UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Ivan Voras

CACHE SERVER FOR DISTRIBUTED APPLICATIONS ADAPTED TO MULTICORE SYSTEMS

DOCTORAL DISSERTATION

Zagreb, 2011

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Ivan Voras

POSLUŽITELJ PRIRUČNE MEMORIJE ZA RASPODIJELJENE APLIKACIJE PRILAGOĐEN SUSTAVIMA S VIŠEJEZGRENIM PROCESORIMA

DOKTORSKI RAD

Zagreb, 2011.

This doctoral dissertation was created at the University of Zagreb, Faculty of electrical engineering and computing, at the Department of control and computer engineering.

Mentor: prof. Mario Žagar, Ph.D.

The doctoral dissertation is comprised of 114 pages.

Doctoral dissertation number ___.

The dissertation evaluation committee:

1. Professor Danko Basch, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb
2. Professor Gordan Gledec, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb
3. Professor Maja Štula, Ph.D.,
   Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split

The dissertation defence committee:

1. Professor Danko Basch, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb
2. Professor Gordan Gledec, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb
3. Professor Maja Štula, Ph.D.,
   Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split
4. Professor Mario Kovač, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb
5. Professor Maja Matijašević, Ph.D.,
   Faculty of Electrical Engineering and Computing, University of Zagreb

Date of dissertation defence: June 13th 2011

Abstract

The Internet is the largest platform for application delivery today, but building scalable Web applications for the global audience is hard. The problem of satisfying the constraints of starting small to minimize initial investments but also adopting an architecture with the potential to allow future growth in the number of users and the complexity of the service does not have an immediately obvious or boiler-plate solution. This dissertation investigates the described problem and proposes solutions which are based on data partitioning techniques, with the focus on introduction of a new cache server designed for modern multi-core CPUs, offering an expanded metadata model for cached records. In order to make use of the new cache server, it also proposes new Web application patterns. The dissertation contains an analysis and evaluation of the introduced models and a comparison with previously available solutions.

Keywords: distributed systems, Web applications, cache, database, concurrency, algorithms

Strukturirani sažetak

Izrada skalabilnih Web aplikacija dostupnih globalnoj publici je zahtjevan zadatak kojeg dodatno kompliciraju kompromisi aplikacijske arhitekture koji su prisutni kada aplikacija započinje korištenjem među manjim krugom korisnika i kasnije eksplozivno proširuje krug korisnika. Ovaj doktorski rad istražuje problem izrade visoko skalabilnih Web aplikacija te predlaže rješenja koja su bazirana na tehnikama raspodjeljivanja podataka, sa fokusom na uvođenje novog poslužitelja priručne memorije koji je posebno osmišljen za višeprocesorska i višejezgrena računala, sa proširenim modelom podataka za pohranu zapisa, te predlaže arhitekture za Web aplikacije koje optimalno koriste napredne mogućnosti novog poslužitelja priručne memorije.

Cilj

Doktorski rad istražuje problem stvaranja visoko skalabilnih Web aplikacija s obzirom na broj korisnika, pomoću uvođenja poslužitelja priručne memorije te prilagodbu arhitektura aplikacija ovom novom poslužitelju. Cilj rada uključuje omogućavanje stvaranja kompleksnih Web aplikacija koje koriste poslužitelj priručne memorije za ostvarenje velikih performansi.

Metode

U ovom doktorskom radu su istražene trenutne prakse i trendovi u izradi Web aplikacija, proučeni problemi skalabilnosti prisutni u Web aplikacijama te postojeće strategije koje doprinose izradi skalabilnih Web aplikacija. Na temelju proučenog postojećeg stanja, u radu su dani zahtjevi za novi poslužitelj priručne memorije sa novim mogućnostima rada u višeprocesorskim i višejezgrenim računalima i novim podatkovnim modelom za opis pohranjenih podataka sa podatkovim strukturama dizajniranima s ciljem da ostvaruju velike performanse pri istovremenom pristupu pohranjenim podacima.

Rezultati

U ovom doktorskom radu je predložen model novog poslužitelja međumemorije na temelju prethodno danih zahtjeva koji omogućava usporedbu više modela višedretvenog rada u pristupu i obradi podataka, visoke performanse pri istovremenom pristupu pohranjenim podacima, te model pohranjenih podataka koji omogućava dodavanje metapodataka u svrhu efikasnog izvođenja grupnih operacija nad podacima. Predložene su nove arhitekture Web aplikacija koje su prilagođene korištenju novog poslužitelja priručne memorije. Izrađena je analiza predloženih modela i implementiranih rješenja te njihova usporedba sa prethodnim rješenjima.

Zaključak

U ovom radu dan je novi model raspodijeljenog poslužitelja međumemorije temeljenog na particioniranju podataka, ostvaren je novi poslužitelj međumemorije posebno osmišljen za moderne višejezgrene procesore , te predložene arhitekture okoline aplikacija na Webu prilagođena novom poslužitelju međumemorije . Napravljeno je vrednovanje modela, poslužitelja i arhitekture okoline aplikacije te usporedba sa postojećim rješenjima.

Ključne riječi: raspodijeljeni sustavi, Web aplikacije, međumemorija, baza podataka, konkurentni pristup, algoritmi

Acknowledgements

I wish to thank my family and friends for support, without which it would have been difficult and pointless to create this work. I also wish to thank my mentor, prof. Žagar, on both academic and personal guidance.

Ivan Voras, June 2011

Table of Contents

1. Introduction

By the year 2011 when this dissertation is written, it is both a cliché and an understatement to claim the Internet has changed the lives of everyone on the planet. As one of the most significant drivers of globalization it is a true candidate for the shortlist of humanity's greatest achievements, present due to its unconstrained nature in all segments of daily lives and business. The Internet is a medium for numerous protocols and services, but one of those services certainly stands out: the World Wide Web, as probably the most important platform for the deployment of distributed applications.

From its humble beginnings [1], the Web has grown into a platform capable of delivering hundreds of billions of dollars [2][3] in e-Commerce and reaching audiences of hundreds of millions of everyday users [4]. Its low barrier to entry has enabled companies to start small and over the course of a few years grow into world-class companies as measured by their revenue stream (e.g. [5][6]). This large growth poses challenges on both the technical aspects of the infrastructure providing the service, which must support the growth, and the financial backing of the enterprise, as typical large-scale data centres carry costs in the orders of hundreds of millions of dollars [7][8] for initial investments and millions of dollars in monthly expenditures for electricity and network equipment [9][10]. Any investments in server and application optimization therefore have immediate influence in practical business matters.

The HyperText Transfer Protocol (HTTP) has shown itself to be adequate as a platform for application delivery, especially when augmented with client-side technologies like JavaScript, but its real-world use requires certain cessions from application creators. Among these the most significant are emulated persistent sessions over the originally non-presistent protocol [11] and the use of still somewhat quirky client-to-server communication channels in the form of Asynchronous JavaScript and

XML (AJAX) [12]. However, these issues have been accepted and worked around in practice, making the concept of Web applications immensely popular and widespread.

Typical Web applications are built in a multi-tiered architecture, with at least an HTTP server (i.e. a *Web server*), an application server (e.g. PHP, ASP, JSP, etc.), and a database server (e.g. MySQL, PostgreSQL, Microsoft SQL Server, Oracle) layers. Though this arrangement is conceptually often fixed, practical applications might merge some of those layers into single entities / processes or add more layers as needed to meet certain application requirements. A notable addition to the basic architecture is the introduction of a cache server with the intention of enhancing performance in typical environments where data is mostly accessed for reading (i.e. *read-mostly* data). Modern deployments of large-scale applications regularly include at least a simple cache server facility, often with huge success[1].

This dissertation is a result of the research in the area of enhancing performance of Web applications by using data caching techniques and includes as one of its results a new cache server designed for modern industry standard server architectures.

1.1. Motivation

The most important type of distributed applications nowadays are Web applications, as they can offer services to an unprecedented number of users and are from this point of view currently the most scalable computer application types in practical use. They pose unique challenges and present unique opportunities for research into the development and deployment of large scale applications. The focus of this dissertation is on performance enhancement of Web applications, which is as an improvement in efficiency directly reflected to possible reduction of the number of servers used for applications and with that, savings in the areas of maintenance, the usage of electricity (for servers, network equipment, cooling and other components of the data server) and even in certain configurations on the network bandwidth required by the applica-

1 Facebook implements the largest publicly described deployment of a Web application with a cache server, reportedly held more than 28 TB of frequently accessed user data in its cache in 2008, spread across more than 800 cache servers [13], which climbed to over 300 TB in 2010 [14].

tions. This performance enhancement will be achieved through research into cache servers and their integration into Web application architectures.

1.2. Research goals and methods

The central problem of this research can be stated as "How to increase scalability in Web applications with acceptable levels of service and minimal cost?" This statement of the problem in turn requires clarification of its parts and the specific discussion of the following topics:

- What is scalability in the context of Web applications?
- What are acceptable levels of service?
- What minimizes cost?

These questions have the effect of practical constrains on this dissertation's results, as its results will have to be justified through them. The research described here has (as a touching point with real-world usage) its application in the "Quilt" Web content management system which is written at our home Faculty, of which the author is one of the principal developers.

The goals of this dissertation are: to create a model for a new highly distributed cache server which is able to make use of modern multi-core CPUs, whose operation is based on data partitioning techniques, to create its exemplary implementation, and to describe Web application architectures which can extract the maximal performance benefits from this cache server. It includes an evaluation of the models, the cache server and the application architectures in comparison with existing solutions.

1.3. Dissertation organization

This dissertation is organized as follows: the introductory chapter 2 describes and discusses current best practices in Web application development. Chapter 3 describes the central problems in more detail which leads to a discussion of possible solutions in chapter 4. Chapter 5 focuses on the requirements for the new cache server and chapter 6 introduces the model for the new cache server. In order to make use of the new cache server, new Web application architectures adapted to it are derived in

chapter 7, with a discussion of the trade-offs in practical implementations of this architecture. Finally, all the presented models and architectures are analysed and evaluated in chapter 8, the possibilities for future work are presented in chapter 9, and the conclusions of the dissertation are presented in chapter 10.

2. Best practices and trends in Web application development

Multi-tiered application architectures have been widely implemented in non-trivial applications because the model is a natural continuation of the idea of modularity in computer programs[2]. The advent of the networked age has simply broadened the medium over which modules can be integrated, which resulted in a virtual ecosystem of distributed applications implemented by modules using what were until recently infeasibly high-level calling conventions with very verbose protocols and data description languages like HTTP and XML [16][17][18]. However, network bandwidth is still a precious resource and network latencies often forbid highly-distributed architectures and architectures with complex protocols within application backbone infrastructure, favouring tighter binary protocols for the serialization of structured data, like Google Protocol Buffers [19], Apache Thrift [20] or BSON [21].

The majority of contemporary Web applications have a generally uniform tiering architecture consisting of:

1. A Web server
2. An application server
3. A database

In this configuration, the Web server is often nothing more than a simple protocol broker, processing and passing HTTP requests to the application server (and optionally serving static and miscellaneous requests such as files containing CSS design, JavaScript code or images directly from file system). The most widely deployed type of

2 As eloquently stated by Schuman in 1974 [15]: "In general, program development does not consist of writing independent procedures, but rather of writing complete packages that may be used as is or selectively incorporated into other packages, thus defining progressively higher-level modules."

the application server is a dynamic language interpreter (or runtime) which is a generic execution environment for user code, providing a set of libraries and a framework for creating Web applications but without specifying or limiting the details of the applications it executes. The database usually implements a relational data model and SQL as an interface to the application.

The methods of communication between these tiers / components are important and have a large influence on overall application performance. The local optimum in architecture for interfacing application servers to the Web server (in the sense that it is widely supported and simple enough while having adequate performance) is the FastCGI protocol [22]. By using FastCGI, the application servers can be implemented as persistent processes running either on the same system as the Web server or on arbitrary connected systems. Performance achieved with with FastCGI is greatly improved over the formerly popular (but only recently standardized) CGI protocol [23] [24] by avoiding process creation and teardown for every HTTP request. The database server and the application communicate with SQL, but the exact communication protocols are not defined beyond general purpose-dictated constraints (e.g. implemented over a connection-oriented protocol similar to TCP). Each popular database implements its own client-server protocol, an API and often its own libraries for communication with applications (as visible e.g. from [25]). The described components can exist in three general arrangements:

1. Completely separated, implemented on different server computers and communicating via TCP or a similar protocol
2. Partially separated, with some components running on the same server, possibly in separate processes, while others are running on separate servers
3. Completely integrated, with all components existing within the same process on a single server

For each of the described arrangements there is a respectable number of active deployments in various Web sites (though of course the scale of deployment varies). One example is the popular stack of Open source projects consisting of the Apache Web

server, the PHP scripting language interpreter and the MySQL or SQLite databases, which can be implemented in either of the three arrangements. Similar arrangements exist with proprietary stacks like Microsoft's (Internet Information Server as Web Server, ASP.Net as the execution environment and the SQL Server as the database) or Oracle's (Oracle Application Server, Oracle Database). Because of the distributed nature of the Web there are many Web application architectures and stacks for various environments (e.g. as described in [26] and [27]), some of which could be considered obsolete when compared to modern products. Even the considerably old CGI protocol remains a popular option, especially in environments where the Web application is peripheral to the product (e.g. software and hardware network appliances, embedded devices managed with a Web interface). The focus of this dissertation however is on the other end of the spectrum: on high-end Web applications built with modern technologies and for global audiences.

2.1. Trends in scalable Web applications architectures

If there would be a need to summarize the recent trends in building large-scale Web applications in as few words as possible, these words would be “Shared nothing architecture.” In the historical debate between the opposite concepts of “Shared everything” and “Shared nothing” architectures, the latter has become the de-facto only architecture used today for implementing globally accessible Web applications, mostly due to its intrinsic property that scalability for a larger number of clients can be achieved by adding cheap servers to the hosting environment and that faults in servers can have only a very localized influence. The Shared nothing architecture was first championed in the field of database architectures in 1986 [28] but has recently found its application in Web applications where it has achieved great popularity and is currently the strategy on which the largest Web application and Web service vendors rely on for scalability [29][30][31][32].

From the point of view of data storage, providing for global applications is challenging. The vastly dominant storage medium is still the mechanical hard disk drive, and while solid state drive (SSD) deployments are growing, they have not yet reached the same order of magnitude as with the conventional mechanical drives [33]. Since ap-

plications for the global audience are accessed frequently and unpredictably, their data access patterns emphasize the slowest operations in mechanical hard drives – seeks. Increased requirements for performance have resulted in wide-spread adoption of various caching techniques where the "hot set" of accessed data is kept in memory. Some of the largest Web applications whose implementation is officially disclosed use cache servers to increase the access performance for the largest part of their working data set. Examples for this are Facebook, which holds more than 300 TB of data in memory caches [14] and Twitter, which implements an architecture which considers memory as its primary data store with the disk-based database storage being secondary (and for which "flushing the cache" would be "catastrophic") [34].

The prevalent type of the memory cache used is a simple key-value data store with automatic expiry, which is exemplified in the Open source product *memcached*, used by some of the largest Web sites such as Facebook, Flickr, LiveJournal, Reddit, Twitter, YouTube and Wikipedia [36]. The successes in implementing this type of cache in highly popular Web applications has contributed to the resurgence in usage of simple, non-relation data models for main data stores, which is focused in the "NoSQL movement" [35][3].

3 The "NoSQL movement" self-defines in [35] as "Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable. The original intention has been modern Web-scale databases. The movement began early 2009 and is growing rapidly."

3. Problems in Web application scalability

The term “scalability” as applied to Web applications can be defined with clarity as describing the simplicity with which the applications can be made accessible to a larger number of users, usually by introducing new hardware resources like CPUs, memory and persistent storage into the system, or in other words how well the application responds to an increased amount of resources available to it. When discussing computer system scalability in general, there are usually two aspects of the problem which are being discussed: *vertical scalability* and *horizontal scalability.* The former is describing the response of the applications to adding more resources to a single computer system (e.g. a local CPU, more RAM modules, more disk drives) while the latter describes the application's response to adding *more computer systems*, assuming that the question of vertical scalability within each system is already solved or irrelevant. This dissertation touches on both aspects of scalability but has a slightly more intense focus on the area of horizontal scalability, which is currently necessary for reaching global audiences.

Internet-facing, publicly available Web applications are unique in that they can literally be made accessible to the whole world, or at least every Internet user on it[4]. This introduces large strains on all important resources of such a system: the CPU load, memory, storage space and network bandwidth, all of which must support all of the users accessing the application or there is a risk of degraded experience for a significant part of the user base, or even all of it. The following sections discuss the problems in scalability of major types of resources in a computer system and their influence on the quality of services delivered to end-users.

4 Currently no more than one third of the world population has Internet access [37][38]. According to some statistics, almost 50% of these users access Google on daily basis [39] and almost 40% access Facebook on daily basis [40].

3.1. Scalability of CPU load

Scalability of CPU usage has been the typical problem of vertical scalability for a long time. It can be described as the efficiency in using available CPU computational power (i.e. not wasting it on “book-keeping” operations or inefficient algorithms). Though historically the problem has been addressed though advancements in algorithms and program architecture, today it shares some points with horizontal scalability in environments where there is more than one active CPU core available to applications on a single system [41].

One of the major tasks of application architecture design today is the design and choice of algorithms which can be applied in programs running on multi-core multi-processor (SMP) computer architectures. In the technological landscape where the Moore's Law was redirected from building faster CPU cores to building more cores on a single chip[5] [42] but the memory, system data buses and even storage have remained centralized and at least partially shared between the CPU cores, the emphasis has shifted over time to developing algorithms which operate on multiple cores but reduce or avoid contention on shared memory regions [43][44][45].

From the application's point of view the “hidden latencies” such as memory and system bus accesses usually manifest as non-specific CPU-related slowdowns. These latencies will, unless a specific distinction needs to be made, be folded under “CPU load.” This discussion assumes that the CPU, buses and memory are matched in technical properties and speed and considers largely out of scope the situations where there is some kind of mismatch between CPU performance and memory or system bus performance (for example where a fast new-generation CPU is used with a slow past-generation memory).

The impact of a CPU overload on a Web application server used (shared) by many users degrades the response time, usually in linear proportion to the factor of overload, for all users of the server. This property makes the effects of CPU overload among the more light-weight of the overload effects. However in some border cases,

5 Moore's law concerns itself with the growth of the number of transistors on a single chip

for example if there is significant contention for application resources, overloading the CPU may have a significantly worse then linear impact on overall system performance.

In order to maximize vertical scalability of available CPU resources, applications need to be designed to make use of multiple CPUs or CPU cores in a computer system, by using techniques of multiprocessing and multithreading. The challenge of horizontal scalability is in extending these techniques to implementation on multiple networked computer systems.

3.2. Scalability of memory

Though the amount of available system memory (RAM) has increased significantly in the past years, the demands on memory have grown to match it. A large portion of this increase simply comes from the increase in the scope of data processed by applications: more messages, larger images, multimedia features and similar demands which come from the demand for better user experience.

Traditionally, memory overload is somewhat mitigated by using virtual memory techniques with disk paging, colloquially called “swapping” but high-performance applications suffer greatly from increased latencies of disk drive-backed memory, leading to severe user experience degradation. In this respect, memory overload is a significantly worse situation than CPU overload and should be avoided at all cost. Additionally, memory is used for local file system caches so the shortage of memory can cause performance and stability problems even if there is strictly enough memory for the programs themselves.

Depending on memory access patterns and operating system virtual memory algorithms, as well as the severity of memory shortage, memory overloads can have an influence on either all users of the Web application or only for some users. Due to the large gap in latencies between “real” memory and disk-based virtual memory, memory shortage manifests in sudden performance drops and, if not handled properly can lead

to “spiral of death” behaviour as the queue of incoming HTTP requests is lengthening while waiting for progressively slower requests to finish executing.

Vertical scalability in memory availability is usually trivially achieved within the applications if the hardware allows memory expansion (though with the exception of high-end systems this expansion requires system downtime). Horizontal scalability of system memory is usually not achievable by itself due to requirements of high-bandwidth and low-latency interconnects, and is usually a consequence of horizontally scaling for CPU load by running programs on multiple servers.

Another aspect of memory scalability is the possibility of memory bandwidth overload, either directly caused by the memory modules themselves or by the design of system buses. While this aspect can have an impact on application performance, contemporary practices in system design tend to pair CPUs, buses and memory modules of similar performance classes [46], minimizing the impact or preventing bandwidth overload.

3.3. Scalability of storage

Storage systems in general have three aspects critical for practical Web application scalability: available storage space, performance expressed in terms of operations per seconds (IOPS – Input Output operations Per Second) and reliability, each having a different influence on overall application performance.

Scalability of storage space is important as the amount of data in the application grows. If the storage hardware or the application architecture do not allow storage space expansion, new data cannot be recorded but access to existing data will generally not be hindered. This situation may prevent signing up of new users or uploading of new data, but existing users may view already present data, usually without major additional problems.

Overloading the storage systems in terms of IOPS is a problem with larger consequences, influencing all users of the same storage service or a device in a way sim-

ilar to CPU overload: in the general case the performance degradation is linear in proportion to the factor of overload. Recently popularized technologies such as flash memory based solid state drives (SSDs) can allow significantly increased IOPS performance but the gains are sensitive to the type of workload and they can degrade over time in which the device is actively used [47].

Vertical scalability of storage (both for available space and for IOPS performance) is usually easily achievable within the applications if the hardware allows expansion. Making use of horizontal scalability requires application designs which allow storage to be implemented across multiple servers or specialized storage devices, while still maintaining coherence and data integrity with respect to business rules.

In addition to storage space and IOPS scalability, an important property of storage systems for the majority of applications is their reliability. While disasters involving CPUs, memory and other core systems can be relatively painlessly recovered by replacing those parts, loss of data is usually a much more serious problem which can more directly lead to business problems such as loss of revenue and customers (users) [48]. Unfortunately, as reliability of storage is achieved by implementing redundancy in various components [49], it often stands as a goal opposite to the goals of increased storage space and IOPS performance, requiring compromises.

3.4. Scalability of networks and internal communication channels

Application tiers need to communicate, and the way this communication is implemented affects application performance and scalability. It is easily observable that this area of system design is often not given the attention it deserves, erroneously assuming near-instant communication between application tiers whether they are implemented on a single server or distributed across many servers in various arrangements. This assumption leads to various “unexplainable” performance problems for applications which are otherwise correctly designed. Commoditization and componentization of modules from which the application tiers themselves are built with can lead to overlooking performance bottlenecks such as using comparatively expensive protocols for internal communication within a server (such as using TPC over localhost instead

of light-weight operating system IPC or shared memory), verbose or complex protocols for communicating latency-sensitive data between servers (such as using ASCII-based or XML-formatted protocols to communicate with database servers) or even using inadequate or unconfigured hardware (such as 100 Mbit/s Ethernet instead of 1 Gbit/s Ethernet).

From the application scalability point of view, the problems are in identifying critical communication paths and implementing them in a way which maximizes both local performance and allows future expansion.

3.5. Scalability of application architecture

Even if all the lower levels are designed correctly, that alone is not enough to ensure adequate scalability. A scalable Web application architecture will adapt to expansion of each of the previously described types of resources, with the goal of maximizing the effective use of any new resources attached to the application. It must respond with increased performance if local computer resources (CPUs, memory and storage) are added as well as allow running application instances on separate computer systems (servers), while also being able to access multiple networked storage servers and other devices.

4. Strategies for scalable Web applications

A beneficial aspect of the HTTP with respect to scalability is that it was originally designed as a stateless protocol in versions up to and including 1.0 [50]. This directly supports the *shared nothing* approach and allows for a trivial scheme of scalable content delivery based on the property that, since no persistent data needs to be kept between HTTP requests, sequential requests can be responded to by separate and unrelated servers, providing that each one of the servers carries the whole data set and implements the same namespace. The simplest practical implementation of this scheme utilizes a feature of the Domain Name System (DNS [51]) by which clients requesting domain name resolution can be pointed to an IP address chosen from a list of possible addresses in a round-robin fashion, which results in a stochastically balanced load between the servers in the list (a "load balanced group").

Dynamic Web content – which would now in most cases be synonymous with Web applications – relies on preserving state between HTTP requests to provide users with a richer set of interactive features. Application state data can be preserved across requests even in HTTP/1.0 by including it as "query" parts of URLs or HTML "form data" in requests. While this approach has the distinction of being very standard-compliant, retaining the basically stateless aspects of the protocol, it is also cumbersome and very inefficient if the application state is large.

To standardize and encourage the creation of more complex applications, the concept of "HTTP Cookies" was introduced [52], allowing applications to either transfer state data with every request or to use more advanced session-keeping facilities of the servers. In either case, the move has proven to be pivotal for the appearance of complex Web applications. As the applications become more complex and with more state data to be preserved, a common technique in preserving the state became trans-

porting only a short, unique and random "session identifier" string in an HTTP cookie (or in older applications as a part of URLs), storing the much larger application state data on the server as "session data" in a database or as a simple file (which is also recommended in [53]). This technique minimizes the amount of data being transferred between the client and the server in each request-response transaction, however it also breaks with the stateless nature of the protocol, removing the possibility of exploiting the trivial *shared nothing* approach in scaling to multiple servers. Additionally, complex applications most likely need access to a larger data set in addition to the user-centric state data, e.g. a database containing business data, a set of files, etc.

Achieving similar levels of scalability with stateful Web applications is possible if all servers acting as part of a load balanced group have equal access to all of the application data (including session data). This requirement can be satisfied by having all servers accessing the same database or the file system or by replicating data between them. This approach is adequate for a smaller number of servers but introduces scalability problems for larger deployments. In the segment of bleeding-edge Web application technology, overcoming this step is what separates small and medium deployments of Web applications and the large-scale or global Web applications.

4.1. Strategies for global scalability

Web sites aiming for global scalability with complex applications must implement *shared nothing* architectures more aggressively. On a high level, users from different parts of the world need to be presented a service with harmonized network conditions, getting around latency sources such as inter-continental network links, congested network links and unstable routing paths [54]. This is achieved by geographically distributing data centres involved in serving the application to the users, together with deploying geographically-dependant DNS service which provides transparent balancing of user requests to the data centre which is closest to the request origin (in terms of network topology and latency) [55].

This strategy introduces the notion of a Web application which appears for all intents and purposes to be monolithic and single-sourced but which is in practice distributed across the globe. It also introduces a problem of global data coherency.

To return to the *shared nothing* ideal architecture, the currently largest Web sites have begun to implement aggressive data partitioning schemes in order to minimize the need for online data synchronization between various parts and layers in their application architecture, with the ultimate goal of handling a single related block of information on as few systems as possible, preferably localized with respect to global distribution.

4.1.1. Case study: Facebook

Facebook as one of the largest Web applications currently deployed has a very pronounced history of "starting small" and upgrading their infrastructure as the number of users increased [56], relying on self-invented techniques based on commodity products to handle scalability. Due to the specific nature of data maintained by Facebook – social graphs which span the whole world – it has not yet deployed geographically distributed data centres for its core data, but makes extensive use of third party content delivery networks (CDNs) for static content (uploaded images, multimedia) [32]. This enables the core application to process the bulk of the applications' dynamic data within its main data centres but handle high-volume data in a distributed fashion.

Facebook's main data centres contain over 30,000 servers total, of which approximately 2,000 are database servers and approximately 1,000 are cache servers (numbers estimated from partial data available for years 2009 and 2010 from cited sources). Since Facebook makes use of CDNs for static data, the majority of the remaining servers are Web application servers.

4.1.2. Case study: Google

Though Google publishes more papers on methods, technology and analysis than any of the big Web-oriented companies [57], less is known about its specific internal infra-

structure. It is known that Google used techniques for building distributed computing systems from the start and relied less on commodity products, building its infrastructure (both software and hardware [58]) with considerable planning. One of the biggest early technical innovations from Google was the "BigTable" system, a highly distributed database which was the forerunner in the application of *map-reduce algorithms* [59] for high volume data processing [30].

Google is known to have highly geographically decentralized architecture, implementing its service through more than 50 world-wide data centres (extrapolated from information available in 2008 [60]) and an estimated more than a million servers [61]. These data centres are used to store and serve both global search index results and user information such as the contents of Google Mail mailboxes. In case the user accesses his information from a "distant" location (i.e. an suboptimal data centre), his data (or a portion of it) is migrated at runtime.

4.1.3. Case study: Twitter

Twitter is one of the youngest globally popular Web applications, but similar to others, it has experienced a large growth in the number of users, requiring a rapid transition from a minimal infrastructure to one needing substantial hardware and software investments [34]. Interestingly for this dissertation, its most recent infrastructure forgoes the model of a classic database-centric architecture for a more memory-based one implemented with the memcached cache server (the database is "only" a backup), and can achieve processing volume of at least 7,000 messages per second on a global scale [62]. Due to the high volume of data, a complete database restart in Twitter is an operation lasting more than 12 hours [63]. Little is known about the specifics of Twitter's current hardware infrastructure but it is large enough to warrant the construction of a custom-built data centre [64].

4.1.4. Service levels

While having a widely distributed and well implemented application can effectively increase its availability [65] on the large scale, this is not true for smaller deployments. In environments where redundancy is not totally pervasive, increasing the

number of components which can break down can in a trivial way decrease the availability of the application.

Introducing redundancy in hardware targets removing single points of failure on the lowest level and spans the entire spectrum from redundancies in the design of the electrical supply to the building, power supply units on the equipment, network connectivity (including active equipment like network switches) to using redundant servers, which themselves are equipped with ECC or Extended ECC RAM (whose importance is attested in [66]), certified server CPUs and disk drives (if applicable). In order to protect from data loss or unavailability, redundant data storage equipment and services are a necessary part of the overall system, manifesting on different levels as using redundancy-increasing RAID levels on disk drives, file systems with built-in replication and database services with built-in replication in various forms. On the application level, redundancy is achieved by running multiple instances of the application code on different servers rather than implemented on the same system[6], but requires that the application supports this type of deployment and can make efficient use of the distributed data storage and services.

4.1.5. Cost

Large-scale Web application deployments rely on using cheap hardware and high levels of automation for bringing down both cost and complexity of implementation [58][14]. Large data centre deployments drive costs down by using mass-produced equipment, commodity industry standard servers based on Intel x86 or AMD architectures [67][68], with standardized components such as storage and network systems. The cost breakdown of data centres indicates that servers themselves take slightly less than 60% of overall cost, while the rest is spent on electrical energy supply and distribution, cooling and other equipment [10].

Any increase in server efficiency – from better hardware to better algorithms – reflects not only as direct savings in the number of servers needed to serve a fixed num-

6 Except in very high-end business-targeted computer systems with mainframe qualities which offer advanced CPU coupling features for application redundancy, which are out of scope for this dissertation.

ber of users but also indirectly on savings on power supply, cooling and supporting equipment. In rapidly growing globally available Web application deployments, switching to a more efficient Web application tier can slow down the rate at which new equipment is acquired [32].

4.2. Strategies for data storage scalability

In order to maximize the locality of data and with it reduce the latencies involved and to directly or indirectly implement *shared nothing* architectures, large-scale Web applications make extensive use of *sharding* and *tiering* techniques, either separately or in a combination. Conceptually, both techniques can be implemented independently of the actual low-level data storage methods (such as database types or storage device architecture) and can be realized either purely as a function of the database (or other storage system) or as a high-level application feature closely tied with business logic [69][70][71][72][73][32].

The term *sharding* has been recently popularized by developers of distributed applications to usually refer to a specific method of data partitioning with the goal of aggressive horizontal scalability where data objects and all their referenced objects reside self-sufficiently on a single system (typically, the criteria for sharding include users, topics and geographic locations). This approach is usually combined with at least slight denormalization of data (to reduce data set complexity while increasing performance) and some duplication of data across data partitions (to ensure complete independence of data partitions), but with the combined benefit of ensuring with high certainty that the method will result in a highly horizontally scalable architecture.

Data storage tiering is an extension of the hierarchical model of memory [74], applied to the topics of scalability. Tiering for scalability includes isolating and/or moving frequently accessed (“hot”) data to better performing (and regularly more expensive) storage while leaving less frequently accessed data on slower, mass storage devices [75]. This principle is the basis of caching, including the use of cache servers in Web applications. When combined with sharding, the technique involves moving

entire self-sufficient data sets to better performing storage systems for the duration of the period of frequent accesses [14].

4.3. The importance of cache servers

Contemporary *cache servers* are specialized memory-only database servers whose primary purpose is providing performance enhancements to complex applications, typically by serving as fast storage for performance-sensitive application data. Depending on the specifics of their usage, they do not necessarily need to be highly optimized for performance (though they usually are) as long as their common operations are faster than the operations the applications would have to perform if the cache servers are not used.

A common use for cache servers is as intermediate cache layers between the application and the database, storing and retrieving data sets which are slow or complex to query directly from the database. In this arrangement, the cache server can be shared by application servers to make effective use of common data cached between application instances, virtually acting as a tier in the hierarchical memory model. A classic problem of all data caching techniques is data expiry, for which the cache servers need to provide adequate support.

Dedicated cache servers (like memcached [36]) are optimized for performance of their most commonly used operations, sacrificing all other database functionalities (such as ACID properties or complex data models). An important aspect of this is the use of main memory for storage as maximal performance is achieved if the whole data set fits in the system RAM (which is not preserved across server reboots or other events which cause discontinuations of the cache server operation). The orientation towards performance also manifests in the implementation of simple data models (in popular cache servers it is exclusively the simple key-value pair model) and in the support for only a restricted set of operations – usually only PUT, GET, DELETE are implemented, with occasional support for atomic increment and decrement on (specially formatted) data record entries [36][76][77].

One of the observations this dissertation explores is that the benefits of cache servers can be exploited in both the technical and the business aspect of Web application deployments. As general-purpose data storage accelerators they can help build better performing applications or they can enable application feature growth with the same performance characteristics. From the business point of view, they can enable savings in the amount of server infrastructure needed to provide services to a certain number of users. Their usage is wide-spread and it would be fair to say that cache servers are one of the principal enablers of today's complex and global Web applications.

4.4. Previous work

Cache servers for Web applications are one of the building blocks of high performance Web applications and are responsible for much of the good performance of some major contemporary Web sites like Wikipedia, LiveJournal, Filckr, YouTube, Digg [36], Zynga, ShareThis [77] and companies like VMWare [76]. Several high-profile state of the art projects with similar features are available with relatively similar features, of which the most important are Memcached, Membase and Redis, used at the previously listed sites respectively. Their most important features and drawbacks are highlighted in Table 1.

Project name	Notes	Features	Drawbacks
Memcached	The first and very popular cache server	Simple key-value store with atomic operations, the most popular Web cache server	Very simple data model, asynchronous replication
Membase	Created as an alternative to Memcached	Network protocol compatible with Memcached, multi-tenancy, replaceable storage layer, optional data persistence	Very simple data model, asynchronous replication
Redis	Created to offer a richer data model	Internal support for complex data types in records (lists, sets, bitmaps, with associated operations), optional data persistence	Single-threaded, only master-slave replication

Table 1: Characteristics of existing major Web cache servers

Common features shared by all projects in Table 1 are the use of the key-value data model as the central model, fast operation provided by in-memory data storage, and

at least some type of asynchronous data replication between servers of the same type. The projects are also uniformly created for the Unix-like (or POSIX) environments, and at least to some extent make use of advanced event-based IO APIs. Of the listed projects, Memcached is by far the most popular cache server and the one with the most active development, so it is suitable as a baseline for comparison in this dissertation.

A goal of this dissertation is to design a Web cache server which would extend the capabilities of existing solutions and address some of their shortcomings.

5. Requirements for a new cache server model

While cache servers are widely deployed in high scale applications, they offer a simplistic and limited data model (making it harder to design applications with complex caching requirements), sometimes incomplete adaptation to multi-processor environment (resulting in underutilization of hardware) and with less attention spent on data consistency in replication (which can cause data loss or even performance problems).

Based on the needs for development and large scale deployment of Web applications, the proposed hypothesis of this dissertation is that the following improvements would make the biggest impact on the efficiency of the cache server and the ease of implementation over the existing solutions:

- A more complex data model, allowing for more complex data queries and for more complex processing to be performed directly on the cache server
- A program architecture optimized for contemporary multi-core processors commonly found in industry standard servers
- A model of durability based on synchronous data replication with predictable performance and data coherency

These improvements to the contemporary cache server architectures are designed to enable faster and more productive development of scalable and feature-rich Web applications. In order to be competitive with the currently widely deployed cache servers, the new model adopts some existing constraints and optimizations: the cache server should effectively be a memory-only database, with a data model considerably simpler than that of general-purpose databases, with simple atomic operations and

without multi-operation transactions. Thus, the new cache server will not attempt to be a general-purpose database but is designed for a specific mode of use.

5.1. Data model and supported operations

The basic form for an addressable database is a store of records made of key-value pairs (a dictionary), where both the key and the value are more or less opaque binary strings. The keys are conceptually treated as *unique* addresses by which the values are stored and accessed. Because of the simplicity of this model, it can be implemented efficiently and it is often used for fast and robust databases [78]. As a simple and robust model, it is often used as an architectural primitive on top of which more complex data models can be built. However, pure key-value databases can be limiting and inflexible and complex applications would benefit from a more complex cache data model, as demonstrated in [79].

The new cache server's data model should be based on the key-value record data model with simple timed expiry at its core but it should extend it with user-defined “tags” so that each key-value data record stored in the cache server can be additionally augmented with an arbitrary number of specially formed record tags. The structure of these tags should also follow the key-value model but with a limited and rigorous format which maintains high performance of common operations: both the tag key and tag value data are to be of strictly enforced data types, namely signed integers. The intent behind the introduction of such limited tags is to enable applications to assign custom metadata to the key-value records (which leads to the possibility of implementing queries which reference not only record keys but also such metadata) while at the same time holding efficiency and performance as key design goals. Such tags can be viewed as a means for classification of cache records for the purpose of extending the flexibility of certain operations. This extension to the basic key-value data model, is simple but hopefully powerful enough to significantly extend the functionality and usefulness of the new cache server, allowing for easier implementation of more complex applications. An overview of the structure of the new records is depicted in Figure 1.

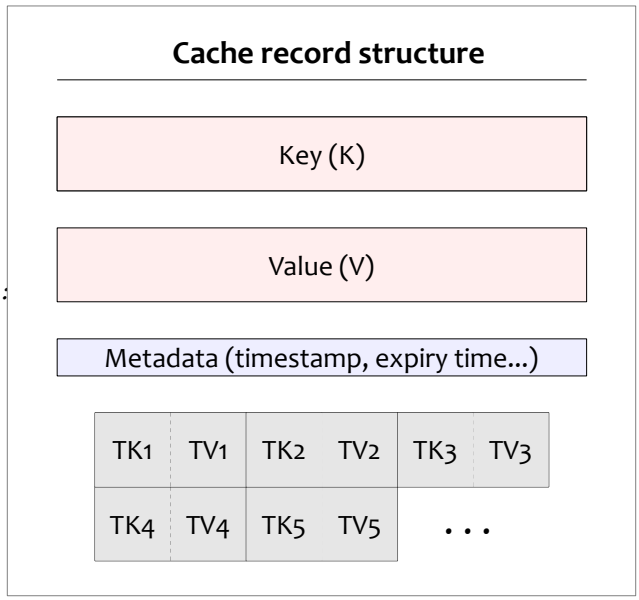

Figure 1 Main elements of the new cache server record

The data model should be accompanied by the additions to the basic set of cache server operations, supporting conditional and ranged queries referencing tags keys and tag values. The choice of the additional supported operations (at least in this phase) was governed by performance concerns. If the records are described as having a key K and value V, and one or more tag keys TK_n and tag values TV_n, the additional operations are described in a pseudo relational query syntax in Table 2.

Additional Cache Operations
PUT (K, V), (TK_1, TV_1) [, (TK_2, TV_2)...]
GET K, V WHERE TK = $TK AND TV IN ($$TV_1$, [$$TV_2$...])
DELETE WHERE TK = $TK AND TV IN ($$TV_1$, [$$TV_2$...])

Table 2: Additional cache operations supported by the new data model

The new operations emphasize the use of tag keys as data types or data groups, with the tag values as specific object identifiers within the type or group. One particularly useful application of this scheme is for describing cached records as belonging to a particular user, page, page object or a business object, enabling queries such as “GET records belonging to a certain object” or “DELETE records belonging to a certain page.” The latter example demonstrates a frequent operation required for efficient cache data expiry; without metadata tagging, relationships such as those between a

page (which is a high-level Web application entity) and individual cached records (which are low-level data objects) would have to be stored either in the application's private data or in another record in the cache server whose updating may lead to race condition errors in concurrent updates due to the simple “PUT & GET” model of operations.

The introduction of tagging in the cache server is expected to be a large step in flexibility for application developers, enabling the development of more complex application features which are supported by a more complex data model in the cache. With it, the cache server can (if needed) become a part of the application instead of a peripheral subsystem not normally accessed from the application business logic.

5.2. Program architecture for multi-core processors

Contemporary industry standard servers are built around SMP and small-scale ccNUMA processor architectures, that is to say either with memory which is uniformly shared across all of the processor cores (illustrated in Figure 2) or with processors with their own locally attached memory (as in Figure 3) but with enforced memory coherency on the hardware level [80][81].

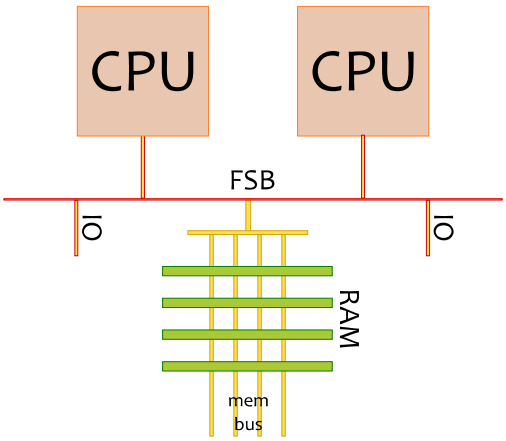

Figure 2: SMP / UMA - Symmetric multiprocessing, uniform memory architecture illustration

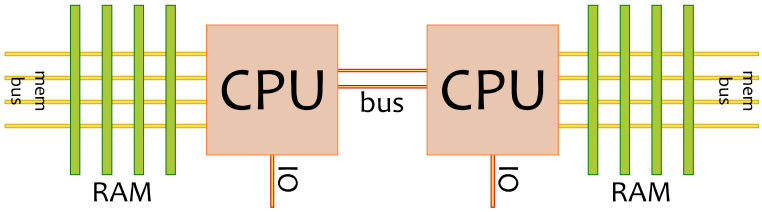

Figure 3: NUMA - Non-uniform memory access architecture illustration

The benefits of the SMP architecture are centred around its relative simplicity and the symmetry of memory and IO access arbitrated by the use of the *front-side bus* (FSB), while the NUMA architecture offers potentially significantly lower latencies in memory and IO access if they reference the particular CPU's memory and IO space. The most common number of processor sockets in industry standard servers is two, but varies from one on the low end to eight on the high end (servers with a higher number of sockets are rare and implement specialized hardware which moves them away from industry standards). The total number of processor cores (not counting technologies such as Hyperthreading) is approaching 64 in high-end servers but is more commonly around 16 [82][83][84].

In order to make use of the multiple available CPU cores, data structures and algorithms must be adapted to allow parallel execution on multiple cores, generally aiming to be as efficient as possible in the face of Amdahl's law [41][85]. This goal is made difficult by real-world constraints on several layers: the application, the operating system and the hardware. Within these, this dissertation will focus on three aspects:

1. Interaction with the operating system,
2. Distribution of tasks and resources across CPU threads of execution, and
3. Allowing parallel access to the cache server structures in memory.

These aspects together govern the program architecture and have a very direct influence on its actual performance. This research has included a study of several variations of the listed aspects and as one of its results the implementation of multiple techniques and algorithms (where applicable) to allow direct comparison between

them. Some of these techniques and algorithms have direct influence on client application architecture patterns.

5.2.1. Interaction with the operating system

The new cache server should be portable across operating systems in order to ease its implementation in various environments and provide easy access to its client applications. The dominant platform for wide scale server deployment (mostly because of free implementations in the form of Linux and BSD variants) is a Unix-like POSIX environment [14][34][56][86], which has influenced the decision to make the new cache server POSIX compliant. Apart from the standard general-purpose application programming interfaces, it should use asynchronous and event-driven IO operations and the *pthreads* API for multi-threading and inter-thread synchronization.

5.2.2. Task distribution across CPU threads

The workings of a cache server can be divided into several distinct groups of tasks, the most important of which are connection handling, network IO handling and query processing. The distribution of these tasks across CPU threads has a large influence on its performance, and some models of this distribution may be more suitable for certain tasks than others. For this reason, the cache server should architecturally support several multithreading models known in literature: single-process event-driven (SPED), staged event-driven architecture (SEDA), asymmetric multi-process event-driven (AMPED) and symmetric multi-process event-driven (SYMPED) [87].

5.2.3. Allowing parallel access to in-memory cache records data

The well known Amdahl's law dictates that the gains from parallelism are in the largest part governed by the tasks which cannot be parallelized. Since the hardware memory model for which the cache server is targeted offers a coherent view on all of the memory to all running threads (as described e.g. in [88]), it is possible to implement a synchronization model which allows parallel read operations and only requires waiting for write operations. The literature describes this topic extensively and from different points of view – from low level methods used to ensure proper ordering of operations (such as critical sections [89], reader-writer locks [90], monitors [91] and

non-blocking algorithms [92]) to more high-level consistency models used for both multiprogramming and distributed computing [93] (of which the most important ones for contemporary highly distributed systems are serializability [94], multi-version concurrency control and eventual consistency [95]). Using some of these techniques, the new cache server should minimize the amount of blocking while accessing shared data, while allowing maximum throughput for (read) operations on non-shared data.

5.3. Durability through data replication

Though the commonly implied goal behind the use of caches of any kind is to improve a system's performance, it is also implied that the original operations themselves *could* be performed even without the caches but with lower performance. However, practical use usually includes complex interactions where this is not so simple. Many advanced computer users are familiar with situations which arise when various levels of operating system caches become inoperable or suboptimally configured (most notably the disk caches) and such situations are proportionally worse on servers with more demanding workloads. Shutting down a cache server for a Web application can be compared to disabling operating system disk caches – most systems would not be able to continue operating with acceptable performance [34]. Additionally, application architecture may not even allow operation without a cache server as that could translate to losing the functionality of an API layer.

Due to the nature of the cache servers, they cannot use slower persistent storage devices to achieve durability of data and must implement durability with systems of comparable speed and capacity – i.e. other cache servers. The most common way of achieving such an arrangement is deployment of identical cache servers which replicate their data over a local network, with the obvious downside of increased latencies (even 10 Gbit/s Ethernet links are very slow compared to internal server buses which offer a few orders of magnitude better performance). With simpler cache servers such replication may be implemented at the application layer by having the application push data (e.g. execute PUT commands) on multiple cache servers and pull data (e.g. GET) from one of them, chosen randomly or in round-robin fashion. Cache record invalidation could be implemented in an analogous way. While it is simple to implement

(and its implementation does not rely on server-side support), this method has a downside of possible temporary loss of data coherency between replicas (i.e. its operations are not atomic).

The new cache server should implement a model with stronger guarantees – synchronous multi-master replication specifically intended for improving durability by replicating data between a small number of servers connected with high-speed network links.

6. The model of a new cache server

Requirements described in Chapter 5 are not met by existing cache server products, which has led me to propose the model of a new cache server which supports the following functionalities:

- **Startup and management.** The cache server is configurable from the command line with at least these options: network parameters, the threading model and resource limits.
- **Communication with client applications.** As communication channels, the cache server offers TCP/IP and Unix domain sockets. The protocol used in this communication emphasizes performance over convenience of operation.
- **Data model**. The cache server offers a data model centred around a key-value records, with the addition of arbitrary integer key-value tags to each record.
- **Cache operations.** The cache server offers the following data operations: simple PUT, GET and DELETE (by one or more keys), tagged PUT, GET and DELETE (described in table 2), simple atomic ADD, SUBTRACT and CMPSET (arithmetic and synchronization operations on a single record).
- **Concurrent access**. The cache server can be used by multiple clients at once, with efficiency and flexibility.
- **Data replication**. The cache server offers synchronized multi-master replication of data between several identically configured cache servers.

In order to implement these functionalities, the cache server needs a careful program design which would allow the features to be implemented efficiently. It can be divided functionally into the following modules and tasks:

1. Operating system interfaces and basic program infrastructure
2. Network setup
3. New connection processing
4. Network IO processing
5. Network protocol parsing
6. Database data structures and algorithms
7. Database query processing
8. Replication processing

The relationships between these modules and tasks are presented in Figure 4. The network setup task, the network connection handler module, the network IO handler module and the replication module interface directly with the operating system while the network protocol parsing module, the database query processing module and the data storage module do not need to communicate with the operating system directly. In order to support multiple multithreading models, the connection handler module, the network IO handler module and the network protocol parsing module communicate (in general) by using synchronized queues, while the remaining modules use direct calls (where applicable).

The modular approach outlined in Figure 4 allows experimentation in the design of the data structures, multithreading models and queuing techniques. Tasks are labelled T1 to T4 and they can include multiple modules (exemplified by task T3 which contains “payload work” done on behalf of the client applications).

Support for multithreading is implemented by carefully decoupling the operation of various tasks. Tasks T2 and T3 are almost completely isolated from the rest of the program by using job queues and as such can be instantiated in an arbitrary number of CPU threads. Task T1 is generally intended to be instantiated at most in one thread because at the lowest level it is dependant on individual server sockets and its workload is not complicated so it cannot be expected to be improved by introducing parallelism (which agrees with other research, e.g. [96]).

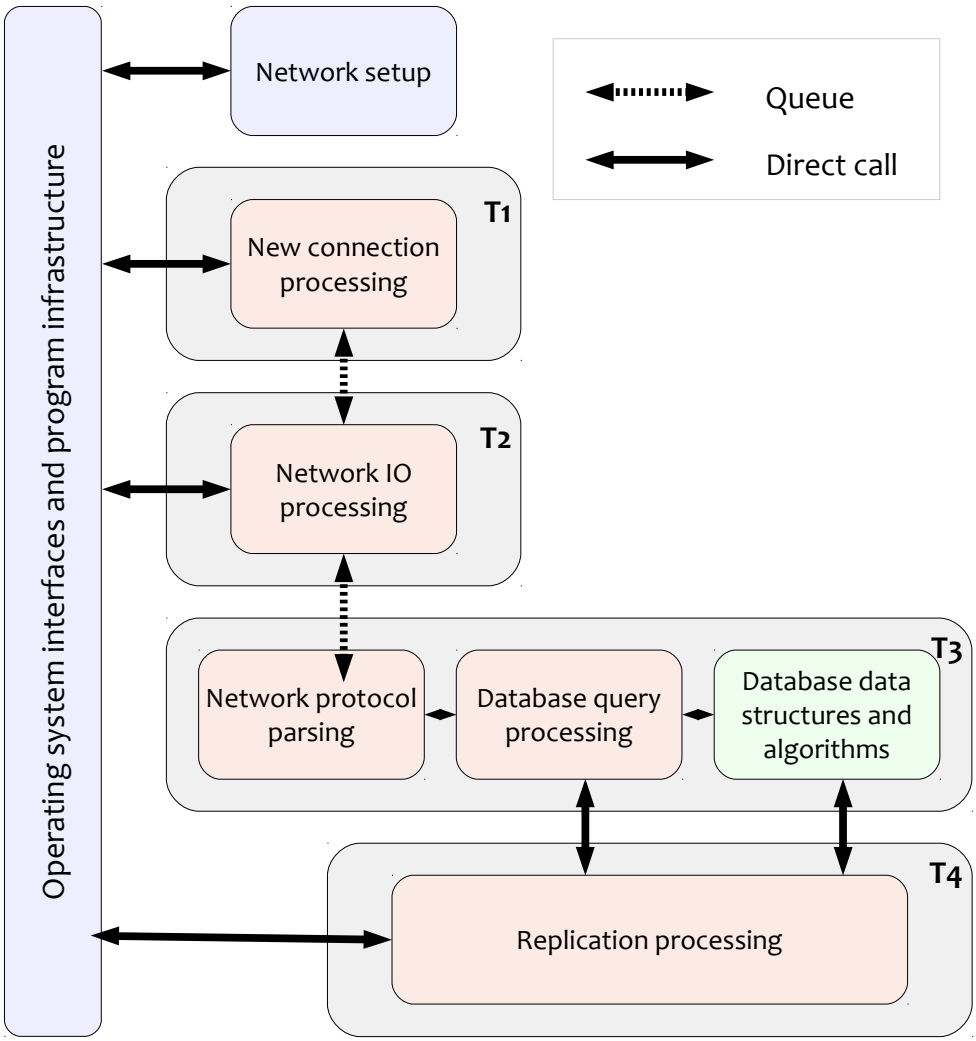

Figure 4: Modules and tasks of the new cache server

Task T4 containing the replication module is a special case, as for reasons of ensuring data consistency it must participate in concurrent access to data structure in response to both local changes and events from remote replicas, for which it instantiates additional threads.

6.1. Interaction between threads

The new cache server uses the POSIX threads (*pthreads*) API for thread management and synchronization. The *pthreads* API is a fairly complete API specification with a

medium level of abstraction for realizing proper multithreaded programs. Of its many supported features, the new cache server uses the Thread, Mutex, Condition Variable and Read/Write Lock parts of the API to achieve controlled parallel execution in multiprocessor environments.

In this dissertation, the term "multithreading model" refers to a distinct way tasks in an IO driven network server are distributed to CPU threads (specifically, tasks T1, T2 and T3 from Figure 4). It generally concerns itself with the general notion of threads dedicated for certain type of tasks, rather than the quantity of threads or tasks present at a time (except for some special or degenerate cases like 0 or 1). A typical network server (and specifically also in the new cache server) has three types of tasks that can be parallelized or instantiated into different CPU threads:

1. Accepting new client connections and closing or garbage collecting existing connections.
2. Network communication with accepted client connections.
3. Payload work that the server does on behalf of the client, which includes generating a response. This task class contains all operations the server needs to perform in response to a command or input received from the network.

These types of tasks map into tasks T1, T2 and T3 in Figure 4. Several models for distribution of such tasks are recognized in practice and literature [97]. At one part of the spectrum is the *single process event driven* (SPED) model, where all three types of tasks are always performed by a single thread (or process), using kernel-provided event notification mechanisms to manage multiple clients at virtually the same time, but actually performing each individual task step sequentially. This model is characterized by the presence of a single "event loop" program construct which receives IO events from the operating system, processes them one by one and blocks waiting for new IO events. On the opposite side of SPED is *staged event-driven architecture* (SEDA), where every task is implemented as a thread or a thread pool and a response to a request made to the client might involve processing in multiple different threads.

In the new cache server the SPED model is implemented within the connection accepting thread (running task T1) with bypassed queueing between it and the task T2, and between tasks T2 and T3, by using direct calls to complete requests. This is made possible by the fact that the number of threads for all tasks is known in advance (at server startup and configuration). In contrast to this mode of operation, SEDA is the generic model where each of the tasks is instantiated as an arbitrary number of threads.

Between the extremes are the *asymmetric multi-process event-driven* (AMPED) model where the network operations are processed in a way similar to SPED (i.e. in a single thread) while only the payload work is delegated to separate threads to avoid running long-term operations directly in the network IO loop and *symmetric multi-process event driven* (SYMPED) which uses multiple SPED-like threads, each processing several client connections. AMPED is achieved in the new cache server by handling network IO (task T2) from the same thread as the connection handler task (task T1) and SYMPED by grouping tasks T2 and T3.

A special class of AMPED is a *thread-per-connection* model (usually called simply the multithreaded or multiprocessing – MT or MP – model) where connection acceptance and garbage collecting is implemented in a single thread which delegates both the network communication and payload work to a separate thread for each connected client. This model will not be specially investigated here as there is a large body of work already covering it and because it exhibits performance degradation as the number of clients rises [98].

This flexibility of the new cache server in testing multithreading models is unique among publicly described software of comparable type.

The experience with multithreading models in [99] led to an interest in certain edge cases – in particular, reducing unwanted effects of inter-thread communication and limiting context switching between threads. This research has also touched on a previously undistinguished variation of the SEDA model where the number of worker

threads exactly equals the number of network threads and each network thread always communicates with the same worker thread, avoiding some of the inter-thread locking of communication queues; this model, previously not specially discussed in the reviewed literature, was given the name SEDA-S (for *symmetric*).

Each of the described multithreaded models can also be implemented with multiprocessing, and in fact some of them are more well known in this variant (the process- or thread- per connection model is well known and often used in Unix environments), but this work is focused on the multithreaded variants. The configuration of tasks and threads in relation to various multithreading models is summarized in Table 3.

Model	New connection handler (T1)	Network IO handler (T2)	Payload work (T3)
SPED	1 thread	In connection thread	In connection thread
SEDA	1 thread	N_1 threads	N_2 threads
SEDA-S	1 thread	N threads	N threads
AMPED	1 thread	1 thread	N threads
SYMPED	1 thread	N threads	In network thread

Table 3: Supported multithreading models

This division of tasks into multithreaded models closely follows the description of the models available in literature, but is of course adapted to this specific program.

6.2. Operating system interfaces and program infrastructure

The new cache server is created in a mix of C and C++ and limits itself to the common Unix-like operating system interfaces, mostly those documented as POSIX standards. It has been successfully tested for portability on two major such environments: Linux and FreeBSD. Of the advanced features offered by the operating system, only POSIX threads and asynchronous network IO are used, making the program self-contained and independent of third-party libraries.

The program is configured from command line arguments. Among the configurable features are the threading model, logging, cache contents dumping and pre-warming, and the list of replication peers. The main program thread performs network setup

and creates other appropriate threads, then waits until all threads exit before ending the process.

6.3. Network setup

The new cache server offers its services over stream-based communication channels: TCP and Unix domain (or “local”) sockets, treated equally. A small number of optimizations are applied to all sockets (client and server): turning off of “Nagle's algorithm” for reduced latencies (in case of TCP) and explicit configuration of network buffers to sizes expecting to hold average incoming and outgoing messages.

6.4. New connection processing

The new connection handler (task T1) is always instantiated in a single thread. Its main workload is asynchronously accepting newly connected client sockets from the operating system (i.e. “listening” on server sockets), configuring them and enqueuing them into the asynchronous network IO delivery mechanism. As a special case, it can enlist the client sockets into its own IO delivery queue, supporting the SPED model.

6.5. Network IO processing

Modern operating systems support both synchronous and asynchronous IO operations (as described from the point of view of how they report their status and completion to the caller) in various forms. Asynchronous, event-based IO operations are more efficient as they push a larger part of the task into the operating system kernel where they can be executed more efficiently and in bulk [100][101]. In effect, this network IO architecture notifies the program when one or more IO events become available for processing. Performance gains resulting from this type of interaction between the program and the operating system are the primary reason why they are used in the new cache server.

The prevailing and recommended model for building high-performance network servers for a long time was based on single process asynchronous and event-driven IO architectures in various forms [98][102][103]. Shortcomings of this model grew as the number of CPU cores available in recent servers increased. Despite the complexities

present in creating multithreaded programs, multithreading programming was inevitably accepted as it allows fuller use of hardware possibilities [104][105].

All network IO in the new cache server (including the network IO within the new connection handler) is performed by using asynchronous IO functions of the operating system. Since this is a performance sensitive area of the program, advanced APIs are used to ensure maximal efficiency – Linux epoll [106] and FreeBSD kqueues [101]. Messages received from clients in task T2 are checked for consistency and queued for delivery (or passed as a direct call in case of SPED and SYMPED) to task T3 for parsing and execution.

An important optimization for efficiency of network IO is in the scaling of the network IO buffers. For optimal buffer sizing, the sizes of cache data requests during regular operation (8 hours on a work day) of www.fer.hr have been investigated, and the results are presented in Figure 5.

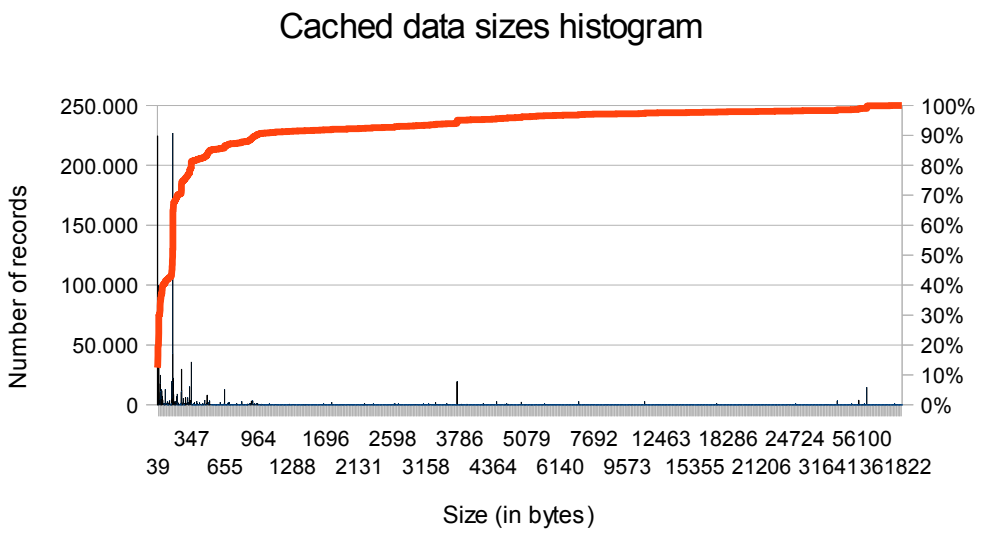

Figure 5: Sample of cached record sizes during regular usage of www.fer.hr

The analysis of cache data requests from Figure 5 showed that approximately 90% of cached data records (per number of records) are smaller than 1000 bytes (and approximately 96% are smaller than 4096 bytes). This information is used to scale the "average record size" and IO buffers in the new cache server to 1 kB and 4 kB, respectively.

6.5.1. Efficient event scheduling

The initial implementation of IO processing in the new cache server mapped the operating system's event-based IO capabilities to program behaviour in a direct and simple way. During the performance evaluation of the new cache server with the described architecture, the analysis of the program IO path revealed that in the case of large transaction per second loads, the distribution of IO events received by the program was inefficient, distributing events received from the operating system one by one to the worker queues. To improve this situation, a different model was created and implemented. In the new approach, events are received from the operating system in as large numbers as possible and distributed to worker threads (depending on the multi-threading model) in bulk, while at the same time avoiding locking operations on the worker queues. This approach significantly increased the complexity of the interaction with worker queues and the associated locking, but has resulted in measurably increased performance in situations with a large number of clients and high loads (as described in chapter 8).

6.6. Network protocol processing

The protocol used for communication between the client applications and the new cache server is designed to be light-weight and requiring minimal parsing. It is a binary protocol working directly with hardware data types (without the need for translations such as for different endianness or alignment requirements), allowing direct access to protocol data inside the network IO buffers. The primary reason for choosing this type of protocol (rather than a text-based protocol or a more descriptive binary protocol) is the performance advantage which is gained by having only minimal protocol parsing. The client and the server can test their compatibility in areas such as endian-ness in the protocol handshake.

The messages used in the protocol are of uniform structure which includes (among other data, detailed in Figure 6) the message type and the message's full size. The existence of the size field enables the server to perform at most one more memory (re)allocation if a message is received which does not fit into expected buffer size.

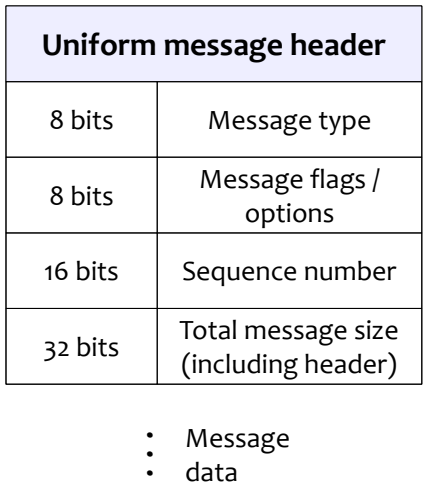

Figure 6: Uniform network message header

All fields are unsigned integers unless otherwise stated. The uniform message header is of minimal size (only 8 bytes) and optimized for efficiency, containing only the essential data to adequately retrieve and fetch the rest of the message. Because of this its design contains some minor compromises: there can be only 256 different message (which is not a significant limitation), only 8 message flags and the message sequence counter overflows every 65536 messages (which is deemed enough to distinguish messages in processing and pair them to their responses even at high message rates). The message data following the header can be variably sized, depending on the payload. A typical message (PUT) is described in Figure 7.

The PUT message, used to insert (or overwrite) a single record into the cache, is a common variably-sized message. It begins with the uniform message header, the number of tags, the sizes of the record key and data (key length is limited to 64 KiB − 1 byte, data length to 4 GiB − 1 byte), the expiration time (expressed in the Unix integer timestamp format) and then contains the variably-sized tags, each a pair of two 32-bit signed integers, following with the key data and the value data.

PUT message	
64 bits	Uniform message header
16 bits	Number of tags (ntags)
16 bits	Key size (ksize)
32 bits	Value size (vsize)
32 bits	Record expiration time (exptime)
ntags*64 bits	ntags pairs of (TK, TV)
ksize*8 bits	Key data
vsize*8 bits	Value data

Figure 7: PUT message structure

The decision to use 32-bit integers for tags instead of 64-bit integers, which are currently natively supported by industry standard servers, was made because 32-bit integers offer a good trade-off between a (compact) data size and the size of the namespace they can represent. This choice can be easily changed at compile-time.

When received, such a message is stored directly and the parsing step only involves calculating and storing additional pointers to the tags, name and value data in additional internal structures, for faster direct access.

6.7. Database data structures and algorithms

The primary concern for the design of data structures and algorithms was their efficiency with respect to two facilities:

1. Quick access to data
2. Maximal concurrency of data access (from the viewpoint of simultaneously connected clients)

The first facility is a necessity for high-performance applications in general, and thus also for cache servers where the high speed of data access is a highly desired feature. The choice of structures and algorithms for data storage has a huge impact on the performance curve of the cache server as the number of stored records increases. A frequent choice for the purpose of fast indexed data access is the tree structure due to its simplicity and applicability to datasets whose size is not known in advance. For this model the "Red-black tree" variant of the structure (originally introduced by [107]) was chosen, as it has the following useful properties [108]:

- Trees with n internal nodes have a height of $O(\log n)$
- All operations on the tree (SEARCH, INSERT, DELETE) have strong worst-case complexity of $O(\log n)$.

Red-black trees trade a somewhat complex implementation for proved complexity bounds, making them suitable for real-time applications[7]. Though with desirable performance characteristics, the Red-black trees (like all balanced trees) are difficult to implement with concurrent write access (INSERT and DELETE) as the nodes need to be shuffled (rotated) based on a balancing criteria, which would require an extensive and very careful design of concurrency control (locking). In order to support concurrent write operations on records, a composite data structure was designed consisting of a hash table whose elements (buckets) are Red-black trees, shown in Figure 8.

This data structure breaks down the need for locking into separate locks for each of the hash table buckets, enabling concurrent write operations on separate trees. Consequently, in such an arrangement, all nodes of a single tree hash to the same

7 Though the new cache server does not specifically attempt to have real-time characteristics, it could have at least soft real-time characteristics if such characteristics are supported by the network IO and if the number of records in the cache and the number of simultaneous clients is known or restricted.

hash value, making the effects of a bad hash function amplified through trees of different height[8].

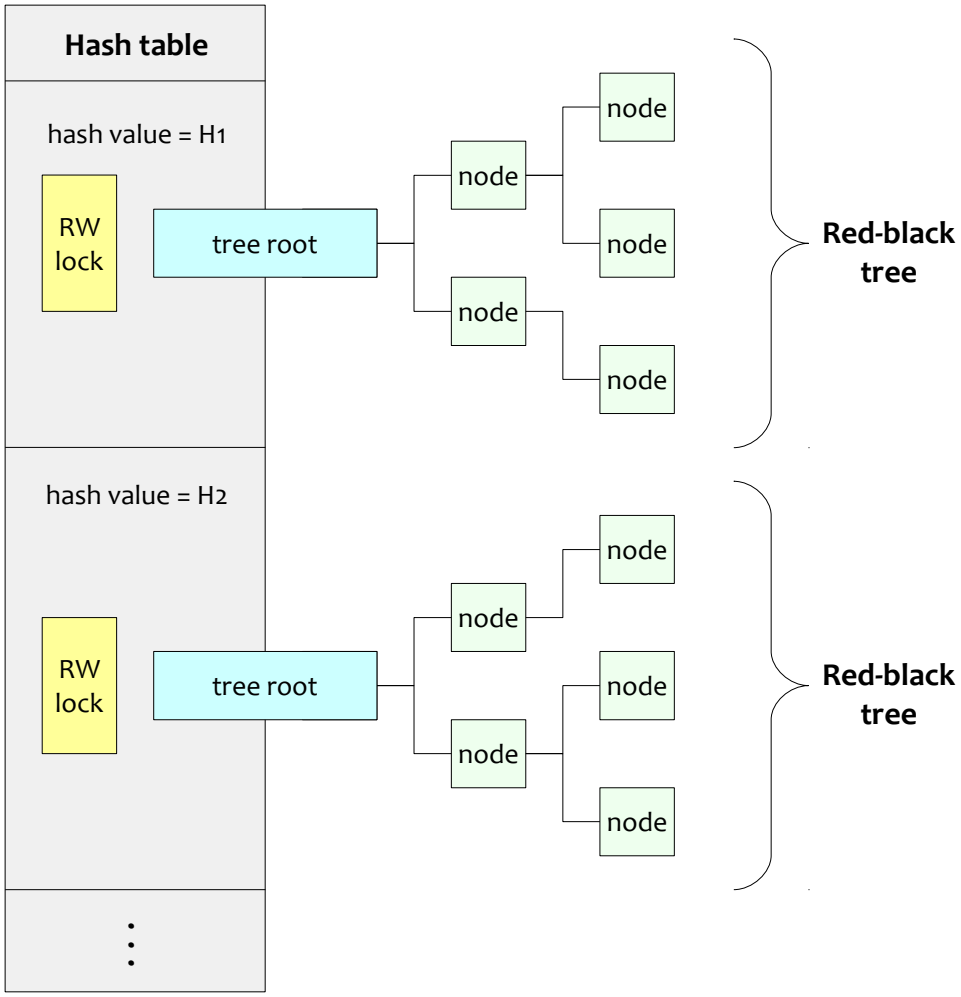

Figure 8: Data structure for cached records indexed by keys (from [99])

The expected algorithm time complexity of this composite structure, without taking into account lock contention artefacts, is $O(\log(n) / H)$ where n is the number of items stored in the composite structure and H is the number of hash table buckets (and also the number of locks).

8 To counter this, a relatively novel but well tested hash function called MurmurHash2 [109] is used.

6.7.1. Expected performance under contention

The data structure locks are implemented with *pthreads* reader-writer locks (also called shared-exclusive). This type of locks is the best match for the purpose as it allows any number of read transactions to be performed concurrently on a tree (which is allowed by the Red-black tree structure as it isn't self-adjusting for read requests as some other tree structures are, notably the Splay tree), while write transactions require exclusive access as usual. As a cache server is ideally a read-mostly database, the ability to execute concurrent read transactions is highly desirable.

Reader-writer locks can be implemented either with reader priority or with writer priority, with respect to behaviour in the case when a locking request of one type arrives for a lock which was already acquired by another thread with the opposite lock type. It is intuitively obvious that since there can be a large number (theoretically infinite) of simultaneous read acquisitions on a lock but only one write acquisition, implementing reader priority would result in writer starvation as the writers would have to wait too long until all readers (including newly arrived ones) release their locks. Writer priority avoids this by implementing a special case where a write lock acquisition causes all further attempts of read acquisitions to be put on a waiting list until that write lock acquisition is performed and released. Inspection of the source code for popular Unix-like operating systems (Linux, OpenSolaris and FreeBSD) confirms that the writer priority scheme is more commonly implemented.

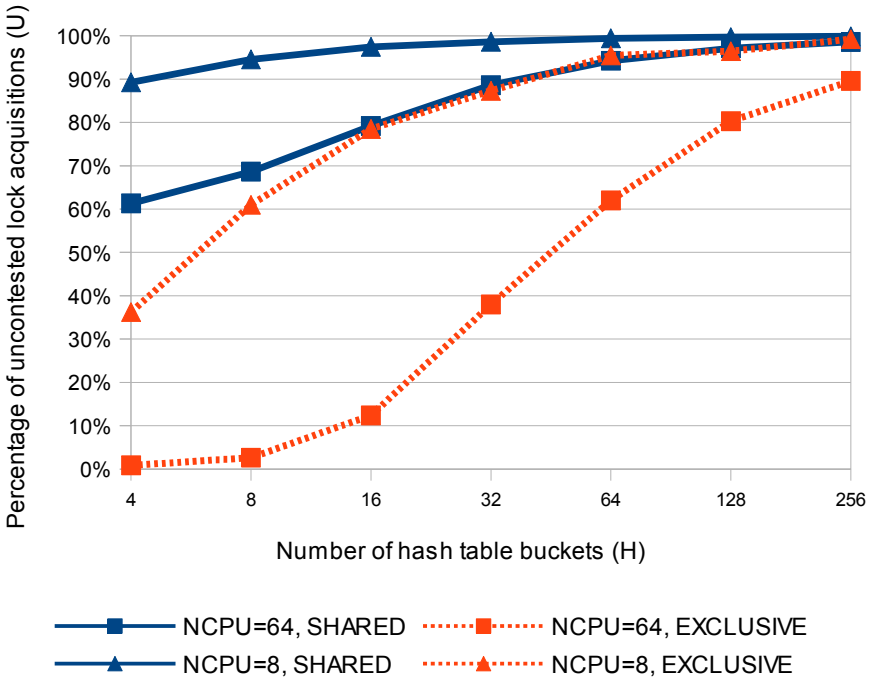

Figure 9: Data structure lock contention with 90% readers and 10% writers (from [99])

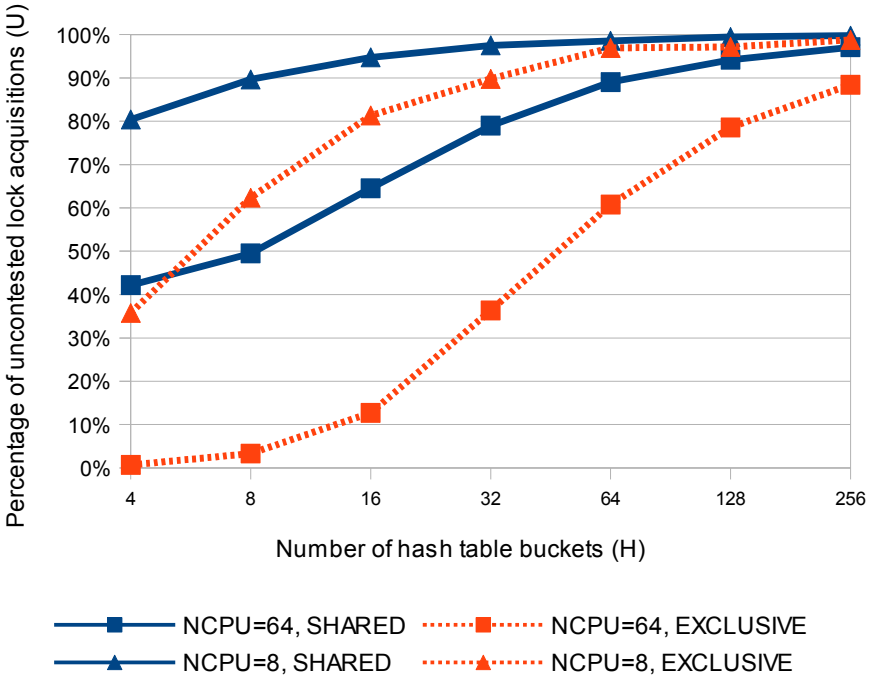

Figure 10: Data structure lock contention with 80% readers and 20% writers (from [99])

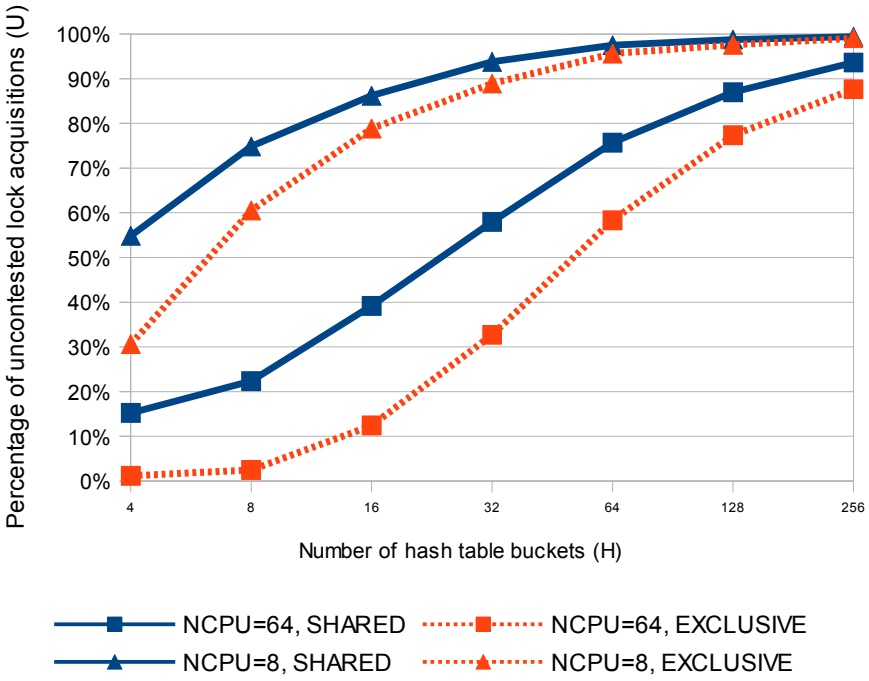

Figure 11: Data structure lock contention with 50% readers and 50% writers (from [99])

In order to study the expected behaviour of the designed data structure with respect to locking contention in multi-core and multi-user environments, a GPSS simulation was created which simulates a system where a large number of transactions is simultaneously arriving to a pool of locks which may be acquired either for shared or for exclusive access. The simulations were run with different ratios of transactions requiring shared and exclusive access and their result was expressed as the percentage of uncontested (fast) lock acquisitions the transactions have been able to perform. At the boundary conditions, a system in which all lock acquisitions are shared would have 100% of uncontested lock acquisitions in all cases (per definition of a shared lock acquisition), while a system in which all lock acquisitions are for exclusive access, the percentage of uncontested lock acquisitions (with multiple parallel transactions) would be nearly 0%. These results were plotted in Figures 9, 10 and 11 for ratios of shared to exclusive locks of 90%:10%, 80%:20% and 50%:50%, respectively, and for two cases of simulated parallel execution: with 8 CPU cores and for 64 CPU cores (limiting the number of transactions which may execute an operation simultaneously). The CPU core counts have been selected to represent those currently widely deployed on

industry standard servers and those estimated to be available, at least in high-end equipment, in the following years.

The results (first published in [99]) show that in the cache-friendly case with 90% read transactions (Figure 9), over 90% of those read transactions can be processed without significant lock contention for all significant hash table sizes (8 and above) for the case of 8 CPU cores. More generally, an interpolation of these results shows that almost 95% of read transactions can be processed without significant lock contention in a system with N CPU cores if the size of the hash table H is equal to or larger than N. Write transactions, requiring exclusive locks, move this percentage down to around 60%. The shapes of the curves in Figures 9, 10 and 11 follow the expectations set by the Amdahl's law (Equation 1) and indicate that this data structure can be expected to allow a very high level of parallelism in practical applications.

$$S(N) = \frac{1}{1 - P + \frac{P}{N}} \qquad \text{Equation 1}$$

In Equation 1, the variable P represents the portion of the task which can be parallelized, and N represents the number of parallel execution units (CPUs). The data points resulting from these simulations can be fitted to functions which are inferred (and generalized) from the original Amdahl's law (Equation 2).

$$U(H) = A + \frac{B}{1 - P + \frac{P \cdot N}{H}} \qquad \text{Equation 2}$$

Note that the common graph representation of Equation 1 presents speedup S as a function of the number of available CPUs N, while graphs in Figures 9, 10 and 11 as well as the function in Equation 2 present the percentage of uncontested locks U as a function of the number of hash buckets H. The similarity with Amdahl's law is thus by analogy, and the variables A and B are free fit factors.

The record tag key-value pairs from data records are also organized to serve as indexes for fast lookup, and in this model they are stored in structures similar in design to the composite data structure storing the records, with the added layer which groups tags with the same key values for increased performance of lookup operations. The tag keys are hashed in a table with shared-exclusive locks, which contain Red-Black trees with tag values, in which each node contains a list of pointers to records that hold the relevant key-value pairs. This structure is optimized for queries which lookup all records containing a particular tag key-value pair or only a specific tag key.

6.8. Database query processing

The complete list of queries supported by the new cache server (extending the list in Table 2) is given in Table 4.

List of the New Cache Server's Data Operations	
1.	PUT (K, V)
2.	GET K1 [, K2, ...]
3.	DELETE K1 [, K2, ...]
4.	ATOMIC_INCREMENT K, N
5.	ATOMIC_CMPSET K, V1, V2
6.	PUT (K, V), (TK1, TV1) [, (TK2, TV2)...]
7.	GET (K, V) WHERE TK = $TK AND TV IN ($TV1, [$TV2...])
8.	DELETE WHERE TK = $TK AND TV IN ($TV1, [$TV2...])

Table 4: Entire list of the new cache server's data operations

Operations 1 through 3 in Table 4 are the usual database operations found in key-value databases, where the PUT operations doubles (atomically) as UPDATE if a record key already exists, completing the CRUD (Create, Read, Update and Delete) set of basic operations on a database [110]. Operations 4 and 5 are present for efficient implementation of some client-side operations such as counters and shared locks, and while they are not in any way required operations, they are a convenient addition implemented by several key-value databases (ATOMIC_INCREMENT variant in [36], [76], [77], ATOMIC_CMPSET variant in [36], [77]). Operations 6 through 8 are the additional operations of the new cache server that are aware and can make use of tags. With regards to locking, all operations acquire locks first and implement the op-

erations next, executing a rollback-and-retry operation if all the necessary locks cannot be acquired for the operation (a variant of Two-phase locking as described e.g. in [94]). A consequence of this locking scheme is that essentially all operations can be considered “atomic” in the sense that operations can not operate on half-completed results of other concurrent operations.

The semantics of all these operations are described in the following sections, grouped by the operation type.

6.8.1. PUT operations

Both forms of PUT operations (numbered 1 and 6 in Table 4) insert a single record in the database, differing only in the presence of tags. Internally, both operations share much of the code path and behave almost the same. In case the operations find an already existing record with the specified key, they will atomically replace it and signal this to the client. The PUT operations require exclusive access to the hash bucket(s) used for record and tag data.

6.8.2. GET operations

GET operations (numbered 2 and 7 in Table 4) retrieve one or a number of records from the cache based on one of the two criteria: either by a list of (one or more) record keys provided by the client, or by a tag key and a list of tag values (zero or more). The GET operations require only shared access to the hash bucket(s) used for record and tag data and as such can be successfully used with high concurrency.

6.8.3. DELETE operations

DELETE operations (numbered 3 and 8 in Table 4) share the form and method of selecting records with the GET operations, except they delete the records instead of returning them. The DELETE operations require exclusive access to the hash bucket(s) used for record and tag data.

6.8.4. Atomic operations

The atomic operations are special in two ways: in the type of the task they perform and in the guarantees they give for the task. Specifically, they semantically involve more than one step in completing the task, and they guarantee that the task will be implemented as if there are no other atomic operations of the same type executing on the same data records at the same time.

The ATOMIC_INCREMENT operation is currently the only operation which interprets the record value in some way instead of treating it as an opaque binary string. It operates only on values exactly 8 bytes in size and treats them as 64-bit integers in the same format (and endianness) as used in the network protocol. It is given a single record key which identifies the record for the operation and a signed 64-bit value which will be added to the record value. It returns the new value to the client.

The ATOMIC_CMPSET operation implements the CMPSET (Compare And Set, also abbreviated to CAS) operation on a data record, without interpreting the data values. It is given a record key and two values. If a record with the given key is found and its value is a binary string which is exactly equal to the first given value, it replaces it with the second given value, leaving tags intact. It returns a success status to its caller indicating if CMPSET was successfully executed.

The atomic operations require exclusive access to the hash buckets of the records they operate on.

6.9. Replication processing

Replication processing in the new cache server introduces a parallel network service in addition to its main client-server service (inactive unless specifically enabled at server startup) named the *replication backend*. This part of the infrastructure communicates with remote instances of the new cache server in a peer-to-peer mode and adds two layers to the overall processing described in previous sections: one for internal lock operations and one for data operations requiring exclusive access, forwarding them to remote instances. In effect, it streams a sequence of data and lock opera-

tions to remote peers, which replay the stream on their local data store. It cooperates with “local” data operations in a uniform way: the Two-phase locking algorithm applies to the appropriate hash buckets on all connected clients before the operations are carried out (illustrated in Figure 12), allowing for synchronous multi-master replication. The replication backend maintains a list of locks acquired by remote peers and performs a rollback operation on them if communication with the peers is suddenly dropped. Data consistency is further guarded by data generation counts per each bucket.

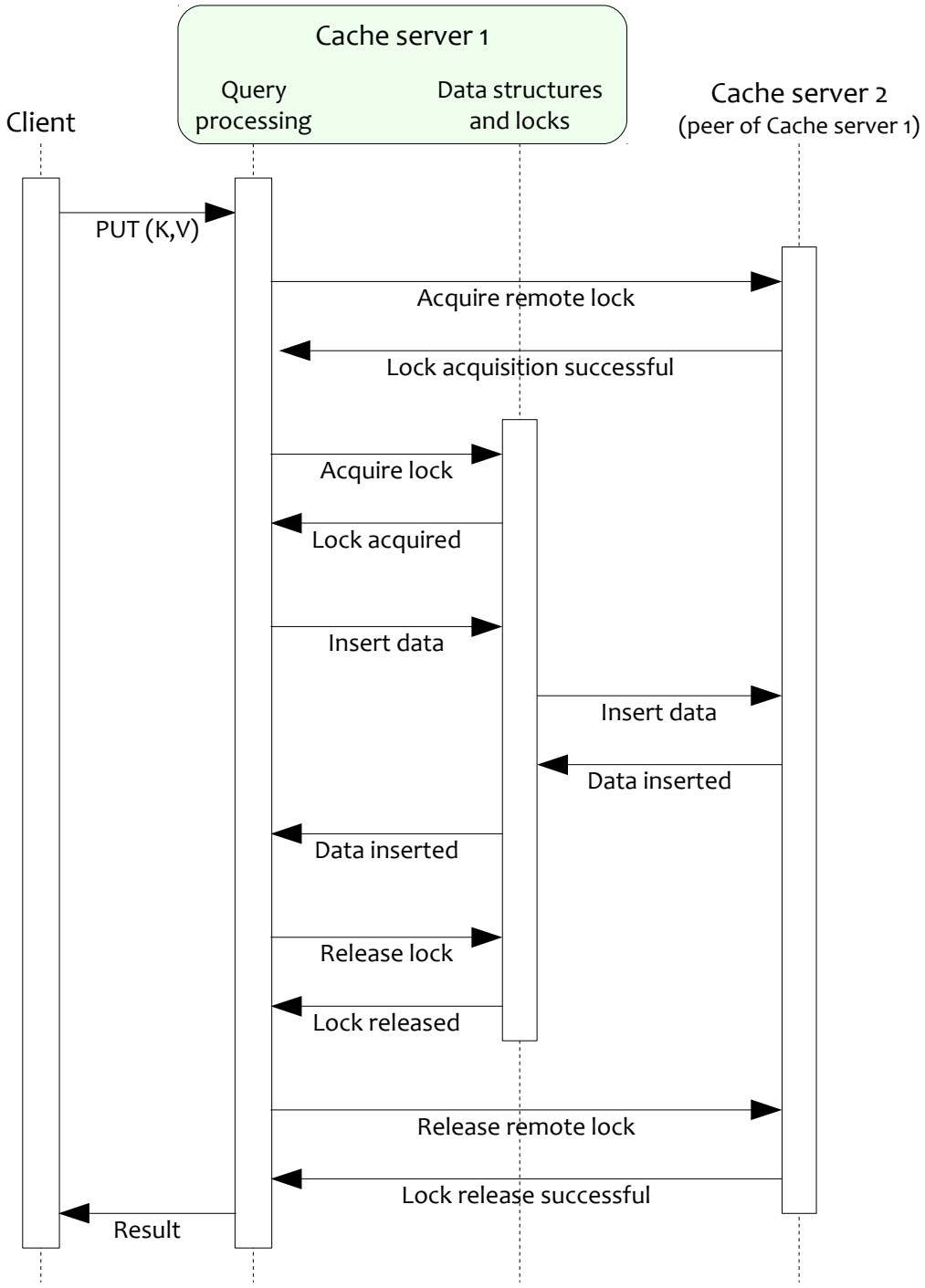

Figure 12: Protocol diagram of a simple PUT operation of the cache server with one replication peer

Because of its ties into the data locking operations, the replication backend can be a performance bottleneck. The volume and the nature of the described stream of operations (even non-exclusive operations need to stream reader locks to their peers) requires a low-latency high-bandwidth network connection. For this reason, the replica-

tion feature is mostly recommended for environments where having a “live” instance of the cache server with current data is more important than performance or for environments where a suitably fast network link can be implemented.

6.10. Client application interfaces

The new cache server is implemented in a combination of C and C++, and the client-server network protocol uses native C data types in the communication channel, so the native client interface for the new cache server is implemented in the form of a C library. As the C language is very wide-spread and universally supported in POSIX environments, used as a foundation for more complex environments and even programming languages, this library can be directly used in higher-level frameworks. One such wrapper, created during the development and implementation of the new cache server, is an adaptor library for the PHP language that exposes most of the lower level C library calls.

The client library offers straightforward functions implementing operations such as “connect”, “put record”, “get record”, “delete record” etc.

7. Web application architecture patterns for high scalability using the new cache server

The new cache server offers a novel approach to several very common cache operations and a strict, coherent model of data replication which may be used for increasing reliability or performance without sacrificing strict data consistency. In order to make the best use of these features in the light of addressing issues from Chapter 3 (Problems in Web application scalability), this chapter proposes a set of additions and improvements to common Web application architectural patterns.

The new cache server is designed to be a service outside the application, not tightly coupled with application code. It resembles conventional databases in that there is a strict separation of duty between it and the application, to which it communicates via a network protocol. In their most basic usage, cache servers operations are meant to replace expensive operations in a system with more light-weight operations, resulting in performance improvements (by the extension of the use of cache memory buffers in computer system design, described in e.g. [111]).

The interaction with cache servers needs to be thought out in advance and carefully implemented to achieve maximal performance benefits. Of particular importance are the methods of cached records' expiry (also called cache invalidation) which, while they can in some cases be simplified to data expiry by timeout, generally require the involvement of the application since it is directly in control of all data that is incoming to and outgoing from the system. Cache data expiry should necessarily be implemented with the following two points in mind:

1. It should only touch the minimal set of data which is in direct need of expiry as to not affect the performance of other data, and
2. It should be exact and thorough, affecting all data in need of expiry to prevent stale data being processed or displayed to the users.

From the point of view of the system as a whole, these two points describe an efficient cache data expiry mechanism. The following sections propose Web application patterns designed to take advantage of the new cache server's data model.

7.1. Cache server as application object cache

The role of object caches in Web applications is to reduce the occurrence of expensive repetitive data processing (including database queries) by using fast storage to persist the objects between Web application invocations. For the purpose of this discussion, “objects” are any entities stored in application memory and do not necessarily have to correspond to object instances from the Object Oriented Programming terminology; it is sufficient that these objects are computationally expensive to construct (or require a large amount of other resources).

Applications of any notable complexity and the number of users typically have a mixture of global or non-session-specific data and session-specific data. Typical non-session-specific data are semi-static Web page content like news articles, attached files and even user comment and forum posts. Contrasted to this, session-specific data is all data which varies in some way depending on the state of the current session (this commonly includes information about the logged-in user). Web pages frequently contain both types of data; for example: a news article in the central place on the page and a welcome banner with the user's full name, or a different set of visible controls and/or different content depending on the user's authorization levels. A typical problem in such environments is keeping the data consistent and fresh with respect to many changes which are dynamically entered into the system by its users, while at the same time avoiding expensive operations that compile and manipulate data into the final HTML document.

The object cache approach to using the cache server involves gathering data which forms Web page objects (e.g. from a database), performing necessary data processing, rendering the HTML presentation of the objects and finally caching either the objects augmented with their HTML presentation code, or just the final HTML code in the cache server. Subsequent HTTP requests that need to include the HTML presentation of an object can simply retrieve it from the cache, avoiding all the previous phases. If needed, the objects' HTML presentation code can be divided into separate parts, such as for the title, the lead text, the main text and the footer, allowing for separate caching of each of the parts. This approach can be used directly with simple key-value databases by using a unique ID (possibly generated from more complex data by using a hash function) as the key. Cache freshness on data update is preserved by deleting (forcibly expiring) cache records for objects which are being updated. However, this approach fails if the objects are not self-sufficient but dependent on other system data and/or other objects as the key-value record does not hold any dependency information, so it is not possible to properly refresh cache records if their dependent data changes. Maintaining a dependency information within the application as another cached object only pushes the problem to another layer, as this record itself needs to be frequently modified while the application is used by multiple simultaneous users. On the other hand, using very coarse forced expiration / cache invalidation rules (such as purging all records from a cache server) can result in seriously inefficient usage of the cache servers and can mitigate potential performance improvements. The new cache server's record tags are intended to address the common instances of this problem in a simple and robust way.

Groups or classes of interdependence can be given their unique integer identifiers and used as tag keys, with particular dependency instances (also converted or reduced to integer IDs) used as tag values. This scheme allows efficient querying and expiring records of a certain dependency class and instance by using operations 7 and 8 from Table 4, with simpler semantics than some of the algorithms described earlier in the literature (such as *Data Update Propagation* described in [112]), which are still robust enough. It also allows record expiry by multiple criteria, as a record can have multiple tags assigned to it. The following sections describe common Web application

object cache patterns and give implementation examples with focus on efficient record expiry.

7.1.1. Inter-object dependencies

A Web application can work with objects which are complex (e.g. a news item has a title, a lead text and a main text), have several forms of presentation (e.g. only title and lead text, a special formatting for mobile devices and a special presentation for the front page) or depend on objects of different type (e.g. a news item's presentation may depend on the size and the type of a possible image gallery attached to it or vice versa). Each of the described dependant objects can be cached as a separate record for reasons of efficiency: it is highly unlikely that a single invocation of a Web application will need to retrieve more than one such object.

For a pattern which ensures invalidation of all related cache records if a top level object changes (in this example a news item object), the suggestion is to introduce a tag key identifier with the notion of "news item" and a tag value with the specific news item's unique database identifier (assuming it is an integer), then attaching this tag to all records depending on the particular news item. Data expiry of all dependant records can be implemented in a single efficient (and atomic, as described in section 6.8) DELETE operation which operates on all records tagged with this particular key-value record tag.

7.1.2. Virtual locality dependencies

Web applications (as well as other application types) can usually provide several ways in which their data may be grouped by its virtual location. Some such "natural" locations for Web application can be pages, forum threads and news posts. The cache server's tagging abilities can be used for efficient management of data grouped in this way.

It is not rare for large Web sites to have thousands of pages – identified by unique path components of the URLs used on the Web site[9]. Complex globally used Web ap-

9 As examples, the main Web site of our Faculty, www.fer.hr, serves over 9,000 separate pages, the

plications can, depending on the specific definition of a "Web page" have millions of different pages. As a dependency source, pages can have a number of properties which if changed require invalidation of dependant objects (usually those presented on the respective pages), such as page layout changes (requiring objects to be resized or repositioned), page design changes and page access permission changes. If the pages are described in a database, containing unique integer identifiers, invalidation can be implemented as for application objects, described in the previous section. If pages are identified only by their paths (as parts of their URLs), the same effect can be achieved by using a good hash function on the URL to generate the tag value. Depending on the number of pages and the quality of the hash function, the number of hash collisions may be negligible. An experiment using the CRC32 function as a hash function on the corpus of 9,161 page paths on the www.fer.hr system found only one collision. Such collisions are non-fatal for cache invalidation, resulting in at most some inefficiency by invalidating more cache records then strictly needed.

7.1.3. User session dependencies

Contents of a Web page may depend on the active user session, whether because it is customized for the user or it depends on certain per-user business rules (such as access permissions). To take advantage of the performance improvements offered by the new cache server, the Web application may cache user-customized content with tags such as the database user ID or the session identifier string. In case a user ID is used, the case becomes similar to that of recording inter-object dependencies, but the usage of session identifiers is more involved.

As the HTTP is after all a stateless protocol with essentially independent request-response pairs, it is up to the Web application or its underlying framework (if any) to maintain session state (described in Chapter 4). Best practices in current Web applications implement session state persistence across multiple HTTP transactions by assigning a short, unique session identifier strings and piggybacking them on regular HTTP requests and responses. The session identifier strings are created as strongly

Web site of the Faculty of Economics servers over 8,000 separate pages, and the Web site of the Faculty of Law holds over 3,500 different pages (all Web sites are using the same Web content management system).

random strings (to discourage session hijacking attacks on security by guessing the identifier) whose length is on the order of 30 characters (usually 32 hexadecimal characters encoding 16 random octets, but this is highly implementation dependant). As such, the namespace of session identifier is usually very sparsely used – even in a Web application with approximately a billion active users this is a difference between 2^{30} and 2^{128}. Given that the new cache server uses integers as tag keys and values, a mechanism for translating session identifiers to integer keys needs to be introduced. This mechanism needs to map session identifiers to tag keys bijectively, prohibiting utilization of a simple hash function for the mapping. The following two program patterns are suggested for efficient use if the new cache server.

The first suggested pattern for efficient implementation of this mapping is to store it in the application's main database, using database-provided operations and algorithms to maintain it. The second suggested pattern is to use the new cache server's atomic operations on a cache server record to calculate the integer session identifier at the session creation time (in its simplest form it can be implemented as an integer counter modulo 2^{31}), at the same time maintaining a set of mapping records which can be queried or created with atomic CMPSET operations. The benefits of the second approach are centred around the avoidance of database use and the use of light-weight cache operations.

Forced data expiry of user session-related records may help efficiency and performance, but also overall system security by forcing expiry of sensitive data from the cache server when the user “logs out” in the Web application.

7.1.4. Cache server as Web application session storage

As a special case of using the cache server for storing session-dependant data, the entire set of session-associated data can be stored in the cache server instead of the main application database or a file system. This mode of operation is offered by most Web application frameworks[10] as a way of achieving data persistence across HTTP re-

10 All the major Web application frameworks support customizable session storage, examples are PHP (http://www.php.net/session), Ruby on Rails (http://guides.rubyonrails.org/security.html), Django (http://docs.djangoproject.com/ en/dev/topics/http/sessions/) and Java Servlets (http:// download.

quests, exposing a simple and persistent environment to application developers and as such is a different case from storing session-dependant data which are under the exclusive control of the Web application. As application frameworks tend to implement serialization and deserialization of session data natively and their interfaces offer "cooked" strings containing serialized application data, session storage often requires only a key-value type of records, easily implementable with high performance with the new cache server.

7.1.5. Cache server for storing application-global data

Large-scale Web applications may be implemented by technologies where application instances are running on many separate systems, without a common central point. Sharing data in such cases is usually detrimental to overall scalability and performance, but if data sharing is needed, a high performance medium utilized for this sharing may reduce the harmful effects up to a point. The new cache server offers high-performance simple key-value storage and some advanced features which can help in this case.

7.1.6. Issues addressed

The use of the new cache server as an object cache primarily addresses issues dealing with performance enhancements over slightly increased code complexity. As such it can be used to improve applications in the areas of CPU load scalability and application architecture scalability. Improvements in CPU scalability can be achieved by making use of the basic cache function of the server, reducing the need for redundant complex operations, and by making use of fast operations offered by the new cache server. Application architecture scalability can be improved by allowing application instances to be run on separate servers while still sharing some data, or using a pool of (possibly replicated) cache servers.

7.2. Cache server as database cache layer

The most common Web application architecture integrates separate environments for the application itself and the database, i.e. separate servers, separate processes and

oracle.com/javaee/5/ap i/javax/servlet/http/HttpSession.html).

separate languages (e.g. PHP, Java, Ruby versus SQL). The application code usually implements at least a part of the "business logic" layer and the back-end of the presentation layer (i.e. creation of HTML documents). The application issues queries to the database, and it responds with results.

Modern databases can implement complex data schemas and hold large volumes of data without significant effort, but accessing such data can involve queries which are complex, have high demands for database server resources and/or have long execution times. Publicly available Web applications, especially if they implement functionalities of a news site, a Web portal site, a blog site or a similar application type where a large number of users accesses essentially the same content, often issue repeated database queries on a data set which while dynamic, changes relatively infrequently on a "human" scale of several minutes, allowing for the implementation of a cache layer in the database access framework of such applications. Such cache layers can operate on simple key-value records by using the (hash of the) SQL query string as the record key and the query result as the record value, which while effective, leads to problem with data expiry. Cache record expiry is the central problem in the described model, especially if information is business-critical and stale information cannot be allowed to reach the users (e.g. in Web sites which track financial applications).

Two patterns which addresses the problem of record expiry while optimally making use of the new cache server are suggested here. The first suggested pattern is to analyse the application schema for tables (or stored procedures, views and other data producing entities) whose data needs to be presented fresh to the users, then tagging cache records holding data (either entirely or in part) from these tables with tags containing table identifiers. When the table data changes, a cache server operation can be issued which purges all dependant data.

The second suggested pattern is more involved and is based on identifying specific database records or groups of records referenced by queries and choosing tags in a way which allows expiry of a smaller number of cache records (instead of all records associated with a table). This approach requires more detailed knowledge of the data-

base schema and the business logic behind the application and can be in effect similar to using the cache server as an application object cache with inter-object dependencies.

Both approaches require tight cooperation from the Web application code, a modification of essentially all code which issues database queries for certain tables or data, and as such may be complex or tedious to implement.

7.2.1. Issues addressed

By using the new cache server for caching database data, improvements can be achieved in CPU load (on the database server) and storage scalability. CPU load is reduced by fetching required data from the cache server instead of passing redundant queries to the application's primary database. Storage scalability is increased on the database side in the same way, by reducing the disk IO load created by the database while working on a large data set. On the other hand, the usage of a cache server which essentially duplicates (or more than duplicates) data from the database can adversely affect memory scalability. In order to reduce memory bloat, cache records may be configured for automatic timed expiry, which would have the effect of keeping only the most frequently used data in the cache server.

7.3. Cache server as primary data store

Certain use cases require high performance data operations but are not very sensitive on data persistence, consistency or structure (one of these is the popular social network messaging platform Twitter, as described in section 4.1.3.) The new cache server provides a convenient data model with high performance which can be used by applications as a primary data store. The following application patterns which can be used to build Web applications with distinct storage requirements are suggested.

The first pattern is to use the new cache server for sharing frequently used (“hot”) data between several different applications or between instances of the same application, with forethought about the volatility of these data. Certain types of data, for example high volume operation and performance logs, sensor data, ephemeral user data

such as user preferences, actions, geographical position and position in virtual environments (games, augmented reality or even the position in a classical Web site's page structure) may be useful or even essential while the application is running but either not worth storing in persistent storage at all or of no use in a disaster scenarios which result in the application server and / or the database server being unavailable. Such data may be stored in the new cache server, while making use of the advanced features of its data model (possibly with patterns described in section 7.1). For example, an application might track a user's position in a virtual shopping mall and use the new cache server as a shared database available to other users for the purpose of accurate presentation and interaction but unless a very detailed log of the user's movements is required, this data does not need to be stored in a persistent database.

Another pattern for the application of new cache server is in cases where the data is reasonably important but the freshness of data is not of the utmost importance. In such cases, data can be stored and operated on while completely stored in the cache server but also periodically copied to a more persistent storage (e.g. a general-purpose database). Using the same example as above, if the user's past movements are worth storing but only at a granularity of one minute, a “checkpoint” process might copy relevant data from the cache server to the database for historical safekeeping.

7.3.1. Issues addressed

By using the new cache server as the application's primary storage, scalability issues pertaining to storage are moved away from simple IO performance and start overlapping with areas of interest of network scalability, memory scalability and general application architecture scalability. Practical implementations will in many cases be constrained by network latency and bandwidth, or in high-end environments with huge amounts of data even by available system memory bandwidth.

7.4. Cache server and application data partitioning

High performance applications or applications needing to cache more data than can practically fit in a single computer system's memory might build upon the new cache server's existing features by introducing a data partitioning layer in front of the cache

server that would distribute records onto a pool of multiple cache server instances (selected by some criteria based on the record key, usually a hash function). In effect, this pattern mimics the behaviour of the hash table data structure by treating whole server instances as buckets, and shares with it both the good sides (the distribution of processing load and data volume across multiple servers) and the bad sides (problems arise when the number of buckets / servers needs to be increased).

A practical complex example of this pattern, with the cache server used as an object cache or a database cache, might use N cache servers as buckets, each of which is replicated M times for reliability, then distribute cache requests among the N servers while further distributing read requests among the M servers of each bucket for performance reasons. The issue of resizing the server pool might be addressed simply by invalidating all cached data on all servers, allowing them to be repopulated in the usual way, or by creating a new server pool of different size, copying the data to the new pool, then gradually switching the applications to use the new pool. The latter approach allows for the applications operate continuously during the migration.

7.4.1. Issues addressed

The ability to make use of a number of cache servers equally and without special cases is a sign of good application scalability. The usage of multiple cache servers (especially in two layers with replication and load balancing of read requests) can have a significant impact on overall CPU scalability of the application. If the cache servers are the primary data store of the application, this approach also addresses the scalability of storage and memory.

7.5. Trade-offs and the limits of applicability of proposed Web application architecture patterns

Though the new cache server is not strictly limited to data caching applications, caching is one of its most likely applications. The concept of data caching relies on the assumptions that the cache operations are in some significant way faster (or better performing with respect to other computer resources) than the original operations they are replacing and that such replacements can be achieved often enough to result in

benefits to the overall system. If the original operation that generates the potentially cacheable data is fast enough or the data is modified frequently enough that the benefits from retrieving the cached data are small compared to the cost of refreshing it in the cache, no significant gains can be obtained from any type of caching.

In order to achieve maximum performance, the new cache server implements a data model centred around the concept of key-value records where both the key and the value are opaque binary strings. Applications using the cache server need to be adapted to this model, i.e. they need to construct these strings for the data they want to store in the cache. Keys can be formed from already present domain-unique data such as database identifiers, possibly with additional components specifying their domain to avoid collision with other domains (e.g. “user-9238”), but the record values often need to contain complex data structures which need to be adapted for the purpose by serialization (or marshalling) into binary strings. If the structures' complexity is high enough that the serialization is a resource-intensive process (or not even practical), the application may not be able to take advantage of this type of caching.

The new cache server extends the key-value data model with record tags which are the central concept for achieving significant efficiency gains for certain operations. This new model is flexible enough, enabling the classification of the cached data which can be used in operations performed on a large number of records, but is itself of a very rigid structure, owing to it being designed for maximal performance. It is therefore conceivable that there can be cases where the model is not suitable for accurately representing the application's data and the trade-offs would introduce unacceptable complexity, inefficiency or imprecision.

Finally, introducing another component in a system means that the failure of this component must be considered to maintain a stable operation. In a simple Web application environment where the main components are the Web server, the application server and the database, introducing the cache server presents a 25% increase in the number of points of failure. If an application is using the cache server only for caching, workarounds for when the cache server is unavailable should be implemented.

8. Analysis and evaluation of the proposed models and architectures

The models used in the design of the new cache server and the architectural patterns proposed for the client applications are verified from several aspects crucial for their applicability:

- Scalability and efficiency of the multithreading models
- Scalability and efficiency of the model of network IO operations
- Scalability and efficiency of the data structures
- Benefits from application architectural patterns with the use of the new cache server

These aspects are analysed in the following sections.

8.1. Analysis of scalability and efficiency of the multithreading models

One of the defining characteristics of the new cache server is the support for a number of different multithreading models for the distribution of its internal tasks. The supported multithreading models, detailed in section 6.1, are SPED, SEDA, SEDA-S, AMPED and SYMPED. These models have different characteristics and their optimal use may be dependant on the exact environment and the task in which they are used. The evaluation of the multithreading models within this dissertation is focused on the efficiency of execution in the basic task of key-value record insertion and retrieval to and from the cache server with the data size kept small in order to exercise the specific edge cases of cooperation in the algorithms. Unless otherwise stated, the performance tests of the models has been carried out on server system with eight CPU cores (of possibly different models), with the cache server and the benchmark client

running on the same system and communicating over local Unix domain socket protocol (to minimize outside influences such as network latencies), and with the benchmark client configured to use 50 simultaneous client threads, 30,000 small records (of 90 bytes average size) and a mixture of 10% write requests and 90% read requests under the FreeBSD 8 operating system.

As described in [87], multithreading models have different performance characteristics but also different resource uses. The performance aspect is best illustrated in Figure 13. The lowest performance, as expected from its lack of support for multi-processor operation, belongs to the SPED model (*Single Process Event Driven*). The model itself is robust and with excellent performance in situations where the cost of "payload" work resulting from a client request over the network (measured primarily in CPU usage but also other resource usage) is negligible when compared to the cost of actual network communication and connection multiplexing. Its design practically guarantees that strictly less than a single CPU core will be dedicated to program activities (not counting operating system kernel activities).

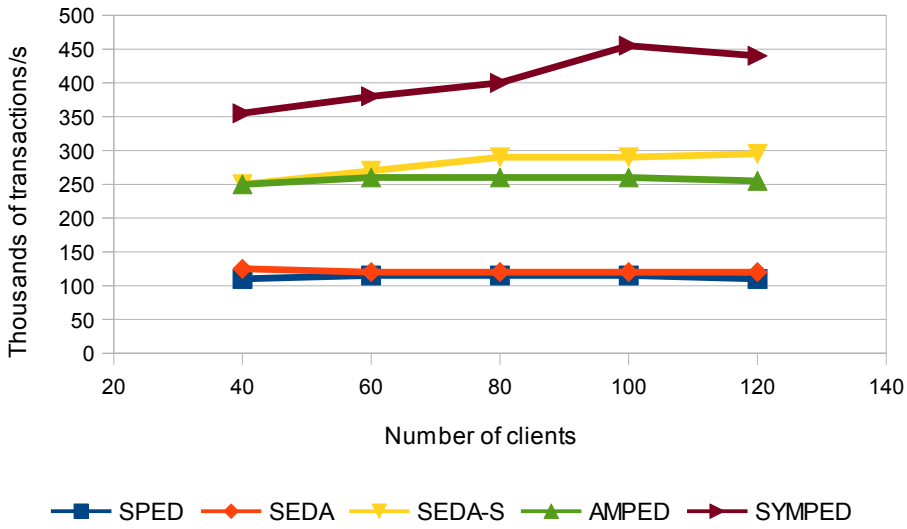

Figure 13: Performance characteristics of different implemented multithreading models (from [87])

An unexpected result is the relatively low performance of the SEDA model (*Staged Event-Driven Architecture*), supported in the new cache server by dividing the net-

work IO and the query processing tasks into an (arbitrary) number of threads, with queue structures for communication between the threads and between the threads and the rest of the program. Each of the query processing threads has its own job queue. In the test whose results are shown in Figure 13, the number of network IO and query processing threads is actually the same: two of each. Analysis of the program behaviour has indicated that the low performance is the result of relatively very asymmetric processing requirements of the test, compared to relatively high cost of multi-threaded task queuing. Effectively, the program and the operating system have spent more time managing the queuing and context switching than doing useful work. After noticing this, an improvement was designed and implemented as the SEDA-S model (until now not specially described in literature). It shares the same basic operation with SEDA, but the number of network IO threads and the query processing threads must be exactly the same, with strong coupling between the pairs of threads which eliminates most of the queue locking, only leaving in place inter-thread notification, and also ensuring better utilization of CPU caches as each of the pairs of threads can be executed on the same CPU. This addition has more than doubled the average performance of the SEDA-S tests compared to plain SEDA.

The AMPED model (*Asymmetric Multi-Process Event-Driven*) uses a single network IO thread which dispatches tasks to an (arbitrary) number of query processing threads (thus the asymmetry). This model effectively concentrates all network operations in a single thread, allowing the operating system to optimize context switching, while delegating the potentially more CPU-intensive tasks to separate threads (in the specific tests shown in Figure 13, there were three query processing threads). Since the number of worker threads can be arbitrary, there is still a relatively large amount of work done in maintaining the job queues and context switching, and the performance of the AMPED model is lower than that of the SEDA-S model.

Finally, the SYMPED model (*Symmetric Multi-Process Event-Driven*) is a variation which can be most concisely described as instantiating multiple threads, each of which is running the SPED “event loop”. It is supported in the new cache server by directly invoking the query processing routines without any queuing from inside the

network threads, which can be instantiated in an arbitrary number. Each of the network threads in this model is assigned network connections from the new connection processing task in the round-robin fashion. This model closely couples processing of data received from the network in a single thread, so the lack of queuing and context switching overheads make it the fastest model by far. It is also scalable in the sense that the instantiated SPED-like threads can be directly distributed across CPUs, and as threads do not themselves force context switching, they can stay bound to specific CPUs, benefiting from efficient use of CPU caches and operating system scheduling, as demonstrated by tests whose results are shown in Figure 14.

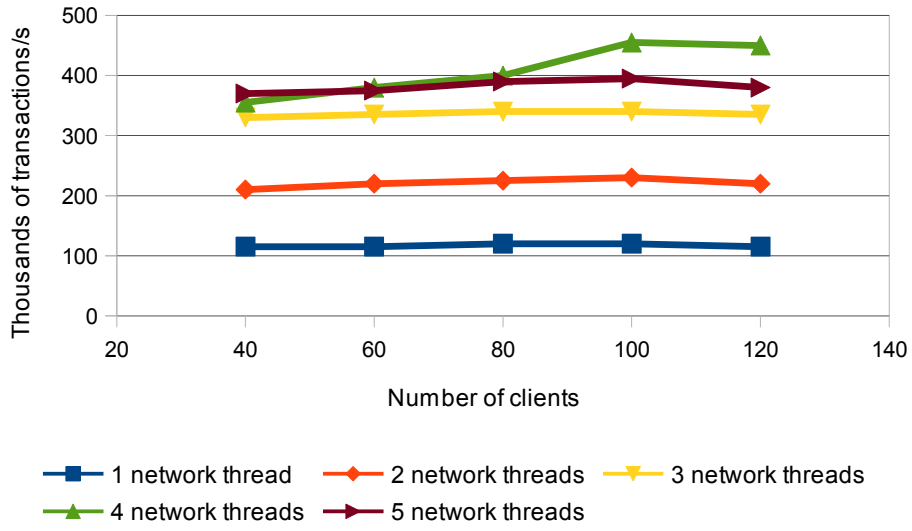

Figure 14: Performance of the SYMPED multithreading model in a 8-core server, while varying the number of network threads on the server (from [87])

In these specific tests, though the system is equipped with 8 CPU cores, because the benchmark clients are executing on the same system as the cache server, improvements in scalability stop after all the CPU cores become saturated. As shown in Figure 14, instantiating 4 SPED threads results in not entirely predictable performance, while with 5 threads the system is oversaturated and performance starts to fall.

8.1.1. Discussion

The performance tests described in this chapter test the behaviour of the multithreading models under specific conditions, where the number of records is relatively small and the number of clients relatively large, exercising the behaviour under a load by a large number of clients on a small database with simple queries.

If the conditions change, for example if the number of records is significantly increased and so does the complexity of queries, the cost of query execution might become large enough that the SPED-like behaviour of SYMPED might introduce noticeable latencies because of their serial processing of network requests. Under such conditions, models which offload query processing to separate threads (SEDA-S and AMPED) would reduce some of the request processing latencies by parallelising network IO and request parsing with query processing.

The overall performance differs significantly with hardware capabilities. Figure 15 shows the results of the test of the SYMPED model with similar parameters as in Figure 14 (4 server threads, 60 clients), on two different hardware configurations.

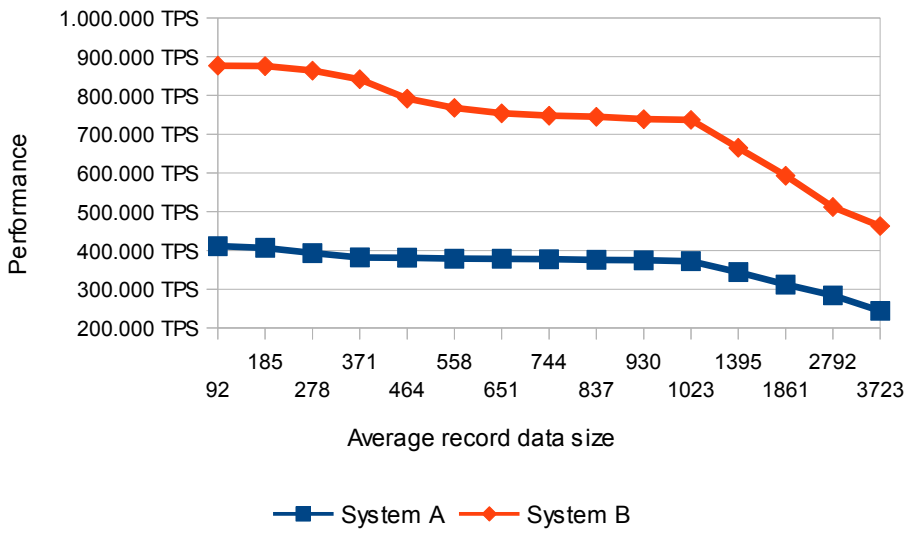

Figure 15: Performance comparison of the new cache server on two different server hardware configurations, depending on record size

In Figure 15, System A contains two Intel Xeon 5430 quad-core CPUs (running at 2.6 GHz), while System B contains a single Intel Xeon 5630 quad-core CPU with Hyper-threading (running at 2.5 GHz), two CPU generations newer than System A and with much faster memory and IO access performance. Despite having less full CPU cores than System A (and is much cheaper), System B delivers more than twice the performance. Comparison of the results presented for System B in Figure 15 with the publicly available results of other cache servers (at the time of writing of this dissertation) suggests that the new cache server's results are in the best of class for software of similar type.

8.2. Analysis of scalability and efficiency of network IO operations

As the new cache server is very fast in doing “payload” work (query processing), the complexity of network IO operations can become the dominating factor in its overall performance. The effects of this are demonstrated in Figure 16, where two cache servers: the new cache server and Memcached 1.4.5 [36], are tested with different access methods: Unix domain sockets (only the new cache server as memcached does not support this access method), TCP over the loopback interface (the localhost address) and TCP over a switched gigabit Ethernet LAN (the client and the server have identical hardware, connected to the same managed Ethernet switch).

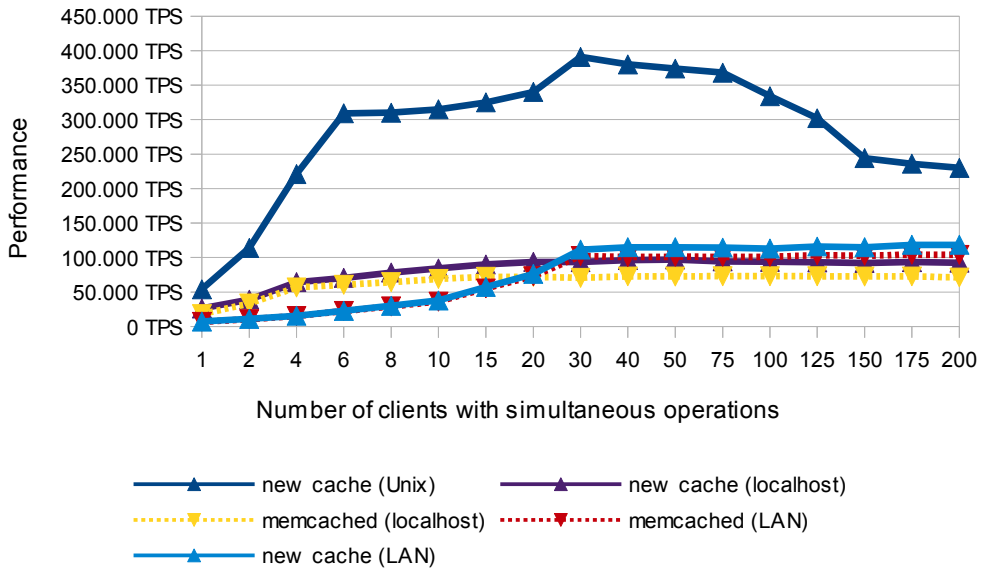

Figure 16: Performance of the new cache server and memcached depending on the access method

The Unix domain socket access method has the lowest overhead, as it can be implemented efficiently within the operating system without the need for complex protocol parsing and routing and so yields the highest performance among all access methods. Its downside is, per its definition, that it can only be used for inter-process communication within a single operating system image. Its peak performance is more than 3.5 times higher than the next fastest access method (TCP over Ethernet) and it can be considered to represent the theoretical peak of communication performance. The other access methods have several unavoidable overheads: TCP/IP protocol parsing, routing (however rudimentary) and more software layers which limit scalability. The operating system used for testing (FreeBSD 8, but the similar situation was observed in Linux) uses a single network thread for processing network IO per network interface in the tested hardware configurations[11], further limiting scalability with a relatively complex and chatty protocol like TCP/IP is. However, the performance is good enough for a comparison between the two cache servers using the same access method. The chart in Figure 17 shows relative performance difference in results

11 More advanced (and more notably, expensive) hardware exists with multiqueue processing which can be supported by operating systems to implement multithreaded network processing. Such hardware was not available for testing during the writing of this dissertation.

between the new cache server and memcached, created from the same data as in Figure 16.

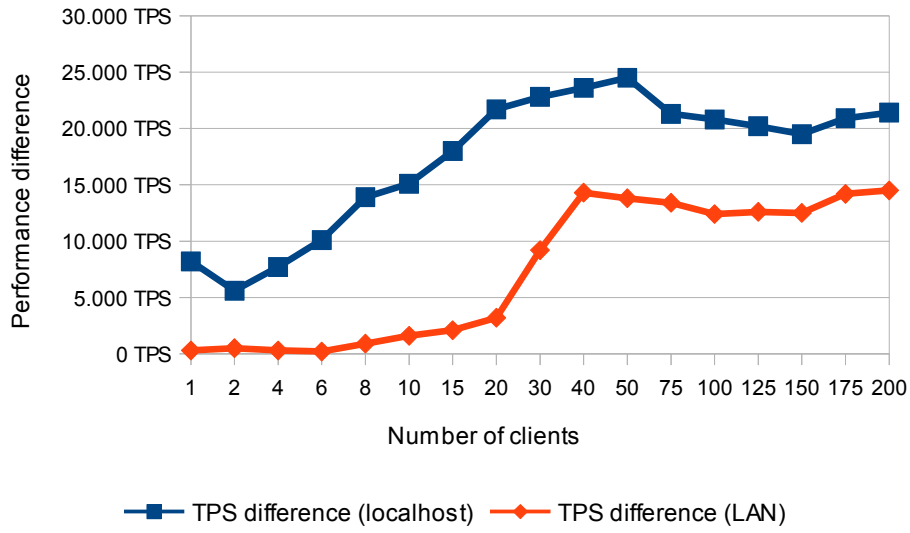

Figure 17: Performance difference between memcached and the new cache server (in favour of the new cache server)

As Memcached implements the SYMPED multithreading model, the same model is also configured on the new cache server, and the number of threads in both servers is set to 6. The new cache server exhibits better performance then memcached in all cases, with improvements as the number of clients increases. The biggest jump in performance happens when the number of clients is between 20 and 50, corresponding to a similar jump with the Unix domain sockets access method from Figure 16. As the number of server threads is in both cases is equal, this indicates better IO event scheduling and more efficient data structures on the side of the new cache server.

An interesting property of these results is the difference between performances of the loopback interface (localhost) and over the (gigabit) local area network. Aside from the constant offset between localhost and LAN performance curves, there is a difference in curve slopes for measurements with up to 50 clients, which can be attributed to the more efficient use of system resources (among others: operating system buffers and CPU caches) when both the client and the server are on the same system.

8.3. Scalability and efficiency of the data structures

Records are stored in the new cache server in a composite data structure consisting of a hash table as the first stage and the Red-black binary tree as the second stage, with a tree rooted in each bucket of the hash table (as the only payload of the buckets). Each of the buckets also contains a reader-writer lock object protecting access to the tree, allowing concurrent write access to individual buckets, or in the best case arbitrary read access to all buckets for all clients (described in detail in section 7.1.1).

The expected time complexity of random access to this structure is $O(\log(n) / H)$ for the structure populated with n records and with H buckets in the hash table. The default size of the hash table is set to 256, both reducing lock contention and increasing record access performance for up to two orders of magnitude over the tree structure alone (though as a constant factor). The result of data structure scalability tests are shown in Figure 18.

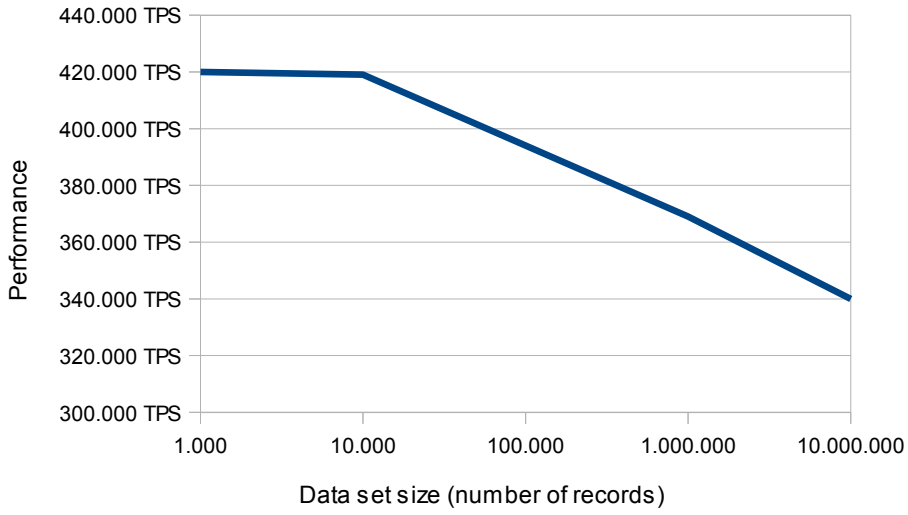

Figure 18: Scalability of the new cache server data structures depending on the number of records in the structure (access over Unix domain sockets)

The tests in Figure 18 were made with 30 simultaneous clients running the common benchmark with a configurable number of records (otherwise configured as in Chapter 8.2). Memory constrains on the server prevented testing with 100,000,000 re-

cords but the attained results very closely match the predicted logarithmic scalability (the better than expected results with 1,000 and 10,000 records are the results of two factors: access method overheads and the data records being highly efficiently cached by the CPU caches).

8.4. Benefits of application architectural patterns with the new cache server

Dynamically generated Web pages which are also visited by a large number of readers without much per-user customization share a significant likeness to static Web pages: their content is not often changed. However, modern Web applications can rarely get away from using static or pre-generated pages as the users (rightly) expect more and more interaction with the Web applications. In this scenario, pages cannot be pre-generated or cached in whole but have to be composed from parts which are static and parts which are dynamic. The new cache server offers a data model which facilitates building complex applications that cache complex objects and in which the complexity of record management and invalidation is significantly reduced.

This property enables the implementation of Web application patterns which are not possible with other cache servers, such as the efficient caching of interdependent objects, but even without it, it is estimated (based on the implementation of the Quilt Web content management system at our Faculty) that the support for basic cache operations can be implemented in Web applications with up to 20% reduced code complexity (measured by the number of lines of code and the intricacy of the cache expiration logic).

8.5. Strategies for global scalability using the new cache server

To visualize the impact the use of the new cache server can have on applications' performance, it is useful to compare the speed of operations of cache servers and a generic SQL relational database using similar query types and an identically sized data set. Such a comparison was made on the same hardware used in tests in Figure 16, with the same data set of 30,000 records used for the original tests, with a client written in C accessing an installation of the PostgreSQL 9.0 database configured for per-

formance, over a Unix domain socket. The database contained a single table with two fields of “TEXT” type, one used as the record key (set as the table's indexed primary key) and the other as its value. The results of the comparison with strictly read operations (i.e. GET) and a single client thread are shown in Figure 19.

The results in Figure 19 indicate that the implementation of any cache server in an application for use as a local cache server (running on the same system) can significantly improve the application's performance. The new cache server offers significantly better performance than Memcached and is almost five times faster than the relational database on the same simple type of queries, using the same access method.

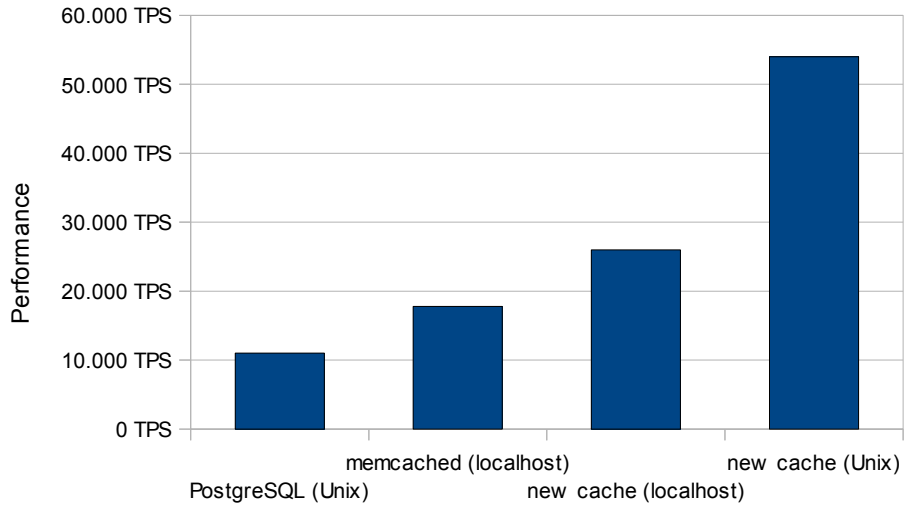

Figure 19: Single-client performance comparison between PostgreSQL, memcached and the new cache server

As a further exploration of these results, the performance of the system consisting of the PostgreSQL database and the new cache server with the Unix domain socket access method with a varying ratio of cached queries to database queries (assuming that a certain percentage of database queries can be completely replaced by cache queries) was extrapolated. The results of this extrapolation are shown in Figure 20.

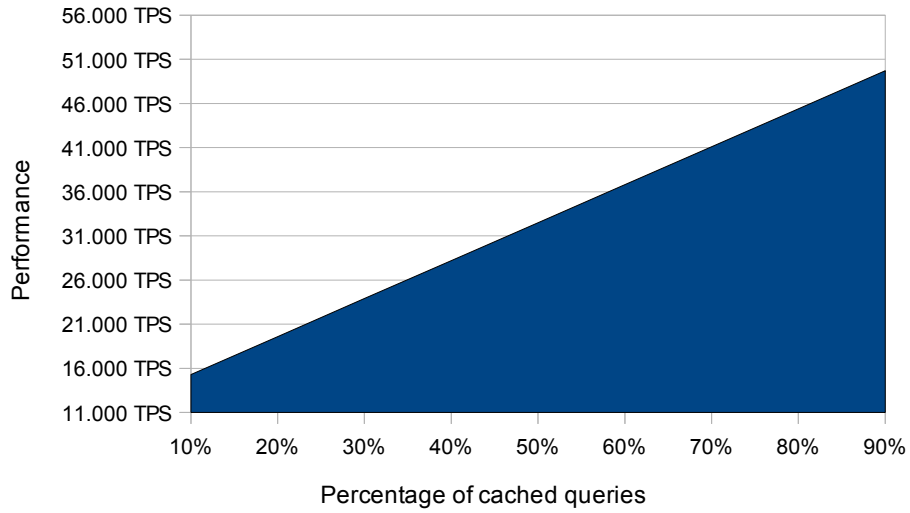

Figure 20: Extrapolated performance depending on the predicted ratio of database queries that are retrieved from the cache instead of the database

From the point of view of a realistic application, this extrapolation is a pessimistic one as the database queries are usually more complex than in this simulation, resulting in larger performance improvements with caching. The performance increases offered by the use of the new cache server can be reflected in the savings on the number of servers used for the application.

The cost of maintaining a data centre with thousands of servers depends on several factors, among which are the cost of air conditioning / cooling, the network equipment and the power supply to the servers. As described in [10], the portion of amortized monthly expenditures directly caused by servers can approach 60%. By implementing a cache scheme which allows the same overall application performance with a smaller number of servers it is possible to introduce cost savings. Figure 21 contains the results of an extrapolation of the portion of monthly amortized server cost based on the referenced work. As cache efficiency increases, the number of total servers used to maintain original performance decreases.

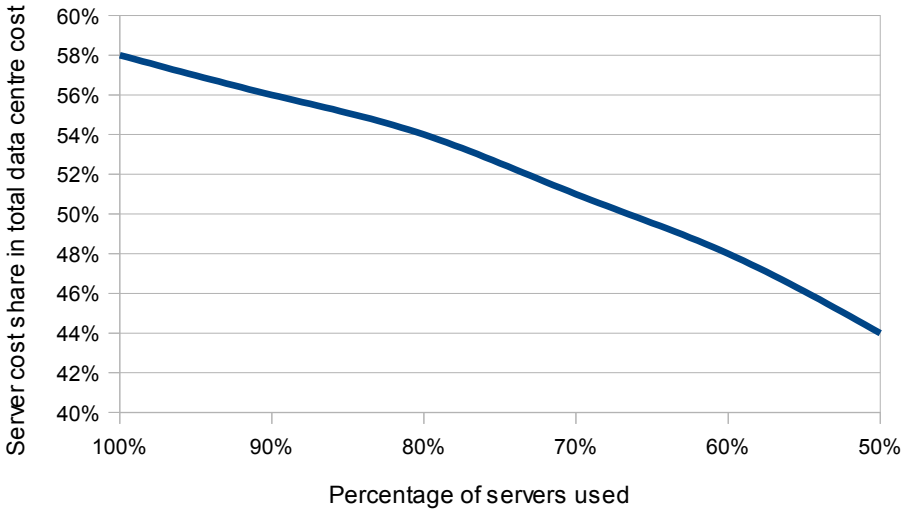

Figure 21: The estimated share of server cost in total data centre monthly cost from a model by J. Hamilton

On the other hand, by keeping the number of servers constant, more users can be serviced with existing infrastructure, which is a direct improvement in scalability. The new cache server can be implemented as a part of the application stack, running on the same system as the Web application for maximum performance, or it can be implemented on separate dedicated servers. In the latter case it is possible to calculate the approximate number of cache servers needed depending on the number of requests per second expected on the Web application servers and the number of cache queries executed during each requests. As an example of this calculation, if the application servers are serving 100 page requests per second and each request causes 50 cache queries to be issued on average, the ratio of Web application servers to cache servers could be (conservatively) estimated to be 20:1 (if all other factors are ruled out such as the availability of memory and the necessary network bandwidth, using the TCP access method). This estimate also plays a role in tracking cache efficiency – it indicates that at most 20 Web application servers can make use of cached or shared data if under full load.

9. Future work

The area of Web application scalability is large and contains many possible avenues of research. Some of these may be explored in the context of the new cache server with the goals of improving its own efficiency and scalability, and others can be applied to solving challenges in general Web application scalability. This section contains brief descriptions of the future work possible as continuations to the work described in this dissertation.

9.1. Improvements in data structure locking

The currently used locking model is very satisfactory, allowing concurrent access in all but the most intensive workloads. This locking model relies on the classical concept of locking objects whose services are provided by the operating system. In parallel with this approach, a new class of algorithms with integrated concurrency control was popularized as “lockless” and “non-blocking” algorithms which do not rely on the operating system-provided locking objects but on hardware-provided atomic operations, pushing contention down to the level of system bus arbitration [113][114]. It is possible that these algorithms may offer better performance in certain areas, such as the job queues between the cache server tasks, leading to better performance in the SEDA and AMPED multithreaded models.

9.2. Improvements in network IO processing

As shown in section 8.2, network communication over TCP/IP introduces a severe performance degradation when compared to the very light-weight Unix domain sockets IPC mechanism. A part of this degradation is due to the relatively larger complexity of the protocol, indicating that a more light-weight protocol may perform better. A good candidate for such a protocol is UDP/IP, but its implementation would require a greater restructuring of the cache server as this protocol is not stream-oriented.

9.3. Explicit use of the NUMA computer model

The NUMA model of computer design is becoming more popular with hardware manufacturers in order to circumvent current technological obstacles in building CPUs with faster operating frequency and high-bandwidth access to memory [115]. The currently widely deployed variant of NUMA (and the one which will in all probability be the standard for the foreseeable future in industry standard servers), cache coherent NUMA (ccNUMA), mostly hides the basic hardware non-uniformity of memory architecture in a level below the operating system,[12] but it cannot always hide one edge case in the way high-performance programs (both user programs and operating systems) access memory: not all memory accesses are of comparable latencies and bandwidth. Modern operating systems and applications cope by adapting process schedulers [117] and memory allocation [80] subsystems which try to establish locality between the program code and the memory it accesses, exposing a mechanism of hints to the applications which can be used to specify affinity for certain hardware combinations. However, this is not enough for complex applications requiring a large amount of memory (such as cache servers and databases in general) as memory accesses are fundamentally unpredictable. Further studies need to be performed to establish if even more complex scheduling (such as for TCP connections with regards to the connecting client and its past behaviour in data accesses) can substantially help achieve better performance or is the complexity of such scheduling too large to be feasible.

9.4. Extension of the cache server to persistent storage

As the tasks of the new cache server are modular and separated by well defined interfaces, the new cache server is suitable for experimentation where its data storage module is replaced by one with a different characteristics – for example, one that would store the data persistently in a file system. As there are several persistent key-value databases available that could fit this purpose (e.g. Oracle's / SleepyCat's BerkeleyDB, GDBM, Tokyo Cabinet), a new data storage module could use one of them directly for persistent data storage.

12 Contemporary implementation of ccNUMA in x86 servers offers uniform-enough view of the memory (by the usage of fast buses and caching) that the AMD Corporation sometimes calls its architecture SUMO – *sufficiently uniform memory organization* [116]

9.5. Improvements in Web application architecture

While it is primarily intended as a cache server, the new cache server can be used for data sharing and storage between multiple Web applications. Further work may reveal new areas of application.

10. Conclusion

This dissertation introduces models and architectures for increasing performance in Web applications centred around the new cache server. The first chapter describes the motivations, the research goals and methods used throughout the work which is described in the dissertation. Web applications are a ubiquitous service on the Internet which have spurred the creation of many new businesses. They are the most globally accessible application type in contemporary computer engineering and their scalability can have a large influence on both business and leisure of their users. The second chapter covers trends and best practices in building modern Web applications, with them emphasis on application scalability, while the third chapter discusses problems in Web application scalability. The problems touch on all general areas of computer systems, from CPU load and memory to storage, network utilization and application architectures.

Chapter four presents an overview of current strategies for both global Web application scalability and the infrastructure used by some of the currently largest companies focused on providing Web application services. It also contains a discussion on the importance of cache servers in modern Web applications and a reviews some existing cache servers. Based on this discussion, chapter five presents the requirements for a new cache server with an improved data model, which would efficiently use the capabilities offered by multi-core server systems and which would implement durability through data replication. Chapter six describes in detail the model for the new cache servers, discussing its internal operation with emphasis on the requirements set in chapter five. The new cache server implements novel functionalities which enable the creation of more complex Web applications. The program architecture patterns which may be used in Web application to maximize the benefits of the advanced features of the new cache server are presented in chapter seven. The largest improve-

ments in Web application scalability are expected from the newly introduced capabilities of tracking inter-object dependencies in the cache and using them for efficient data expiry, but the cache server can also be used as a facility for sharing data between applications (or applications' instances) or as a primary data store.

Chapter eight presents an analysis of the proposed models, both for the new cache server and for the Web applications wishing to make optimal use of the new cache server. The results are encouraging, and the new cache server performs better than its closest and most similar cache server, Memcached. By using the new cache server, scalability can be increased and server costs reduced in large deployments. Finally, chapter nine presents possible future areas of research based on the continuation of themes of this dissertation.

This dissertation has introduced a novel model of a cache server which uses the techniques of data partitioning to achieve high performance and scalability when accessed by a large number of clients. It has presented an implementation of the described model, which allows the exploration of certain parameters of the said model such as the multithreading model and the network access method, and presented novel Web application architecture patterns designed to take advantage of the new cache server. The dissertation has included the analysis and the evaluation of the cache server model and the Web application architectures in comparison with existing solutions.

11. Bibliography

1. Tim Berners-Lee, *Weaving the Web*, Texere Publishing, 2001
2. Forrester Research, *Western European Online Retail Forecast, 2009 To 2014*, 2010
3. Forrester Research, *US Online Retail Forecast, 2009 To 2014*, 2010
4. Facebook: "Facebook Statistics", Address: *http://www.facebook.com/press/info.php?statistics*, published in 2010, visited on 2010-10-04
5. All Facebook: "Facebook's Revenue to Surpass $1.2 Billion This Year", Address: *http://www.allfacebook.com/facebooks-ad-revenue-to-surpass-12-billion-this-year-2010-08*, published in 2010, visited on 2010-10-04
6. Google: "Google Investor Relations", Address: *http://investor.google.com/earnings.html*, published in 2010, visited on 2010-10-04
7. The Wall Street Journal: "Facebook Picks Site for Data Center", Address: *http://online.wsj.com/article/SB10001424052748703848204575608613663031230.html*, published in 2010, visited on 2010-10-05
8. Data Center Knowledge: "Google's Data Center Spending Soars", Address: *http://www.datacenterknowledge.com/archives/2010/10/15/googles-data-center-spending-soars/*, published in 2010, visited on 2010-10-05
9. Data Center Knowledge: "Facebook: $50 Million a Year on Data Centers", Address: *http://www.datacenterknowledge.com/archives/2010/09/16/facebook-50-million-a-year-on-data-centers/*, published in 2010, visited on 2010-10-05
10. James Hamilton: "Overall Data Center Costs", Address: *http://perspectives.mvdirona.com/CommentView,guid,57aec6e4-18e8-4cbe-a05a-890b0b0fd2fd.aspx*, published in 2010, visited on 2010-10-05
11. R. Fielding et al: "RFC 2616: Hypertext Transfer Protocol - HTTP/1.1", Address: *http://tools.ietf.org/html/rfc2616*, published in 1999, visited on 2010-10-06
12. L.D. Paulson, "Building rich web application with Ajax", in *Computer*, IEEE Computer Society, 2005
13. Facebook: "About Facebook", Address: *http://www.facebook.com/topic.php?uid=207734924035&topic=11783*, published in 2010, visited on 2010-10-06
14. T. Cook: "A Day in the Life of Facebook Operations", Address: *http://velocityconf.com/velocity2010/public/schedule/detail/13103*, published in 2010, visited on 2011-01-16
15. S.A. Schuman, "Toward Modular Programming in High-Level Languages", in *ALGOL Bulletin*, ACM, 1974
16. Dave Winer: "XML-RPC Specification", Address: *http://www.xmlrpc.com/spec*, published in 1999, visited on 2010-10-07
17. W3C XML Protocol Working Group: "SOAP/1.2", Address: *http://www.w3.org/TR/soap/*, published in 2007, visited on 2010-10-07
18. Pingdom: "REST in peace, SOAP", Address: *http://royal.pingdom.com/2010/10/15/rest-in-peace-soap/*, published in 2010, visited on 2010-10-07
19. Google: "Protocol Buffers - Google Code", Address: *http://code.google.com/apis/protocolbuffers/*, published in 2010, visited on 2011-01-04
20. Apache Foundation: "Apache Thrift", Address: *http://incubator.apache.org/thrift/*, published in 2010, visited on 2011-01-04

21. 10gen: "BSON - Binary JSON", Address: *http://bsonspec.org/*, published in 2010, visited on 2011-01-04
22. Open Market: "FastCGI: A High-Performance Web Server Interface", Address: *http://www.fastcgi.com/drupal/node/6?q=node/15*, published in 1996, visited on 2010-10-08
23. R. McCool: "Server Scripts (e-mail message to the www-talk@w3.org mailing list)", Address: *http://1997.webhistory.org/www.lists/www-talk.1993q4/0485.html*, published in 1993, visited on 2010-10-10
24. D. Robinson et al: "RFC 3875: The Common Gateway Interface (CGI) Version 1.1", Address: *http://tools.ietf.org/html/rfc3875*, published in 2004, visited on 2010-10-08
25. The PHP project: "PHP: Database Extensions", Address: *As visible e.g. in the number of different APIs and styles in the PHP's collection of database extensions documented at http://www.php.net/manual/en/refs.database.php*, published in 2011, visited on 2011-03-08
26. J. Hunter and W. Crawford, *Java Servlet Programming*, O'Reilly & Associates, 1998
27. R. Radhakrishnan and L. Kurian John, "A Performance Study of Modern Web applications", in *Lecture Notes in Computer Science / Euro-Par'99 Parallel Processing*, Springer Berlin / Heidelberg, 1999
28. M. Stonebraker, "The Case for Shared Nothing", in *Database Engineering*, IEEE Computer Society, 1986
29. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", in *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, USENIX, 2004
30. F. Chang, J. Dean, S. Ghemawat et al, "Bigtable: A Distributed Storage System for Structured Data", in *Proceedings of the Seventh Symposium on Operating System Design and Implementation*, USENIX, 2006
31. D. Farino: "Behind the Scenes at MySpace.com", Address: *http://www.infoq.com/news/2009/02/MySpace-Dan-Farino*, published in 2009, visited on 2010-10-11
32. A. Agarwal: "Scale at Facebook", Address: *http://www.infoq.com/presentations/Scale-at-Facebook*, published in 2010, visited on 2010-10-12
33. P. Prince, "When will Flash memory take off in enterprise environments?", in *Proceedings of the Flash Memory Summit 2010*, Conference ConCepts 2010
34. E. Weaver: "Improving Running Components at Twitter", Address: *http://blog.evanweaver.com/articles/2009/03/13/qcon-presentation/*, published in 2009, visited on 2010-10-12
35. NOSQL Databases: "NOSQL Databases", Address: *http://nosql-database.org/*, published in 2010, visited on 2010-11-20
36. Memcached: "memcached - a distributed memory object caching system", Address: *http://memcached.org/*, published in 2010, visited on 2010-11-20
37. Internet World Stats: "World Internet Usage Statistics News and World Population Stats", Address: *http://www.internetworldstats.com/stats.htm*, published in 2010, visited on 2010-12-01
38. ITU: "Increased competition has helped bring ICT access to billions", Address: *http://www.itu.int/net/pressoffice/stats/2011/01/index.aspx*, published in 2011, visited on 2011-01-28
39. Alexa Internet: "Google.com Site Info", Address: *http://www.alexa.com/siteinfo/google.com*, published in 2010, visited on 2010-12-01
40. Alexa Internet: "Facebook.com Site Info", Address: *http://www.alexa.com/siteinfo/facebook.com*, published in 2010, visited on 2010-12-01
41. M.D. Hill and M.R. Marty, "Amdahl's Law In the Multicore Era", in *Computer*, IEEE Computer Society, 2008

42. J. Rattner, "Multi-core to the masses", in *Proceedings of the 2005 conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society 2005
43. S.P. Dandamundi, "Reducing hot-spot contention in shared-memory multiprocessor systems", in *Concurrency*, IEEE Computer Society, 1999
44. W.N. Scharer, III and M.L. Scott, "Advanced contention management for dynamic software transactional memory", in *Proceedings of the 2005 ACM symposium on Principles of distributed computing*, ACM, 2005
45. Xiaohuang Huang et al., "XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines", in *Proceedings of the 2010 IEEE International Conference on Computer and Information Technology*, IEEE Computer Society 2010
46. R. Shiveley and L. Mead: "The New Economics of Mission-Critical Computing - an Intel Corp. whitepaper", Address: *http://www.intel.com/Assets/en_US/PDF/whitepaper/323911.pdf*, published in 2010, visited on 2011-01-05
47. N. Agrawal et al, "Design tradeoffs for SSD performance", in *USENIX 2008 Annual Technical Conference*, USENIX Association, 2008
48. P. Fallara, "Disaster recovery planning", in *IEEE Potentials*, IEEE, 2004
49. P. Chen et al, "RAID: High-Performance, Reliable Secondary Storage", in *ACM Computing Surveys*, ACM, 1994
50. T. Berners-Lee et all: "RFC 1945: Hypertext Transfer Protocol - HTTP/1.0", Address: *http://tools.ietf.org/html/rfc1945*, published in 1996, visited on 2011-01-10
51. P. Mockapetris: "RFC 1034: Domain Names - Concepts and Facilities", Address: *http://tools.ietf.org/html/rfc1034*, published in 1987, visited on 2011-01-11
52. D. Kristol et al: "RFC 2109: HTTP State Management Mechanism", Address: *http://tools.ietf.org/html/rfc2109*, published in 1997, visited on 2011-01-11
53. D. Kristol et al: "RFC 2965: HTTP State Management Mechanism", Address: *http://tools.ietf.org/html/rfc2965*, published in 2000, visited on 2011-01-11
54. H. Pucha et al, "Understanding network delay changes caused by routing events", in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM, 2007
55. V. Cardellini et al, "Geographic load balancing for scalable distributed Web systems", in *Proceedings of the 8th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE, 2000
56. Data Center Knowledge: "The Facebook Data Center FAQ", Address: *http://www.datacenterknowledge.com/the-facebook-data-center-faq/*, published in 2010, visited on 2011-01-15
57. Google Inc.: "Google Research", Address: *http://research.google.com/*, published in 2011, visited on 2011-01-16
58. S. Shankland: "Google uncloaks once-secret server", Address: *http://news.cnet.com/8301-1001_3-10209580-92.html*, published in 2009, visited on 2011-01-15
59. J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", in *Communications of the ACM*, ACM, 2008
60. Pingdom: "Map of all Google data center locations", Address: *http://royal.pingdom.com/2008/04/11/map-of-all-google-data-center-locations/*, published in 2008, visited on 2011-01-16
61. B. Barrett: "Google's Insane Number of Servers Visualized", Address: *http://gizmodo.com/5517041/googles-insane-number-of-servers-visualized*, published in 2010, visited on 2011-05-20
62. J. Dawn: "Celebrating a New Year with a New Tweet Record", Address: *http://blog.twitter.com/2011/01/celebrating-new-year-with-new-tweet.html*, published in 2011, visited on 2011-02-28

63. J. Paul: "Twitter & Performance: An update", Address: *http://engineering.twitter.com/2010/07/twitter-performance-update.html*, published in 2010, visited on 2011-02-28
64. J. Paul: "Room to grow: a Twitter data center", Address: *http://engineering.twitter.com/2010/07/room-to-grow-twitter-data-center.html*, published in 2010, visited on 2011-02-18
65. M. Glotzbach: "Destination: Dial Tone -- Getting Google Apps to 99.99%", Address: *http://googleenterprise.blogspot.com/2011/01/destination-dial-tone-getting-google.html*, published in 2011, visited on 2011-01-16
66. B. Schroeder, E. Pinheiro and W. D. Weber, "DRAM errors in the wild: a large-scale field study", in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ACM 2009
67. I. King: "Intel's Server Demand Outweighs Tablet Shortcoming", Address: *http://www.bloomberg.com/news/2011-01-13/intel-first-quarter-sales-forecast-tops-analysts-estimates-shares-climb.html*, published in 2011, visited on 2011-01-18
68. Top500.Org: "TOP500 List - November 2010 (1-100)", Address: *http://www.top500.org/list/2010/11/100*, published in 2010, visited on 1011-01-18
69. T. Hoff: "An Unorthodox Approach to Database Design : The Coming of the Shard", Address: *http://highscalability.com/unorthodox-approach-database-design-coming-shard*, published in 2009, visited on 2011-01-21
70. L.A. Barroso at al, "Web search for a planet: The Google cluster architecture", in *IEEE Micro*, IEEE, 2003
71. R. Miller: "Database Sharding Helps High-Traffic Sites", Address: *http://www.datacenterknowledge.com/archives/2007/04/27/database-sharding-helps-high-traffic-sites/*, published in 2008, visited on 2011-01-21
72. H. Li et al, "Pfp: parallel fp-growth for query recommendation", in *Proceedings of the 2008 ACM conference on Recommender systems*, ACM 2008
73. J. Waldo, "Scaling in Games & Virtual Worlds", in *ACM Queue*, ACM, 2008
74. A. Aggarwal et al, "A model for hierarchical memory", in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ACM, 1987
75. D. Vagnerov, "A reinforcement learning framework for online data migration in hierarchical storage systems ", in *The Journal of Supercomputing* , Springer, 2008
76. Redis: "Redis", Address: *http://redis.io/*, published in 2011, visited on 2011-01-22
77. Membase: "Membase", Address: *http://www.membase.org/*, published in 2011, visited on 2011-01-22
78. M. Olson et al, "Berkeley db", in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, USENIX 1999
79. Ivan Voras, Mario Žagar, "Web-enabling Cache Daemon for Complex Data", in *Journal of Computing and Information Technology*, , 2008
80. P. Kaminski: "NUMA aware heap memory manager", Address: *http://developer.amd.com/Assets/NUMA_aware_heap_memory_manager_article_final.pdf* , published in 2009, visited on 2011-01-24
81. R. Singhal, "Inside Intel Next Generation Nehalem Microachitecture", in *Intel Developer Forum*, Intel 2008
82. J. Jackson: "AMD Server CTO: Core Wars Will Subside", Address: *http://www.pcworld.com/article/207892/amd_server_cto_core_wars_will_subside.html*, published in 2010, visited on 2011-01-25
83. AMD: "AMD Opteron™ 6000 Series Platform", Address: *http://www.amd.com/us/products/server/processors/6000-series-platform/pages/6000-series-platform.aspx*, published in 2010, visited on 2011-01-25
84. Intel: "White paper: Intel® Xeon® Processor 7500 Series impact analysis", Address: *http://communities.intel.com/docs/DOC-4997*, published in 2010, visited on 2011-01-25

85. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *AFIPS '67 Proceedings of the Spring 1967 joint computer conference*, ACM 1967
86. J. Corbet: "KS2009: How Google uses Linux", Address: *http://lwn.net/Articles/357658/*, published in 2009, visited on 2011-01-28
87. Ivan Voras, Mario Žagar, "Characteristics of Multithreaded Models for High-performance IO Driven Network Application", in *Proceedings of the Africon 2009 International Conference*, 2009
88. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", in *IEEE Transactions on Computers*, IEEE, 1979
89. E. W. Dijkstra, "Solution of a problem in concurrent programming control", in *Communications of the ACM*, ACM, 1967
90. P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent control with “readers” and “writers”", in *Communications of the ACM*, ACM, 1971
91. C. A. R. Hoare, "Monitors: an operating system structuring concept", in *Communications of the ACM*, ACM, 1974
92. M. P. Herlihy, "Impossibility and universality results for wait-free synchronization", in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, ACM 1988
93. S. V. Adve, "Shared memory consistency models: a tutorial", in *IEEE Computer*, IEEE, 1996
94. P. A. Bernstain, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987
95. W. Vogels, "Eventually Consistent", in *ACM Queue*, ACM, 2008
96. M. Crovella, R. Frangioso and M. Harchol-Balter, "Connection scheduling in web servers", in *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, USENIX 1999
97. D. Pariag et al, "Comparing the performance of web server architectures", in *ACM SIGOPS - Operating System Review*, ACM, 2007
98. J. K. Ousterhout, "Why Threads are a Bad Idea (for most purposes)", in *Proceedings of the 1996 USENIX Technical Conference*, USENIX 1996
99. Ivan Voras, Danko Basch, Mario Žagar, "A High Performance Database for Web Application Caches", in *Proceedings of the 14th IEEE Mediterranean Electrotechnical Conference*, 2008
100. V. Pai et al, "Flash: An efficient and portable Web server", in *Proceedings of the 1999 USENIX Annual Technical Conference*, USENIX, 1999
101. J. Lemon, "Kqueue: A generic and scalable event notification facility", in *Proceedings of BSDCon 2000*, BSDCon 2000
102. G. Banga, J. C. Mogul and P. Druschel, "A scalable and explicit delivery mechanism for UNIX", in *Proceedings of the 1999 USENIX Technical Conference*, USENIX 1999
103. F. Dabek et al, "Event-driven programming for robust software", in *Proceedings of the 10th ACM SIGOPS European Workshop*, ACM 2002
104. R. von Behren, J. Condit and E. Brewer, "A scalable and explicit delivery mechanism for UNIX", in *Proceedings of the 2003 workshop on Hot Topics in Operating Systems*, USENIX 2003
105. Peng Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives", in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, USENIX 2007
106. D. Libenzi: "/dev/epoll Home Page", Address: *http://www.xmailserver.org/linux-patches/nio-improve.html*, published in 2001, visited on 2011-01-30
107. R. Bayer, "Symmetric binary B-Trees: Data structure and maintenance algorithms", in *Acta Informatica*, Springer-Verlag, 1971

108. R. Sedgewick and L. J. Guibas, "A dichromatic framework for balanced trees", in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*, IEEE 1978
109. A. Appleby: "SMHasher & MurmurHash", Address: *http://code.google.com/p/smhasher/*, published in 2011, visited on 2011-02-07
110. J. Martin, *Managing the data-base environment*, Prentice-Hall, 1983
111. A.J. Smith, "Cache Memories", in *ACM Computing Surveys*, ACM, 1982
112. A. Iyengar, "Design and performance of a general-purpose software cache", in *Proceedings of the 1999 IEEE International Conference on Computing and Communications*, IEEE 1999
113. B.N. Bershad, " Practical considerations for non-blocking concurrent objects", in *Proceedings the 13th International Conference on Distributed Computing*, IEEE 1993
114. M. M. Michael and M. L. Scott, "Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing", in , ACM 1996
115. B. Verghese et al, "Operating system support for improving data locality on CC-NUMA compute servers", in *ACM SIGOPS Operating Systems Review*, ACM, 1996
116. D. Watts and R. Moon, "System x3755 Technical Introduction", technical document published by IBM, 2007
117. T. Li et al, "Efficient operating system scheduling for performance-asymmetric multi-core architectures", in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ACM, 2007

12. Indexes

Illustration Index

Index of Tables

13. Biography

Ivan Voras was born in Slavonski Brod, Croatia in 1981. He received his dipl.ing. degree in Computer engineering in 2006 from the University of Zagreb Faculty of electrical engineering and computing (FER) in Croatia. Since 2006 he has been employed by the Faculty as an Internet Services Architect and has enrolled in the PhD program in the same year. He has participated in research projects at the Department of control and computer Engineering, and his current research interests are in the fields of distributed systems and network applications, with a special interest in performance optimizations. As a graduate and post-graduate student he has received several Faculty and national-level awards for his work, and has been a four-time participant in Google's Summer of Code program. He is an active member of several Open source projects and is a regular contributor to the FreeBSD operating system.

Contact e-mail address: ivan.voras@fer.hr.

Životopis

Ivan Voras je rođen 1981. u Slavonskom Brodu, u Hrvatskoj, gdje je završio osnovnu školu i prirodoslovno-matematičku gimnaziju. U 2006. godini je diplomirao na Fakultetu elektrotehnike i računarstva u Zagrebu, sa titulom diplomiranog inženjera računarstva. Te iste godine se zapošljava na Fakultetu kao Arhitekt usluga na Internetu pri Centru informatičke potpore i upisuje poslijediplomski doktorski studij. Tijekom doktorskog studija sudjelovao je u znanstvenim projektima pri Zavodu za automatiku i računalno inženjerstvo, a istraživački interesi su mu u područjima raspodijeljenih sustava i mrežnih aplikacija, sa posebnim naglaskom na optimizaciju performansi. Tijekom dodiplomskog i poslijediplomskog studija primio je nekoliko nagrada na fakultetskoj i državnoj razini za svoj rad u ovim područjima te je četiri puta sudjelovao u programu Google Summer of Code. Član je nekoliko Open source projekata i jedan od redovitih suradnika na operacijskom sustavu FreeBSD.

E-mail adresa za kontakt: ivan.voras@fer.hr