workload behavior such that the book popularity followed a Zipf distribution (with $\alpha = 1$), which appeared in a study that observed a major online bookstore's data characteristics.[11]

For our tests, we used two servers with dual-processor Pentium III 900-MHz CPUs and 1 Gbyte of memory. We used the Apache 2.0.49 Web server with PHP 4.3.6 in our edge server, PostgreSQL 7.3.4 for our database management system, and PgPool to pool database connections (http://pgfoundry.org/projects/pgpool/). We emulated a WAN between the edge server and the origin server by directing all network traffic to an intermediate router that ran the NIST Net network emulator (http://snad.ncsl.nist.gov/itg/nistnet/). This router delays the packets sent between different servers to simulate a realistic WAN. For the rest of this article, we refer to links via NIST Net with 50-Mbps bandwidth and 100-ms latency as WAN links, and 100 Mbps and zero latency as LAN links. These values are considerably optimistic because Internet bandwidth usually varies a lot and is affected by network congestion. We chose these values to model the best network conditions for a CDN built on an Internet backbone, but they're the least favorable conditions for showing any data caching or replication system's best performance. Figure 2 shows our experimental setup. We didn't emulate wide-area latency between clients and the edge

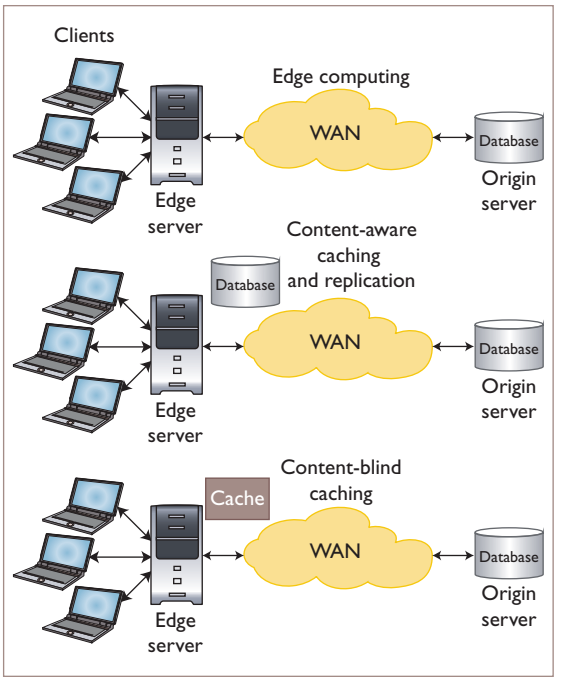

Figure 2. Architecture of the evaluated systems. This setup guarantees fair and reproducible performance comparison between the different systems.

server: this is a constant that would equally apply for all systems.

We started all our experiments with a cold cache. We warmed the system up for 20 minutes, after which we took measurements for a period of 90 minutes. Because we didn't want the effect of cache replacement algorithms to affect CAC and CBC performance, we didn't constrain the edge server's storage capacity — that is, we didn't restrict the cache repository's size. At the outset, this might look advantageous for caching systems, but in our experiments, we found that the amount of disk space required for cache repository was, at most, only 20 percent of the entire database. In all our experiments, we varied the request load (expressed in terms of the number of active client sessions) and measured the end-to-end *client latency*, which is the sum of *network latency* (the time the request spends traversing the WAN) and *internal latency* (the time the request spends generating query responses and composing the subsequent HTML pages).

Performance Results

Figure 3 shows our results. For RUBBoS, CBC performed the best in terms of client latency (except under low loads) whereas EC performed the worst. The reason for CBC's superior performance with RUBBoS was twofold. First, RUBBoS's workload exhibits high temporal locality (yielding a cache hit ratio of up to almost 80 percent) thereby avoiding WAN latency. Second, for CBC, the query execution latency incurred in generating a query response was much lower than that of REPL (or CAC) because the caching system avoided database query planning and execution latency. This let CBC sustain a higher load than REPL and CAC. EC performed worse than the other architectures because each data access incurred a WAN latency, and a single-origin server handled all requests. However, REPL performed marginally better than CBC during low loads because each query was answered locally thereby avoiding any WAN latency. Moreover, during low loads, the internal latency incurred in generating a query response was lower than the network latency incurred in answering a query.

In our experiments, CBC and CAC had almost the same hit ratio, despite the fact that CAC's merged storage theoretically allows more hits than CBC. The reason was that CAC's merged storage is most beneficial for workloads with many range queries and queries with multiple predicates. However, in RUBBoS, most queries are exact lookup queries, which don't benefit from the flexibility that CAC's query containment tests offer. We also attributed the latency increase for CAC to the increased overhead of query containment, cache management (inserting and invalidating caches), query planning, and execution.

EC performed the worst for TPC-W, whereas REPL performed the best. In this case, CBC and CAC perform relatively poorly because the TPC-W benchmark workload exhibits poor temporal locality, which yielded a hit ratio of at most 35 percent in our experiments. REPL performed better because the edge servers can answer each read query locally. CBC performed better than CAC because the former's cache hit ratio was only marginally lower compared to the latter. Again, this was because TPC-W doesn't fully exploit CAC's query containment features. Due to space constraints, we won't present detailed discussions of our results here. Extensive discussions and more results for multiple edge servers using weak consistency mechanisms appear elsewhere.[10]

Discussion

From our experiments, we found no clear winner. For applications whose query workload exhibited

high locality (such as RUBBoS), CBC performed the best. CAC didn't perform as well as expected, mostly because the tested query workloads didn't fully exploit CAC's query containment features. For applications that have a predominant load of such queries, we believe CAC will outperform CBC systems. For applications that exhibit poor locality (such as the TPC-W benchmark), data-replication schemes perform better than CBC. We conclude that no single solution performs best for all Web applications and workloads.

Choosing the Right Strategy

Because different techniques are optimal for different applications, Web designers should choose them after carefully analyzing their Web application's characteristics. In general, the best strategy is the one that minimizes the application's end-to-end client latency. The end-to-end latency is affected by parameters such as the cache hit ratio (page cache, CAC, or CBC), application server execution time, and database query execution time. (A detailed description of a model for estimating a multitiered Web application's end-to-end latency appears elsewhere.[12]) Although we can measure parameters such as execution time via server instrumentation and log analysis, measuring the cache hit ratio for strategies such as CBC and CAC is harder. Ideally, we want to estimate the possible hit ratio of different caching strategies without having to run each of them. To this end, we propose the concept of *virtual caches* (VCs).

A VC behaves just like a real cache except that it stores only metadata, such as the list of objects in the cache, their sizes, and invalidation parameters — objects themselves aren't stored. By applying the same operations as a real cache, a VC can estimate the hit rate a real cache would offer with the same configuration, but because the VC stores only metadata, it requires less memory. To measure the hit ratio of a cache that can hold millions of data items, for example, the virtual cache's size would be on the order of a few megabytes. A system could use two such VCs to determine the effective hit ratio for CAC and CBC. (Note that to implement a virtual CAC, you must also implement the query containment checker into the VC along with simple put, get, and invalidation operations.) A Web administrator can thus determine the hit ratios that different caching techniques offer.

By definition, a VC's hit ratio should be the same as a real cache because they both perform the same operations. Our experiments with virtual GlobeCBC, fragment caches, and CAC also confirm this. Moreover, compared to static trace-driven analysis, a VC is more effective due to its ability to instantly measure cache hit ratios online. Once we obtain the hit ratios for different techniques and execution times for servers at dif-

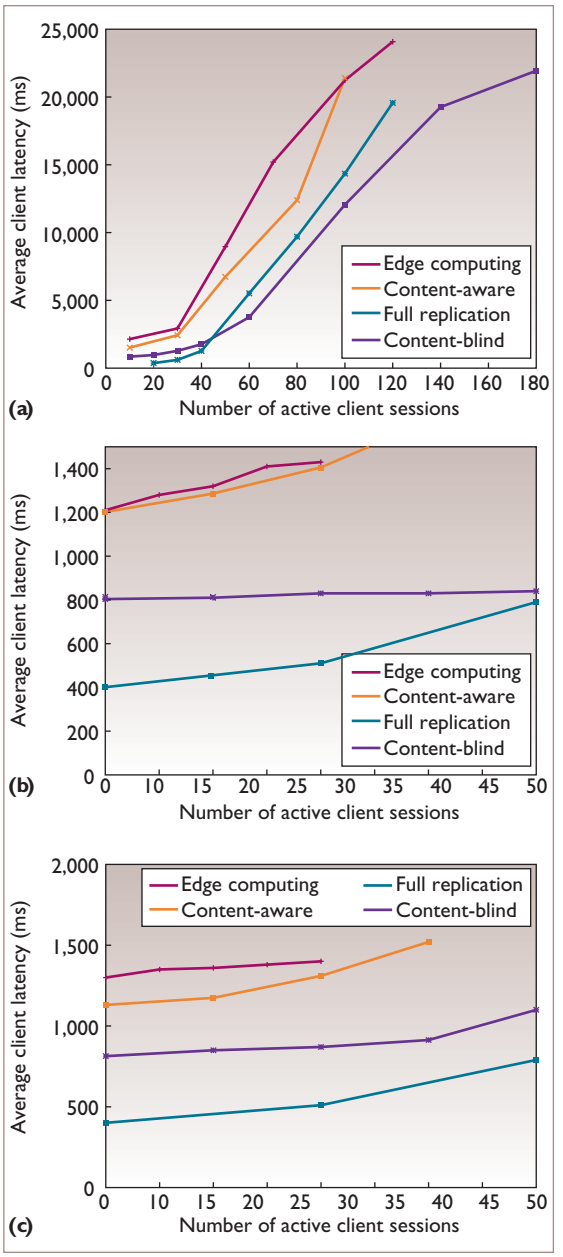

Figure 3. Performance results. (a) RUBBoS benchmark, (b) TPC-W browsing, and (c) TPC-W ordering. Content-based caching performs the best for RUBBoS whereas full database replication is the best for TPC-W.

ferent tiers, we can estimate the application's end-to-end latency if we were to deploy it with different techniques.[12] Subsequently, we can choose the technique that offers the least end-to-end latency to host the application.

Note that Web practitioners can do this selection process during the initial phases of application deployment and revise it periodically, if necessary. In such cases, different VCs must run (with the application) only during the decision-making periods. However, if the application experiences frequent workload changes, we envisage running these VCs continuously and performing adaptations more frequently.[12]

Currently, we're building and evaluating a prototype system that enables dynamic provisioning and reconfiguration of multitier Web applications. The prototype uses a combination of end-to-end analytical model and virtual caches to determine the optimal resource configuration for a given application.

References

1. J. Challenger, P. Dantzig, and K. Witting, "A Fragment-Based Approach for Efficiently Creating Dynamic Web Content," *ACM Trans. Internet Technology*, vol. 5, no. 2, 2005, pp. 359–389.
2. M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the Scalability of a Large Web-Based Shopping System," *ACM Trans. Internet Technology*, vol. 1, no. 1, 2001, pp. 44–69.
3. M. Rabinovich, Z. Xiao, and A. Agarwal, "Computing on the Edge: A Platform for Replicating Internet Applications," *Proc. 8th Int'l Workshop Web Content Caching and Distribution*, Springer-Verlag, 2003, pp. 57–77.
4. K. Amiri et al., "DBProxy: A Dynamic Data Cache for Web Applications," *Proc. Int'l Conf. Data Eng.*, IEEE CS Press, 2003, pp. 821–831.
5. C. Börnhvd et al., "Adaptive Database Caching with DBCache," *Data Eng.*, vol. 27, no. 2, 2004, pp. 11–18.
6. C. Olston et al., "A Scalability Service for Dynamic Web Applications," *Proc. Conf. Innovative Data Systems Research* (CIDR), *ACM Press*, 2005, pp. 56–69.
7. E. Cecchet, "C-JDBC: A Middleware Framework for Database Clustering," *Data Eng.*, vol. 27, no. 2, 2004, pp. 19–26.
8. C. Plattner and G. Alonso, "Ganymed: Scalable Replication for Transactional Web Applications," *Proc. 5th ACM/IFIP/USENIX Int'l Conf. Middleware*, Springer-Verlag, 2004, pp. 155–174.
9. C. Amza, A. Cox, and W. Zwaenepoel, "Conflict-Aware Scheduling for Dynamic Content Applications," *Proc. 5th Usenix Symp. Internet Technologies and Systems*, Usenix Assoc., 2003; www.usenix.org/events/usits03/tech/amza.html.
10. S. Sivasubramanian et al., *GlobeCBC: Content-Blind Result Caching for Dynamic Web Applications*, tech. report IR-CS-022, Vrije Univ., 2006.
11. E. Brynjolfsson, Y.J. Hu, and M.D. Smith, "Consumer Surplus in the Digital Economy: Estimating the Value of Increased Product Variety at Online Booksellers," *Management Science*, vol. 49, no. 11, 2003, pp. 1580–1596.
12. S. Sivasubramanian, G. Pierre, and M. van Steen, "Towards Autonomic Hosting of Multi-Tier Internet Applications," *Proc. Usenix/IEEE HotAC-I Workshop*, 2006; www.aqualab.cs.northwestern.edu/HotACI/program.html.

Swaminathan Sivasubramanian is a PhD candidate in the Computer Systems group at Vrije Universitiet, Amsterdam. His thesis research focuses on building middleware and techniques that enable scalable hosting of multitiered software systems. Sivasubramanian has an MS in computer engineering from Iowa State University and a BE in computer science from Anna University, India. He is a member of the IEEE and the ACM, and is a co-editor of *IEEE DSOnline*'s Web systems section. Contact him at swami@cs.vu.nl.

Guillaume Pierre is an assistant professor in the Computer Systems group at Vrije Universiteit, Amsterdam. His research interests focus on Web-based systems. Pierre has an MSc and a PhD in computer science from the University of d'Evry-val d'Essonne, France. He is a member of the IEEE and is an editorial board member for *IEEE DSOnline*. Contact him at gpierre@cs.vu.nl.

Maarten van Steen is a full professor in the Computer Systems group at Vrije Universiteit, Amsterdam, where he heads a research team developing large-scale distributed systems. His research interests also include peer-to-peer and gossip-based distributed systems. Van Steen has an MSc in applied mathematics from Twente University and a PhD in computer science from Leiden University. He is a senior member of the IEEE and a member of the ACM. Contact him at steen@cs.vu.nl.

Gustavo Alonso is a professor in the Department of Computer Science at the Swiss Federal Institute of Technology in Zurich (ETHZ). His research interests include Web services, grid and cluster computing, databases, workflow management, sensor networks, data streaming, pervasive computing, and dynamic aspect-oriented programming. Alonso has an MS and a PhD in computer science from the University of California, Santa Barbara. Contact him at alonso@inf.ethz.ch.