Chapter 5
Replicating for Performance: Case Studies

Maarten van Steen and Guillaume Pierre

Abstract In this chapter we take a look at the application of replication techniques for building scalable distributed systems. Unlike using replication for attaining dependability, replicating for scalability is generally characterized by higher replication degrees, and thus also weaker consistency. We discuss a number of cases illustrating that differentiation of replication strategies, for different levels of data granularity, is needed. This observation leads us to conclude that automated replication management is a key issue for future research if replication for scalability is to be successfully deployed.

5.1 Introduction

Building scalable distributed systems continues to be one of the more challenging tasks in systems design. There are three independent and equally important perspectives on scalability [20]:

- **Size scalability** is formulated in terms of the growth of number of users or data, such that there is no noticeable loss in performance or increase in administrative complexity.
- A system is said to be **geographically scalable** when components can be placed far apart without seriously affecting the perceived performance. This perspective on scalability is becoming increasingly important in the face of distributing a service across the Internet.
- **Administrative scalability** describes the extent to which a system can be put under the control of multiple administrative organizations without suffering from performance degradation or increase of complexity.

In this chapter, we will concentrate on size and geographical scalability, in particular in relation to the perceived performance of a system. More specifically, we are interested in scalability problems that manifest themselves through performance degradation. To keep matters simple, in the following we will refer to scalability in only this more narrow context.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 73–89, 2010.

To address scalability problems, there are essentially only two techniques that we can apply. Following the terminology as proposed in [5], we can **partition** the set of processes and the collection of data those processes operate on, and spread those parts over different nodes of the distributed system. An excellent example of where this scaling technique has been successfully applied is the Web, which can be viewed as a huge, distributed information system. Each Web site is responsible for handling its own part of the entire data set, allowing hundreds of millions of users to access the system simultaneously. As we will discuss later, numerous sites need further partitioning as a single machine can not handle the stream of requests directed to them.

Another illustrative example of where partitioning has been successfully applied is in the Internet's Domain Name System. By October 2008, the entire name space had been partitioned across an estimated 11.9 million servers[1]. These servers collaborate in resolving names, and in such a way that many requests can be handled simultaneously. However, an important reason why DNS generally performs so well, is also because much of its data has been **cloned**, or more formally, **replicated**.

Cloning processes and associated data is useful for addressing geographical scalability problems. The principle is simple: by placing services close to where they are needed, we can reduce performance degradation caused by network latencies, and at the same time by placing a service everywhere it is needed, we address size scalability by dividing the load across multiple servers. In the following, we shall often use the term replication instead of cloning.

A main issue with replication is that it requires each update to be carried out at each replica. As a consequence, it may take a while before all replicas are the same again, especially when updates need to be carried out at many replicas spread across a large network such as the Internet. More problematic is when multiple updates need to be carried out concurrently, as this requires global synchronization if we wish to guarantee that in the end the replicas are indeed the same. Global synchronization requires the execution of an agreement protocol. Such an execution is generally not scalable: too many parties may need to communicate and wait for results before an update can be finally committed. An important consequence is that if we apply replication as a scaling technique, then we generally need to compromise on consistency: copies cannot be kept the same at all time.

This observation is not new. For example, it is well known among architects of very large Web-based systems such as Amazon, Google, and eBay that scalability can be attained only by "embracing inconsistency"[2]. A keyword here is **eventual consistency**: in the absence of further updates, replicas will converge to the same state (see also [34]). Accepting eventual consistency as the best possible option is needed when dealing with cloned services. The problem is that there is no way that one can guarantee the combination of strong consistency, availability, and coping with partitionable networks at the same time. This so-called CAP conjecture was postulated by Eric Brewer in 2000 and proved correct two years later [9]. For large-

[1] `http://dns.measurement-factory.com/surveys/200810.html`

[2] eBay's Randy Shoup at his presentation at Qcon, London, 2008.

scale distributed systems, it simply means that one cannot guarantee full systemwide consistency of update operations unless we avoid cloning services.

Unfortunately, there are no general, application-independent rules by which we can specify to what extent inconsistencies can be tolerated. In other words, replication for scalability is inherently coupled to the access, usage, and semantics of the data that are being replicated. For example, caching in DNS generally works because name-to-address bindings are relatively stable, allowing caches to be refreshed only once every few hours. Such dependencies, in turn, have led to a myriad of solutions regarding replication strategies. In addition, determining the appropriate granularity of the data to be replicated turns out to be crucial.

In this chapter, we will take a closer look at replication as a scaling technique, and in particular consider those situations in which scalability can be achieved only if the replication degree is relatively large. Such replication is necessarily coupled to applications, but also requires that we can tolerate inconsistencies between replicas. For these reasons, we follow an approach by discussing several cases, each dealing in its own with inconsistencies. In particular, we will argue that in order to achieve performance, we need to automatically decide on (1) which data needs to be replicated, (2) at which granularity, and (3) according to which replication strategy.

To keep matters simple, we assume that updates are coordinated such that write-write conflicts will not occur. In effect, this means that concurrent updates are serialized such that all replicas will process all updates in the same order. This assumption is realistic: in many practical settings we see that potential conflicts on some data set are avoided by having a coordinator for that data set. This coordinator sees all write requests and orders them accordingly, for example, by first timestamping requests before they are passed on to replicas. Furthermore, we focus on scalable Web-based distributed systems, which makes it easier to compare the various trade-offs regarding replication for scalability. Note that many issues we bring up are also applicable to other types of distributed systems, such as large-scale enterprise information systems. We ignore replication for wireless distributed systems, including large-scale systems based on sensor networks, mobile ad hoc networks, and so on. These type of distributed systems are becoming increasingly important, but often require specific solutions when it comes to applying scaling techniques.

In the remainder of this chapter, we start with discussing the large variety of possible replication strategies in Section 5.2. This is followed by a discussion on the data granularity at which these strategies must be applied in Section 5.3. Different forms of consistency guarantees are discussed in Section 5.4, followed by replication management (Section 5.5). We come to conclusions in Section 5.6.

5.2 Replication Strategies

A **replication strategy** describes **which** data or processes to replicate, as well as **how**, **when**, and **where** that replication should take place. In the case of replication for fault tolerance, the main distinguishing factor between strategies is arguably *how* replication takes place, as reflected in a specific algorithm and implementation (see also [35]). Replication for scalability also stresses the *what*, *where* and *when*.

Moreover, where algorithms for fault-tolerance replication strategies are compared in terms of complexity in time, memory, and perhaps messages, the costs of a replication strategy employed for performance should be expressed in terms of *usage* of resources, and the trade-off that is to be made concerning the level of consistency.

The costs of replication strategies are determined by many different factors. In particular, we need to consider replica placement, caching versus replication, and the way that replicated content is updated. Let us briefly consider these aspects in turn (see also [30]), in order to appreciate replica management when performance is at stake.

5.2.1 Replica Placement

Replica placement decisions fall into two different categories: decisions concerning the placement of *replica servers*, and those concerning the placement of *replicated data*. In some cases, the decisions on server placement are irrelevant, for example, when any node in a distributed system can be used for replica placement. This is, in principle, the case with data centers where the actual physical location of a replica server is less important. However, in any distributed system running on top of a large computer network such as an intranet or the Internet, where latencies to clients and between servers play a role, server placement may be an important issue and precedes decisions on data placement.

In principle, server placement involves identifying the K out of N best locations in the network underlying a distributed system [22, 25]. If we can assume that clients are uniformly distributed across the network, it turns out that server placement decisions are relatively insensitive to access patterns by those clients, and that one need only take the network topology into account when making a decision. An obvious strategy is to place servers close to the center of a network [3], that is, at locations to which most communication paths to clients are short. Unfortunately, the problem has been proven to be NP-hard, and finding good heuristics is far from trivial [12]. Also, matters become complicated when going to more realistic scenarios, such as when taking actual traffic between clients and servers into account [10].

Once replica servers are in place, we have the facilities to actually place replicated data. A distinction should be made between client-initiated and server-initiated replication [31]. With server-initiated replication, an **origin server** takes the decision to replicate or migrate data to replica servers. An origin server is the main server from which content is being served and where updates are coordinated. This technique is typically applied in Content Delivery Networks (CDNs) [23], and is based on observed access patterns by clients.

Client-initiated replication is also known as client-side caching. The most important difference with server-initiated replication is that clients can, independently of any replication strategy followed by an origin server, decide to keep a local copy of accessed data. Client-initiated replication is widely deployed in the Web for all kinds of content [11]. It has the advantage of simplicity, notably when dealing with mostly-read data, as there is no need for global coordination of data placement. Instead, clients copy data into local caches based completely on their own access patterns. Using shared caches or cooperative caches [1], highly effective data replication and placement can be deployed (although effectiveness cannot be guaranteed, see [36]).

As an aside, note that caching techniques can be deployed to establish server-initiated replication. In the protocol for the Akamai CDN, a client is directed to a nearby proxy server. The proxy server, configured as a traditional Web caching server, inspects its local cache for the referred content, and, if necessary, first fetches it from the origin server before returning the result to the client [19].

5.2.2 Content Distribution

Once replicas are in place, various techniques can be deployed for keeping them up-to-date. Three different aspects need to be considered:

State versus function shipping: A common approach for bringing a replica up-to-date is to simply transfer fresh data and overwrite old data, with variations based on data compression or transferring only differences between versions (i.e., delta shipping). As an alternative to this form of passive replication, a replica can also be brought up-to-date by locally executing the operation that led to the update, leading to active replication [28]. This form of update propagation is known as function or operation shipping, and has proven to be an alternative when communication links are slow [17].

Pull versus push protocols: Second, it is important to distinguish between protocols that *push* updates to replica servers, or the ones by which updates are *pulled in* from a server holding fresher updates. Pushing is often initiated by a server where an update has just taken place, and is therefore generally proactive. In contrast, pulling in updates is often triggered by client requests and can thus be classified as on-demand. Combinations of the two, motivated by performance requirements, is also possible through leases by which servers promise to push updates until the lease expires [6].

Dissemination strategies: Finally, we need to consider which type of channels to use when delivering updates. In many cases, unicasting is used in the form of TCP connections between two servers. Alternatively, multicasting techniques can be deployed, but due to lack of network-level support we generally see these being used only at application level in (peer-to-peer) content delivery networks [37]. Recently, probabilistic, epidemic-style protocols have been developed as an alternative for content delivery [14, 7].

Clearly, these different aspects together result in a myriad of alternatives for implementing replication strategies. Note also that although such implementations could also be used for replicating for fault tolerance, emphasis is invariably on efficiently delivering content to replica servers, independently of requirements regarding consistency.

5.2.3 Strategy Evaluation

With so many ways to maintain replicas, it becomes important to compare and evaluate strategies. Unfortunately, this is easier said than done. In fact, it can be argued that a blatant omission in the scientific approach to selecting replication strategies is a useful framework for comparing proposals (although such an attempt has been made [13]). The difficulty is partly caused by the fact that there are so many performance metrics that one could consider. Moreover, metrics are often difficult if not impossible to compare. For example, how does one compare a replication strategy that results in low perceived latencies but which consumes a lot of bandwidth, to one that saves network bandwidth at the cost of relatively poor response times?

An approach followed in the Globule system (and one we describe below), is to make use of a general cost function (which is similar to a payoff or utility function in economics). The model considers m performance metrics along with a (nondecreasing) cost $c_k(s)$ of the k^{th} metric. The cost $c_k(s)$ is dependent on the deployed replication strategy s. Combined, this leads to a total replication cost expressed as

$$rep(s) = \sum_{k=1}^{m} w_k c_k(s)$$

where w_k is the (positive) weight associated with making costs $c_k(s)$. With this model, it becomes possible to evaluate and compare strategies, with the obvious goal to minimize the total costs of replication. Note that there may be no obvious interpretation in what the total costs actually stand for. Also, it is up to the designers or administrators of the system in which data are being replicated to decide on the weights. For example, in some cases it may be more important to ensure low latency at the cost of higher usage of bandwidth. Besides latency and network bandwidth, typical performance metrics include used storage, energy consumption, monetary costs, computational efforts, and the "cost" of delivering stale data.

5.3 Replication Granularity

We now take a closer look at a number of cases where replication is used to improve the scalability of a system. In all cases, the improvement comes from adapting the system in such a way that it can simultaneously support several replication strategies, and differentiate among these strategies for smaller units of data than before. Concretely, in our first example, we will demonstrate that supporting a replication strategy on a per-page basis for sites storing static Web pages leads to higher scalability and better performance. In our second example, this kind of differentiation and higher granularity will be shown to also benefit cloning of Web services. As a last example, we take a look at an extensive analysis of Wikipedia traces. The overall conclusion is that replicating for performance requires differentiating replication strategies for smaller data units than is presently common.

5.3.1 Example 1: Content Delivery Networks

An important class of large-scale distributed systems is formed by content delivery networks (CDNs) Internet. Specific content, such as a collection of Web pages, is serviced by what is known as an origin server. As mentioned before, an origin server is responsible for handling updates as well as client requests. Also, it provides replica servers with content to which client requests can then be redirected. The size of a typical CDN may vary between a few tens of servers to tens of thousands of servers.

In order to guarantee availability and performance, replication of content plays a key role in any CDN. Besides the general issues discussed above concerning where to place replicas and how to keep them up-to-date, it turns out that the granularity of the data to consider for replication is equally important. For example, applying a single replication strategy to an entire Web site leads to much worse performance than replicating each Web page separately according to a page-specific strategy. Furthermore, even for seemingly stable Web sites, we have found that access patterns originating from a site's clients, change enough to warrant continuous monitoring and adaptation of per-page replication strategies. We briefly report on one such study.

Pierre et al. [21] conducted experiments to examine to what extent differentiating replication strategies could make a difference in the overall performance of a Web site. To that end, they considered several sites consisting of only static Web pages. Experiments were conducted by choosing a single replication strategy for an entire site, as well as experiments in which each document, i.e. Web page, would be subject to its own replication strategy. In the experiments, clients were traced to their autonomous system (AS), measuring latency as well as bandwidth. In addition, they kept an accurate account of updates on documents. Using these data, a *what-if* analysis was performed using a situation in which so-called **edge servers** were assumed to be placed in the various ASes as sketched in Figure 5.1.

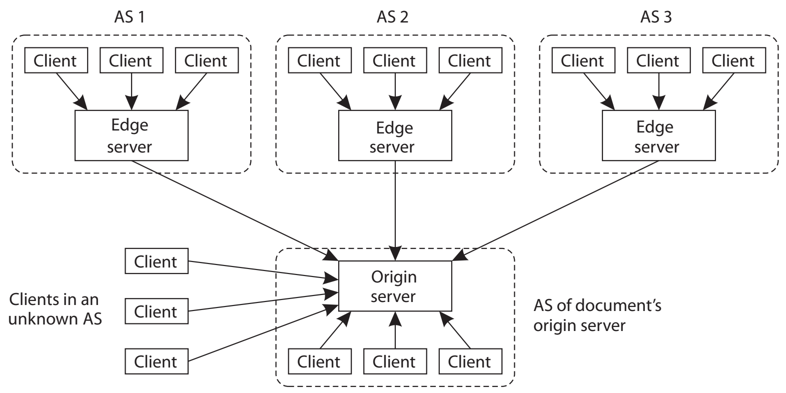


Fig. 5.1 Set-up of the CDN experiment with Web sites having static Web pages.

Table 5.1 Evaluated caching and replication strategies.

Abbr.	Name	Description
NR	No replication	No replication or caching takes place. All clients forward their requests directly to the origin server.
CV	Verification	Edge servers cache documents. At each subsequent request, the origin server is contacted for revalidation.
CLV	Limited validity	Edge servers cache documents. A cached document has an associated expiration time before it becomes invalid and is removed from the cache.
CDV	Delayed verification	Edge servers cache documents. A cached document has an associated expiration time after which the primary is contacted for revalidation.
SI	Server invalidation	Edge servers cache documents, but the origin server invalidates cached copies when the document is updated.
SUx	Server updates	The origin server maintains copies at the x most relevant edge servers; x = 10, 25 or 50
SU50+CLV	Hybrid SU50 & CLV	The origin server maintains copies at the 50 most relevant edge servers; the other edge servers follow the CLV strategy.
SU50+CDV	Hybrid SU50 & CDV	The edge server maintains copies at the 50 most relevant edge servers; the other edge servers follow the CDV strategy.

Clients for whom the AS could not be determined were assumed to directly contact the origin server when requesting a Web page. In all other cases, client requests would be assumed to pass through the associated AS's edge server. With this setup, the caching and replication strategies listed in Table 5.1 were considered.

The traces were used to drive simulations in which different strategies were explored, leading to what is generally known as a *what-if analysis*. As a first experiment, a simple approach was followed by replicating an entire Web site according to a single strategy. The normalized results are shown in Table 5.2. Normalized means that the best results were rated as 100. If bandwidth for a worse strategy turned out to be 21% more, that strategy was rated as 121. Stale documents returned to clients are measured as the fraction of the different documents requested.

What these experiments revealed was that there was no single strategy that would be best in all three metrics. However, if the granularity of replication is refined to the level of individual Web pages, overall performance increases significantly. In other words, if we can differentiate replication strategies on a per-page basis, optimal values for resource usage are much more easily approached. To this end, the cost-based replication optimization explained above was explored. Not quite surprisingly, regardless the combination of weights for total turnaround time, stale documents, and or bandwidth, making replication decisions at the page level *invariably* led to performance improvement in comparison to any global replication strategy. Moreover, it turned out that many different strategies needed to be deployed in order to achieve optimal performance. The study clearly showed that differentiating replication strategies at a sufficient level of granularity will lead to significant performance improvements. The interested reader is referred to [21] for further details.

Table 5.2 Normalized performance results using the same strategy for all documents, measuring the total turnaround time, the fraction of stale documents that were returned, and the total consumed bandwidth. Optimal values are highlighted for each metric.

	Site 1			Site 2		
Strategy	**Turnaround**	**Stale docs**	**Bandwidth**	**Turnaround**	**Stale docs**	**Bandwidth**
NR	203	**0**	118	183	**0**	115
CV	227	**0**	113	190	**0**	**100**
CLV	182	0.61%	113	142	0.60%	**100**
CDV	182	0.59%	113	142	0.57%	**100**
SI	182	**0**	113	141	**0**	**100**
SU10	128	**0**	**100**	160	**0**	114
SU25	114	**0**	123	132	**0**	119
SU50	102	**0**	165	114	**0**	132
SU50+CLV	**100**	0.11%	165	**100**	0.19%	125
SU50+CDV	**100**	0.11%	165	**100**	0.17%	125

5.3.2 Example 2: Edge-Server Computing

The previous example dealt only with static Web pages. However, modern CDNs require replication of dynamic pages and even programs [23, 24]. In general, this means that the architecture needs to be extended to what is known as an edge-server system. In such a system, the origin server is supported by several servers situated at the "edge" of the network, capable of running a (partial) replica of the origin server's database, along with programs accessing those data. There are essentially four different organizations possible, as shown in Figure 5.2.

The simplest organization is to clone only the application logic to the edge servers, along with perhaps some data. In this case, requests are processed locally, but if necessary, data are still fetched from the origin server. This scheme is typically used to address size scalability by reducing the computational load of the origin server. However, it will not be sufficient if performance costs are dominated by accesses to the database. If data has been copied to the edge server, it is assumed to be mostly read-only and any updates can be easily dealt with offline [24].

With full replication, the database at the origin server is cloned to the edge servers along with the logic by which data are accessed and processed. Instead of fully cloning the database, it is also possible to clone only those parts that are accessed by the clients contacting the particular edge server. In practice, this means that the origin server needs to keep track of access traces and actively decide which parts of the database require replication.

An alternative scheme is to deploy content-aware caching. In this case, the queries that are normally processed at the origin server are assumed to fit specific templates, comparable to function prototypes in programming languages. In effect, templates implicitly define a simple data model that is subsequently used to store results from queries issues to the origin server. Whenever a query addresses data

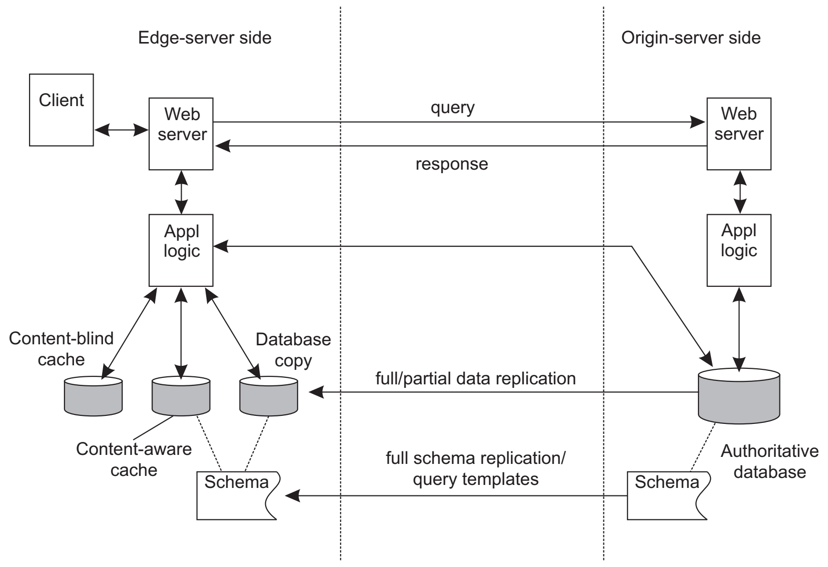


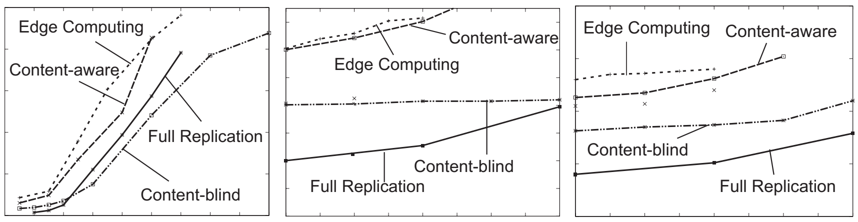
Fig. 5.2 Different ways to organize edge-server computing.

that has already been cached, the result can be retrieved locally. To illustrate, consider a query for listing all books by a given author of which the result is cached at the edge server from which the query originated. Then, when a subsequent query is issued for listing all books by that same author, but for a specific time frame, the edge server need, in principle, only inspect its local cache. This approach is feasible only if the edge server is aware of the templates associated with the queries.

Finally, edge servers can follow a content-blind caching scheme by which a query is first assigned a unique key (e.g., through hashing the query including its parameter values), after which the results are cached under that key. Whenever the exact same query is later issued again, the edge server can look up the previous result from its cache.

All these schemes require that edge servers register for updates at the origin server. In principle, this means that the cloned data at an edge server can be kept identical with that stored at the origin server. For scalability purposes, updates may not be propagated simultaneously to all edge servers, but instead an update is delivered only when there is a need to do so. This may happen, for example, because cloned data are requested by an edge server's client.

For scalability purposes, it is often convenient to let the edge server decide when updates are actually fetched from the origin server. In effect, an edge server will allow its clients to operate on stale data for some time. As long as clients are unaware that some updates have taken place, they will rightfully perceive the cloned data to be consistent. This approach toward delaying update propagation has been used in the file sharing semantics of the Coda distributed file system [15]. Problems



(a) Flashcrowd (b) Web browsing (c) Web ordering

Fig. 5.3 Performance of edge-server systems for different workloads. The x-axis shows increased client-side browsing activity, whereas the y-axis shows response times.

with such schemes occur when clients are allowed to switch between edge servers. In that case, it may happen that a client observes a version D_t of same data at one edge server, and later a *previous* version D_{t-1} of that data at another edge server. Of course, this is not supposed to happen. One solution that has been extensively explored in the Bayou system, is to support client-centric consistency models [32]. Simplifying matters somewhat, these models guarantee that data are kept consistent on a per-client basis. In other words, an individual client will always see the same or fresher data when issuing requests, regardless through which edge server it is accessing that data. However, guaranteeing client-centric consistency requires keeping track of what clients have accessed, which imposes an extra burden on edge servers.

Having a choice from different replication and caching strategies immediately brings up the question which strategy is the best one. Again, in line with the results discussed for simple CDNs, there is no single best solution. Sivasubramanian et al. conducted a series of trace-driven simulations using different workloads for accessing Web services. The results of these experiments are shown in Figure 5.3, of which a detailed report can be found in [29]. Again, what these studies show is that differentiating strategies is important in order to achieve higher performance. In addition, edge-server computing also puts demands on which edge servers clients are allowed to access in order to circumvent difficult consistency problems. Similar results have been reported by Leff et al. [18] in the case of Java-based systems, and distributed objects [8].

5.3.3 Example 3: Decentralized Wikipedia

As a final example, consider the increasingly popular Wikipedia system. This system is currently organized in a near-centralized fashion by which traffic is mostly directed to one of two major sites. Each site maintains a database of so-called wikis: a collection of (hypertext) pages marked up in the *wikitext* markup language. The Wikipedia system provides a Web-based interface by which a Wiki document is returned in HTML form for display in standard Web browsers.

A serious problem for the Wikimedia organization hosting the various wiki's is that the increase in traffic is putting a significant burden on their infrastructure. Being a noncommercial and independent organization means that financial support is

Table 5.3 Wikipedia workload analysis and impact for decentralization.

Type of page	% Requests	Strategy to follow
Pages that are read-only in practice and are mostly read (>75%) in default HTML format.	27.5%	HTML caching or replication with the degree of replication depending on the popularity of the page.
Pages that are almost read-only and have a significant fraction (>25%) of reads in alternate formats.	10.9%	Wikitext replication in combination with HTML caching. The degree of replication should depend on the popularity of the page.
Maintained pages that are mostly read (>75%) in default HTML format.	46.7%	HTML replication with a replication factor controlled by the popularity of the page.
Maintained pages that have a significant fraction (>25%) of reads in alternate formats.	8.3%	Wikitext replication with a replication factor controlled by the popularity of the page. HTML caching can be considered if the read/save ratio is considerably high.
Nonexisting pages.	8.3%	Negative caching combined with attack detection techniques.

always limited. Therefore, turning over to a truly collaborative, decentralized organization in which resources are provided and shared by the community would most likely significantly relieve the current infrastructure allowing further growth.

To test this hypothesis, Urdaneta et al. conducted an extensive analysis of Wikipedia's workload, as reported in [33]. The main purpose of that study was to see whether and how extensive distributed caching and replication could be applied to increase scalability. Table 5.3 shows the main conclusions.

Although most requests to Wikipedia are for reading documents, we should distinguish between their rendered HTML forms and data that is read from the lower-level wikitext databases. However, it is clear that there are still many updates to consider, making it necessary to incorporate popularity when deciding on the replication strategy for a page. Surprisingly is the fact that so many nonexisting pages are referenced. Performance can most likely be boosted if we keep track of those pages through negative caching, i.e. storing the fact that the page does *not* exist, and thus avoiding the need to forward a request.

5.4 Replicating for Performance versus Consistency

From the examples discussed so far, it is clear that differentiating replication strategies and considering finer levels of replication granularity in order to improve performance will help. However, we have still more or less assumed that consistency need not be changed: informally, clients will always be able to obtain a "fresh" copy of the data they are accessing at a replica server. Note that in the case of content delivery networks as well as edge-server computing, we made the assumption that clients will always access *the same server*. Without this assumption, maintaining client-perceived strong consistency becomes more difficult.

Of course, there may be no need to sustain relatively strong consistency. In their work on consistency, Yu and Vahdat [38] noted that consistency can be defined along multiple dimensions:

Numerical deviation: If the content of a replicated data object can be expressed as a numerical value, it becomes possible to express the level of consistency in terms of tolerable deviations in values. For example, in the case of a stock market process, it may be allowed to let replicas deviate to a maximum of 1% before update propagation is required, or likewise, that values are not allowed to differ by more than $0.02.

Numerical deviations can also be used to express the number of outstanding update operations that have not yet been seen by other replica servers. This form of consistency is analogous to allowing transactions to proceed while being ignorant of the result of *N* preceding transactions [16]. However, this type of consistency is generally difficult to interpret in terms of application semantics, rendering them practically useless.

Ordering of updates: Related to the number of outstanding update operations, is the extent to which updates need to be carried out in the same order everywhere. Tolerating deviations in these orders may lead to conflicts in the sense that two replicas cannot be brought into the same state unless specific reconciliation algorithms are executed. Consistency in terms of the extent that out-of-order execution of operations can be tolerated is highly application specific and may be difficult to interpret in terms of application semantics.

Staleness: Consistency can also be defined in terms of how old a replica is allowed to be in comparison to the most recent update. Staleness consistency is naturally associated with real-time data. A typical example of tolerable staleness is formed by weather reports, of which replicas are generally allowed to be up to a few hours old.

This so-called **continuous consistency** is intuitively simple when dealing with deviations in the value of content, as well as in the staleness of data. However, practice has shown that as soon as ordering of operations come into play, application developers generally find it difficult to cope with the whole concept of data (in)consistency. As mentioned by Saito and Shapiro [27], we would need to deal with a notion of *bounded divergence* between replicas that is properly understood by application developers. Certainly when concurrent updates need to be supported, understanding how conflict resolution can be executed is essential.

Researchers and practitioners who have been working on replication for performance seem to agree that, in the end, what needs to be offered to end users and application developers is a perception of strong consistency: what they see is always perceived as what they saw before, or perhaps fresher. In addition, if they are aware of the fact that what they are offered deviates from the most recent value, then at the very least the system should guarantee eventual consistency. This observation had already led researchers in the field of distributed shared memory (DSM) to simplify the weaker consistency models by providing, for example, software patterns [4]. In other cases, only simple primitives were offered, or weaker consistency was supported at the language level, for example in object-based DSM systems, which provided an workable notion of weak consistency (called *entry consistency* [2]).

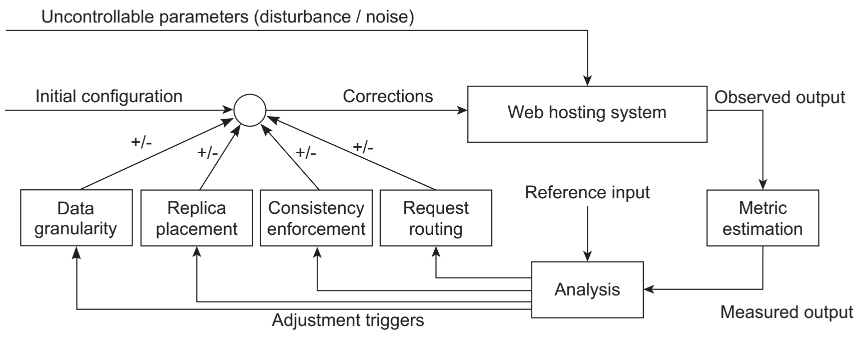


Fig. 5.4 The feedback control loop for automated replica management.

5.5 Replication Management

What all these examples illustrate is that when applying replication for performance, there is no single best solution. We will need to take application semantics into account, and in general also stick to relatively simple consistency models: well-ordered updates and eventual consistency (be it in time or space).

This brings us immediately to one of the major issues in replication for performance: because acceptable weak consistency is dependent on application semantics, we are confronted with a serious replication management problem, which is now also application dependent. By replication management we mean deciding on a replication strategy and ensuring that the selected strategy can be implemented (e.g., by ensuring that appropriate replica servers are in place). As we have discussed above, not only do we need to choose from multiple strategies, we also need to figure out at which level of data granularity we should differentiate strategies.

Manually managing replication for performance in large-scale systems is a daunting task. What is needed is a high degree of automated management, effectively meaning that we are required to implement a feedback loop as shown in Figure 5.4. The control loop shows four different adjustment measures: replica placement (*where to replicate*), consistency enforcement (*how and when to replicate*), request routing (*how to route requests to replicas*), and deciding on data granularity (*what to replicate*).

Notably the deciding on the granularity of data is important for efficient analysis and selection of strategies. For example, by grouping data items into the largest possible group to which the same strategy can be applied, fewer comparisons to reference input is needed thus improving the throughput of feedback.

However, the real problem that needs to be addressed in this scheme is the realization of the analysis component. In content delivery networks such as Globule where data items have an associated origin server, this server is an obvious candidate to carry out the analysis. Doing so will lead to a natural distribution of the load across the system. In this case, an origin server simply logs requests, or collects access traces from replica servers that host content it is responsible for.

Such a scheme cannot be universally applied. Consider, for example, the case of a collaborative, decentralized Wikipedia system. Unlike content delivery networks, there is no natural owner of a Wikipedia document: most pages are actively maintained by a (potentially large) group of volunteers. Moreover, considering that extensive replication is a viable option for many pages, many requests for the same page may follow completely independent paths, as is often also the case in unstructured peer-to-peer networks [26]. As a consequence, knowledge on access patterns is also completely distributed, making analysis for replication management more difficult in comparison to that in content delivery networks.

There seems to be no obvious solution to this problem. What we are thus witnessing is the fact that replication for performance requires differentiating replication strategies at various levels of data granularity and taking application semantics into account when weak consistency can be afforded. However, this replication management requires the instantiation of feedback control loops of which it is not obvious how to distribute their components. Such a distribution is needed for scalability purposes.

5.6 Conclusions

Replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two is reflected by the naturally higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication. In this chapter, we have argued that replication for performance requires automated differentiation of replication strategies and at different levels of data granularity.

In many cases, this automated differentiation implies the instantiation of decentralized feedback control loops, an area of systems management that still requires much attention. If there is one conclusion to be drawn from this chapter, it is that research should focus more on decentralized replication management if replication is to be a viable technique for building scalable systems.

Acknowledgements This chapter could not have been written without the hard research work conducted by a number of people at (one time working at) VU University. We thank Michal Szymaniak, Swaminathan Sivasubramanian, and Guido Urdaneta for their contributions.

References

1. Annapureddy, S., Freedman, M., Mazieres, D.: Shark: Scaling File Servers via Cooperative Caching. In: Second Symp. Networked Systems Design and Impl. USENIX, USENIX, Berkeley, CA (May 2005)
2. Bershad, B., Zekauskas, M., Sawdon, W.: The Midway Distributed Shared Memory System. In: COMPCON, pp. 528–537. IEEE Computer Society Press, Los Alamitos (1993)
3. Bondy, J., Murty, U.: Graph Theory. Springer, Berlin (2008)

4. Carter, J., Bennett, J., Zwaenepoel, W.: Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. ACM Trans. Comp. Syst. 13(3), 205–244 (1995)
5. Devlin, B., Gray, J., Laing, B., Spix, G.: Scalability Terminology: Farms, Clones, Partitions, Packs, RACS and RAPS. Tech. Rep. MS-TR-99-85, Microsoft Research (Dec. 1999)
6. Duvvuri, V., Shenoy, P., Tewari, R.: Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In: 19th INFOCOM Conf., pp. 834–843. IEEE Computer Society Press, Los Alamitos (Mar. 2000)
7. Eugster, P., Guerraoui, R., Kermarrec, A.M., Massoulié, L.: Epidemic Information Dissemination in Distributed Systems. IEEE Computer 37(5), 60–67 (2004)
8. Gao, L., Dahlin, M., Nayate, A., Zheng, J., Iyengar, A.: Application Specific Data Replication for Edge Services. In: 12th Int'l WWW Conf., ACM Press, New York (2003)
9. Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. ACM SIGACT News 33(2), 51–59 (2002)
10. Ho, K.-H., Georgoulas, S., Amin, M., Pavlou, G.: Managing Traffic Demand Uncertainty in Replica Server Placement with Robust Optimization. In: Boavida, F., Plagemann, T., Stiller, B., Westphal, C., Monteiro, E. (eds.) NETWORKING 2006. LNCS, vol. 3976, pp. 727–739. Springer, Heidelberg (2006)
11. Hofmann, M., Beaumont, L.: Content Networking: Architecture, Protocols, and Practice. Morgan Kaufman, San Mateo (2005)
12. Karlsson, M., Karamanolis, C.: Choosing Replica Placement Heuristics for Wide-Area Systems. In: 24th Int'l Conf. on Distributed Computing Systems, Mar. 2004, pp. 350–359. IEEE Computer Society Press, Los Alamitos (2004)
13. Karlsson, M., Karamanolis, C., Mahalingam, M.: A Framework for Evaluating Replica Placement Algorithms. Tech. rep., HP Laboratories, Palo Alto, CA (2002)
14. Kermarrec, A.M., Massoulié, L., Ganesh, A.: Probabilistic Reliable Dissemination in Large-Scale Systems. IEEE Trans. Par. Distr. Syst. 14(3), 248–258 (2003)
15. Kistler, J., Satyanaryanan, M.: Disconnected Operation in the Coda File System. ACM Trans. Comp. Syst. 10(1), 3–25 (1992)
16. Krishnakumar, N., Bernstein, A.J.: Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. ACM Trans. Database Syst. 4(19), 586–625 (1994)
17. Lee, Y.W., Leung, K.S., Satyanarayanan, M.: Operation Shipping for Mobile File Systems. IEEE Trans. Comp. 51(12), 1410–1422 (2002)
18. Leff, A., Rayfield, J.T.: Alternative Edge-Server Architectures for Enterprise JavaBeans Applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 195–211. Springer, Heidelberg (2004)
19. Leighton, F., Lewin, D.: Global Hosting System. United States Patent, Number 6,108,703 (Aug. 2000)
20. Neuman, B.: Scale in Distributed Systems. In: Casavant, T., Singhal, M. (eds.) Readings in Distributed Computing Systems, pp. 463–489. IEEE Computer Society Press, Los Alamitos (1994)
21. Pierre, G., van Steen, M., Tanenbaum, A.: Dynamically Selecting Optimal Distribution Strategies for Web Documents. IEEE Trans. Comp. 51(6), 637–651 (2002)
22. Qiu, L., Padmanabhan, V., Voelker, G.: On the Placement of Web Server Replicas. In: 20th INFOCOM Conf., Apr. 2001, pp. 1587–1596. IEEE Computer Society Press, Los Alamitos (2001)
23. Rabinovich, M., Spastscheck, O.: Web Caching and Replication. Addison-Wesley, Reading (2002)
24. Rabinovich, M., Xiao, Z., Aggarwal, A.: Computing on the Edge: A Platform for Replicating Internet Applications. In: Eighth Web Caching Workshop (Sep. 2003)
25. Radoslavov, P., Govindan, R., Estrin, D.: Topology-Informed Internet Replica Placement. In: Sixth Web Caching Workshop, Jun. 2001, North-Holland, Amsterdam (2001)
26. Risson, J., Moors, T.: Survey of Research towards Robust Peer-to-Peer Networks: Search Methods. Comp. Netw. 50(17), 3485–3521 (2006)

27. Saito, Y., Shapiro, M.: Optimistic Replication. ACM Comput. Surv. 37(1), 42–81 (2005)
28. Schneider, F.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Comput. Surv. 22(4), 299–320 (1990)
29. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: Analysis of Caching and Replication Strategies for Web Applications. IEEE Internet Comput. 11(1), 60–66 (2007)
30. Sivasubramanian, S., Szymaniak, M., Pierre, G., van Steen, M.: Replication for Web Hosting Systems. ACM Comput. Surv. 36(3), 1–44 (2004)
31. Tanenbaum, A., van Steen, M.: Distributed Systems, Principles and Paradigms, 2nd edn. (translations: German, Portugese, Italian). Prentice-Hall, Upper Saddle River (2007)
32. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., Welsh, B.: Session Guarantees for Weakly Consistent Replicated Data. In: Third Int'l Conf. on Parallel and Distributed Information Systems, Sep. 1994, pp. 140–149. IEEE Computer Society Press, Los Alamitos (1994)
33. Urdaneta, G., Pierre, G., van Steen, M.: Wikipedia Workload Analysis for Decentralized Hosting. Comp. Netw. (to be published 2009)
34. Vogels, W.: Eventually Consistent. ACM Queue, pp. 15–18 (Oct. 2008)
35. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding Replication in Databases and Distributed Systems. In: 20th Int'l Conf. on Distributed Computing Systems, Taipei, Taiwan, Apr. 2000, pp. 264–274. IEEE (2000)
36. Wolman, A., Voelker, G., Sharma, N., Cardwell, N., Karlin, A., Levy, H.: On the Scale and Performance of Cooperative Web Proxy Caching. In: 17th Symp. Operating System Principles, Kiawah Island, SC, Dec. 1999, pp. 16–31. ACM (1999)
37. Yeo, C., Lee, B., Er, M.: A Survey of Application Level Multicast Techniques. Comp. Comm. 27(15), 1547–1568 (2004)
38. Yu, H., Vahdat, A.: Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. ACM Trans. Comp. Syst. 20(3), 239–282 (2002)