aiCache, Inc.

www.aiCache.com

Version 6.x

aiCache V6 User Guide

At aiCache, we take pride in positive environmental impact of our technology. Millions fewer tons of pollutants enter Earth's atmosphere due to tremendous reduction of server footprint, made possible by our product.

Please consider taking this one step further and do not print this Guide out. Keep it in electronic form instead. It is easy to use, easy to search and it always stays up to date at http://aicache.com/pdf/adminguide.pdf . Thank you !

Table of Contents.

aiCache End User License Agreement.

THIS IS A LEGAL AGREEMENT ("AGREEMENT") BETWEEN YOU AND AICACHE, ("LICENSOR"). PLEASE READ THIS AGREEMENT CAREFULLY. BY INSTALLING AND/OR USING THIS SOFTWARE, AS A RESULT OF PURCHASE AND/OR TRIAL DOWNLOAD, YOU AGREE, ON BEHALF OF YOURSELF AND YOUR COMPANY (COLLECTIVELY "LICENSEE"), TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, YOU ARE NOT PERMITTED TO USE THE SOFTWARE.

1. License Grant. Licensor grants to Licensee a worldwide, nonexclusive, nontransferable, royalty free license to use the aiCache software (the "Software"). Licensee is permitted to make a single copy of the Software for backup purposes. Except as expressly authorized above or as permitted by applicable law, Licensee will not: copy, in whole or in part, Software or any related documentation; modify the Software; reverse compile, reverse engineer, disassemble or reverse assemble all or any portion of the Software; rent, lease, license, sublicense, distribute, transfer or sell the Software; or create derivative works of the Software. Licensee obtains no rights in the Software except those given in this limited license.

2. Ownership. The Software, any related documentation and all intellectual property rights therein are owned by Licensor, its affiliates and/or its suppliers. The Software is licensed, not sold. Copyright laws and international copyright treaties, as well as other intellectual property laws and treaties, protect the Software. Licensee will not remove, alter or destroy any copyright, proprietary or confidential notices placed on the Software or any related documentation. Licensee agrees that aspects of the Software, including the specific design and structure of individual programs, constitute trade secrets and/or copyrighted or patented material of Licensor, its affiliates and/or its suppliers. Licensee agrees not to disclose, provide, or otherwise make available such trade secrets or material in any form to any third party without the prior written consent of Licensor. Licensee agrees to implement reasonable security measures to protect such trade secrets and material.

3. NO WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW AND SUBJECT TO ANY STATUTORY WARRANTIES THAT CANNOT BE EXCLUDED, THE SOFTWARE AND ANY RELATED DOCUMENTATION ARE PROVIDED TO LICENSEE "AS IS." LICENSOR MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND EXPRESSLY DISCLAIMS AND EXCLUDES TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW ALL REPRESENTATIONS, ORAL OR WRITTEN, TERMS, CONDITIONS, AND WARRANTIES, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY AND NONINFRINGEMENT. WITHOUT LIMITING THE ABOVE, LICENSEE ACCEPTS THAT THE SOFTWARE MAY NOT MEET LICENSEE'S REQUIREMENTS, OPERATE ERROR FREE, OR IDENTIFY ANY OR ALL ERRORS OR PROBLEMS, OR DO SO ACCURATELY. LICENSEE USES THE SOFTWARE AT HIS/HER OWN RISK. This Agreement does not affect any statutory rights Licensee may have as a consumer.

4. EXCLUSION OF CONSEQUENTIAL AND OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR, ITS AFFILIATES OR ITS SUPPLIERS BE LIABLE TO LICENSEE, LICENSEE'S CUSTOMERS, OR OTHER USERS, FOR damages of any kind including, WITHOUT LIMITATION, DIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE LICENSE OF, USE OF, OR INABILITY TO USE THE SOFTWARE (INCLUDING, WITHOUT LIMITATION, DATA LOSS OR CORRUPTION, ECONOMIC LOSS, LOSS OF ACTUAL OR ANTICIPATED PROFITS, LOSS OF CONFIDENTIAL INFORMATION, BUSINESS INTERUPTION, LOSS OF PRIVACY, FAILURE TO MEET ANY DUTY OF REASONABLE CARE OR NEGLIGENCE) EVEN IN THE EVENT OF THE FAULT, TORT, STRICT LIABILITY, BREACH OF CONTRACT, BREACH OF STATUTORY DUTY OR BREACH OF WARRANTY OF LICENSOR, ITS AFFILIATES OR SUPPLIERS AND EVEN IF LICENSOR, ITS AFFILIATES OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR SUCH DAMAGES WERE FORESEEABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTIAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY.

5. LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR'S LIABILITY EXCEED THE LICENSE FEE PAID BY LICENSEE OR US$5.00, WHICHEVER IS GREATER. THIS LIMITATION OF LIABILITY AND RISK IS REFLECTED IN THE PRICE OF THE SOFTWARE. NOTWITHSTANDING THE FOREGOING, NOTHING IN THIS AGREEMENT SHALL EXCLUDE OR LIMIT LICENSOR'S LIABILITY TO LICENSEE FOR ANY LIABILITY THAT CANNOT, AS A MATTER OF APPLICABLE LAW, BE EXCLUDED OR LIMITED.

6. INDEMNIFICATION. BY ACCEPTING THIS AGREEMENT, LICENSEE AGREES TO INDEMNIFY AND OTHERWISE HOLD HARMLESS LICENSOR, ITS OFFICERS, EMPLOYEES, AGENTS, SUBSIDIARIES, AFFILIATES, SUPPLIERS AND OTHER PARTNERS FROM ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES ARISING OUT OF, OR RELATING TO, OR RESULTING FROM LICENSEE'S USE OF THE SOFTWARE OR ANY OTHER MATTER RELATING TO THE SOFTWARE.

7. Termination. Licensor may immediately terminate this Agreement and the license granted hereunder if Licensee fails to comply with the terms and conditions of this Agreement. Upon such termination, Licensee must immediately cease using the Software, destroy or delete all copies of the Software and upon the request of Licensor, certify the destruction or deletion of the Software. Licensee may terminate this Agreement and the license granted hereunder at any time by destroying or deleting all copies of the Software. ALL DISCLAIMERS,

LIMITATIONS OF LIABILITY AND ANY OTHER PROVISIONS INTENDED TO SURVIVE TERMINATION WILL SURVIVE ANY TERMINATION AND CONTINUE IN FULL FORCE AND EFFECT.

8. International Trade Compliance. The Software and any related technical data is subject to the customs and export control laws and regulations of the United States ("U.S.") and may also be subject to the customs and export laws and regulations of the country in which it is installed.

9. Governing Law. The laws of the State of Nevada, United States, without regard to conflicts of laws principles, govern this Agreement. If applicable law does not permit the parties to agree to the governing law, the laws of the country in which Licensee installs or acquires the Software govern this Agreement. To the extent permitted by applicable law, any dispute arising under this Agreement or relating to the Software shall be resolved by a court of proper jurisdiction in Nevada, United States. Licensee and Licensor irrevocably submit to the jurisdiction of such courts and waive any and all objections to jurisdiction a party may have under applicable law. Notwithstanding the foregoing, if the Software is installed or acquired in the People's Republic of China, any dispute arising under this Agreement or relating to the Software shall be resolved by binding arbitration, held in Las Vegas, Nevada, United States, under the Judicial Arbitration and Mediation Services (JAMS) International Arbitration Rules.

10. Legal Effect. This Agreement describes certain legal rights. Licensee may have other rights under applicable law. This Agreement does not change Licensee's rights under applicable law if such laws do not permit the Agreement to do so.

11. Miscellaneous. This Agreement constitutes the entire agreement between Licensor and Licensee and governs Licensee's use of the Software, superseding any prior agreement between Licensor and Licensee relating to the subject matter hereof. Any change or modification to this Agreement will only be valid if it is in writing and signed on behalf of Licensor and Licensee. A failure by either party to enforce its rights under this Agreement is not a waiver of those rights or any other rights it has under this Agreement. The parties disclaim the application of the United Nations Convention on the International Sale of Goods. The terms of this Agreement are severable. If any term is unenforceable for any reason, that term will be enforced to the fullest extent possible, and the Agreement will remain in effect. The Software and any related technical data are provided with restricted rights. Use, duplication, or disclosure by the U.S. Government is subject to the restrictions as set forth in subparagraph (c)(1)(iii) of DFARS 252.227-7013 (The Rights in Technical Data and Computer Product) or subparagraphs (c)(1) and (2) of 48 CFR 52.227-19 (Commercial Computer Product – Restricted Rights), as applicable. To the extent permitted by applicable law, Licensee may not assign this Agreement, and any attempted assignment will be null and void. aiCache, the aiCache logo, and other aiCache names and logos are the trademarks of aiCache Technology LLC or its affiliates.

12. Contact Information. Any inquiries regarding this Agreement or the Software may be addressed to Licensor contact found on the website, www.aicache.com.

Introduction to aiCache.

aiCache is a unique software product that creates a better user experience by increasing the speed and availability of your site. aiCache accomplishes this by offloading request processing from the web, application and database tiers, reducing code complexity and the cost for servers, space, power and cooling.

Typical web site setup overview.

Before we discuss the numerous benefits and advantages aiCache has to offer, let us spend few quick minutes reviewing typical web site setup and associated challenges.

A typical infrastructure setup of an Internet web site might contain some/all of the following major components:

- **Web Servers** - such as Apache, Microsoft IIS and similar. When a number of servers are used, this component is frequently called the *web farm*. The reasons for having a number of these servers are speed & availability. Using a single web server creates a single point of failure and a single server is typically not capable of coping with significant amount of web requests. It is not uncommon to see web farms composed of dozens of web servers.

The web servers receive HTTP *requests* from end-user's web browsers and serve HTTP *responses* back, in the form of either static files, obtained from a file system - such as images, Javascript, CSS or dynamically generated responses. These web servers are called ***origin servers*** henceforth, as it is where the Web content originates.

- **Application or Middleware Servers**: for example, JBoss App Server, Tomcat Web Container[1], BEA WebLogic, IBM Webshere or custom-written software. These servers execute business logic functions such as customer profile editing, rendering of a blog page, product search, shopping cart functionality etc. The reasons for having a number of these servers are similar to those for "farming" of web servers.

- **Back-end Servers**: such as Database Servers (e.g. MySQL, Oracle, Microsoft SQL Server). This is where persistent information, such as customer profiles, product information, message board articles, site's editorial content might be stored. Frequently the database itself might be partitioned ("sharded") or setup as write master and a number of read replicas, resulting in multiple database servers. Sometimes information store is not a database, but a file system, such as NFS (prevalent in Unix/Linux environments) or CIFS (MFST Windows-based setups).

[1] Java, Sun, Solaris are registered Trademarks of Sun Microsystems Inc; Weblogic is registered Trademark of BEA Inc; Websphere, AIX are registered Trademarks of IBM. Apache , Jakarta, Tomcat are registered Trademarks of Apache Software Foundation; Oracle is a registered Trademark of Oracle Inc; IIS, SQL Server are registered Trademarks of Microsoft Inc;

- **Shared file store**: such as NFS or CIFS file servers or filers. This is frequently used to share common files – configuration files, binaries, static web content and backup data.

- **Search engines/appliances**: such as Google, Sphinx, Lucene, Fast etc . These are used to execute search requests against site's content – such as editorial content, user blogs/posts, video/image metadata etc.

Rather frequently, application server functionality is "embedded" right into web servers. Examples include Apache/Tomcat (Apache webserver connected to Tomcat servlet/JSP container via JK connector), Apache/PHP with PHP interpreter embedded into Apache webserver, IIS/ASP (Microsoft ASP) and ColdFusion via the web server connector.

Some web sites include custom back-ends. For example, a web site dedicated to news might rely on proprietary systems that provide some kind of news wires APIs, a web site serving financial quotes might rely on market data feeds and so on.

There are, of course, much simpler web site setups than that, typically consisting of a single web server. Such web server might co-reside (share hardware) with database software and result in a "3-in-1" setup. Such simplified setups are rarely adequate for sites with any volume of traffic.

We are not mentioning another important infrastructure component - the network. It typically encompasses Internet uplinks, routers, firewalls, load balancing hardware and network switches. This Guide assumes that network capacity and its performance, across all of the layers/components, is sufficient to support required volume of traffic.

Common challenges experienced by web sites with heavy traffic.

When subjected to heavy user traffic, typically manifested by large number of requests (typically measured as RPS – requests per second) and simultaneous client connections, the common web site setup as described in previous section has a number of associated issues.

End-user experience suffers, especially with dynamically generated content, which **is often slow to generate and stresses all or most of components of the infrastructure** – from web servers to app servers to the backend databases.

It is not uncommon to see scenarios where dynamically generated web pages take a few seconds to be generated on the fly, each such request requiring creating a new process on web servers, compilation of code and/or instantiation of hundreds of new objects on app servers and execution of multiple, fairly expensive queries on DB side. Even when serving static content off a file system, the resulting disk IO might place heavy burden on web server. Imagine having to serve hundreds of requests per second, each requiring similar resources and you can easily see why and how web sites might get slow or melt down outright.

Quite often the majority of static and dynamically generated content is the same for significant percentage of web site visitors, but there is **no way to share it across different users** in satisfactory and flexible way.

The logic that drives content generation **often relies on a-process-per-request model**, meaning a new system process/thread needs to be established and maintained for each request. This places a heavy burden across infrastructure, requiring increase in number of servers and associated spend on power, cooling, space and

maintenance. Higher server CPU loads and increasing memory utilization, resulting from such dedicated-process-per-request architecture routinely bring even most powerful servers to their knees, resulting in untold hours of downtime, unhappy users, declining customer retention, shorter site visits and lost revenues.

Quite frequently the web servers, already busy processing user requests, are also tasked with **HTTPS traffic encryption, a very CPU-intensive activity**.

Some sites **do not scale horizontally**, due to backend or other limitations and simply cannot be made to scale by adding extra HW, without going through major overhaul of code base and design, yet there's no time or money in budgets to allow for such overhauls.

In order to tackle the performance problems, all too often companies first resort to **buying more HW** – more web servers, more application servers, database servers and so on. Often at quite a steep acquisition cost and at ever increasing expense and run rates resulting from having to manage ever growing environments, expand existing or move to new datacenters, acquire more power and cooling capacity etc. **It is quite common to see situations when existing datacenter or hosting cage capacity is exhausted and then expense grows ever more out of control**.

Faced with numerous limitations (spend, space/power/cooling in datacenters) companies frequently share servers between different web sites and applications. It is **exceedingly hard to control quality of code in the time of smaller TTM requirements** and as a result, **faulty code may bring down a whole number of web sites and servers**. It could be as simple/direct as overloading of the servers that the faulty code is running on (along with all other applications sharing the same server) or indirect – by stressing out backend database system that is shared with other application.

When there's an outage in such a complex, shared web setup (it is not uncommon to see hundreds of servers in today's Datacenters serving some of the busiest web sites), **it is often hard to pinpoint the problem application** due to limited instrumentation available in applications/code. The simple question of "*which application is running slow and bringing the site (or whole shared web farm) down right now?"* becomes very hard to answer. Resorting to complete restarts of web farms is a common attempt at remediation – only to see the web servers go down almost instantly with no easy way to identify just what is ailing the setup and restore the service.

When and if the culprit is finally identified, it still is **very hard to isolate/disable the failing application** to let other applications to stay up, due to inevitable interdependencies.

It is equally **hard to have up-to-the-second information on current traffic**: volume (requests per second, bytes in/out, different request types), distribution across web sites (which one is being hit the hardest, what is the spike ?) and response times for different applications (which one is the slowest: customer login, product search, news search ?) . While a limited subset of this information can be distilled from log files collected on web servers, it is a time consuming, all-hands-on-deck process that is never real-time.

Heavy traffic sites routinely serve millions of web requests a day ("hits") and generate very large log files that are then processed for various purposes. Due to the sheer size of these log files, **log file management** becomes an **unpleasant chore** and all too often web servers simply run out of disk space and stop serving requests.

It is not uncommon for hackers to target web serves in their attacks. **Proprietary and wide-ranging architectures of web sites make it much harder to maintain adequate security**. Security holes are often first

discovered and exploited by hackers, leaving web site owners to wait for days or weeks for security patches to be issued by respective software vendors.

aiCache to the rescue.

Let's consider a popular news web site. Most of web pages on such web sites typically contain information that is updated (published) quite frequently throughout the day. In order to guarantee freshness of content, pages are likely to be generated dynamically upon request. This response generation process is typically slow and consumes resources on web servers, app and database servers alike.

Yet even with these seemingly non-stop changes and breaking news to deliver, it can be shown that **most pages can be cached for some period of time (from few seconds to few minutes or more), still keeping the content sufficiently fresh**, while greatly enhancing end-user experience and dramatically reducing the load on overall web site infrastructure.

Same can be said of most types of web sites on the Internet:

- Community and social networking sites – where 95%+ of access is read-only, such as reading of messages or blog posts, browsing of user galleries or popular profile on a social networking site. Each view of these items typically requires establishment of a client connection, spawning of a process on the web server and application server, interpretation and execution of some logic, a connection to database server, a number of SQL queries - often resulting in numerous disk IO and physical reads, before the end result is obtained, processed and send to the requestor. While absolute majority of content hasn't changed since it was requested by a previous viewer some milliseconds ago, it is regenerated anew for each and every request . This type of content is a perfect candidate for caching - instead of generating this content anew for each visitor.

- Busy E-commerce sites where users browse popular items before selecting some into a shopping basket and checking out. Again, most of the activities are read-only access to assorted product data – to obtain description, pictures, customer feedback and inventory of different products. These don't change that frequently at and should be cached instead of regenerating pages on the fly.

- Media web sites that offer access to popular videos – often seeing tens of thousands of visitors trying to access a particular video when it goes "viral", all in span of just few minutes, generating exactly the type of traffic capable of melting down even largest web sites. Again, the pattern is very common - a page is displayed with some basic metadata about a particular video, may be along with user comments and ratings - the screen is identical for each and every user that is trying to view the actual video. This type of content, too, is a perfect candidate for caching - instead of generating this content anew for each visitor.

- News or news aggregation web sites, serving editorial content, news wire stories, financial market quotes or blogs to visitors. On a day when a major story breaks, world sporting record is broken or financial markets see a major move, such web sites are likely to see a very significant increase in traffic, as visitors flock to the site's home page, read news articles, view stock quotes, search for news

moving the markets etc. Being able to cache these types of pages for just a few seconds often means a difference between a healthy website with brisk response times and a total meltdown.

aiCache is a web site and web application acceleration product. It takes these basic ideas of caching frequently requested content and brings them to the next level. Along with enhancing end-user experience and dramatically slashing load on, and requirements to, the web site infrastructure, aiCache also delivers much richer set of features and functionality that is not available with any other products - we shall explain these in great detail later in this guide. aiCache's heritage can be traced back to 2001, when the product made first appearance.

aiCache is placed in front of traditional web servers. As a result of such placement, aiCache is capable of intercepting and responding to user requests before they ever hit origin web servers, App or Database servers, helping to solve or mitigate most of the typical issues mentioned previously. In other words, by deploying aiCache you can address most, if not all of your web site's problems at a minute fraction of cost, compared to brute-force buy-more-HW approach or equally expensive and time consuming code changes and site redesigns.

The most noticeable benefit that deployment of aiCache offers is the ability to cache frequently accessed content, so that the regular chain of "Web Servers - App Servers – Backend Servers" is accessed less frequently. It results in dramatic improvements in terms of web site performance and tremendous reduction of load across all components of your infrastructure.

aiCache is a Linux application, custom written in C for maximum performance and flexibility. It runs on most popular and ubiquitous Linux distributions. It utilizes the *EPOLL* mechanism, available only on Linux, that offers unmatched scalability and performance for network IO, translating to ability to serve tens of thousands of simultaneous clients off a single server (solving the famous C10K dilemma) with virtually zero overhead. aiCache is a *right-threaded* application, utilizing a very limited number of threads (processes) – typically well under a dozen, to do most of the processing. This is in marked contrast to some other products that rely on a single process or large pools of thousands of processes or threads, dedicating a thread to each client request, a design that doesn't scale well in our opinion.

Test results show that under favorable conditions (fast clients, smaller, cacheable responses, sufficient network bandwidth) single aiCache instance running on Intel(tm) Nehalem processor system is capable of serving more than 260,000 RPS, that's around **1 billion requests per hour** !

And lastly, aiCache is developed by a team that's been at the forefront of designing, building and maintaining some of the world's busier web sites – encompassing pretty much every single imaginable type of web site and web technology. It is from this experience that features of the product were conceived, developed and refined – a rather distinguishing feature indeed.

aiCache features at a glance.

- Dynamic caching & sharing of web content, including GET and POST requests.
- RAM-based caching for unsurpassed response times. aiCache generates almost no disk IO.
- Ability to function as HTTPS end point, forwarding decrypted traffic to origin servers.

- Excellent scalability and high performance, tested to excess of 260,000 RPS per server.
- Flexible document freshness control, including cookie-driven control .
- Cluster support - run a centrally managed set of aiCache servers.
- Cluster-aware on-demand response-driven cache expiration .
- Fallback mode to keep your site up even in case of backend failures.
- Origin-Server-Of-Last-Resort feature, to further help server traffic in case of backend failures.
- Selective log suppression and decimation.
- Automated time-based or size-based log file rotation.
- Three different ways to monitor aiCache: cluster-aware CLI, Web and SNMP, with extremely rich set of statistics available.
- Full support for regular expressions for pattern matching.
- Feature rich command line interface (CLI): full set of statistics, inventory check, document expiration and removal, log file rotation and much more.
- Load balancing of requests across a number of "origin" web servers with 3 different load balancing metrics supported.
- Monitoring health of origin servers, including content matching.
- Response preload/pre-fetch.
- Comprehensive security features, including cluster-aware, multi-layered defense against Denial-of-Service attacks and protection against malicious requests .
- Flexible Mobile processing
- Flexible Geo processing
- User-Agent-based redirection
- Built-in comprehensive Health Monitor
- Configurable on-the-fly compression
- Advanced alerting warns you before there's user impact
- Extend aiCache with user-provided logic via aiCache plug-ins.
- Inject static (file-system based) content into accelerated sites

- Simply and automate configuration management in distributed setups
- And many more unique features that we shall cover in depth in this document

Document Conventions.

Throughout this document, you will see examples of operating system commands, aiCache command line interface (CLI) commands, snippets from configuration files, and screen captures. Font and color changes are employed so that these could be clearly distinguished from the surrounding text.

We use terms *visitor, user, and customer* to refer to web site visitors. Such users visit web sites by directing their web browsers (Internet Explorer, Firefox, Chrome, Opera , Safari etc) to URLs that are hosted by web sites.

We use term "*origin (web) server*" to refer to web server(s) that are being *accelerated* by deploying aiCache.

We use terms "w*eb document*", *"response"* to refer to files stored on, or content generated by, origin web servers and subsequently requested by web site's visitors. Such w*eb documents* or *responses* include, but are not limited to, dynamic pages generated using JSP/Servlet, PHP, .Net and other technologies, static HTML, JavaScript, Content style sheets, images in various formats, PDF files, Flash animations etc.

We use term "*cacheable*" request/response to refer to responses that can be cached/shared via aiCache. Opposite to such requests are "*0-TTL"*, *"non-cacheable*" requests/responses - these cannot and/or should not be cached because they contain private data or because caching these doesn't offer any significant benefits. Yet, as you shall discover, aiCache offers significant benefits even for traffic that cannot be cached. Some of the benefits include request/response and connection offloading, response preload/pre-fetch and real-time traffic reporting and alerting.

Key new features in aiCache V6.

Compared to prior versions, version 6 of aiCache offers following key new features and benefits:

- Support for HTTPS. Now you can have aiCache manage HTTPS traffic, while, optionally, communicating via HTTP with origin servers, relieving them from encryption chores and administration overhead.

- Support for customer logic via flexible, yet very easy to code, plugin architecture. Now you can have arbitrary number of your own plugins examine incoming requests and tell aiCache to drop, redirect, rewrite, change the TTL of requests or provide complete response bodies. aiCache automatically collects and reports all of the plugin statistics, without having to code for it.

- Better scalability on multi-core systems with more than 4 CPU cores. aiCache has been tested to over 260,000 Keep-Alive client req/sec. While very few sites ever will push this kind of traffic, but massive scalability is there should you ever need it.

- Multiple listen ports, both HTTP and HTTPS, can now be specified. You can tie a listen port to a website or still have aiCache tie incoming requests to website via regular logic.

- Support for "*origin servers of last resort*". Now, in addition to regular aiCache failback feature, you can try filling requests against dedicated content-replica servers, should attempts to obtain responses from regular origin servers fail.

- Increased set of collected statistics, at global, website and pattern levels. As usual, you can see the stats via Web, CLI and SNMP interfaces.

Version 6 supports V5 configuration file syntax, so it is a simple drop-in replacement. A number of V6-specific settings have been added, explained later in this Guide.

Effective Jan 2010 aiCache Ltd stops all new feature development for V5 of aiCache and places V5 into maintenance mode, where we continue to address minor bugs. We recommend all of our customers to switch to aiCache V6.

Example Web Site.

As we discuss features and configuration of aiCache, we shall refer to a fictional web site setup, so that settings, commands and configuration files referred to throughout the rest of this document make more sense.

This fictional web site is **www.acmenews.com**[2]. The AcmeNews is in the business of providing news to a large number of users. AcmeNews prides itself in being first to most stories, entertainment or financial, aiming to have shortest story publishing time. Some content on the site, such as breaking news and section fronts, is refreshed rather frequently – every minute, to stay up to date. Some other content is more static and is only refreshed once an hour, once a day and once a week.

AcmeNews supports community features, allowing users to comment on articles. When a page (news wire story, editorial content etc) is displayed, user comments are shown below the actual story. AcmeNews prides itself in allowing its users to enjoy real-time discussion thread updates - as soon a new topic or a response is posted, the content updates to reflect the changes.

In addition to dynamic content, the web content includes large number of Javascript files, CSS files and images, with file sizes ranging from 50B (for simple one-pixel images) to 50KB+ for some Javascript files and higher quality images.

Internally the site is composed of a number of origin servers (the web farm), with the names of **www1.acmenews.com**, **www2.acmenews.com** and so on, to be able to cope with large number of Web hits the site gets. The exact mechanism used to load balance user requests across origin servers is not important. Examples include: multiple DNS A records or CNAME aliasing, combined with dynamic DNS service, such as Dynect or similar, load balancers - such as F5, A10, Netscaler, Alteon, HA Proxy etc.

All of the content served from **www.acmenews.com** is delivered via HTTP protocol, as opposed to the secure form of it, HTTPS. At the same time AcmeNews maintains subscriber profiles for users that want get access to paid subscriptions. In order to allow users to sign-up for these services in a secure fashion, a separate secure web server is setup with the name of **secure.acmenews.com**. This server uses HTTPS to communicate with the subscribers, encrypting all the traffic between the users and the server, so that users' private information is protected while in transit.

Business Development has been busy thinking of more features to add to the site, to attract even more visitors - things like user comments on news articles, dedicated message boards, video section. A dedicated mobile web site is also in works, aimed at attracting mobile clients.

Yet as the popularity of the AcmeNews grows, resulting in more page views, larger number of unique visitors and longer visits, the site finds its response time starting to climb and load on all of the site's infrastructure components increasing. Outages become all too frequent. Environment footprint keeps growing with more and more servers going into the racks, requiring more expense for HW and SW purchases, additional

[2] Any resemblance to real web sites is unintentional.

space, power and cooling. But even throwing money at the problem doesn't seem to work anymore and there's a feeling that a radically different approach is in order as the current path is not sustainable.

And this exactly when the technology team, after days of research and brainstorming, comes up with an idea and … drum roll …. aiCache enters the scene.

Prerequisites.

Understanding your Web Setup.

As the product is very easy to install and configure, chances are you will not need any special help. Basic understanding of the web site setup is helpful. Typical questions that arise during the installation of aiCache include document freshness criteria - as in what can be cached and for how long, origin server configuration (what are the web servers that run the site and how is user traffic directed at them: DNS, load balancers) and cookie/session management issues.

You might elect to install our product yourself, which requires some fairly basic Linux knowledge. Alternatively, you can contact us for a quote on Professional Service engagement - a turn-key package aimed at helping you to get up and running, straight from the source or one of our many partners worldwide.

Server Platform.

You need to have or be ready to obtain a 64 bit platform – based on Intel or AMD CPUs, to run our product on. aiCache requires Linux OS and almost any recent 64 bit distribution will do. 64 bit requirement is there so that you can *use* more than 4GB of RAM per process – which in turn provides more memory for caching of content. The exact amount of RAM you'd need depends on your cache footprint – number and size of cached responses that are stored in the cache.

For example, let's assume you want to cache 1000 active articles that average 50KB in size – such set will require about 50MB of RAM - a very small footprint. 100,000 pieces of content with same average size will require 5 GB of RAM - still perfectly reasonable size. To help with keeping the memory footprint under control, aiCache has a number of tricks up its sleeve, including advanced cache cleaner logic. You can configure just how aggressive you want it to be - we shall have more to say about it later in this document.

As you already know, aiCache is *right-threaded*[3], to take advantage of today's multi-core systems. It runs as a single multithreaded process. You specify number of worker threads you want to have, ideally matching

[3] As opposed to single-threaded (cannot benefit as effectively from multiple CPUs) or process/thread-per-connection designs (exhausts resources quickly with large number of requests/connections).

that number to number of available CPU cores (default number of worker threads is 4). As a ballpark guide, a single, modern CPU core is capable of driving about 20,000 non-keep-alive cacheable RPS. So if you have 4 cores and configure 4 aiCache worker threads, you should expect to be able to handle around 80K RPS, assuming most content is cacheable *and you don't exhaust your network capacity*. Likewise, 8 threads and 8 CPU cores can get you close to 160K RPS and so on. Very few sites push anywhere close to these kind of RPS, but the capacity is there should you ever need it.

Being right-threaded also means that no matter the number of connected clients (30 or 30 thousand) or request/sec (50 RPS or 250,000 RPS) you're serving, you will still have only one process running with configured threads - be ready to be amazed at extremely low CPU utilization that aiCache servers exhibit even when under high loads.

aiCache is capable of fully utilizing gigabit network interfaces, so you are never constrained by aiCache and instead, it is likely that the bottlenecks, if any, will be found at uplink, firewall, network and load balancing level instead.

Please note aiCache is also available as ready-to-go Cloud images, including Amazon AWS image. We strive to offer aiCache as a Cloud product available with other popular Cloud providers and integrators, such as RightScale, 3Tera and others - please check the aicache.com web site for the latest information on aiCache in the Cloud.

Operating Systems and Software Pre-requisites.

Almost any modern Linux distribution will do , as long as it is 64bit kernel, version 2.6.9 or newer. While it doesn't require any additional software or special tricks, as usual you are likely to benefit by using most up-to-date production version of 64bit 2.6+ Linux operating system with, as usual, the most recent recommended security and performance patches applied. aiCache has been tested on Ubuntu 8 and 9, Fedora 10, Open SUSE 11, Red Hat 5.x and CentOS 5.x distributions, all in 64 bit mode.

aiCache requires reasonably recent Glib shared library (2.16.4 or later), Zlib shared library (any reasonably recent version will do), OpenSSL shared library (v0.9.8+) and PCRE shared library**.** All of the libraries that aiCache requires are normally pre-installed during OS installation and require no additional action on your part**.** Libraries are shared, meaning they are not statically compiled into the aiCache binary but are instead resolved and loaded dynamically during run time.

aiCache distribution carries within itself two different aiCache binaries - with and without HTTPS support. This way, if your server lacks OpenSSL libraries and you don't require HTTPS, you can use HTTPS-free aiCache binary.

SNMP integration feature requires Net-SNMP server software. Again, chances are it's been installed as part of OS install, if not you need to follow installation instructions for your flavor of Linux. We cover SNMP integration in-depth later in this document.

aiCache is also available as ready-to-go Cloud images, including Amazon AWS images. We strive to offer aiCache as a Cloud product available with other popular Cloud providers and integrators, such as RightScale, 3Tera and others - please check the aicache.com web site for the latest information on aiCache in the Cloud.

Please note that high-traffic sites might to adjust some of the default values for assorted network parameters/settings, we describe these settings and the recommended values elsewhere in this Guide.

Red Hat 5, Fedora 8 & derivatives warning.

Some Red Hat and Red Hat-derived distributions, such as Fedora 8, Red Hat 5.x and CentOS 5.2 might come with an older version of GLib libraries and require download, compilation and install of newer GLib. You can obtain most recent GLib library at http://www.gtk.org/download-linux.html . Please follow standard practice of running *configure, make, make instal* . Don't worry - the installation of an updated GLib should not take longer than few minutes of your time. Use a package manager of your choice or download and compile the GLib by hand.

When manually compiled and installed, GLib installs into **/usr/local/lib** . To make sure aiCache loads these newer libraries, not the default and older version, you can set **LD_LIBRARY_PATH** to point to **/usr/local/lib** before you start aiCache. You can create a custom shell wrapper for aiCache, including this and other settings, such as **ulimit** etc, in it . For example:

```
# Increase number of allowed file descriptors
ulimit -n 124000
# enable core file
ulimit -c 1000000000
# point to directory with GLIB libraries in it
export LD_LIBRARY_PATH /usr/local/lib
# start aicache
/usr/local/aicache/aicache -f aicache.cfg -l 234324434.lic
```

aiCache won't start unless proper version of GLib library is installed on the system. A tell-tale sign of an outdated GLib library version is an error message during startup that points to a g_* function being not found. To address this please install newer version of GLib, as per instructions above.

Likewise, if you try to start HTTPS-enabled version of aiCache on a server without proper OpenSSL libraries, it will refuse to start. aiCache distribution carries within itself two different aiCache binaries - with and without HTTPS support. This way, if your server lacks OpenSSL libraries and you don't require HTTPS, you can use HTTPS-free aiCache binary.

Installation.

Please note that aiCache is also available as ready-to-go Amazon AWS image. We strive to offer aiCache as a Cloud product available with other Cloud providers - please check the web site for the latest information on aiCache in the Cloud. If you choose to with an aiCache Cloud image, you don't have to download aiCache binary or acquire aiCache licenses- it is all taken care of already, all you need to do is to order and start an official aiCache instance.

Network Setup

Assuming you want to install aiCache on dedicated server(s), you must be able to connect the aiCache server(s) to the network so that the aiCache can reach the origin web servers and in turn can be reached by Internet users. Usually it means that there must be spare ports on your network switches and you have spare IP addresses to assign to these new aiCache server(s).

Depending on volume of traffic you need to handle, you might elect to setup a single network connection on aiCache servers or multi-home them instead. Consider using NIC-teaming if you have redundant network switches to make aiCache's network connection highly available. Later in this guide we describe a VRRP-based HA setup, with redundant, hot-hot aiCache servers assuring maximum uptime.

If you decide to co-locate aiCache software on the same server where an existing origin web server resides, then there are no changes required to the network setup. You will only need to change the port the actual web server listens on a to a different one, as the original port 80 will now be claimed by aiCache. Alternatively, you can use multiple IP addresses (aliases) on the same server, bind you origin web server to one of them, and configure aiCache to use a different one. Now, both origin server and aiCache can share the same port number.

aiCache distribution file.

The product usually comes in the form of a small Linux *tar* archive file that you obtain by direct download from our web site or receive in the mail on some form of media, such as CD. Contained within the tar archive are 2 aiCache binaries: one compiled with HTTPS support and one without , example configuration file, README file, SNMP integration script, convenience install script, alert email script and Admin Guide in Adobe PDF format. Also included is a *plugin* directory - containing plugin header file, plugin build script and example plugin C source file.

After obtaining the distribution file, you can *un-tar* it in a temporary directory - for example in /tmp.

```
cd /tmp
tar -xvf aicache.tar
cd  aicache
```

aiCache binaries.

The aiCache distribution contains 2 aiCache binaries: **aicache** and **aicache_https**, compiled without and with HTTPS support, respectively. Otherwise, both files are identical in their functionality. We provide aiCache

binary (**aicache**) compiled without HTTPS support in case your system lacks OpenSSL libraries - in which case you won't be able to start the HTTPS-enabled aiCache binary (**aicache_https**).

Clearly, if you require HTTPS support, you must run HTTPS-enabled binary and have OpenSSL libraries installed. Your license file should also have https option enabled (all Cloud images are enabled for HTTPS support and require no license file).

Installation.

To ease the installation, aiCache distribution includes a simple installation script - **install.sh**, that automates most of the setup steps described below. The script creates user and group **aicache:aicache, /usr/local/aiacache and /var/log/aicache** directories and copies distribution files to **/usr/local/aicache**. If you choose to run the installation script to automate the install chores, you can skip to "Production License File" section after running the **install.sh** script. The install script normally takes about one second to complete.

Alternatively, you might elect to perform the required steps manually, if that suits your requirements better. Please refer to README file in the distribution tar file for step-by-step install instructions. For your convenience the same README file is included as an Appendix to this document.

You need to select a location on host server's file system - aiCache installation directory, where to install the aiCache binary. aiCache distribution weighs in well under 1MB in size, so you don't need much space to install it.You might use the same directory or a different one to install aiCache configuration and license files. You can use any name for installation directory, but we recommend using standard name of **/usr/local/aicache**. To simplify matters to also recommend keeping the configuration and license files in the same directory, possibly under **/usr/local/aicache/conf** subdirectory.

We recommend to select a different location (directory) for your log files, for example **/var/log/aicache**, as you will most likely end up with a number of log files and you do not want them to clutter your main installation directory. We also recommend that you configure logging directory on a drive or partition with ample free space, especially for the sites with heady traffic. If you expect to serve 1000+ RPS, please make sure the disk IO performance is adequate to write the log files. You can also use selective log suppression and log decimation features of aiCache to dramatically reduce amount of log information aiCache generates. Unless you need to collect it, consider disabling processing and logging of Referrer and User-Agent information (see logging section later in this document) - this too, saves significant amount of disk space.

For convenience purposes, you can create a *symlink* in aiCache installation directory, for example **/usr/local/aicache/logs** pointing to **/var/log/aicache** - it is rather a common setup. Spend some time setting up automated log file rotation. A common practice is to configure aiCache to rotate it's log files, on the fly, just past every midnite and move the rotated log files to a different location, where they are stored in accordance with your log file retention policies. See the dedicated "Logging" chapter later in this Guide for more information on aiCache access and error logging capabilities.

Production License File.

The aiCache distribution might include a demo-mode license file, called something like **2352352452.demo.** This demo license allows you to test *full* aiCache functionality for 30 days or so. Do not modify the license file in any way as it will render the license invalid.

Unless you're running aiCache as aiCache's own Amazon AWS image or other approved Cloud aiCache images, you will need to obtain a license file before you can use aiCache in *production* mode. To issue a license file we require the output of **ifconfig** command, executed with the name of your production network interface.

For example, when you execute:

```
/sbin/ifconfig eth0
```

you shall see output similar to the one below:

```
eth0 Link encap:Ethernet  HWaddr 00:ee:dd:bb:0c:0e

 inet addr:192.168.168.8  Bcast:192.168.168.255  Mask:255.255.255.0
 inet6 addr: fef0::eee:4ddd:xxx:c0e/64 Scope:Link
 UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
 RX packets:51421 errors:0 dropped:0 overruns:0 frame:0
 TX packets:85637 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
```

To issue a license, we only require HW (MAC) address of your production network interface – in the example above it is contained in the first line of the output. Please note that your HW address will be different from the one shown in example. When sending the HW address (aka MAC address) to us, please send it as it appears in the output of **ifconfig** command, colons and all, preserve the case and do not modify it in any way.

After the license is issued, you shall receive a small file, named something like **1827364490.lic** Feel free to look inside - it is a simple text file. Do not modify the license file in any way as it will render the license invalid.

Copy the file into the directory where you installed aiCache binary and config file - in our case it was **/usr/local/aicache**. To start aiCache you must specify both the name of a configuration file and the license file, for example:

```
cd /usr/local/aicache
./aicache -f acmenews.cfg -l 1827364490.lic
```

aiCache checks the license file upon startup. Should any problems be encountered, you will see an error message. Please refer to "Running aiCache" section for more information on aiCache option, testing configuration file syntax etc.

Feel free to create a shell wrapper script for aiCache startup, so that you don't have to specify all of the parameters every time you start up the server. For example:

```
export LD_LIBRARY_PATH /usr/local/lib
ulimit -c  1000000000
unlimit -n 128000
```

```
/usr/local/aicache/aicache -f myconfig.cfg -l 3245234523454.lic
```

When running Amazon AWS aiCache image or other Cloud aiCache images, license file is not required. All cloud images are pre-enabled for HTTPS and require no action on your part.

The aiCache cloud binary is pre-enabled and pre-installed by aiCache Ltd and requires no license. As with most Cloud images, you will be billed per unit of resource (time or bytes downloaded/uploaded), of actual usage of aiCache. As rates are subject to change, make sure to check with your Cloud Provider to obtain the most up to date rates, terms and conditions.

Configuring aiCache: look Ma, no XML !

Configuring aiCache requires editing a single, simple text-based configuration file with an editor of your choice**.** The configuration file is a good old simple "*name value*" text file, not an XML file.

aiCache is extremely rich with features and it is your responsibility to configure aiCache to benefit your web site the most. You can use included example configuration file as a starting point (template) for you own configuration files. In most cases, to get up and running all you need to specify in the configuration file are a web site name (such as *www.acmenews.com*) and IP addresses of the one of more origin web servers, it is that simple.

Most admins get heartburn just thinking about having to learn about configuration files. Relax, aiCache is very different. Just to illustrate how simple and very short aiCache configuration file could be, here's an example and fairly minimalistic, yet fully functional, aiCache configuration file:

```
server
log_directory /tmp

website
hostname www.acme.com
pattern / simple 10
origin 1.2.3.4
```

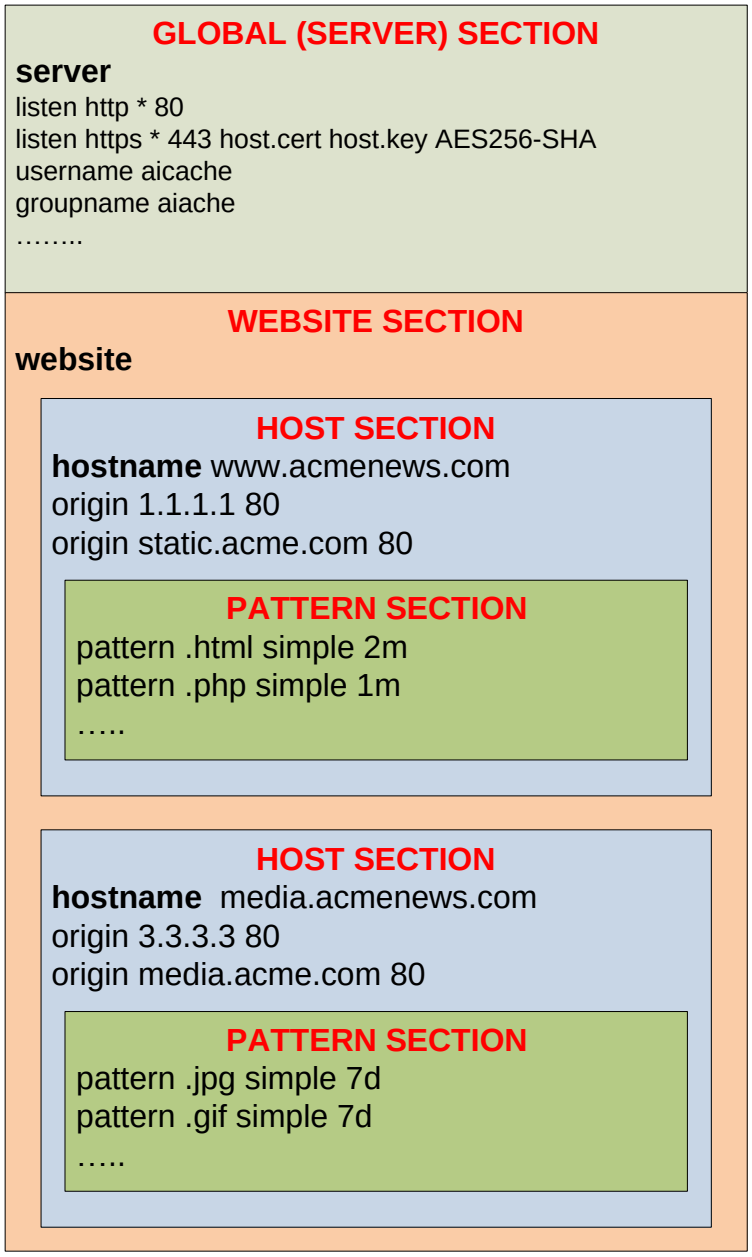
Configuration File Sections.

The configuration directives in configuration file must be grouped in certain order (sectioned). As long as you follow the provided template you'd be just fine - please refer to the picture on the next page to visualize the ordering of the sections.

aiCache configuration file **must contain, in this order:**

- A required single **global** (server) section - it is identified/started by a "**server**" directive. This section contains global configuration settings. This section must be followed by
- A required single **website** section - it is identified/started by a line containing "**website**", followed by
- At least one (required) or more of **host** sections, identified/started by a "**hostname value**" directive. Host section(s) contain website-specific configuration settings. Each host section must specify at least one origin server. It can also contain optional
- **pattern** section containing pattern matching rules

Pattern sections are optional, but since these are how you configure caching rules, most configuration files do contain a number of pattern settings.



You might elect to order website-specific configuration settings in some logical order that makes sense to you. For example, keep origin server directives together, do same for patterns. You might decide to keep origin server directives at the end of each host section or in the beginning, following the **hostname** directive - it is up to you.

However, a place where ordering of configuration settings is important is the pattern section where you specify URL matching patterns. aiCache tries to match request URLs to these patterns in the order that they are found in configuration files. When the first match is obtained, this matching process stops and appropriate action is taken. It means that you might want to place more common patterns toward the top of the pattern list.

Likewise, put the least common pattern toward the bottom of the pattern list. While not a must, this tactic is likely to improve performance. Exclusionary, *everything-but* patterns, should be placed *before* the patterns they are intended to override.

Similarly, if you accelerate a number of busy websites on a aiCache server, place definition of the most frequently accessed sites toward the top of the configuration file.

If you desire to break up your aiCache configuration into a number of smaller configuration files, instead of having one large configuration file, you can do so via one or more of **include file_name** directives. These directives can be placed anywhere within configuration file. Upon encountering an **include** directive, aiCache opens the file that it references and continues parsing process with the content of included file. The included file can, in turn, provide a number of **include** directives of its own.

You might use **include** directive to break up configuration so that configuration of each accelerated website is stored in its own configuration file or, perhaps, store pattern sections in their own files etc. aiCache allows for nesting of **include** directives up to 10 levels deep.

When running aiCache in clustered setup, you'd typically want to have each clustered aiCache server have same configuration file. This too, can be accomplished with **include** directive - simply put settings that makes each aiCache server unique, into a separate file and then use **include** directive pointing to that file. One setting that is unique to each clustered aiCache server is the list of peers (on each node, it excludes node's own IP).

Configuration line format.

Each *configuration* line in aiCache configuration file must be in the following format:

```
parameter_name[space]param_value1[space]param_value2[space][#comment]…"
```

Most parameters take a single value, some are just *flags* and require no value. A few take up to 7 values. Parameter *names* must be in lower case.

Some configuration settings have reasonable default values and if you deem these to be right for your setup, you don't have to specify such settings in the configuration file. Please see the configuration settings table below for what parameters have default values.

Empty lines are allowed in the configuration files, as are *dedicated comment* lines If you need a *dedicated comment* line, start it with a "#" (pound) symbol. You can also *add* a comment to configuration setting line - add such comment after last "#" symbol. For example:

```
# comment line all by itself, preceded by and  followed by an empty line
```

```
max_os_ka_reqs 20    # comment added to a configuration setting.
```

Please note that there might be cases where you want to have # (hash symbol) considered as a valid part of a configuration setting (match or rewrite pattern etc), as opposed to having it treated the usual way – as start of a comment.

To accomplish that, terminate the configuration line with an additional "#" (hash sign), followed by optional comment. This way all of the "#" symbols in the configuration line, except the last one, will be treated as valid part of respective configuration setting. In other words, should multiple # symbols be discovered in configuration line, aiCache will treat the ***last*** **#** as start of comment, all previous **#** symbols (if any) will be properly treated as parts of respective settings.

For example:

```
...

rewrite /topnews.html /news.asp#top
# The line above won't work, aiCache will treat #top as comment
# and it will be stripped out

rewrite /topnews.html /news.asp#top  # Rewrite to #top
# The line above is proper way to do it, comment starts with "last" hash on the line
# and #top is properly included into the rewrite pattern
```

Simple, exact and regular expression patterns.

To configure the caching time (aka TTL or time-to-live) for different URLs on your site, you can use "**exact**", "**simple**" or "**regular-expression**" pattern types. While *regexp* patterns are much more powerful in their matching abilities, in most cases *simple* patterns are all you need.

Exact pattern is a string that is tested for **exact** match against request's URL . For example:

```
pattern / exact 10m
```

matches URL that consists of "/" and nothing else. You won't be using exact patterns too frequently as they are very limited in their matching ability. You'd most likely use these as a way to catch a few very specific URLs that are hard or too expensive to accommodate for with other types of patterns - most noticeably the "/".

Simple pattern is a string that will be looked for in incoming URLs and if that string is present *anywhere* within the request's URL, a match is declared. In other words, a partial match is performed with URL serving as a "haystack" and simple pattern being a "needle".

For example:

```
    pattern breakingnews.html simple 10m
    pattern /static_content/ simple 7d
```

declares all URLs that contain **breakingnews.html** to be cacheable and assigns TTL of 10m. It also specified that responses to any requests with URL containing **/static_content/** are to be cached for 7 days. The latter pattern would apply to URLs like **/static_content/images/1x1.gif , /static_content/css/mainstyle.css, /static_content/js/menu.js** and so on. So if static content on your site uses this or similar type of naming convention with a common prefix to URLs, you can match most of it via a single **pattern** directive.

Due to partial nature of matching of simple patterns, be aware that a *simple* pattern "*acme*" matches *acmenews.html, acme.php, notacme.asp*.

Regexp patterns (regular expressions) are used when you cannot accomplish what you need with *simple* or *exact* patterns.

There is a wealth of information freely available on the Web about regular expressions and you can study some of it. In most cases you will know enough about these *regexps* by simply looking into the example configuration files.

A regular expression pattern can be anchored to the beginning of line by using caret sign: **^aiCache** will match anything that starts with **"aiCache".** A dollar sign anchors pattern to the end of line: **tor$** matches any string that ends with **"tor".**

You can see that some symbols have special meaning in regexps. It means also that to match such symbols literally, one has to escape them. For example, to match period sign ".", followed by **"js"** we would use **\.js.** Pretty simple stuff, yet it is much more powerful than that as you will discover if you do read more on the subject.

A frequently asked question is how to make a regular expression pattern case insensitive. To do that, simply prefix the (required part of) pattern with **(?i)**. As usual, you can use aiCache's simple "pattest" tool to test your patterns:

```
$pattest -p '(?i)aaa' -s AAA
[Success]: Matched.
 Pattern: >(?i)aaa<
 String:  >AAA<

$pattest -p 'aaa' -s AAA
[Failed]: Not matched.
 Pattern: >aaa<
 String:  >AAA<
```

Please note that you cannot use just "/" as a pattern (regexp type or simple type) - as it is likely to match almost all of the URLs on your site - probably, not what you want to happen. For much more information on the PCRE syntax, google for it or visit this link: http://library.gnome.org/devel/glib/stable/glib-regex-syntax.html

Pattern testing tool.

To ease creation and testing of regular expression match and rewrite patterns, aiCache distribution includes a small command line tool: **pattest**. The installation script places it into the same directory as aiCache binary, but you can move it as you see fit - for example, you can move it to **/usr/local/bin**, a commonly used directory for assorted binaries.

The tool allows both regexp *match* testing and more complicated regexp *rewrite* testing. Right at the command line you specify a pattern to use for matching, a string to match the pattern against and optional rewrite string. We advise use of single quotes to enclose all of parameters, to prevent possible interpretation and substitution/expansion of parameters by Linux shell. Here are some quick examples:

```
# ./pattest -p '^abc' -s abcdef
[Success]: Matched.
 Pattern: >^abc<
 String:  >abcdef<

# ./pattest -p '^abc' -s zabcdef
[Failed]: Not matched.
 Pattern: >^abc<
 String:  >zabcdef<

# ./pattest -p '(abc).*' -s abced -r 'ZZZ'
[Success]: Rewrote original string:
>abced<
 To new string:
>ZZZ<
 Using pattern:
>(abc).*<
 And replacement string:
>ZZZ<

# ./pattest -p '(abc).*' -s abced -r 'ZZZ=\1'
[Success]: Rewrote original string:
>abced<
 To new string:
>ZZZ=abc<
 Using pattern:
>(abc).*<
 And replacement string:
>ZZZ=\1<
```

URL match actions.

When a request URL matches a pattern, one or more of the following actions are taken:

- URL TTL might be assigned (including TTL value of 0, declaring request non-cacheable)
- URL might be blocked

- URL might be redirected
- URL might be rewritten
- URL might be rewritten and redirected
- URL might have "do not log" flag set (log suppression)
- URL might have some/all of its parameters discarded (URL caching signature parameter busting)
- URL might have a cookie and its value added as part of URL's cache signature
- URL might have request's User-Agent value added as part of URL's cache signature
- URL might have request's User-Agent rewritten and added as part of URL's cache signature
- URL might have request's Accept-Language value added as part of URL's cache signature
- URL might have its TTL set to 0 if a certain cookie is present in the request
- URL might allow for one or more cookies from origin server response to be cached
- Configured customer-provided plugin code can be executed

When no match is found for request's URL, the request is declared to be non-cacheable. In other words, aiCache never caches a response unless it is configured to.

Please note that the action list above is not a complete list, as we continuously enhance the product with more features.

Example configuration file.

To reiterate, you configure all of the aspects of aiCache behavior via single, simple, text based, non-XML *name-value* configuration file. You can name it to your liking, but the whatever name you choose must be passed to aiCache in the command line (-f parameter). An example configuration file, called **example.cfg,** is provided in distribution and you can use it as a starting point.

The configuration settings are very straightforward and very few of them are required in most cases, resulting in configuration files that contains just few lines of settings. However, aiCache prides itself in an incredible flexibility and this is why there're so many available options.

Should aiCache encounter a setting/directive it doesn't understand/support, an error message is printed out to the console and/or error log file, pin-pointing the offending configuration line, so you can correct it. When such warning is printed out, you either have provided a valid setting name but didn't specify proper number of required arguments or you have specified an invalid setting name.

Alternatively, you can start aiCache with "**-t**" option, to test your configuration file line-by-line. For more details, please see "Starting aiCache" section.

If you desire to break up your aiCache configuration into a number of smaller configuration files, instead of having one large configuration file, you can do so via one or more of **include file_name** directives. These directives can be placed anywhere within configuration file. Upon encountering an **include** directive, aiCache opens the file that it references and continues parsing process with the content of included file. The included file can, in turn, provide a number of **include** directives of its own.

You might use **include** directive to break up configuration so that configuration of each accelerated website is stored in its own configuration file or, perhaps, store pattern sections in their own files etc. aiCache allows pretty nesting of **include** directives up to 10 levels deep.

When running aiCache in clustered setup, you'd typically want to have each clustered aiCache server have same configuration file. This too, can be accomplished with **include** directive - simply put settings that makes each aiCache server unique, into a separate file and then use **include** directive pointing to that file. One setting that is unique to each clustered aiCache server is the list of peers (on each node, it excludes node's own IP).

Just to illustrate how simple and very short aiCache configuration file could be, here's an example and fairly minimalistic, yet fully functional, aiCache configuration file:

```
server
log_directory /tmp

website
hostname www.acme.com
pattern / simple 10
origin 1.2.3.4
```

As you can see, barely a handful of configuration file lines are required to have a working setup. In the example above, we define log directory of **"/tmp**" and we specify a single pattern that declares that all of the URLs are cacheable for 10 seconds. It also declares a single origin server. You can also see that most of the configuration settings have rather obvious, easy to remember and self-explanatory names.

Let's now take a look at a more evolved example configuration file. Just for illustration purposes, the example below contains a larger number of configuration settings. Once again, most configuration files will be much simpler than the example below.

```
############################  Server Section ########################
server # REQUIRED
#
listen http * 80
listen https * 443 host.cert host.key AES256-SHA:RC4-MD5 server_port 80
if_name eth0 # REQUIRED
server_name aiCache5x
admin_ip *
admin_port 111
admin_password secret
username test
groupname test
work_dir /usr/local/aicache
alert_dir /usr/local/aicache/alerts
logdirectory /var/log/aicache # REQUIRED
pid_file /var/run/acme.aicache.pid
log_healthcheck
logtype extended
accesslog access
dump_bad_req
debug_clip 1.2.3.4
stay_up_on_write_error
errorlog error
drop_user_agent
drop_refferer
```

hdr_clip X-Forwarded-For

refresh_hdr_clip

max_header_size 2000

max_body_size 50000

chunked_resp_size_hint 33000

maxkeepalivereq 10

maxkeepalivetime 10

maxclientidletime 4

max_os_resp_time 4

snmp_stat_interval 4

max_os_ka_connections 2

max_os_ka_req 4

max_os_ka_time 10

max_post_sig_size 128

min_gzip_size 4000

compress_json

compress_xml

max_sig_size 512

max_post_sig_size 128

client_linger 0

os_linger 0

silent_400

cache_cleaner_interval 320

hard_cache_cleaner_interval 7200

max_log_file_size 100000

logging dedicated

shutdown_grace_wait 5

refresh_website

logstats

stat_url aiCachestat

keep_http10 on

table_stat_url aiCachetable

pend_url aiCachepend

alert_email support@a.b.c

alert_bad_req_sec 20

alert_max_cache_entries 10000

alert_client_conn_max 30000

alert_client_conn_min 5

alert_os_conn_max 20

alert_os_fails_sec 2

alert_req_sec_max 2500

alert_req_sec_min 10

alert_os_rt 2000

x_os_header X-aiCache-OS

x_ttl_header X-aiCache-TTL

peer 1.2.3.4

peer 1.2.3.5

peer 1.2.3.6

peer_prefix mysecretpeerprefix

##########################Website Section ##############################

```
website # REQUIRED
hostname www.acmenews.com # REQUIRED
cname bbs.acmenews.com
cname mobile.acmenews.com
wildcard cms
sub_hostname content.acme.com
no_retry
404_redirect_location http://acmenews.com/404grace.html
 500_redirect_location http://acmenews.com/500grace.html
prefetch_url  /slowGetAdCall?page=home&area=top 100 gzip
prefetch_http_header  User-Agent TestUser Agent V1.0
prefetch_conn_close
min_gzip_size 4000
min_gzip_size 4000
compress_json
compress_xml
ignore_case
ignore_ims
sig_ua
ua_sig_rewr Browser111 b1
fallback
forward_clip
logstats
decimate_log 100
decimate_bad_log 1000
```

```
    conn_close
    max_os_resp_time 4
    max_url_length 128
    ignore_no_cache
    sig_ignore_body
    max_os_ka_connections 2
   max_os_ka_req  4
   max_os_ka_time  10
    healthcheck URL HTTP 10 10
    alert_email
    alert_max_cache_entries 53451
    alert_max_bad_os 3
    alert_os_fails_sec 4
    alert_req_sec_max 3434
    alert_req_sec_min 3000
    alert_os_rt 5000
############################## origin Servers ####################
# At least one must be specified for each defined hostname #
    origin 127.0.0.1 8888 1
    origin 127.0.0.1 8889 2
    origin static.acme.com 80
############################## Patterns ##########################
    pattern .html simple 120  ignore_query
    os_tag 3
```

label Pattern matching all of the .html files

param_partial_match

redirect_location http://promo.acme.com

0ttl_cookie customer_id

sig_cookie connection_type

rewrite images /media/images

rewrite_http_only

max_os_resp_time 4

sub_hostname media.acme.com

max_url_length 128

no_retry

post_block

ignoreparam param1

pattern forumdisplay regexp -120

conn_close

block

drop

request_type both

ignore_case

pattern / exact 1h

sig_ignore_body

dump_req_resp

Server/global section.

We start with "*server*", a.k.a *"global"* settings section – such settings are applicable to the *whole* server, as opposed to the *per-website* parameter section. Some of the parameters defined in the global section are *inherited* by website settings but may be *overridden* (set to different values) in the website sections. Website-level setting always supersedes global-level setting. Likewise, some global and/or website-level settings may be *overridden* (set to different values) in the pattern sections. Pattern-level setting always supersedes both global-level and website-level setting.

The first configuration setting in aiCache configuration file must be *server* directive. You can begin config file with empty or comment lines, but the *first directive* must be "*server*".

Global (system-wide) settings.

server	Starts server (aka global) section of the configuration file. Required. Parameters: none. Default: none
if_name eth0	Specifies a valid interface name that aiCache will run on. Required. Parameters: interface name. Default: none
listen http * 80	IP address, port number and protocol that server listens on. You can specify as many of these as your setup requires. HTTPS port require additional parameters - see dedicated "Listen Ports" section. Default: any IP address defined on the server, port 80.
admin_ip * **admin_port 2233** **admin_password secret**	IP address, port number and password for CLI (Admin) server. Default: any IP address defined on the server, port 2233, "secret"
username aicache **groupname aicache**	Identity to assume after startup. Server will assume the user and group ID that are provided, to limit its privilege level. Default: none, aiCache runs as superuser (root) - not recommended. Make sure specified user and group exist in /etc/passwd and /etc/group files.
num_files 128000	As an alternative to running a "ulimit -n NNNN" command *before* starting aiCache, you can let aiCache set it its own limit on max number of open files. Default: not set . aiCache warns if you set this number lower than 2000 .
work_dir /usr/local/aicache	Specifies aiCache working directory that aiCache chdir()s to after startup. Default: whatever directory aicache binary is started from.
alert_dir /usr/local/aicache/alerts	Specifies directory that aiCache uses to spool alerts files to (see

	section on Alerting). Default: /usr/local/aicache/alerts
pid_file /var/run/aicache.pid	When set, aiCache will write it's PID to the specified file, upon startup. Optional. Make sure aiCache user can write to this file.
cfg_version 03102010-001	When set, aiCache will report this in both Web and CLI statistics screen, to assist in ascertaining what version of configuration file is presently in effect.
logdirectory /var/log/www	Specifies directory where log files reside Required. Default: none . More Information here.
stay_up_on_write_error	When set, server will continue to service requests even when it runs out of space in log partition (directory). Default: aiCache terminates on write errors .
logtype extended	Specifies access log format. Choose between “extended” and “apache” Default: extended More Information here.
accesslog access	Name of access log file. Default: access
errorlog error	Name of error log file. Default: error
log_healthcheck	When specified, health check requests will be logged in access log file. Default: no health check logging
logging dedicated	When specified, each accelerated website logs into its own, dedicated access log file, named access-hostname. Default: all of the website log into a single shared access file. More Information here.
drop_referrer	When specified, aiCache will not process and/or log Referrer header value. We recommend having drop_referrer configured - it will reduce the size of the access log files rather significantly. If you need to collect Referrer field for log crunching, do not specify this setting. Default: Referrer is collected and logged.
drop_user_agent	When specified, aiCache will not process and/or log User-Agent header value. We recommend having it turned on - it will reduce the size of the access log files very significantly. Default: User-Agent is collected and logged
dump_bad_req	When specified, aiCache writes bodies of bad requests to files named /tmp/bad_req.PID.NNNN. No more that 120 bad request bodies can be written out per minutes to avoid resource over-utilization. Can be used to troubleshoot bad requests. Default: bad request bodies are not written out.
logstats	When specified, global statistics will be written out to a dedicated

	statistic log file every 5 seconds. Default: no statistics logging
debug_clip 1.2.3.4	When set, aiCache captures and writes out both requests and responses (up to 32KB of) with matching client IPs - after they are sent to/received from origin server. Can be a helpful troubleshooting tool.
maxclientidletime	When specified, a valid request header must be obtained from client within this many seconds. Default: 5 seconds
maxclientbodytime	When specified, a valid and complete request must be obtained from client within this many seconds. Default: 60 seconds.[4]
max_post_sig_size 128	When specified and/or set to a value and POST caching is enabled, only POST bodies smaller than are used as part of response signature. Default: 256 bytes. See POST caching section for more information
max_log_file_size 100000000	When specified, size-based log file rotation is enabled. Access log file will be auto-rotated when it grows above this size (bytes). Default: no size-based log file rotation
maxkeepalivereq 10	When specified, client Keep-Alive connections are allowed to serve up to this number of requests. Can be set at both global and WS levels, latter overriding former. Default: 20
maxkeepalivetime 10	When specified, client Keep-Alive connections are allowed to stay connected in idle state up to this number of seconds. Really busy web sites commonly set this at 5 seconds. Can be set at both global and WS levels, latter overriding former. Default: 10
max_os_resp_time 4	When specified, responses from origin servers must complete within set amount of seconds. Default: 10.
max_os_ka_connections 2	When specified, enables HTTPS keep-alive connections to origin servers and limits number of such connections to the specified number, per origin server. Default: OS KA connections are disabled. Global setting can be overidden at website level.
max_os_ka_req 20	When specified, keep-alive connections to origin servers are limited to serving no more than specified number of requests before being closed. Default: 10 . Global setting can be overidden at website level. Set to 0 to disable keep-alive connections to origin servers.

[4] It is not a good idea to use Accelerator as front-end for large user downloads (say 10MB+) - as it will consume too much RAM. 60 sec default value is more than adequate for a typical form data (few tens of KB at most).

max_os_ka_time 5	When specified, keep alive connections to origin servers are limited in to total duration (from open till final use and close) to no more than this amount of seconds. Default: 5 . Global setting can be overidden at website level.
client_linger 0	When specified, client connection SO_LINGER option is enabled and set to the specified number. Default: client SO_LINGER is disabled. Please make sure you understand the side effects before enabling this option.
os_linger 0	When specified, origin server connection SO_LINGER option is enabled and set to the specified number. Default: origin server SO_LINGER is disabled. Please make sure you understand the side effects before enabling this option. See "Have mercy upon thy origin servers."
client_tcp_no_delay	When specified, TCP_NODELAY option is set for client connections. It has effect of turning off Nagle TCP algorithm. Default: TCP_NODELAY is not applied.
os_tcp_no_delay	When specified, TCP_NODELAY option is set for origin server connections. It has effect of turning off Nagle TCP algorithm. Default: TCP_NODELAY is not applied.
shutdown_grace_wait 5	When executing graceful shutdown, aiCache stops accepting *new* connections for this many seconds, while allowing *existing* requests finish processing, and then exits. Default: 5
refresh_website	When set, for 2nd and following requests that aiCache obtains from a keep-alive client connection, it rematches Host: header to obtain matching website. Used to address broken load balancing setups. Default: off (not specified)
snmp_stat_interval 5	When set, aiCache generates SNMP statistics file every this many seconds. Must be set for SNMP integration to work. Default: off, SNMP integration is disabled . SNMP Monitoring.
silent_400	When set, aiCache does not send error responses in response to invalid/malformed/oversized requests. Instead it drops such client connections silently and instantly. Default: off, error responses are sent in response to malformed/oversized requests.
orig_err_resp	Specifies how aiCache treats error status codes it receives from origin servers in response to cacheable requests. When set, original responses are delivered, when not set (default), aiCache overwrites all error responses with status > 400, except 401,404 and 407, with its own,

	abbreviated versions. This is done to preserve network bandwidth.
stat_url aicachestat **table_stat_url aicachestattable**	These are URLs you use to get access to aiCache's self-refreshing web pages that contains aiCache global or website-specific statistics. You may set it so that only people that know it can see your statistics. See "Self-refreshing Web monitor" section for more information on this feature. Default: accelstattext, accelstattable Self-refreshing Web Monitor
pend_url aicachepend	This is URL you use to get access to aiCache's self-refreshing web page that contains aiCache global or website-specific pending requests. You may set it so that only people that know it can see your statistics. See "Self-refreshing Web monitor" section for more information on this feature. Default: accelpendtext Self-refreshing Web Monitor
alert_email support@a.b.c	When set, automatic global alerts are generated and sent to this email address . Default: not provided, no global alerts are generated. Please note that for alerting to work, you must enable statistics logging via logstats directive. Websites can have their own alerting set, acting on different set of criteria and emailing to different email addresses (see website section). Please see a dedicated chapter on automated email alerting feature of aiCache.
alert_bad_req_sec 20	When set, an alert is generated when number of bad RPS exceeds this number . Default: not set. Please see a dedicated chapter on automated email alerting feature of aiCache.
alert_max_cache_entries 20000	When set, an alert is generated when number of cached responses exceeds this number . Default: not set
alert_client_conn_max NNN	When set, an alert is generated when total number of client connections exceeds this number . Default: not set
alert_client_conn_min NNN	When set, an alert is generated when total number of client connections is less than this number . Default: not set
alert_os_conn_max NNN	When set, an alert is generated when total number of origin server connections exceeds this number . Default: not set
alert_bad_resp_sec NNN	When set, an alert is generated when total number of failed responses from origin servers exceeds this number . Default: not set
alert_req_sec_max NNN	When set, an alert is generated when number of RPS exceeds this number . Default: not set
alert_req_sec_min NNN	When set, an alert is generated when number of RPS is less than this

	number . Default: not set
alert_os_rt NNN	When set, an alert is generated when origin server response time is more than this number (milliseconds) . Default: not set
alert_humane	When specified, no alerts are generated between midnight and 7am local time.
alert_exclude_pat	When specified, current date/time is compared to the specified patterns and if match is found, no alerts are generated.
mail_path (obsolete!!!)	Obsolete! Used for email alerting. Points to mail binary. Default: /usr/bin/mail
chunked_resp_size_hint 33000	When receiving origin server responses with Transfer-Encoding: Chunked, aiCache sets receiving buffer to this number of bytes. It is done to improve performance at cost of possible memory over-allocation. Default: ~33 000 B (32KB)
x_ttl_header X-aiCache-TTL	When set, aiCache returns response's TTL to clients via custom HTTP header that is specified as the parameter (X-aiCache-TTL in this example). Value of 0 is returned for non-cacheable responses. Default: not set, response's TTL is not inserted into the response header.
x_os_header X-aiCache-OS	When set, aiCache returns origin server IP and port number to clients via custom HTTP header that is specified as the parameter (X-aiCache-OS in this example). Default: not set, origin server IP and port number are not returned as part of response header.
peer 1.2.3.4 **peer .....** **peer .....**	Identifies a peer - another aiCache server that accelerates same web site. Default: not set, no peers are defined. For more information about peers please read Clustering aiCache.
peer_prefix mysecretpeerprefix	Peer command prefix. For more information about peers please read Clustering aiCache. Default: xaicachepeer .
cache_cleaner_interval N **hard_cache_cleaner_interval M**	Set cleaner run intervals, in seconds. See chapter on aiCache Cache Cleaner Logic.
keep_http10 [on|off]	Set it to enable/disable aiCache from overwriting HTTP/1.0 requests to HTTP/1.1. When set to "off", aiCache rewrites the minor version of HTTP requests - which might speed up processing of responses from

	origin servers. Default: set, HTTP/1.0 is not rewritten as HTTP/1.1
hdr_clip X-Forwarded-For	When set, aiCache will parses forwarded Client IP from request's HTTP header and log it in the access log file . See dedicated section on the subject. Default: not set, no processing takes place.
refresh_hdr_clip	When set, aiCache re-obtains (re-parses-out) forwarded Client IP for each request within a Keep-Alive connection. See dedicated section on parsing of forwarded Client IP for more information. Default: not set, assuming aiCache is configured to process forwarded Client IPs, it will only do so for first request within a Keep-Alive client connection and reuse it when logging subsequent requests within the same Keep-Alive client connection.
enable_http10_gzip [on|off] **enable_http10_keepalive**	These allow to override default treatment of HTTP/1.0 request. Enabling either one is risky. See dedicated chapter on HTTP/1.0 requests.
fallback_4xx	When set, aiCache tries to serve stale cached response, if one is available, upon receiving response with 4xx error code from Origin Server. Default: 4xx returned back, instead of falling back to previously cached response.
count_4xx_as_bad	When set, aiCache counts 4xx responses from origin servers as bad (same way as 5xx responses are counted). This way you can observe/monitor and alert on 4xx responses. Default: 4xx responses are nod added to bad response stats.
disable_host_normalization	When set, aiCache preserves the original reques't Host header case and then performs case-sensitive match to defined **hostname**s, **cname**s and **wildcard**s. Default: host matching is case-insensitive.
debug_request_cookie	When set and aiCache running with "-D" command line option, diagnostic messages are printed out as aiCache parses out request cookies. Use it when troubleshooting cookie-related issues. Default: off.
debug_response_cookie	When set and aiCache running with "-D" command line option, diagnostic messages are printed out as aiCache parses out response cookies. Use it when troubleshooting cookie-related issues. Default: off.
debug_ws_match	When set , aiCache logs request's Host header value when it cannot be matched to any of the defined websites. Default: off.
max_sig_size	When request's cache signature (minus possible sig_cookie and sig_ua) exceeds this value, request is declared non-cacheable. Default: 1024 characters.

decimate_log 100	When set, aiCache decimates request logging by the specified factor (for example, when set to 100, every 100th request is logged). Default: log decimation is disabled, every request is logged
decimate_bad_log 100	When set, aiCache decimates request logging by the specified factor (for example, when set to 100, every 100th request is logged) – for response codes of 400 and higher. Default: log decimation is disabled, every request is logged

Website-specific settings.

We continue with "*website" settings.* Such settings are applicable to a particular website, as opposed to the *global* settings. Some of the parameters defined in the *website* section might override the *global* settings of same name. Some website-level *settings* always, in turn, might be overridden by *pattern*-level settings of same name.

website	Begins website (as opposed to global) section of the configuration file. Required. Default: none
hostname www.acmenews.com	Value of Host: HTTP request header that matches this website. Required, at least one hostname section must provided. Default: none Please note aiCache matches incoming request's Host header to **hostname** in case-insensitive manner.
logstats	When set, per-website statistics is collected and written out to a dedicated file named "stats-*hostname*" in log directory. Default: per-website stats gathering is turned off. Please note that for alerting to work, you must enable statistic logging via logstats directive.
min_gzip_size 4000	When specified, responses with Content-Type of text/*** and applcation/x-javascript and of size equal or larger than the specified number of bytes are compressed-on-the-fly. Default: on-the-fly compression is disabled.
fallback	When set, if a refresh of a cached response fails, the older (previous) cached version, if one is available, is served back to the clients (as opposed to sending of an error response). Default: fallback is disabled; error response is served when refresh fails. See "Forced fallback" for an alternative solution.
httpheader hdr_name hdr_value **httpheader .....** **httpheader .....**	These, in addition to original request line, form request header for cacheable HTTP/1.1 requests. Use example configuration file as guidance for what headers to configure. The values might contain spaces. You can add an arbitrary header to cacheable requests - this can be used to pre-authorize requests when Basic authentication is used, etc. Optional.
httpheader0 hdr_name hdr_value **httpheader0 .....** **httpheader0.....**	These, in addition to original request line, form request header for cacheable HTTP/1.0. requests. Use example configuration file as guidance for what headers to configure. The values might contain spaces. You can add an arbitrary header to cacheable requests - this can be used to pre-authorize requests when Basic authentication is

	used, etc. Optional
healthcheck url match NN MM	When specified, each origin server for this website is sent a health check request for "url", response is matched for "match". Such HC requests are sent every NN seconds and the response must be obtained within MM seconds. In case of error (no response within MM or response data that doesn't match "match" string) origin server is temporarily disabled. Default: origin server health checking is disabled
disable_os_tag_hc	When specified, tagged origin servers are not health-checked. Default: all origin servers health checked when healtcheck_url is defined.
conn_close	When specified, client Keep-Alive is disabled for this website. Default: client Keep-Alive is enabled.
origin 127.0.0.1 8888 1 **origin static.acme.com 80**	Specifies an origin server: DNS name or IP address, optional port number and optional origin server tag. Required. Default: none . At least one origin server must be specified for each accelerated website.
max_log_file_size	Enables size-based log file rotation. Same meaning as in global section, but applied to this particular website. Default: size-based log rotation is disabled.
maxkeepalivereq 10	When specified, client Keep-Alive connections are allowed to serve up to this number of requests. Can be set at both global and WS levels, latter overriding former. Default: 20
decimate_log 100	When set, aiCache decimates request logging for this website by the specified factor (for example, when set to 100, every 100th request is logged). Default: log decimation is disabled, every request is logged
silent_400	When set, aiCache does not send error responses in response to invalid/malformed/oversized requests. Instead it drops such client connections silently and instantly. Default: off, error responses are sent in response to malformed/oversized requests.
decimate_bad_log 100	When set, aiCache decimates request logging for this website by the specified factor (for example, when set to 100, every 100th request is logged) – for response codes of 400 and higher. Default: log decimation is disabled, every request is logged

forward_clip [header name]	When set, aiCache forwards Client IP addresses to origin server, as a request header. Default: CLIP is not forwarded
os_persist	When set, aiCache pins/persists clients to origin servers. See a dedicated section on Client-OS-Persistence. Default: OS persistence is disabled, request are fanned out across all of available origin servers in accordance with configured load balancing metric and os tags. Not recommended for use with DNS-specified origin servers.
no_retry	When set, aiCache doesn't retry failed requests. Default: total of 3 attempts are made to obtain a good response in case of GET requests, POST requests are not retried
cache_auth_req	When set, aiCache allows caching of responses to requests with Authorization headers. In other words, presence of Authorization header in request doesn't reset TTL to 0. Default: such requests/responses are not cacheable, TTL is always set to 0.
sig_cookie	When set, aiCache adds value of HTTP request's matching cookie to the signature of cached response. This can be used to cache and serve different responses, for the same URL, based on a cookie value. You can also set this at pattern level, latter overriding former. Multiple sig_cookie could be specified. Default: cookies are not used in cached response's signature.
sig_header	When set, aiCache adds value of HTTP request's matching header to the signature of cached response. This can be used to cache and serve different responses, for the same URL, based on a header value. You can also set this at pattern level. Default: header values are not used in cached response's signature.
sig_language	When set, aiCache adds value of HTTP request's Accept-Language header to the signature of cached response. This can be used to cache and serve different responses, for the same URL, based on user's language preference. Default: Accept-Language header is not used in cached response's signature.
sig_ua	When set, aiCache adds value of HTTP request's User-Agent header to the signature of cached response. This feature can be used to serve different responses based on browser used by the client. Default: UA header is not used in cached response's signature.
ua_sig_rewr	When set, aiCache rewrites/reduces value of HTTP request's User Agent string and add it to the signature of cached response. This feature can be used to accommodate for mobile clients/devices. Default: UA is not modified/used in cached response's signature.

ignore_case	When set, aiCache makes signatures of cached responses for this website case-insensitive , converting them to lower case . Default: signatures are case sensitive
max_os_resp_time 4	When specified, responses from origin servers must complete within set amount of seconds. Overrides global setting of the same name. Default: 10.
max_url_length NNN	When set, requests with URL length exceeding this limit are declined (with 414 or silently dropped if silent_400 is set) . Default: no limit is emposed.
ignore_no_cache	When specified, X-nocache headers from origin servers are ignored, requests are not retried.
sig_ignore_body	When set, aiCache doesn't append cacheable POST request's body to the request's signature . Applied to all requests to this website.
cname bbs.acmenews.com **cname account.acmenews.com**	Defines one or more of alternative names for this website - as a convenience measure . Do not abuse it - sometimes it is better to create a whole new, different website - as then you'd have better control over it and better view of its traffic. Please note aiCache matches incoming request's Host header to **cname**(s) in case-insensitive manner.
wildcard media **wildcard images**	Defines one or more of wildcard alternative names for this website - as a convenience measure . While **cname**s (above) are matched in exact-match fashion, **wildcards** are matched in partial manner . So **wildcard** of **media** would match media01.acme.com mediafarm.acme.com , server01.media.acme.com and so on. Please note aiCache matches incoming request's Host header to **wildcard**(s) in case-insensitive manner.
cc_obey_ttl	When set, aiCache wont clear out fresh cached responses. Default: CC cleans out fresh cached responsewhen there was no access to it since previous cache cleaner run.
cc_disable	When set, Cache Cleaner logic is disabled for this website. Default: CC logic is enabled, subject to cc_obey_ttl.
cc_inactivity_interval	Cache cleaner can remove cached responses that saw no access in this many seconds. Default: set to same value as cache cleaner interval
send_cc_no_cache	When set, aiCache sends "Cache-Control: no-cache" HTTP header in responses to requests with negative TTLs or when origin server response indicates that response should not be cached.

send_cc_cache *or* **add_cache_control**	When set, aiCache sends "Cache-Control: max-age=NNN" HTTP header in responses to requests with positive TTLs.
404_redirect_location	When set, aiCache redirects to this location when it receives 404 responses from origin servers.
500_redirect_location	When set, aiCache redirects to this location when it receives 500+ responses from origin servers and/or when aiCache generates its own 500 response.
ignore_ims	When set, aiCache disregards conditional If-Modified-Since header in requests . No 304 responses are generated. Default: IMS is processed and 304 responses are generated when appropriate.
max_os_ka_connections 2	When specified, enables keep-alive connections to origin servers and limits number of such connections to the set number, per origin server. Default: OS KA connections are disabled. Overrides global setting .
max_os_ka_req 20	When specified, keep-alive connections to origin servers are limited to serving no more than set number of requests before being closed. Default: 10 . Overrides global setting . Set to 0 to disable keep-alive connections to origin servers
max_os_ka_time 5	When specified, keep alive connections to origin servers are limited in to total duration (from open till final use and close) to no more than this amount of seconds. Default: 5 . Overrides global setting .
prefetch_url /slowGetAdCall?page=home&area=top 100 gzip **prefetch_http_header AAA BBB** **prefetch_conn_close**	Pre-fetch response directive. See a dedicated section on response prefetching/preloading later in this guide.
alert_email support@a.b.c	When set, automatic website-specific alerts are generated and sent to this email address . Default: not set, no website alerts are generated and all the alert setting below have no effect. Please note that for alerting to work, you must enable statistic logging via logstats website-level directive.
alert_max_cache_entries 20000	When set, an alert is generated when number of cached responses exceeds this number . Default: not set
alert_bad_resp_sec	When set, an alert is generated when total number of failed responses from origin servers exceeds this number . Default: not set

alert_max_bad_os	When set, an alert is generated when total number of failed origin servers exceeds this number . Default: set to 0 so even failure of a single OS causes an alert to be generated.
alert_req_sec_max	When set, an alert is generated when number of RPS exceeds this number . Default: not set
alert_req_sec_min	When set, an alert is generated when number of RPS is less than this number . Default: not set
alert_os_rt	When set, an alert is generated when origin server response time is more than this number (milliseconds) . Default: not set
alert_client_conn_max NNN	When set, an alert is generated when total number of client connections exceeds this number . Default: not set
alert_client_conn_min NNN	When set, an alert is generated when total number of client connections is less than this number . Default: not set
alert_os_conn_max NNN	When set, an alert is generated when total number of origin server connections exceeds this number . Default: not set
alert_humane	When specified, no alerts are generated between midnight and 7am local time.
alert_exclude_pat	When specified, current date/time is compared to the specified patterns and if match is found, no alerts are generated.
leastconn	When specified, aiCache uses least connections load balancing metric for this website. Please see a dedicated chapter on available load balancing metric (round robin, weighted distribution and least connections)
enable_https_os_ka	When set, aiCache enables HTTPS OS keep-alive connections. By default HTTPS OS keep-alive connections are disabled, even when **max_os_ka_connections** global setting is set. Please test before enabling in production.
fallback_4xx	When set, aiCache tries to serve stale cached response, if one is available, upon receiving response with 4xx error code from origin servers. Overrides global setting of same name. Default: 4xx response is served to the client, instead of falling back to previously cached response.
count_4xx_as_bad	When set, aiCache counts 4xx responses from origin servers as bad (same way as 5xx responses are counted). This way you can

	observe/monitor and alert on 4xx responses. Overrides global setting of same name. Default: 4xx responses are nod added to bad response stats.
cache_4xx	When specified, 4xx (except 400) responses are cached in accordance with TTL of matching request pattern. Default: 4xx responses are not cached.
ua_keep_pattern Windows/sXP [keep|redirect] **ua_pattern Windows/sXP [keep | redirect]** **.....**	Requests with matching User-Agent string are accepted by aiCache for regular processing, while non-matching one are directed. See section on UA-based redirection for more information. Default: no UA-driven redirection takes place.
ua_ redirect_host	Specifies the host to redirect non-matching UA requests to . Default: no UA-driven redirection takes place. However this setting must be provided if ua_keep_pattern is specified.
ua_keep_cookie [cookie name]	Configures aiCache to issue a cookie to streamline UA-driven redirection processing. Default: cookie name is set to xaikeeperua. Set to "disable" to disable UA cookie processing.
ua_redirect_host_only	Configures aiCache not to send the original URL in the redirection response. Default: original URL is appended.
ua_keep_length	Configures aiCache to *keep* requests whose User-Agent headers are shorter than certain length. To configure this, set **ua_keep_length** value at website level. Default: disabled.
recheck_ua_keep_cookie	See section on UA-based redirection for more information.

Pattern settings.

Within the *website* section is located *the pattern section*, containing definitions of what constitutes cacheable or non-cacheable content. We dedicate a separate table to explanation of the parameters in this section. Some settings in the pattern section may override both global and website-level settings of same name.

pattern *pat* exact|simple|regexp TTL [ignore_query] [no_log]	Identifies new pattern: matches to “pat”, perform "exact", “simple” or "regexp" matching . Time-To-Live (caching time) is set to TTL .

All of the above must be in a single line.	TTL is specified in seconds or, to simplify, you can end the TTL's value with "m" for minutes, "h" for hours or "d" for days. Specify negative TTL to disallow downstream caching. When ignore_query is specified, query parameters are ignored when forming cached response *signature* (see separate explanation later in this document). When no_log is specified, matching responses are not logged in access log file (log suppression). Default: query is not ignored and is used as part of cache signature; requests/responses are logged
ignoreparam param1 **[ignoreparam param2]** **.....**	Identified parameter (named param1) is removed from cached response's signature (see separate explanation later in this document). Multiple ignoreparam lines can be provided, each with a different parameter name. Default: no parameter is removed from request's query string.
param_partial_match	When specified, ignoreparam parameter name matching is partial. Use it with care, but it has potential to speed up matching and removal of ignoreparams. You can also use it to remove parameter that have semi-random names with a common prefix.Default: ingoreparams are matched by exact match.
request_type both	Can be set to "get", "post" or "both". Default: pattern is matched to GET requests only, POST requests are not matched by default.
0ttl_cookie customer_id	When specified, presence of a cookie with the specified name in a request will disable caching of the request, even when pattern match indicates non-0 TTL. More than one can be specified. Default: caching-busting cookies are ignored, TTL is determined by pattern matching alone.
0ttl_cookie_pat SESS.+=	When specified, presence of a matching pattern in request's Cookie header will disable caching of the request, even when pattern match indicates non-0 TTL. More than one can be specified. Default: caching-busting cookie patterns are ignored, TTL is determined by pattern matching alone.
pass_cookie cookiename **[pass_cookie cookiename2]** **.....**	When specified, cookies with the specified names are stored in the cached response (allowed to pass-through from origin server to cached responses). Default: cached responses never contain Set-Cookies, these are always filtered out, even when origin server responses have Set-Cookie

	headers.
redirect_location http://promo.acme.com	When specified, matching requests receive a 302 redirect response, with the specified location provided in Location: header. Default: redirect is disabled
redirect_5xx	When specified, matching requests that result in 5xx error during processing, receive a 302 redirect response, with the specified location provided in Location: header. Overrides website-level error redirects. Default: redirect_5xx is disabled
redirect_4xx	When specified, matching requests that result in 4xx error during processing, receive a 302 redirect response, with the specified location provided in Location: header. Overrides website-level error redirects. Default: redirect_4xx is disabled
os_tag tag_number	When specified, matching requests are to be filled only from origin servers that have matching origin server tag . Default: origin server tagging is disabled, all origin servers are eligible. Tag number must be non-zero. There must be an at least one origin server defined with matching tag, aiCache does not enforce this check. Tag_number must be less than 254.
decimate_os_tag tag_number	When specified, matching requests are to be filled, in decimated fashion, only from origin servers that have matching origin server tag For example, when set to 10, every tenth request will be filled from an OS with matching OS Tag, all other matching requests will be filled from "regular" OS. Default: tagged decimation is disabled.
block	When specified, matching requests receive a 403 Forbidden response. Default: block is disabled
label	A way to assign a descriptive label to pattern. The label will be displayed in the pattern-level statistics display and made available via SNMP. Default: no label
drop	When specified, matching requests are silently and immediately dropped, without any error response being sent to requesting client. Default: drop is disabled
ua_redirect Windows/sCE http://mobile.acmenews.com	User-Agent-based redirection rules. When the regexp pattern matches User-Agent HTTP header, client browser is redirected to http://mobile.acmenews.com The match pattern must be in regular expression format (this is done to accommodate for possible spaces in User Agent strings).
fallback	When specified, matching requests are served stale cached response

	(if one is available) when refresh of a cached response fails. Default: aiCache serves error response when refresh fails. Please see "forced fallback mode" for an alternative way to serve content when origin servers are down.
match_http_only	Pattern won't match unless client connection is HTTP.
match_https_only	Pattern won't match unless client connection is HTTPS.
match_min_url_length	Pattern won't match unless URL length at least this long. **Default: not set.**
match_max_url_length	Pattern won't match unless URL length is smaller than this value. **Default: not set.**
add_body_url_length	Length of request's body, if any, is added to the url length for the purpose of matching to **match_min_url_length** and **match_max_url_length** . **Default: not set, request body is not accounted for in comparing to min and max URL length.**
rewrite from_pattern to_string	When specified, matching requests will have their URLs rewritten and possibly redirected. **from_pattern** is regexp based. **to_string** can use *backreferences* - see dedicated section on URL rewriting and redirection for more information.
rewrite_http_only **rewrite_https_only**	When specified, **rewrite** is performed only for matching connection type, HTTP or HTTPS. Can be used to rewrite from HTTP to HTTPS and vice-versa, without causing a rewrite loop. Alternatively, you can use **match_http_only** or **match_https_only** pattern-level settings (see above).
conn_close	When specified, client connection is closed after serving of matching requests . Use it to reduce number of open connections when you know you serve "one-timer" requests. Default: unless Connection: close is specified in request, connection is re-used in Keep-Alive fashion (more requests/response are allowed to be served over such connections).
retry_min_resp_size **retry_max_resp_size**	When specified, responses to matching patterns are checked to see if response size is less than min or larger than max setting. If it is , aiCache will retry the request up to 3 times trying to obtain a properly-sized response. Should all 3 attempts fail, the failing response is returned as-is, unless a previous , stale, cached response is available - in which case the stale response in served instead. Default: no size checks are enforced on response bodies. See "Forcing Retries" section later in this document for more information on this feature.

no_retry_min_resp_size **no_retry_max_resp_size**	When specified, responses to matching patterns are checked to see if response size is less than min or larger than max setting. If it is , the failing response is returned as-is, without retries, unless a previous , stale, cached response is available - in which case the stale response in served instead. Default: no size checks are enforced on response bodies. See "Forcing Retries" section later in this document for more information on this feature.
ignore_case	When set, aiCache makes signatures of matching cached responses case-insensitive , converting them to lower case . Default: cache signatures are case sensitive
max_os_resp_time 4	When specified, responses from origin servers, matching this request pattern, must complete within set amount of seconds. Overrides global and website setting of the same name. Default: 10.
max_url_length NNN	When set, requests with URL length exceeding this limit are declined (with 414 or silently dropped if silent_400 is set) Default: no limit is emposed.
no_retry	When set, aiCache doesn't attempt to retry failed requests. Default: total of 3 attempts are made to obtain a good response, unless request is of POST type. POST requests are not retried by default.
sig_ignore_body	When set, aiCache doesn't append cacheable POST request's body to the matching request's signature.
sig_language	When set, aiCache adds value of HTTP request's Accept-Language header to the signature of cached response. This can be used to cache and serve different responses, for the same URL, based on user's language preference. You can also set it at website level. Default: Accept-Language header is not used in cached response's signature.
sig_cookie cookie_name	When set, aiCache adds value of matching HTTP request's cookie to the signature of cached response. This can be used to cache and serve different responses, for the same URL, based on a cookie value. You can also set this at webste level, but pattern-level setting takes precedence. Multiple sig_cookie could be specified. Default: cookies are not used in cached response's signature.
post_block	When specified, POST requests that match this pattern are blocked (with 403 response or silently and instantly dropped when silent_400 is set). Default: POST requests are not blocked. Make sure to enable the pattern for POST matching via **request_type both**
post_drop	When specified, POST requests that match this pattern are silently

	and instantly dropped Default: POST requests are not dropped. Make sure to enable the pattern for POST matching via **request_type both**
cache_4xx	When specified, 4xx (except 400) responses are cached in accordance with TTL of matching request pattern. Default: 4xx responses are not cached. Overrides website-level setting of same name.
bad_response	When specified, responses to matching requests are reported as "bad" - which fact is reported via Web, CLI and SNMP interfaces. It can also be alerted on. Default: not set
error_stat_ignore	When specified, 400+ and 500+ code responses to matching requests are not added up to bad response statistics. Use to preclude known bad requests from skewing up error statistics (such as requests for missing favicon adding up to 404 error count). Default: not set
dump_req_resp	When specified, matching request (up to 32KB of) and response (up to 128KB of) are written out to /tmp directory, under /tmp/pattern_req.dump.PID.CONN filename where PP is aiCache process ID and CONN is connection number. Use connection number to reference the request and response to entries in aiCache log file. Useful it for debugging. Default: not set
send_200_resp	When specified, matching requests will be replied to with an empty status-200 response. This is useable for pattern-based alerting, described later in this document.
ua_url_rewrite	When specified, you can rewrite the request's URL based on the value of the *rewritten* UA string. Available only in mobile-enabled version, explained in detail later in this Guide. See **ua_pattern** setting elsewhere in this Guide, for a different method of redirecting requests based on the value of the *original* UA value.
geo_url_rewrite ….	When specified, you can rewrite the request's URL based geo targeting. See Geo Targeting section for more information
disable_gzip	Disable compression for this pattern.
header_pattern name pattern TTL	You can override pattern's regular TTL with a custom value, based on a value of a response header. Default: not set

Listen Ports.

Within the *global* section may be located one or more of optional **listen** directives. These specify over what IP addresses and ports aiCache accepts incoming connections. When no *listen* directives are provided, aiCache attempts to accept connections on any local IP address, port 80 - the standard HTTP server port.

An arbitrary number of listen ports/IP addresses can be specified. If you specify an IP address different from "*", the specified IP address must be configured on aiCache server - for example, as an interface's primary or an alias IP address.

Here's an example configuration specifying 4 listening HTTP ports, from 80 to 83:

```
listen http * 80
listen http * 81
listen http * 82
listen http * 83
```

Normally, you'd configure aiCache with one or more of websites to accelerate. aiCache matches incoming requests to a configured website by comparing request's *Host header* to configured website hostnames. Alternatively, you can tell aiCache to shortcut such matching logic by specifying a website right in the **listen** directive. aiCache must be able to resolve each of thusly specified websites to a valid configured website. For example:

```
listen http * 80 www.acme.com
listen http * 81 images.acme.com
listen http * 82 news.acme.com
```

Instead of using a wildcard IP address of * (asterisk), you can specify particular IP addresses. For example, let's say you have configured 4 different IP addresses on an aiCache server: 1.1.1.1, 1.1.1.2, 1.1.1.3, 1.1.1.4. Exact method to configure a number of different IP addresses on a Linux server is outside of scope of this guide, but you can use *aliases* or simply have a number of different network interfaces. Now you may specify the following:

```
listen http 1.1.1.1 80 www.acme.com
listen http 1.1.1.2 80 images.acme.com
listen http 1.1.1.3 80 news.acme.com
listen http 1.1.1.4 80 news.acme.com
```

To configure aiCache for HTTPS, there's a tad more work to do. We go into details of HTTPS configuration later in this guide, in "Configuring HTTPS" chapter. However, as far as *https listen* directive is concerned, it looks much like the *regular http listen* directive, but it requires a few extra settings:

- Required certificate file name
- Required host's private key file name
- Optional list of ciphers you want to enable

For example, we configure aiCache for 2 HTTPS listen ports, specifying websites right in the *listen* directive:

```
listen https 1.1.1.1 443 acme.cert acme.key AES256-SHA:RC4-MD5 login.acme.com
listen https 1.1.1.2 443 check.cert check.key ALL checkout.acme.com
```

Alternatively:

```
listen https * 443 acme.cert acme.key AES256-SHA:RC4-MD5
```

When you specify non-standard ports for either HTTP or HTTPS listening endpoints, you must have some means of directing traffic to such non-standard ports without inconveniencing the end users. You cannot expect them to go to your site as http://acme.com:8080 or https://logic.acme.com:444. Some form of load balancing HW or SW solution in front of aiCache, can be used for such purpose.

aiCache is capable of serving a very large number of different websites off a single HTTP IP address and listen port tuple. Situation is different with HTTPS, as aiCache ties a website certificate and a private key to each HTTPS listen port. So if you must serve multiple HTTPS websites via a single aiCache instance, you will require a number of IP addresses, one per HTTPS website. Alternatively, you might use wildcard certificates to alleviate some of these restrictions. See dedicated HTTPS chapter for more information.

You can also specify a special **admin** listen port type. For example:

```
listen admin * 5001
listen admin 192.1.1.1 1234
```

The admin type ports only respond to special type requests (described later in this Guide) – such as *webstats*, *peer* requests and such. Admin ports do not match to any defined websites and can do not serve "regular" webpages/traffic. They exist only to serve special traffic and might come in handy in special occasions when you want to create such *special* listen ports that are separate from *regular* ports.

Network performance and scalability considerations.

As you might already know, TCP/IP specification limits number of open network connection for any given endpoint (an IP:port *tuple*) to about 64,000 connections. Should you ever require more than this many *simultaneously open* connections, you must enable additional *listen* end points - specifying a different IP address or port number (usual caveat emptor applies if you want to use a non-standard port number).

On a busy server you might also be required to modify the following OS-level settings:

- Increase system-wide number of max file descriptors
- Reduce TIME_WAIT interval

- Increase number of per-process open file descriptors

Due to aiCache's multiplexed IO architecture, it won't have difficulties maintaining very high numbers of open connections - assuming you have proper amount of RAM in your servers and properly configure the OS to support large number of open connections. So experiment with client keep-alive settings to have manageable number of open connections while delivering maximum benefits to your clients. We have more to say about keep-alive connections later in this Guide.

Likewise, should you find that you're fully saturating a single networking interface with HTTP or HTTPS traffic, you might need to enable an additional network interface. Of course, you'd only find yourself in this situation when you have very large egress capacity to the Internets. For example, if your hosting setup has 5 Gbps of Internet BW available, having 2 aiCache servers, each with a single gigabit interface, you won't be able to fully use up (saturate) your uplink pipes and might need to add additional network interfaces to your aiCache servers or buy additional aiCache servers. Alternatively, you might opt to go with 10Gbps interfaces on your aiCache servers, some form of bonded NIC teaming etc.

CLI Server.

aiCache provides a built-in Command Line Interface (CLI) server. You can simply *telnet* to the CLI port and issue a number of CLI commands. By default, CLI connections are accepted over any local interface (IP address), but you can specify a particular IP address to force aiCache to accept connections only over certain interfaces when more than one is enabled. Typically you would specify internal and/or trusted IP addresses for this service to listen on. You can also bind CLI service to the “loopback” IP address: 127.0.0.1, so that CLI can only be reached from the aiCache's host server itself.

It is prudent to make sure Admin CLI can only be reached from trusted systems – having it open to the whole Internet is probably not a very good idea. However, chances are your existing network setup is such that this restriction is enforced by default.

For more information on CLI please see dedicated CLI section later in this document.

aiCache handling of requests and responses, enforcing timeouts.

aiCache does it's best to protect both the origin servers and the site's visitors. One of the facets of this protection is enforcement of assorted timeouts. Specifically, aiCache expects visitors to provide a valid HTTP request in reasonable (and configurable) amount of time. This way an attacker would not be able to just open a large number of idle HTTP connections to aiCache.

Likewise, aiCache expects origin servers to deliver a valid HTTP response within reasonable (and configurable) amount of time. This way, should origin server fail, aiCache can retry the request against a different origin server or possibly serve a previous version of cached response etc. Otherwise, both aiCache and most importantly, the visitor would just keep waiting on a bad origin server. Outside of inconveniencing the visitor, such buildup of connections would normally have a detrimental impact on overall site infrastructure.

In the way of specifics, when it comes down to client requests, aiCache expects a valid request header to arrive within 5 seconds (configurable via **max_client_header_time** globally and/or at website level). Next, aiCache expects request body (ie a *complete request)* to arrive within 60 seconds (configurable via **max_client_body_time** globally and/or at website level). Both values are overly generous and you should consider reducing them. However, if your site allows clients to upload large files, consider increasing **max_client_body_time** to better suit your needs.

When dealing with responses from origin servers, aiCache expects a complete and valid response to arrive within 20 seconds (configurable via **max_os_resp_time,** globally, at website or pattern-level). This value should work for most sites, but you might consider increasing it in case of slower origin servers. Likewise, when you know that no response should take longer than 2 seconds, you might consider decreasing the value.

aiCache and large requests and/or response.

aiCache does it's best to improve performance of your site by offloading connection handling from your origin servers. One of the ways it goes about it is by completely consuming entire requests before forwarding them to origin servers. Likewise, it completely consumes (buffers in RAM) entire response before forwarding it to clients.

You can see that such RAM buffering might become ineffective in case of oversized requests or responses. Most regular web sites have nothing to worry about, but if you allow large uploads or download, say in excess of 20MB per, you should consider delegating these away from aiCache and onto dedicated subdomains, say **ul.acme.com.** You also should explore deploying some form of BW management for such upload or download domains.

Origin Servers.

Configuring Origin Servers.

aiCache acts as an *intermediary* between users/visitors on the Internet and your actual**,** *a.k.a origin***,** web servers. It is the *origin servers* that aiCache uses to obtain responses from. It is easy to see that aiCache has to know how to get to the origin servers, for each configured website.

Origin servers can be specified via number IP addresses or DNS names. We highly recommend using numeric IPs instead of DNS names. This way you can have more control of just where the origin traffic is sent, eliminate a possible point of failure and remove possible DNS latency. You will also have more control over configuration, for example, changing origin servers won't require modification of DNS data and waiting for the changes to propagate.

When origin servers are configured via DNS names, you must assure aiCache server is properly configured for DNS resolution. Usually it means setting up /etc/resolv.conf file with one or more of valid DNS servers/resolvers.

In our example main configuration file origin servers are configured as follows:

```
origin 127.0.0.1 8888
origin 127.0.0.1 8889
origin 127.0.0.1 8890
origin origin.acme.com 80
```

At least one origin server needs to be specified for each accelerated website. Same origin server can be used in different websites – using same name/IP and port number or different ones. In some cases, you might choose to specify same origin server more than once even for a given website, if you want to have aiCache drive proportionally more requests to it - but consider using *weighted* load balancing metric instead.

When more than one origin server is specified for a web site and unless configured otherwise, aiCache uses these origin servers in a *round-robin* fashion, where each healthy server gets equal share of traffic. In the example above, with 3 origin servers configured, each origin server receives one third of traffic. Other load-balancing (distribution) metrics are available, including *least connections* and *weighted distribution* - we describe them in the next section. Please note that we don't recommend using *least connections* and *weighted distribution* with origin servers that are specified by DNS name.

You can see the number of requests that each origin server has received and other important origin server stats via Web, CLI or SNMP interfaces. If so configured, aiCache can also monitor health (status) of origin web servers .

Load Balancing Metrics: round-robin, least connections and weighted.

Round-Robin.

As you know by now, simple *round-robin* is the default load balancing metric used by aiCache. When so configured, each origin server gets share of traffic proportional to number of times it is specified as OS for a given website. When each OS is only specified once, each OS gets equal share of traffic (1/3 of it, with 3 OS specified as in example above).

This is the metric we suggest you use when all of the origin servers (for a given website) are of about same capacity and configuration and can cope with traffic on equal terms. In some cases, you might choose to specify same origin server more than once for a given website, if you want to have aiCache drive proportionally more requests to it - but you may consider using *weighted* load balancing metric instead.

Please note that aiCache does not support OS tags when configured for least connections or weighted load balancing metrics. To employ OS tagging, you must use default, round-robin load balancing metric.

Least-Connections.

Instead of default *round-robin* load balancing metric, you can configure aiCache for *least connections* metric, where next request is sent to origin server that is processing the fewest number of requests, at the moment of decision making. This metric has potential to automatically load balance requests in a fair fashion, when origin servers are of different capacity, build and configuration. To configure a given website for least connection load balancing metric, specify

```
leastconn
```

in the respective website's section. When configured for least connections, number of outstanding connections to each origin server are shown via Web interface and reported via SNMP.

Please note that we don't recommend using *least connections* and *weighted distribution* with origin servers that are specified by name.

Weighted.

And lastly, you can configure aiCache for *weighted round-robin*, when you manually specify the weight for each origin server. The weight is specified as last numeric parameter in the **origin** configuration line - after server port and server tag. Both server port (even if it is default port of 80) and server tag (even if you want to have default tag of 0) must be specified, if you want to provide weight value - as otherwise aiCache won't know which value is port, which is tag and which is weight.

Let's say you specify 3 origin servers, with weights of 10,20 and 30. It means that out of every 60 requests (10+20+30), first server will get 10 of them, second 20 and third 30. Please note how in the example below we

have specified both the port numbers and the OS Tag of '0' for all three Origin Servers ! Both server port (even if it is default port of 80) and server tag (even if you want to have default tag of 0) must be specified, if you want to provide weight value - as otherwise aiCache won't know which value is port, which is tag and which is weight.

Please note that we don't recommend using *least connections* and *weighted distribution* with origin servers that are specified by DNS name.

```
origin 127.0.0.1 8888 0 10
origin 127.0.0.1 8889 0 20
origin 127.0.0.1 8890 0 30
```

aiCache doesn't enforce any limit on the value of weight, but we suggest to keep it reasonable. aiCache does its best to smooth out the flow of requests to weighted origin servers, but this approach has its limits when significantly different weights are assigned to origin servers.

To reiterate, to setup *weighted* load balancing, all you need to do is to specify weights for each origin server, there's no separate directive you need to provide outside of providing weights.

aiCache complains loudly and exits when a website is configured for both least connection and weighted load balancing, as it is a fairly nonsensical configuration.

Please note that aiCache does not support OS tags when configured for least connections or weighted load balancing metrics. To employ OS tagging, you must use default, round-robin load balancing metric.

aiCache processing of origin servers specified via DNS names.

aiCache has a special thread dedicated to out-of-band DNS resolution of DNS-defined origin servers. The thread runs in the background and periodically resolves OS DNS names to numeric IPs via regular DNS resolution. It is done so that aiCache is not burned with overhead of DNS resolution when it needs to access origin servers – such resolution occurs *out-of-band*.

You configure how frequently aiCache runs such DNS resolution, at per-website basis, via **dns_interval** setting. Specified in seconds, the default value is 120 seconds (2 minutes).

Consider the following example:

```
website
hostname www.acme.com
dns_interval 600
…
origin origin.acme.com 80
```

Upon startup, aiCache resolves all and any DNS-defined origin servers. Assuming **origin.acme.com** has 5 DNS "A" records defined, 5 OS will be defined, used and reported.

Every 600 seconds (10 minutes) aiCache re-runs DNS resolution for this website, attempting to resolve **origin.acme.com** to a list of DNS "A" records. As long as the same 5 "A" records are returned, there are no changes to OS configuration that was established during the startup.

If an additional record is returned, it is added as an additional OS for www.acme.com. It is also reflected in the aiCache error log file.

When a previously-defined "A" record disappears, it is marked as "DNS-disabled" and is not used in OS capacity (no traffic is sent to it). However, the record of that OS is kept by aiCache – so you can see its statistics etc. It is also reflected in the aiCache error log file.

When a previously-disabled "A" record re-appears, it is marked as "DNS-enabled" and is set to again receive its share of traffic. It is also reflected in the aiCache error log file.

aiCache also reports when a website has any DNS-defined OS. Time of last DNS resolution is also logged and reported. All DNS-defined OS are reported likewise.

To assist in initial DNS setup, aiCache offers global **debug_dns** flag setting. When set, copious amounts of DNS debug information are written out to aiCache error log file.

Clearly, for aiCache to be able to perform DNS lookups, DNS resolution must be properly configured on the server (at the operating system level). While explaining such setup is outside of scope of this manual, most of the time all you need to do is to modify */etc/resolv.conf* to point to one or more of DNS servers.

Monitoring Health of origin Servers.

When configured so, aiCache periodically tries to retrieve a *health-check URL* from each origin server. Upon success the content of the response body, if any, is optionally matched against *health-check match string.* Please note that simple *partial* string match is performed, not the regular expression matching. Only when both steps: retrieval of the response in set amount of time *and* content match, are successful, is the origin server declared healthy to serve content.

If either action fails, the origin server is declared unfit to serve content and is not used by aiCache until it passes a following health check. Origin server failing a health check and passing it after a failure are logged in error log file, complete with time stamp and origin server's IP and port number. You can also configure aiCache to alert via email whenever an origin server fails a health check - see "Alerting" section for more information.

Each website can be configured with its own health check URL and match string by placing health check directive under the respective hostname directive. For example:

```
website
hostname www.acmenews.com
healthcheck  /servertest.aspx  good  30  10
```

In this example aiCache is configured to retrieve **/servertest.aspx** URL from each origin server defined for www.acmenews.com, every 30 seconds. The returned content is matched against "good". Origin servers are

allowed a fairly generous 10 seconds to respond to the query. Should an origin server fail to return matching content in the allowed amount of time, it is declared down - which fact is logged in the *error* log file. 30 seconds later, the same sequence of steps is attempted again. Should a failed origin server pass the check this time, it is declared healthy (which fact is logged in the *error* log file) and put back into origin server "rotation" and is allowed to serve content again .

The URL you specify for health check and the match string are important. Ideally, you'd choose a URL that executes some logic that tests that all of the components in the chain: the origin web server, the application server(s), if any, and the backend (databases and similar), if any, are operational and healthy. The URL could be an actual production, public URL that is normally accessed by visitors or custom coded logic dedicated to health checking only. For example, such health check logic could connect to a DB server, execute a query and produce output containing "good" when all of the steps complete successfully.

The whole response body is matched to see if it contains the specified string anywhere within it. If you don't want to match the response body to any string, specify the match string as "HTTP" or "NULL" and aiCache won't perform any matching. This is a poor choice however, and it is much better to have a more meaningful testing and matching as described above.

When executing a health check request, aiCache generates a HTTP/1.1 request - complete with *Host* header set to website's hostname. It means that origin servers must be able/configured to resolve that hostname to a valid configured website.

aiCache reports the number of health checks that each origin server has passed and failed, via all 3 interfaces: Web, CLI and SNMP. Review this statistic as it might reveal misconfigured and/or underperforming servers - these will normally exhibit higher failed health check counts.

By default, health checking is disabled, but we strongly recommend enabling this feature. This way, aiCache detects origin server failures and attempts to shield client requests from adverse effects caused by failed servers.

For example, let’s imagine an origin server that is not responding and is timing out requests. When aiCache knows about such malfunctioning origin server as a result of it failing a health check, it avoids sending requests to it and can do one of the following instead:

- send client requests to surviving origin servers
- send request to origin server of last resort
- serve stale cached content

On the other hand, if aiCache has OS health checks disabled and cannot detect OS failure, it may send client requests to a dead server and then have to enforce timeout on the response time and only after such timeout, take a corrective action. In the ensuing seconds of waiting for response, the client just might abandon the request and go to a different website.

You might encounter a situation when you use *tagged* OS and discover that the regular health check URL doesn’t work for these – for example, resulting in 404 responses and such. To rectify, you can modify the HC match string to something that is always present in any HTTP response (for example, “HTTP”) or disable HC checking for tagged OS via **disable_os_tag_hc** website-level setting. It is a flag and requires no value.

Monitoring Health of aiCache Servers.

If you need to monitor aiCache servers themselves, for purposes of alerting or you need to configure health checks to "ping" aiCache servers from load balancers and such, you must make sure the health check requests that aiCache receives from these probing agents have a valid Host HTTP header, matching one of the websites that aiCache is configured to accelerate.

In general, aiCache might be configured to accelerate a number of websites: say a.com, b.com and c.com. At any given point in time aiCache might be unable to serve some of these sites, due to failure of all origin servers, while continuing to serve other sites just fine. This is why it is important to make sure any external health checks against aiCache carry a proper Host header - to discern what sites are still available and what are not.

In addition to such HTTP probing of aiCache servers, we also advise to use additional monitoring methods to periodically probe the usual set of OS-level tests: CPU utilization, disk space, swapping etc.

aiCache exposes an extremely rich set of statistics via SNMP and we most strongly advise to collect some of the aiCache SNMP counters as well. aiCache SNMP integration is described later in this Guide.

Cacheable vs. non-cacheable content, why very large TTLs are not always a good thing and auxiliary content versioning.

As far as aiCache is concerned, content can be divided into 2 broad categories: **cacheable** and **non-cacheable**[5]. It also helps to think of *cacheable* content as one that can be *shared* and *non-cacheable* content as one that *should never be shared or offers no merits when cached*.[6]

Certain content on your web site might change frequently, as often as once every 5 second or once a minute. Other content does not change that frequently. Some changes once every hour, some change once every day and so forth.

We must configure aiCache to obey our "document freshness" rules to maximize the benefits of caching: improving overall performance including performance as perceived by visitors, reducing load against the site's infrastructure and, ultimately, allowing for reduction of footprint and significant monetary savings .

In our www.acmenews.com example most content on the site is cacheable and we can reasonably expect to have cache hit ratios as high as 95%+, depending on the traffic pattern.

---

[5] We also call it 0-TTL content (zero-TTL).

[6] With some clever tricks you can configure per-user caching, but doing so will have rather limited benefits.

For example, if 100 RPS come in for the home page and we cache it for 20 seconds, we will serve 20*100 - 1 = 1999 requests out of cache and only 1 from origin. As a result we would have reduced the traffic to the origin server by ... drum roll ... nearly 2000 times, for this particular URL ! The resulting *cache hit ratio* is, for all practical purposes, cool 100%. Of course, if we only get 10 RPS for that URL, then reduction will "only" be a factor of 200, but still the same near 100% . And we do have customers with caching ratios in excess of 99%. Clearly, if most content can be cached in similar fashion, we can reduce the load on origin infrastructure close to zero RPS!

As aiCache, by default, encourages *downstream* caching for the same amount of time as the TTL value, it helps to extend caching of Content Style Sheets (CSS), image and JavaScript (JS) files to may be few hours or even days, so as users visit your site, their browsers don't have to refresh auxiliary content every time they go to a different page.

Yet as we go about configuring the caching rules, it is important not to be carried away. As you can see, caching frequently requested URLs for only 20 seconds already delivers most of the benefits - instantaneous response to users and nearly complete elimination of load on your web, DB and App servers.

But if you allow caching for say 1 day, you lose ability to have the content refresh itself within reasonable amount of time. So if and when you need to modify some cached content on the origin servers, it will take up to 24 hrs for it to get refreshed by visitor browsers, even after you expire it on aiCache servers. Clearly situation to be avoided if you do make frequent changes to your auxiliary content! While you can always force expiration of content in aiCache via CLI, there's no such easy way to do so at visitor browsers, stopping short of publishing a giant "Hit CTRL-F5 NOW!!!" banner on your home page, clearly not a very good idea.

Another reason not to jack up TTLs for cacheable content is less obvious, but equally important. Imagine that due a mistake or malfunction somewhere within origin server infrastructure, a broken or invalid content is delivered to, and cached by aiCache and visitor browsers. Again, if TTL is set too high, it will take a long time for that erroneous document to be purged and refreshed.

So the moral of the story is: don't go overboard with TTLs, set them so that you retain ability to make changes to your content. The TTLs configured in 30m range will allow a typical visitor to enjoy the benefits of his/her browser caching content locally for the duration of their visit to your site, while still preserving your ability to modify such content within reasonable amount of time.

One can argue that images are a different story and will be partially right. Images that never ever change can in fact be cached for an extended period of time (1 week+). Examples include the common 1x1.gif and things of that nature. Yet photographs are different and the same caveats apply: if you need to have ability to recall/change/expire an image within reasonable amount of time, do set reasonable TTLs. Of course there're ways to make unwelcome images to disappear from your site's content by simply editing them out of HTML, but we hope you still get our point.

And as a side advice - about the only way to guarantee whatever changes you make to JS and CSS do take immediate effect is to *version* them. In other world, you'd need to make sure the SRC location changes with every release. You can add a YYMMDD suffix to the names of your JS and CSS files or resort to a different form of versioned and consistent naming convention.

It just keeps getting better: aiCache benefits with non-cacheable content.

The *non-cacheable requests* are always forwarded *verbatim* to the origin servers and obtained responses are fed back to the requesting clients, *verbatim*[7], without being cached or shared. Even in this situation, with no benefits of caching possible, use of aiCache offers a *very important advantage: it offloads the task of dealing with the client network connections away from the web servers***,** where each of client connections requires a dedicated process that lives for the duration of connection, to aiCache – which uses extremely efficient, zero-overhead processing of requests/responses. Do not underestimate this benefit – it can be very significant!

To illustrate, let's imagine a client consuming larger responses from a web site over a slower/congested connection. Without aiCache front-ending such requests/responses, most existing web servers have to dedicate a whole separate process/thread to sending and/or receiving of this data to such slower client, even when the actual generation or post-processing of the response is very fast. Such dedicated process will need to be maintained for the duration of such connection, which could be 10 and more seconds.

Imagine more than a hundred responses like that being fed to the clients at the same time and you can probably see how most web farms would have a problem in such a situation. What if you have a few tens of thousands of connected users - the situation only gets worse ! And chances are that the code that generates the responses also maintains an application server and a DB connection for the duration of the response, only further compounding the problem and propagating even more load to your application and database Servers.

Now, with aiCache front-ending the traffic, the situation is very different. It is the aiCache that obtains a complete request from a client, makes sure it is a valid request and only then, virtually instantaneously, feeds it to an origin server.

Similarly, when on origin server is ready with a response, aiCache consumes it virtually instantaneously, not tying up the origin server for much longer time like a regular client would. After obtaining a complete response from origin server, aiCache then feeds the response to the requesting clients, using its extremely efficient, zero-overhead network logic.

In addition to this offloading benefit, aiCache also offers industry's most comprehensive set of DOS protection countermeasures, see a dedicated DOS chapter elsewhere in this manual.

The list doesn't stop there: additional benefits that aiCache offers for non-cacheable requests include optional on-the-fly compression and expanded reporting of and alerting on, wide range of additional statistics, *all in real time*, including number of connections, request/sec, response processing time, etc .

To summarize: even if your web site serves significant amount of non-cacheable content, aiCache still has so much to offer !

[7] aiCache might perform some request/response header modifications if so configured.

aiCache processing of cacheable content.

Of course, you obtain the most benefits when allowing aiCache cache *cacheable* content (try saying that 3 times in a row fast!). Let's discuss the most important phases that aiCache goes through when processing such cacheable requests.

First-Fill.

The cacheable content is obtained from one of the configured origin servers the first time a response is requested by a visitor and it is preserved in aiCache's in-memory cache for the so called "time to live" (TTL) period of time. During this period of time the response is considered to be fresh and, upon future requests, is served directly from the aiCache's RAM-based cache without having to retrieve it again from the origin servers. And when the response comes directly from aiCache's response cache, it **completely eliminates all and any load** that your web site components, such as web, application and database servers, would be subjected to otherwise.

Since aiCache's response cache is RAM based, serving of cached response generates no disk IO, outside of access log (and even that can be suppressed or decimated).

Refresh.

When a previously cached response becomes stale and aiCache receives a request for it, a fresh copy of the content is obtained from the origin web servers and again, is stored in the aiCache's cache. It is then used to satisfy user requests for another TTL interval. This cycle repeats for as long as the aiCache server is operational.

The normal behavior when discovering a stale cached response is as follows: the first request to discover a stale response, requests a fresh response from an origin server. Subsequent requests that discover the stale response are put in "waiting line" of sorts, awaiting the refresh that was requested by the first request, to finish.

When refresh takes a while to complete, the queue of waiting requests can grow large and each request in that queue will have to wait the entire time till the refresh response is obtained from an origin server.

You can override this behavior by setting **no_wait_refresh** setting at website level. With this setting in effect, it is only the first "unlucky" request that discovers the stale response that requests a refresh and waits for it for it to complete. The subsequent requests are replied to with "stale" data – which behavior you might find acceptable for your situation. When the fresh response data does finally come, the response cache is refreshed with it.

Lets say you cache some URL for 10 seconds and it takes a 2 seconds to refresh it. Assuming 100 RPS for this URL (1000 over span of 10 seconds), normal behavior results in 800 requests satisfied instantaneously with cached response data. 200 requests will have to wait 2 seconds to obtain the new data when stale response is detected.

With **no_wait_refresh** set, 800 requests are satisfied instantaneously with fresh cached response data, 200 requests are satisfied instantaneously with stale cached response data (up to 2 seconds stale) and only 1 request would have to wait the entire 2 seconds for its response. If this is an acceptable behavior for your particular setup, consider enabling it to minimize the response time.

aiCache reports the number of requests satisfied in this fashion as "no-wait misses" in both global and website statistics sections vi aWeb and CLI interfaces.

Irrespective of **no_wait_refresh ,** should the request for the fresh copy of cacheable content fail, you can configure aiCache to serve back previous (stale) version of cached content – instead of serving an error response. By doing so you will , in effect, shield your visitors from origin server failures.

Handling of non-200 origin server responses to cacheable requests.

Most HTTP responses have "200 OK" response code. When a non-200 response is obtained in response to a cacheable request, aiCache can be configured to retry the request and/or serve stale cached content. If all of the retry attempts fail and no stale content is available, aiCache then does one the following, depending on response status:

- 302, 301 redirects: response is sent to all requesting clients, response is cached in accordance with TTL setting
- 401,404,407 - response is sent to all requesting clients, but is not cached
- any other responses > 300, > 400: response is overwritten with short version, unless **orig_err_resp** setting is set in which case no overwriting takes place, and is sent to all pending clients. Such response is not cached.

Basically, aiCache does its best to not cache bad responses and to minimize the overhead of sending error responses to clients.

You can configure aiCache to only accept 200 response code for cacheable requests by setting **retry_non200**. When non-200 response code is received, aiCache will execute complete retry-fallback sequence .

```
pat /news.html exact 10
retry_non200
```

About 401, 407 responses .

We most strongly recommend to configure your site, aiCache and origin servers, to not require authentication for cacheable requests (ones that match a pattern with non-zero TTL). In other words, do not make content that requires authentication, cacheable. Whenever a request comes in that has Authorization HTTP header set, it is assigned TTL of 0 and always forwarded to origin server, even if it matches a cacheable

pattern and there's a valid, fresh, cached response available. Any response obtained to a request with TTL of zero, as you know by now, is never cached/shared.

When a cacheable request comes in, requiring first fill and/or refresh, and origin server replies back with a 401-status or 407-status response that has WWW-Authenticate HTTP header in it, the 401 response is returned to the requesting client(s).

When received by clients, presence of WWW-Authenticate HTTP header results in browsers prompting the users for username and password and when they are obtained, a request is sent again, this time carrying Authorization header. Presence of that header in request makes that request and its response non-cacheable - just as you would expect aiCache to process non-cacheable requests and responses.

If you want to enable caching of responses that require Authentication, you can use **httpheader** website configuration setting to add a pre-cooked Authorization header to all cacheable requests when a first fill or refresh is required - thusly "fooling" origin server into thinking that clients are pre-authenticated, for example:

```
httpheader  Authorization  Basic  ZWRtbW12Zm5Ymmzy
```

As a result the origin servers will not respond back with 401/407 responses and requests/responses can now be cached. Such setup only works with Basic authentication and doesn't work with Digest authentication.

You can capture appropriate value to use for Authorization **httpheader** by using an HTTP header capturing tool, such as HTTPHeaders, Firebug, Fiddler etc.

Best practices to maximize benefits of caching.

Here're some suggestions to maximize the benefits of caching:

- **Allow for prolonged caching of javascript (.js), content style sheets (.css), images (.png, .jpg, .gif etc)** and other auxiliary content, that is not changed frequently. Few days for auxiliary images and few hours for CSS and JS files are a good estimate here. Remember, aiCache serves cacheable content in a way that allows it to be ***properly*** cached by visitor's browsers and this will lead to even more benefits, eliminating the need for conditional requests (If-Modified-Since and If-Match) and 304 responses altogether.

- Sometimes, *caching even for few seconds makes a tremendous difference*. Imagine a URL that receives 100 requests a second. Without aiCache each and every one of these will head straight to the origin servers and most likely past that as well – right to app servers and DB servers. If you enable caching for just 5 seconds, then origin server will only see 1 request every 5 second. That is 500-time reduction of traffic to your origin servers and the rest of your infrastructure. Could be a difference between a site that cannot stay up even with dozens of origin servers (and matching number of App and/or DB servers) and a site whose footprint can now be reduced to say just 2-3 origin servers ! Meaningful results can be obtained even with 1-2 second TTLs for the busier URLs.

- Normally it will be responses to GET requests that we configure for caching. However, some of sites use POST requests for their search forms and such. *aiCache allows for caching of POST requests*,

although you must be careful with these, to prevent personal data from becoming shared by mistake – we have more to say about it later in this document.

- Consider employing some form of versioning for auxiliary content if you want to have full control of your web pages. For example, if a web page references a Javascript file called **script.js** and you need to modify that file, you might not be able to guarantee that all of your site visitors will see the new file content right away - as an older, cached version **script.js** might still out there in corporate and ISP HTTP proxies and in local browser caches. A very simple fix to change the SRC attribute of the Javascript file to **scriptV2.js** and rename the file itself to match this name. Now the change is instantaneous and assured right after you publish the HTML page that references this JS file.

It is important to mention that with aiCache, you do not have to modify anything on your origin web servers, App or DB server nor do you have to ask your Dev team to write any code to handle freshness control - that crowd always has plenty of projects to work on as is. This allows *you* to control freshness of the content with great ease and minimum overhead, all configured, on-the-fly, with no user impact, at a single source.

aiCache handling of conditional HTTP requests.

To maximize benefits of caching, eliminate unnecessary traffic and simplify handling of requests and responses, aiCache employs certain logic that we shall explain here. But let us first provide certain basic information about conditional requests and responses.

HTTP has a number of optional request and response headers that are known as "conditional" headers. For example, when an HTTP response is delivered from a web server to a requesting client, some or all of the following headers could be sent back by the responding web server:

- **Etag** (for example "Etag: 0924385FDCACCC"). This can be thought of as response's *digital fingerprint* of sorts .
- **Last-Modified** (for example "Last-Modified: Jan 1 2001 01:01:01 EST").
- **Expires** (for example "Expires: Jan 1 2001 01:01:01 EST")

The idea behind these response headers is to associate certain information about the response bodies that can then be used by the requesting client to formulate *conditional* requests for the URLs when client are in need of requesting of the URLs again.

For example, client can say: *give me a response for this URL, if the response's Etag is different from certain value,* or: *give me a response for this URL, if the response has been modified since a certain point in time*. The web servers can then look at these conditional headers and serve the complete new response – such as when response's **etag** is indeed different from what client has specified in the request, or the response's **Last-Modified** time is different from one specified in the request.

Or, if the condition has not changed, a much abbreviated version of HTTP response could be sent back. Instead of a complete new response, literally a very brief "nothing has changed" response is sent back. Web server assumes that client browser still has a local copy of the response and since response has not changed at the web server, client should just use its local copy. For example, instead of sending back 100KB of data, less than a 100 bytes are now sent to the client, a 1000-fold decrease in traffic!

So with **Etag** and **Last-Modified** headers sent in the responses, client would typically qualify all subsequent requests for the same URLs with certain matching qualifiers, as if telling web servers – send me a responses to this URL request only if conditions have changed.

The **Expires** header is a bit more straightforward. Incorporated in the response it simply tells the requesting client that it can safely reuse (locally cache) the response till the specified date and time. Client doesn't need to follow up with subsequent conditional requests, for as long as local time is less than **Expires** time as specified in the response.

aiCache takes this optimization ideas one step further. Unless otherwise configured, it eliminates/strips out both the **Etag** and **Last-Modified** conditional response headers for cacheable content. It does so by making sure these are never propagated from origin servers to the requesting clients. It also never generates such headers on its own.

Additionally, aiCache eliminates/strips out **Expires** response header, as sent by origin servers, for cacheable content. In leu of that value, aiCache formulates its own value for the **Expires** header and that is what is stored and sent to the clients, for cacheable content.

For example, assuming certain URL is declared to be cacheable for a day, aiCache would calculate proper **Expires** value, upon first caching of the response, as "time now + 1 day" and use the resulting value in the **Expires** header.

The end result is encouraging of caching and reuse of content by the clients (browsers, intermediary proxies and caches) without having to burden both the clients and the aiCache server with generation and processing of conditional requests.

Enabling forwarding and processing of Etag validators for cacheable responses.

Normally, aiCache removes Etag headers from cached responses and does not provide 304-Not-Modified responses for requests with "If-None-Match" headers. It does so in order to eliminate all conditional requests.

However, if you so require, you can instruct aiCache to store/forward Etag headers for cached responses and/or process requests with If-None-Match headers.

To configure aiCache to store and forward to requesting clients the cacheable response's Etag header, set **forward_os_etag_header** setting at website level. It is a flag and requires no value.

To configure aiCache to process If-None-Match conditional logic, set **process_etag** flag at website level. It is a flag and requires no value. When so configured, aiCache will look into incoming requests to see if "If-None-Match" conditional request header is present. If it is present and cached response contains matching Etag header, as obtained in the response from an origin server, a "304" Not-Modified response will be issued to the requesting client.

Please test carefully before enabling such processing and forwarding of Etag headers so that it doesn't impact your site.

When responding with 304 responses, aiCache will add ETag header to the response, assuming the origin server provided the ETag header when serving the original response back to the aiCache.

Overriding pattern TTL based on response header value.

Normally, aiCache obeys caching TTL rule as set by matching pattern. Sometimes, you might find yourself configuring a aiCache for a website that requires certain flexibility in assigning TTL based not only on

the *request* URL (these you match via *patterns*), but also on what *response* looks like. In this particular case, TTL might depend on values of certain *response* headers.

For example, let's assume there's a URL pattern of **/content_ID/NNNN** , where NNNN is some random number. These URLs, when requested, can return HTML – which you would like to cache for 10 secons, CSS that you could cache for 1 day or images which you could cache for 1 week. But you cannot make the determination of the TTL until you receive the actual response and can analyze the Content-Type header.

Enter the **resp_header_pattern_ttl** setting. It is pattern level setting that requires exactly 3 parameters: header name, header matching pattern and the *override* TTL. Here's how you can use it to configure what you require in the example above

```
…
pattern /content_ID/ simple 10 # default cache of 10 secs

resp_header_pattern_ttl Content-Type css 1d # but cache CSS for 1 day
resp_header_pattern_ttl Content-Type image 1w # and cache images for 1 week
resp_header_pattern_ttl Content-Type java 1w # cache JS for 1 week
```

In the example above, we use Content-Type response header, but you can match on arbitrary response header – so that you could have origin servers return some custom header that you could act on.

For such setup to work, the URL pattern must be cacheable. The overriding TTL could be larger or smaller than that of the enclosing pattern but it must be a non-zero positive number, followed by optional TTL convenience modifiers - "d" for day, "w" for week and so on..

The header matching pattern is a regular expression pattern, so you have full power of regexp here.

aiCache reports *override* TTLs in the output of CLI *inventory* command, but it is the matching pattern's TTL that is reported in the access log file. The access log file does report actual cache hit or miss, taking into account the override TTL value.

Please note that a given URL should always return response of a certain Content-Type, otherwise unpredictable behavior will take place.

TTL-bending when under heavy load.

Normally, aiCache obeys a caching TTL rule as set by matching pattern. Sometimes, you might find yourself in a situation when heavy load on your site is driving response time and load to origin servers and you'd like to temporarily increase the TTLs, to reduce the load and survive the onslaught of traffic. You can certainly manually edit the configuration file, modify (increase) the defined TTLs and reload the configuration for these changes to take effect. But then you need to remember to restore the old settings back when the load subsides and clearly, such manual changes require presence of an operator.

aiCache comes to the rescue yet again, with a website-level setting called **ttl_scale_factor**. It takes 2 required parameters: integer scale factor and activation response time, in that order. For example:

```
website
hostname news.acme.com
ttl_scale_factor 5 2000

pattern news.php simple 10
```

With these settings, should averaged out website response time exceed 2000 msec, the TTLs will be automatically and temporarily increased by factor of 5. So the pattern of **news.php** will be cached for 50 seconds, instead of 10, should response time exceed 2000 msec (2 seconds). All of the defined cacheable patterns will have their TTL increased by the same factor. When response time drops below 2000 msec, the TTL will be atomically restored back to its original value of 10 seconds and likewise, all of the defined cacheable patterns will have their TTL restored back to the original value. All of this magic will happen automatically, without any operator involvement.

Please that aiCache analyzes response times every few seconds . It also logs both setting of the scale factor and resetting of it, in the shared error log file.

Watch-folder, file-driven content expiration.

As you know by now, aiCache offers a number of ways to forcefully expire cached content – including CLI, response-driven expiration and hand-crafted, special expiration URLs. It also provides a simple web page for form-driven expiration.

Additionally, aiCache provides file-driven expiration functionality. You specify an expiration watch directory, by setting global setting of **exp_watch_dir**. For example:

```
exp_watch_dir /usr/local/aicache/exp_watch_folder

admin_email content@operations.acme.com
```

aiCache then checks content of that folder every 5 seconds. Any files found within that folder are assumed to contain content expiration instructions, one per line, in the following format:

```
WEBSITE_NAME   REGULAR_EXPIRESSION_PATTERN
```

For your convenience, aiCache allows and ignores empty and comment lines. For example

```
# Expire News
www.acme.com   NEWS
# Expire Sports
www.acme.com   SPORTS
www.acme.com   .gif
```

As aiCache processes these instructions, resulting output is logged to error log file. It can also be emailed, by setting **admin_email** global level setting. The expiration email contains timestamp, identifies each

expiration file as it is processed and reports how many entries have been expired per pattern. As usual, aiCache only spools the outbound email into the **alert_dir** and it is up to an external script to pick up and dispatch the email to proper address. An example script, alert.pl, is provided within aiCache distribution file.

It is up to you just how you place expiration files into the expiration watch directory. You can use push or pull method of your choosing – FTP, SCP, RCP, RSYNC etc.

Please note that aiCache instance that has obtained an expiration file in its watch folder, doesn't communicate the content of this file to its aiCache peers (should any be defined). In other words, such file-based expiration is not peer aware and you need to make sure each aiCache instance is *fed* the same expiration file, to assure expiration of content throughout aiCache cluster.

aiCache removes each expiration files from **exp_watch_dir** after it processes them, one by one. You can name the files anything you like, may be coming up with some kind of naming convention that makes sense to you. As usual, please assure aiCache user has full rights to the directory, so it could both read and remove the expiration files.

Preventing caching of responses.

When aiCache finds no matching pattern for the request's URL, it declares the request/response non-cacheable. In other words aiCache can only cache the requests that match caching patterns with non-zero TTLs, non-matching requests are never cached.

Another way to prevent Web documents from being cached is to explicitly assign TTL of 0 to certain patterns. Typically examples of such URLs would include dynamic, frequently accessed personalized content or URLs that might "clash" with cacheable URLs.

For example, let's consider a URL:

www.acmenews.com/viewmyprofile.jsp?userid=1234

This URL displays customer profiles, which are unique and private to each user. We definitely want to avoid this response from being cached for a number of reasons. First of all, it does not make sense to cache this response since it cannot be shared amongst different users and we would not realize any performance gain from doing this. Secondly users would not be happy, so say the least, if we were to share their private information with other users of our web site. Therefore if we do not include a pattern that matches this URL in the configuration file, then this document is never cached/shared. Another way to accomplish that would be to put a matching pattern into the configuration file. For example:

```
pattern viewmyprofile.jsp simple 0
```

You must always be careful not to allow caching of private user information. As you fine-tune the configuration files and make changes to TTL of *shareable* documents, no harm can be done to your users from the privacy point of view. However if you do not pay attention and somehow allow for sharing of private user data you might be in trouble with your users. See troubleshooting section for more information on how this can happen and how to avoid from happening in the first place.

aiCache has much more to offer in the way of caching controls, we will discuss it later in this Guide.

URL rewriting and rewrite-redirection.

Occasionally you might have a need for URL rewriting. For example if you re-arrange location of your site's images from **/images** to **/media/images** you might want to catch any "stray" requests for anything under **/images** and change that to **/media/images**. So if you detect a request coming in for **/images/1x1.gif**, you want to change that to **/media/images/1x1.gif** . This kind of on-the-fly rewriting might come handy when you have references to the relocated content scattered throughout your site, including static and dynamically generated content and want to make sure you don't break any of these pages .

A different example: say you decide to move all of the auxiliary content from **www.acmenews.com** to **media.acmenews.com** . So now, when a request comes to **www.acmenews.com,** requesting anything under **/css, /js , /images**, you want such requests be redirected to **media.acmenews.com ,** while keeping (or changing) the original link. For example a request for **www.acmenews.com/image/1x1.gif** might be redirected to **media.acmenews.com/media/images/1x1.gif .**

Another example - dealing with dynamic content. A URL of code that renders forum pages might be changed from **displayforum.php?forum=123** to **showtopic.jsp?fid=123** . Yet you want to make sure any bookmarks that your visitors might have made before the change, continue to work after the switch.[8]

These are the situations where you might need to use *URL rewrite* and *URL rewrite-redirect* features of aiCache. To configure, specify **rewrite** setting under matching pattern section. The rewrite directive takes 2 parameters: the **from pattern** and the **replacement string**.

To accommodate for 1st example:

```
pattern /images simple 30m
rewrite /images /media/images
```

In case of such URL rewrite, the new rewritten URL inherits the TTL setting of the matching pattern.

[8] Clearly you should not rely on such rewriting to deliver most of your content, but use it instead a temporary patch facilitating seamless transition, temporary support of legacy code/users etc. Overuse of rewrites is known to cause mudslides, earthquakes and 10-hour long outages.

To force *rewrite and redirection*, as opposed to *rewrite only*, the replacement string must start with **http:// or https://.** To accommodate for second example:

```
pattern /image simple 30m
rewrite /image http://media.acmenews.com/media/images
```

In case of redirect, the TTL setting ceases to have any meaning, yet still must be provided to conform to the pattern definition syntax.

Normally aiCache issues a "302" redirect. To change to "301" redirect instead, set **redirect_301** flag at the pattern level.

Moving to more complex URL rewriting examples - use of pattern *grouping* in **from_patterns** and *back-references* in *replacement* patterns. You can group matching symbols in the **from_pattern** by enclosing a part of the matching pattern in parenthesis. You can then refer to these groups by using a special notation:

N (backslash-number)

in the *replacement* pattern. \0 stands for the whole matched string, \1 for the first defined group and so on. To accommodate for the third example:

```
pattern displayforum.php simple
rewrite displayforum.php\?forum=(\d+) showtopic.jsp?fid=\1
```

As with all regexp patterns, special characters must be escaped in the *from_pattern* - this is why we have a backslash in front of question mark in *from_pattern*. We capture the *forum* numeric parameter by using the (\d+) grouping pattern and then refer to that captured group in the replacement pattern by using a *back-reference* **\1**. You can have more than one group defined in the *from_pattern* and referenced in the *replacement* pattern.

Normally, rewrites would happen for both HTTPS and HTTP requests. However, you can limit the rewrite to particular type of requests, HTTP or HTTP, by using **rewrite_http_only** and **rewrite_https_only** flags:

```
hostname store.acme.com
...
pattern secure.jsp exact 0
rewrite secure.jsp https://store.acme.com/secure.jsp
rewrite_http_only
```

The example above effectively rewrite-redirects to secure, HTTPS-protected URL. Unless **rewrite_http_only** setting is specified, the rewrite might cause infinite rewrite-redirect loop. Likewise, you might want to safeguard certain URLs so that they are only requested over HTTP, using similar technique.

Alternatively, you can use **match_http_only** or **match_https_only** pattern-level settings .

The replacement patterns are very flexible and can accomplish much. But yet again, do not overuse the rewrites to where your site becomes an example in obfuscation techniques, use it only when necessary. Every now and then do summon all the strength you can and drop older rewrite rules so you don't accumulate too many of them. We've seen sites with hundreds of rewrite rules that went back 3-4 years ! None of these were in use anymore yet still they littered the config files. Think twice before using rewrites to catch and correct for user FFS (fat finger syndrome) .

To help with fine-tuning, testing and troubleshooting of rewrite patterns, you can enable rewrite logging by specifying **log_rewrite** setting (place it in *global* or *website* sections). The original and rewritten URLs will be logged in the *error log file* as rewrites take place. After arriving at working rewrite patterns, we recommend to turn the rewrite logging off.

You can also use *pattest* binary that comes in aiCache distribution to test pattern match and rewrites.

Rewrite-redirect (scalpel) feature is different from *plain redirect* (sledge hammer) feature . For any given match pattern, the latter always redirects to the same location, no matter what was in the original URL, while former allows for much more intelligent handling.

After rewriting request's URL, aiCache doesn't attempt to re-match the new, rewritten URL to the list of defined patterns. Specifically, if you want to assign certain TTL value to a pattern, assign it to the to-be-rewritten pattern.

Please note that when you enable UA-driven URL rewriting and redirection (see below) or use **ua_pattern** settings, explained later in this document, the UA-driven URL rewriting and redirection happens ***first - as in it takes precedence over "plain" URL rewriting*.**

Decimated rewriting.

You can configured aiCache to rewrite a controlled portion of the requests – as opposed to each and every one. The pattern-level setting is called **rewrite_decimate .** It requires a single numeric parameter that it uses as rewrite decimation factor via modulo division. For example, when set to 10, every 10th request is rewritten (10% of requests).

```
pattern displayforum.php simple
rewrite displayforum.php\?forum=(\d+) showtopic.jsp?fid=\1
rewrite_decimate 20 # rewrite 1/20 or 5% of matching requests
```

URL escaping.

Certain older browsers might break URLs by not properly escaping request URLs. For example, international sites might send URLs with non-ASCII symbols. Conforming browsers are supposed to “escape” such symbols by replacing them with “%XX” notations, where XX is the symbol’s hex value.

In order to deal with non-conforming browsers, you might opt to instruct aiCache to force url escaping on inbound requests, by setting **escape_url** flag at global level of the aiCache configuration file. The setting is a flag and requires no value.

By default, the following characters are considered safe and are not escaped in URLs:

```
:/?#[]@%!$&'()*+,;=
```

You can provide your own list of safe characters by setting **escape_url_safe_chars** global level setting, for example:

```
server
…
escape_url_safe_chars :/?[]@%!$&'()*+,;=
```

Since **escape_url_safe_chars** is likely to contain a hash mark # - which is normally reserved for start-of-comment in aiCache, no comments are allowed in the same configuration file line. But you can provide a comment line before or after this line, in usual fashion.

Support for intelligent handling of mobile and desktop versions of the websites.

Supporting desktop and mobile versions of web sites.

Many businesses have a requirement to support both *desktop* and *mobile* versions of their web site(s). In some cases, it done in a fashion where mobile traffic is (re)directed to a *different* website, for example m.acme.com, while sending desktop traffic is sent to the usual www.acme.com.

Some sites would offers both mobile and desktop versions of their content *unified* under *same* domain – for example www.acme.com, tailoring output to match device capabilities.

Mobile site would normally render content in a simplified way that is more suitable for limited screen size and limited Javascript, HTML and CSS standards support in mobile browsers. Some sites distinguish between different mobile devices, offering more content rich versions of the content for more capable devices while offering the simple version to the less capable devices.

Likewise, due to limited screen sizes of mobile devices, sites cannot serve the same number of ads per page, for mobile users, as opposed to much larger number of ads per desktop versions of the site. This simple fact can become point of contention with many a marketing department. Business people would rather serve a desktop page to an end user as opposed to delivering mobile version of same, due to higher ad revenues.

To support rendering of device-specific content, you need to have some kind of user-agent detection logic deployed on your web site(s), analyzing user agents strings in incoming requests and tailoring the output and/or redirecting user traffic as appropriate. For example, when a request coming for www.acme.com's home page is detected to be coming from an Android mobile device, a response is sent to redirect the user browser to m.acme.com. Alternatively, no redirect response is sent back and instead, a mobile version of the page is rendered right by the www.acme.com.

With two different site serving content, similar logic can be deployed at m.acme.com and when request is detected as coming from a desktop browser, the response is sent back that redirects the browser to go desktop version of the site at www.acme.com.

There are a number of challenges related to supporting such *duality* of content representation to different device types.

Reliable detection of mobile devices.

As simple as it sounds, this is less than trivial issue. Most detection methods rely on analysis of the request's User-Agent header. Two different techniques could be used: pattern-driven UA matching and dictionary-driven UA-matching. While pattern method is self-explanatory, the dictionary-driven one relies on some kind of database that contains, verbatim, letter-for-letter, all UA strings that you need to act on.

Once source of such information could the Internet – there're a number of commercial and open source projects that have such UA DB available.

No matter the method, as new devices seem to appear every week, you will need to update your matching logic to stay current. Not only new devices are released frequently, most devices are released by a whole number of vendors, each such device having its own UA string. As browsers and device ROMS get updated and patched, even more UA strings spring into existence. Be prepared to spend some time keeping your UA patterns and dictionaries up-to-date.

aiCache offers industry's leading support for high-performance and seamless device detection using both of these methods, so that you don't have to do it at your application level.

Deploying device detection logic.

If you were to let the code on origin servers decide what version of content to serve, you'd need to pass requests straight to origin servers, so that they could make the decision about device type, effectively negating all benefits of content caching. Clearly a non-starter !

aiCache offers industry's leading support for high-performance and seamless device detection so that you don't have to do it at your application level .

Having a strategy for handling of search bots/spiders.

Most sites don't want to allow spidering/indexing of mobile site's content and instead would rather prefer Googles of the world indexing their main desktop sites. Again, it boils down in most cases to revenue-per-page issue we described above. When someone runs a search and your site pops up in the results, they'd want the link to point to desktop version.

So you would probably want to prohibit spidering and indexing of your mobile site's content – for example by hosting appropriate robots.txt file on your mobile site.

When a request from a spider comes to your main site, you need to make sure to let it through and not redirect it to the mobile version of the site by mistake.

You can easily configure aiCache to handle the bots as special cases so you don't have to do it in your code. Or alternatively, rely on aiCache's default handling of unknown devices – no rewrites or redirects are issued by your desktop site and the request "stays" with the main site.

Supporting different URL structure.

Some site have their desktop and mobile versions served from completely different environments – the code, the backend databases, the datacenters - some or all of these could be different. But most importantly, the URL structure is likely to be different too.

For example, desktop version of acme.com could serve "US News" section as www.acme.com/news_front. The same section is served by mobile site as m.acme.com/content/node_id=123 . Notice how very different the two URL are. Outside of the home page, which is hopefully served as "/" in both cases, dozens of other URLs pointing to assorted section fronts, product listings, published stories etc, could be completely different.

Now imagine a mobile user sharing a link to m.acme.com/content/node_id=123 with a desktop user. When opened in a desktop browser, this URL is likely to result in a PNF (404 Page Not Found) error, unless it is somehow intelligently rewritten to www.acme.com/news_front

For example, let's imagine that you provide 10 different section fronts on the mobile site via URLs like m.acme.com/SECTION_NAME, for example m.acme.com/world_news, m.acme.com/politics , m.acme.com/finance etc.The same section fronts are rendered via completely different looking URLs on the "main" site.

You'd need to come up with a way to provide 2 way mapping between these two sets of URLs, if you want to be able to catch and redirect both mobile and desktop users in appropriate way. This in itself could be a strong argument for use of unified CMS for both sites, one that uses same URLs for the same content - and as you know aiCache offers support for it too.

aiCache offers industry's leading support for seamless device-driven URL rewriting – so the proper version of an article or a section front is delivered to proper devices! As a matter of fact, it is so advanced, even when you have a number of different sites serving different versions of content (ie m.acme.com and www.acme.com) you can fold both mobile and desktop sites under one site, while making sure everything works 100% and proper content is served to proper devices!

Supporting unified mobile/desktop site.

Some sites have their desktop and mobile versions served from same content management system (CMS) – *unified* or *folded* under the same www site. Such setup has challenges of its own. Imagine an iPhone user requesting home page, for example www.acme.com/ Most CMSs would render a simplified version of the home page, taking into amount limited screen size and limited capabilities of the mobile device (iPhone in case, but it could be an Android device or Windows mobile).

At about the same time, a different user requests the home page, but this time she is using a desktop browser. In response to the request, the CMS would render out completely different looking page from one that the same CMS delivered in response to the request made from an iPhone.

Likewise, there could be requests for the home page, coming from assorted search bots – in response to which you might want to deliver a different version of content.

Now think about thousands of different URLs that your site provides and needing to tailor output of most of them to the capabilities of the requesting device.

You might have a sinking feeling in your stomach just about now, thinking to yourself – there's no way this could be all handled and cached by aiCache. All of these requests will now have to go straight to my origin servers and there's no way my infrastructure can possibly handle this much traffic !

aiCache offers full support for device-intelligent caching in this scenario, with no custom programming required on your part !

Letting users have a choice.

So it might be clear to most that a lesser mobile device should be served a simplified mobile version of the requested page. What about 7" Android tablet – what version should this one receive – same simplistic page, a full blown desktop version or something in the middle ?

How about 10" Android Tab or an iPad ? What about Windows 8 tablet – these will likely support all of the latest HTML/CSS/JS and Flash standards just as good as their desktop counterparts. In addition to defaulting users to presentation (mobile vs desktop) of your choosing, how about letting users decide what site they want to see?

And you guessed it, aiCache supports that too!

aiCache's method and apparatus of supporting device-specific seamless and transparent content selection, caching and filling .

Here's the logic that aiCache uses to deliver on the Holy Grail of desktop/mobile-and-everything-in-between intelligent content serving: Divide et impera.

Before we can conquer, we must divide. There are about 16000 different user agent strings in existence today and the number is growing every day. aiCache can use 2 different methods of UA matching: pattern-driven and/or dictionary-based exact-UA-match-driven to "compress" this madness down to more manageable number of User-Agent-derived ***tags.***

To do so, you can define, at website level, a series UA *patterns* along with their tags. Likewise, a setting could be set pointing to a file that matches complete UA strings to tags.

Again, aiCache could use both pattern-based matching and exact UA matching, in that order. When UA could not matched through neither of the methods, a UA tag of "default" is assigned to the request.

aiCache runs the UA-matching logic only when told so, instead doing it for each and every request. While you might want to run this logic for home page URL, you don't want to run it for hundreds of URLs that request auxiliary content – JS, CSS and image files. You configure aiCache to run the UA matching logic for selected requests by setting **ua_tag_process** flag at pattern level.

Note that you *must* set **ua_tag_process** flag at pattern level to have aiCache apply UA tagging logic.

Again, when request's UA string is matched to a **ua_tag_pattern** or to an exact UA string (defined in **ua_tag_file),** aiCache then ***tags*** the request with the tag name that you provide. When no match could be found, a tag of ***default*** assigned to the request. You will see how you can use the *default* tag later.

To make it easier for the origin servers to decide what version of content to render, aiCache sends the matched tag to the origin servers via **X-UA-Rewrite** header.

For example, you can decide to render your content in 3 different stylings: **mobile_simple**, **tablet** and **default** desktop style. As desktop styling is most common, aiCache doesn't require a custom tag for it, but let's use **mobile_simple** and **tablet** as two other *tags*.

Now, we tell aiCache how to match User-Agent string to each of these tags: 2 explicit tags and one default.

To configure UA-to-tag patterns, simply define them, at website level, via **ua_tag_pattern** For example (incomplete list)

```
    ua_tag_pattern     .*BlackBerry8.*    mobile_simple
    ua_tag_pattern     .*iPhone.*    mobile_simple
    ua_tag_pattern     .*iPad.*    tablet
```

To configure UA-to-tag exact-matching, simply define them in a separate file, where each line contain tag and complete UA, in that order, separated by single white space. Lines starting with # are ignored.

To make it easier for the origin servers to decide what version of content to render, aiCache sends the matched tag to the origin servers via **X-UA-Rewrite** header.

You can derive such file from a source of your choosing. It is up to you to secure proper rights and access to the UA database – some are free, some require a payment etc. You might be able to receive a complete list of known US agents from your analytics provider. Beware that the list undergoes daily changes – as new devices, browsers and browser patches, ROM versions etc are introduced to the market.

After obtaining a DB containing the User Agent strings, you can process the file using a custom script and output the file format that aiCache expects. For example, using WURFL or similar device description file, you can look at screen size attribute and assign tags based on that value. Possibilities are endless – you can look for certain device type, CSS or JS support, support for HTML5, Flash - but again it is up to you to produce the output in the format that aiCache expects and as you will see, the format is very simple

For example (incomplete list with just 2 entries) place the following into a file called *useragentfile.* Note in the example below how the exact UA matching is specific to a particular release of Firefox 13 browser , version 13.0.1 . It is likely there will be dozens of different version of Firefox 13, each requiring its own exact UA matching string. Likewise, the particular UA string we provide for Ipad, is likely to go through many minor changes, each requiring a new UA string. It is easier to match for such UA by using UA pattern matching, as opposed to exact, "letter-for-letter" matching. But we digress.

```
    # Match firefox 13.0.1 to "default" tag
    default Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101 Firefox/13.0.1
    # match Ipad to "tablet" tag
    tablet Mozilla/5.0 (iPad; U; CPU OS 3 2 like Mac OS X; en-us) AppleWebKit/531.21.10
(KHTML, like Gecko) version/4.0.4 Mobile/7B367 Safari/531.21.10
```

Next we point to that file via **ua_tag_file** setting, at website level. For example :

```
    ua_tag_file     useragentfile
```

Now, you need to tell aiCache "run the UA matching logic for these requests" by setting **ua_tag_process** flag at pattern level. This way aiCache runs the UA-matching logic only when told so, instead doing it for each and every request. While you might want to run this logic for home page URL, you don't want to run it for thousands of URLs that request auxiliary content – JS, CSS and image files.

Note that you *must* set **ua_tag_process** flag at pattern level to have aiCache apply UA tagging logic.

With **ua_tag_process** flag set at pattern level, aiCache could use both pattern-based matching and exact UA matching, in that order.

When UA could not matched through neither of the methods, a UA tag of "default" is assigned to the request. You can then use the "default" tag to drive URL rewriting, as described below. You can also explicitly assign default tag to the request through pattern or exact UA matching.

To make it easier for the origin servers to decide what version of content to render, aiCache sends the matched tag to the origin servers via **X-UA-Rewrite** header.

When needing to fill a UA-tagged request from an origin server, request's original User-Agent string will be forwarded to the origin server, including "default"-tagged request. You can configure aiCache not to forward User-Agent header for requests that are tagged as "default" by setting **no_ua_default_tag** at website or pattern level.

aiCache executes the tagging logic, as described above, for both cacheable and no-cacheable request, as long as matching pattern has the **ua_tag_process** flag set. Likewise, the tag is forwarded to the origin servers for both cacheable and no-cacheable requests alike and could be acted on by the server-side code.

However, the UA tag takes on special meaning when ***cacheable*** requests are concerned – in that it becomes a part of request's cache signature, allowing to have unique cached responses stored for same URL but different UA Tags.

You can go about serving mobile and desktop user in a number of different ways. The rest of configuration depends on your setup:

Different sites (www.acme.com and m.acme.com), different URL structure.

We'd want to catch what we think are mobile requests, on the "main" site and rewrite-redirect them to the mobile site. Likewise, we want to catch what we think are desktop requests at mobile site and rewrite redirect them to the desktop site.

Please be aware that a setup like this might really stretch your sanity to a limit – as you need to catch and rewrite all of the URLs of significance that need to be matched across both content management systems.

For example, let's imagine that you provide 10 different section fronts on the mobile site via URLs like m.acme.com/SECTION_NAME, for example m.acme.com/world_news, m.acme.com/politics , m.acme.com/finance etc.The same section fronts are rendered via completely different looking URLs on the "main" site.

You'd need to come up with a way to provide 2 way mapping between these two sets of URLs, if you want to be able to catch and redirect both mobile and desktop users in appropriate way. This in itself could be a strong argument for use of unified CMS for both sites, one that uses same URLs for the same content - and as you know aiCache offers support for it too.

Let's concentrate on desktop site for now. First let's match, to the best of our abilities, all of the known mobile devices to **mobile** tag. You can use both pattern and exact matching methods as described above.

Internet has many mobile-matching patterns, ready for use – but do be warned that they need constant upkeep, due to new devices, new ROMs, new versions of browsers appearing daily.

As mentioned earlier, it is often time *more important* to err on the side of keeping the requests on "main", aka desktop site, as opposed to redirecting a request to mobile site.

Next, in the matching pattern section, you can specify an exact match string, that is to be matched against the UA tag (in our case, these will be **mobile**), along with URL rewrite string. For example, we catch requests from mobile devices – by matching to the tag of **mobile**, directed at /news.html and /technews.html on the main site and rewrite/redirect them to point to matching URLs on the mobile site. The setting is called **ua_url_rewrite** and you'd use it at pattern level:

```
pattern /news.html simple 30
ua_tag_process
....
ua_url_rewrite mobile .+ http://m.acme.com/render?id=22

pattern /technews.html simple 30
ua_tag_process

....
ua_url_rewrite mobile .+ http://m.acme.com/render?id=31
```

Note that you have full power of regular expressions in the rewrite pattern, so you capture part of the original URL and use that in the rewritten URL.

We're not tagging the requests from any other browsers, so they will be processed (cached etc) as usual.

Clearly, you'd want to catch and properly rewrite-redirect all URLs of significance – but not more than that. For example, you wouldn't want to apply this processing to the static content etc, as it is not likely to be ever requested from your main site by any of the mobile browsers.

The **ua_url_rewrite** setting requires 3 mandatory parameters and accepts up to 3 optional ones. It has the following format:

ua_url_rewrite TAG match_pattern rewr_pattern [TTL] [OS_TAG] [sub_hostname]

As you can see, you can specify(override) TTL, specify OS Tag, so that matching requests could be filled from different origin servers and lastly, specify a different Host header to be sent to the origin servers when a response is requested.

All 3 are optional, but you must specify the preceding parameters when you want to specify OS Tag or sub_hostname, so that aiCache knows which value is which. So in order to specify OS_Tag, you must specify TTL - even if matches the pattern's TTL.

If you were to specify sub_hostname value, you then must specify both the TTL and OS Tag – even when you simply want to use default OS tag of 0.

Remember that when UA could not matched through neither **ua_tag_pattern**(s)or **ua_tag_file**, a UA tag of "default" is assigned to the request. The tag will only be set when **ua_tag_process** flag is set at pattern level. You can then use the "default" tag to drive URL rewriting, as described above.

Different sites (www.acme.com and m.acme.com), same URL structure.

This scenario is rather unlikely, but configuring it is very easy. We still match mobile requests and tag them with **mobile .** The only difference from above is that we don't rewrite the URLs and instead, forward users to the same URLs on different site.

Normally aiCache issues a "302" redirect. To change to "301" redirect instead, set **redirect_301** flag at the pattern level.

```
pattern /worldnews.html simple 30
ua_tag_process

....
ua_url_rewrite mobile .+ http://m.acme.com/worldnews.html


pattern /technews.html simple 30
ua_tag_process

....
ua_url_rewrite mobile .+ http://m.acme.com/technews.html
```

Same site (www.acme.com), same URL structure, same origin servers.

Again, configuring it is very easy. We still match mobile requests and tag them with **mobile .** And that is it! aiCache will still cache content as per your configuration. To make sure content is cached so that differently-tagged versions of it don't collide, the UA **tag** is added to the cached request signature.

To make it easier for the origin servers to decide what version of content to render, aiCache sends the matched tag to the origin servers via **X-UA-Rewrite** header. Complete User-Agent string is also forwarded to the origin server. It is up you what you want to base your content-rendering decision making on: the **tag** that aiCache conveniently forward for you, or the entire User-Agent string.

Same site (www.acme.com), different URL structure, different origins.

This setup allows you to fold(unify) what presently are two different websites – the main and the mobile, under the same site, even if they are running different CMS and are hosted on different servers, possibly out of different datacenters.

Please be aware that a setup like this might really stretch your sanity to a limit – as you need to catch and rewrite all of the URLs of significance that need to be matched across both content management systems. You will also need to decide when to do *simple* URL rewrite the URLs in the transparent way - so that URL doesn't change in the browsers' address bar, as opposed to when to *rewrite-redirect* so that URL changes in the browsers' address bar.

As usual, do your best to match known mobile UA to **mobile** tag. Again, you can have a number of tags and very evolved matching, we're simplifying here.

Next we rewrite the URLs that are different between desktop and mobile URL structure. You can rewrite "behind the scenes" to where visitors are not aware of the rewrites or issue redirect to the proper URLs instead, it is up you. Possibilities are endless.

We also specify a different OS Tags in the rewrites so that mobile content is filled from different origin servers .

In addition to overriding the TTL and assigning a different OS tag, we can also ask to modify request's host name, when a fill is needed from an origin server.

```
pattern /USnews.html simple 30
ua_tag_process

....
ua_url_rewrite mobile .+ /render?id=123 0 2
# Notice how we override TTL to 0 and OS tag of 2 is specified as last param above


pattern /technews.html simple 30
ua_tag_process

....
ua url rewrite mobile .+ /render?id=223 0 2 m.acme.com
# Notice how TTL is set to 0, OS tag of 2 is specified as AND
# we request the hostname to be changed to m.acme.com


# Main site's origin, default tag of 0 implied
origin 1.1.1.1
origin 1.1.1.2

# Mobile site's origin, notice different network and we specify os tag of 2
origin 2.2.2.1 80 2
origin 2.2.2.2 80 2
```

Same site (www.acme.com), same URL structure, different origins.

Very similar to the setups above. Keep the URLs intact, while specifying different os tags.

Overriding TTL, OS Tag and Host header based on UA tag.

In the 3 scenarios above, you might want to override the TTL for certain UA tags, possibly in addition to selecting different origin servers. For example, when rendering pages for mobile devices, you might want to render the ads into the page server-side and disable caching for such pages, while still caching the same URLs for desktop browsers, as these render ads "client-side", using Javascript.

The TTL is optional 4th parameter in the **ua_url_rewrite** directive. It is to follow the rewrite_to pattern.

The OS_Tag is optional 5th parameter in the **ua_url_rewrite** directive. It is to follow the TTL value so that if you were to request an os_tag, make sure to the specify the TTL even if you were to repeat the pattern's default TTL value.

The substitution host name is optional 6th parameter in the **ua_url_rewrite** directive. It is to follow the TTL and OS tag values so that if you were to specify a sub hostname, make sure to the specify the TTL and OS tag even if you were to repeat the pattern's default TTL value and the default OS tag of 0.

For example:

```
# Default caching of 30 secs
pattern /USnews.html simple 30
ua_tag_process

....
ua_url_rewrite mobile .+ /render?id=123 10 0 m.acme.com
# But modify to 10 seconds when serving mobile users, use default os tag of 0
# and rewrite the hostname to m.acme.com
```

Dropping requests based on UA tag.

To drop requests, match UA to special tag of **drop.** Enable processing for the URLs where you want to apply this drop logic, via **ua_tag_process** at pattern level.

Simplify UA tagging with default tag.

You can explicitly tag requests with special tag of *default*. aiCache also implicitly assigns the same tag when it could not obtain a UA match via neither **ua_tag_pattern**s or **ua_tag_file** (matching is attempted in this exact order).

Letting to specify explicit matching to *default* tag might simplify and/or speed up you matching logic. For example, you might want to match known desktop user agents to **default** before continuing to match the remaining patterns.

While aiCache adds the ***tag*** to the cached response signature – and you can observe that by using CLI *inventory* command, the tag of ***default*** is skipped so as not to lengthen the length of the cache signature.

Letting users have a choice.

You can force-assign a tag to a request by using a cookie, this way aiCache won't attempt to obtain a tag based on the value of the User-Agent string and will use the tag value from the cookie instead.

Use **ua_tag_cookie** setting to specify the cookie name. It is up to you when and how to set this cookie, if ever. When setting the cookie, we recommend using the session cookie that expires when user closes their browser. Otherwise the user will be pegged to a desktop or a mobile site for an extended period of time.

You can provide a link saying "Choose main site" and "Choose mobile site" in the page headers. When clicked, custom server-side or client-side JavaScript logic could be then be executed – that sets appropriate cookie value.

While not obvious, you can delegate setting of the UA tags entirely to your server-side or client-side code, if you so desire. Simply set the **ua_tag_cookie** to the value of your choosing and provide matching tag-driven rewriting rules via **ua_url_rewrite** setting .

Modifying UA tag patterns and file content.

aiCache uses low overhead logic to allow for speedy matching of User Agents to tags. Should you need to modify the configuration of either the patterns or exact UA matching, you can do so via usual zero-downtime CLI reload. aiCache will purposefully leak some amount of RAM upon each such reload.

You will recover that RAM back when you perform complete restart of the aiCache.

[Deprecated] UA-driven URL rewriting and rewrite-redirection.

This feature is only available in mobile-enabled version of aiCache. It is deprecated starting with 6.267, use UA tagging instead. When both features are combined, results will be unpredictable.

Occasionally you might have a need for User-Agent-specific URL rewriting. For example when request is coming from iPhone, for ***news.html,*** you might want to redirect such request to ***iphonenews.html***, while requests coming for the same URL from Blackberries, you want to send to ***berrynews.html*** and so on.This is where you can use UA-driven URL rewrites. These are configured in two steps.

First, you need to configure UA rewrites. We're trying to "compress" hundreds of different mobile user agent string into a more manageable and much smaller set (see elsewhere in this Guide for more detailed explanation of this feature). For example:

```
ua_sig_rewr     .*BlackBerry8.*    berry
ua_sig_rewr     .*iPhone.*    iphone
ua_sig_rewr     .*Android.*    android
```

When **ua_sig_rewr** are specified, the rewritten UA string becomes part of response's signature - for cacheable responses. You can see that by using inventory CLI commands.

Next, in the matching pattern section, you can specify a wildcard match string, that is to be matched against the rewritten UA string (in our case, these will be *berry*, *iphone* or *android*) and URL rewrite string. For example:

```
pattern /news.html simple 30
....
ua_url_rewrite iphone /news.html /iphonenews.html
ua_url_rewrite berry  /news.html /berrynews.html
ua_url_rewrite android /news.html /androidnews.html
```

You can also configure a rewrite to an absolute URL by configuring the rewrite string so that the rewritten URL starts with **http:** or **https: .** In this case, matching requests will be ***redirected*** to the specified location.

Please note that both URL match and URL rewrite strings are regexp strings, so you have full power of regular expressions to accomplish complex rewrites, but the rewritten-UA match string is a simple wildcard search string and NOT a regexp string. Of course the UA-driven URL rewrites can only happen when User-Agent is provided in the request header.

As usual, you can use provided **pattest** tool to test your match and rewrite patterns. To see before and after URLs, specify **log_rewrite** setting in global or website section of the configuration file. Please note that cacheable response signature is formed from the original request URL, but it is the modified URL that gets logged in the access log file.

Please note that when you enable both UA-driven URL rewriting/redirection (see below) and use **ua_pattern** settings, explained later in this document, the UA-driven URL rewriting and redirection happens ***first - as in it takes precedence over "plain" URL rewriting*.**

Please take care to not-redirect, by mistake, assorted search Bots and similar. For example, redirecting Google search Bot to a mobile version of your site will result in Google search results for your content pointing to your **mobile** servers - probably not what you'd want.

As a safety trigger of sorts, you can instruct aiCache to *keep* requests whose User-Agent headers are shorter than certain length. To configure this, set **ua_keep_length** value at website level.

Rewriting request's Host header.

Occasionally you might have a need to rewrite request's Host HTTP header before request is sent to an origin server. You can accomplish it via **sub_hostname** website or pattern-level setting. If both are set, pattern-level setting takes precedence.

Let's imagine that your site relies on a third party site, **api.somewhere.com** for some kind of functionality. But you want to send client traffic to a different site/domain, one under your control, **api.acmenews.com**, while applying all of the aiCache benefits to the request/response traffic: caching, real time reporting and alerting.

There are a number of uses you might find for this functionality - mostly in case of emergency, for temporary workarounds, obtaining aiCache benefits with traffic to provider APIs and things of that nature. Please do not abuse this feature for any illicit purposes.

Please note that this is different from CNAME DNS records that sometimes are used for somewhat similar, but different function. With Host rewrite, the **api.somewhere.com** will receive requests with Host header specified as **api.somewhere.com**, whereas with CNAMA aliasing, the destination web site receives the *original* Host header. You might be able to request your API provider to setup their servers/application to respond to **api.acmenews.com,** but it might be a time consuming endeavor or something that is not supported.

In general, when a request's specifies a valid hostname, that is matched to a website, through direct hostname match, matching of one defined **cname** or matches one of defined **wildcard**, it is the request's Host header that is forwarded to origin server, unless **sub_hostname** setting is set.

Please note that for non-cacheable requests, aiCache forwards all and any Set-Cookie headers from origin servers back to the requesting clients. However, if origin servers respond with a *domain-specific* Set-Cookie header(s), these will be ignored by the requesting browsers due to domain name mismatch. In other words, clients might think they are accessing **api.acmenews.com**, while Set-Cookie response header instructs to create a cookie for **api.somewhere.com**. Such Set-Cookie response headers would be ignored by browsers and cookies won't get set. aiCache doesn't rewrite Set-Cookie domains for doing so could create a black hole with most disastrous consequences.

Host-header-driven URL rewriting .

Occasionally you might have a need to rewrite incoming request's URL based on the value of request's header. This feature can be used, for example, to accommodate for vanity URLs that are used frequently to shorten longer URLs or to accommodate for special promotions.

For example, let's imaging Acme.com running promotions for spring product catalog and Nikey running shoes. So ads go out, advertizing spring.acme.com and nikey.acme.com.

Yet in reality, when users go to these sites, we want to redirect:

spring.acme.com --> store.acme.com/catalog.aspx?collectionid=23423&promoid=232

nikey.acme.com --> sportinggoods.acme.com/catalog.aspx?collectionid=57465&promoid=434

aiCache allows to set such vanity sites with ease. Start by creating an aiCache website and specify both spring.acme.com and nikey.acme.com as **cnames** and specify, via **host_url_rewrite** pattern-level settings the desired redirection:

```
    website
    hostname store.acme.com
    cname spring.acme.com
    cname nikey.acme.com
    ....

    pattern / simple 0 # To match all of the URLs

    # Below is a single line !
    host_url_rewrite spring .+
http://store.acme.com/catalog.aspx?collectionid=23423&promoid=232

    # Below is a single line !
    host_url_rewrite nikey  .+
http://sportinggoods.acme.com/catalog.aspx?collectionid=57465&promoid=434
```

As you can see, the **host_url_rewrite** is a pattern-level setting, that takes 3 required parameters:

- a simple (non regexp) match string - to be used to match against request's host header.

- a regular expression that is to be matched against the original URL string.

- another regexp - a substitution string that will replace the request's URL string.

In order for *redirection* to happen, you must start the rewritten URLs with **http://** or **https:// .** As usual, as regular expressions are used in match and substitution strings, you can have very elaborate matching, including regexp grouping and back references.

Normally aiCache issues a “302” redirect. To change to “301” redirect instead, set **redirect_301** flag at the pattern level.

Support for Geo-driven processing of requests.

Introduction to basics of Geo-targeting.

Many businesses have a requirement to modify appearance of content they serve based on geographic location of the requesting user. Some of the typical uses of such geo-aware content serving are customizing pages so that, for example, US visitors would see one version of content, while those coming from elsewhere in the world would receive international version of content.

Another common use that most of us have been exposed to on the Internet, is serving of ads that are appropriate for your location . For example, if there's some sort of a sale promotion in visitor's zip code, an Ad serving system could deliver the relevant promotion ad to you. There's no limit to extent of customization that can be performed based on such geo-targeting.

Geo-locating the requesting user.

The geo-targeting systems need to figure out your (geo) location, before any geo-targeting can take place. Most of the time, a database of sorts is used, that ties your IP to a location. In other words, your IP address is the key into the DB and the result is a record that contains a plethora of geo-information, such as Country, State/Region, City, Postal/ZIP code etc.

The Geo databases of today are incredibly accurate. The accuracy is achieved by using of a number of methods, some of which are obvious and some are quite interesting in their own right. For example, a common technique used is so called triangulation. Having a number of set points (servers) with known locations, a number of pings are issued to a particular IP and based on ping latencies, quite an accurate determination of that IP's location can be made.

Geo Database.

There're a number of vendors that sell Geo DBs and/or provide Geo APIs. aiCache uses an integration to MaxMind Geo DB (www.maxmind.com). Effectively, aiCache contains code that, when so configured, can query MaxMind DB – obtaining the geo information (aka geo-tag) about requests. You can then use the said geo-tag to perform a wide range of actions – in fact, identical to those you can perform with UA-based request processing as described in previous section. You can block, redirect and rewrite requests, change TTL and/or OS servers and so on, described in length later in this section.

It is your responsibility, as aiCache customer, to contact MaxMind directly and secure a right to/purchase their Geo City Database and be in compliance with their License. aiCache is in no way affiliated with MaxMind and cannot provide you with a version of the MaxMind Geo DB.

Effectively, all aiCache requires in the end is a .dat file that you point to via a simple configuration directive. You are also advised to periodically refresh the file with a newer version to keep current.

MaxMind provides freely downloadable "lite" versions of their Geo DB files . Again, it is your responsibility to be in compliance with terms of use for any of MaxMind products.

Configuring aiCache Geo-processing.

You must legally obtain and suitably place, onto your system, both the MaxMind shared library, **libGeoIP.so** and a MaxMind DB file. As a courtesy to our customers, we provide a pre-compiled MaxMind library, as part of aiCache distribution file. You can also compile your own version of the library or obtain one from MaxMind – in order to upgrade to later version or if pre-compiled GeoIP library is not compatible with your Linux distribution.

In order to compile your own library, download the "C" API from MaxMind, untar it on a Linux system, enter the resulting directory and perform the regular "./configure; make; make install"

There's a number of global settings, one of which is mandatory: **geo_dat_file .** It must point to a valid MaxMind City .dat file. For example:

```
server
…
geo_dat_file GeoLiteCity.dat
```

By default, the geo-tag contains only Country information in it. You can tell aiCache to add up to 3 additional components to the geo-tag: region/state, city and postal/zip code. You can configure the total number of components via **geo_parts** global directive, must be between 1 (country only) and 4 (all four components).

The unmodified geo-tag has following format:

COUNTRY;REGION;CITY;POSTAL_CODE

For example:

US;NY;New York;10001

US;;;

US;NY;;

As you can see, you can end up with a great number of different geo-tags and much like with UA-driven request processing, you will most likely want to "compress" this number down to a much smaller amount. For example, you can decide to tag all of the users in the US as "USA" and the rest of the world as "WORLD".

Should lookup of the GeoIP database return no results, you can assign a default geo tag by using **geo_default_tag** website-level setting.

To perform such reduction of geo-tags you need to use **geo_tag_pat** (similar to **ua_tag_pat**). In the example below we say that whenever the geo-tag contain US, we will reduce it to USA. All other geo-tag values will get reduced to WORLD.

aiCache doesn't apply geo-processing to all incoming requests, you must configure it at pattern level by setting **geo_tag_process** flag.

```
server
…
geo_dat_file GeoLiteCity.dat
geo_parts 4

website

hostname news.acme.com

geo_tag_pat US USA # if full geo tag contains "US", rewrite it to USA
geo_tag_pat .  WORLD # everything else gets reduced to WORLD

geo_default_tag USA # if geo lookup fails, assign this default tag

pattern / exact 120
geo_tag_process
geo_url_rewrite USA .+ http://us.acme.com # When Geo Tag matches USA
geo_url_rewrite WORLD .+ http://world.acme.com # When it matches WORLD
```

You can then configure aiCache to perform various actions based matching of geo-tag, via **geo_url_rewrite** pattern level setting. The example above tells to rewrite-redirect incoming request for "/", when US-based visitor is detected, to http://us.acme.com .

The **geo_url_rewrite** directive has the following format:

```
geo_url_rewrite EXACT_GEO_TAG URL_MATCH URL_REWRITE [TTL] [OS_TAG]
```

The GEO_TAG must be an *exact* match to the detected geo-tag. The URL_MATCH regex pattern is their so that you could capture selective portions of the URL and use them in the URL_REWRITE regex pattern.

Should the rewrite pattern start with http or https, aiCache will perform rewrite-redirect (302 or 301, you can configure which way you want it), as opposed to doing an internal URL rewrite.

The TTL and OS Tag are optional – you can set either or both. Should you provide OS tag, make sure to specify the TTL value, even if it is the same one already set at the pattern level.

Note that we didn't specify the **geo_parts** setting, so that only requestor's Country is identified, the remaining 3 fields (region, city and postal code) are blank.

aiCache, upon successfully geo-matching incoming request, forwards the resulting, possibly reduced geo-tag to origin server via a request header. You can configure it via **geo_header_name** global-level setting. It defaults to X-Geo-Tag.

It will also forward the "full" version of the geo tag to the origin server via a request header. You can configure it via **geo_complete_header_name** global-level setting. It defaults to X-Geo-Full-Tag.

Likewise, aiCache issues geo cookies back to the requesting user. You can configure it via **geo_cookie_name** and **geo_complete_cookie_name** global-level setting. These default to **geo_tag_cookie**

and **geo_full_cookie** respectively. These are only returned for non-cacheable responses in order to avoid caching them.

Both sending of the geo tags via header values to the origin server and returning them to the client, via cookies, only happens for 0 TTL requests or, for cacheable requests , when geo tag is used as part of the cache signature. This way both the origin servers and the requesting clients are notified about geo tags in "safe" fashion: only when aiCache is told that response may vary based on the geo tag, will the geo tag be sent. Otherwise, origin servers may vary output based on the geo tag , which aiCache would then cache without taking the geo tag itself into account, which would be wrong behavior. Likewise, tagging clients with such cookies, could result in same erroneous behavior unless geo tag is used in the signature.

When **geo_cookie** cookie is set in incoming request, aiCache skips the DB lookup and instead, uses the cookie value as geo-tag. This way, you can affect processing by setting the geo-tag via server or client-side code, by setting this cookie to appropriate value.

You can force aiCache to ignore this cookie and do a "hard" DB-lookup to determine user geo-tag, by setting **geo_cookie_override** global level setting.

Overriding TTL and OS Tag based on Geo tag.

You might want to override the TTL for certain Geo tags, possibly in addition to selecting different origin servers. The TTL is optional 4th parameter in the **geo_url_rewrite** directive. It is to follow the rewrite_to pattern.

The OS_Tag is optional 5th parameter in the **geo_url_rewrite** directive. It is to follow the TTL value so that if you were to request an os_tag, make sure to the specify the TTL even if you were to repeat the pattern's default TTL value.

For example:

```
# Default caching of 30 secs
pattern /news.html simple 30
geo_tag_process

....
geo_url_rewrite USA .+ /render?content=123 10 5
# Rewrite the URL  and set TTL to 10 seconds when serving US users, set os tag of 5
```

Dropping requests based on Geo tag.

To drop requests, match geo tag to special tag of **drop.** Enable processing for the URLs where you want to apply this drop logic, via **geo_tag_process** at pattern level.

Modifying request's cache signature using geo-tag.

You can set aiCache to append the geo-tag as an additional component to the cached response signature by setting pattern-level flag of **geo_tag_sig**

This way, for example, you can have US version of home page cached separately from the same page for other users. Or you can store cached NY content differently from the one for FL users. It implies that origin servers/code would render the same request URL differently based on the value of geo-tag header, as forwarded by aiCache. Exercise common sense when using this feature.

Geo and mobile tag processing order .

Should you configure both geo and mobile tag processing for a given URL (pattern), the geo-tag processing takes precedence and happens first. Should geo-tag processing result in rewrite-redirect, the mobile logic never sees the request.

Should geo-tag processing not act(rewrite-redirect etc) on the request, the mobile logic will next have a crack at processing of the request. You can see that this can lead to some complicated decision making so exercise common sense when setting it up.

You can, as an extreme example, have origin servers serve custom content based *both* on geo and mobile tags. So, desktop users in US will have one version of content, different from one served to iPad users in US and so on. Effectively, origin servers would need to act both on UA and Geo Tags as forwarded by aiCache, while aiCache would modify the signatures of cached responses by including *both* tags into the cache signatures. Troubleshooting this kind of setup won't be much fun and you need to control this complexity based on your comfort level.

Testing Geo-processing .

Basically, it boils down to "tricking" aiCache into processing your requests as if they were coming from different parts of the world/country etc and then observing if the geo rules you've established function to your liking.

There're two basic ways to accomplish that:

- Use proxies – that would relay your requests to aiCache so that aiCache sees the proxy's IP address instead that of your own. There're a number of free proxies around around the world, you can look these up using your favorite search engine

- Use **hdr_clip** (described in detail elsewhere in this Guide) setting. It allows you to forward client IP as a request header. With this setting in effect, you can then use a tool such as *ab* or *wget* to send requests with varying client IPs

Lastly, use **log_rewrite** setting to see how aiCache processes your requests.

Configuring Client-to-Origin Server Persistence.

Please note that OS persistence is not recommended for use with DNS-defined origin servers. If you choose to use it, unpredictable results might ensue. Use of origin servers defined by IP addresses is recommended when you need to use OS persistence.

Some sites might have a need to "pin" clients to specific origin servers. For example, a client A might need to be pinned to origin server 1, client B to origin server 2, while client C might be served by any available origin server.

The reasons for such requirement may vary but a common theme to them is a notion of "session" and "session state". Let's imagine an E-Commerce site where a visitor is in process of populating a shopping basket with assorted items. If such a site is served via a number of origin servers, then there must be a provision that all of the origin servers that such client might access during a shopping visit, know of all the items that are placed into the shopping basket. Otherwise, as different origin servers are accessed, items will seem to disappear and reappear at random, a most frustrating situation for a shopper, as you can imagine.

Let's agree on some tech speak. We state that such shopper creates a *session state* as soon as (s)he logs in and/or when first item is placed into the basket. The *session state* includes basket items. And as long as shopper in our example is free to move between different origin servers, the *session state must be somehow replicated* between all of them so it is available on any of them. Now, doing it reliably and quickly is no easy task - with significant implications both on site's infrastructure, site's code and bottom line. [9]

Fortunately, there's a much easier way to address this dilemma - we simply make sure that after a certain activity takes place, the user is *pinned* to an origin server. Another common name used for this technique is *origin server persistence or sticky connections*. When used, a given visitor always connects to a single origin server. Other visitors connect to other origin servers, so in the end the load is still evenly distributed across all of the origin servers.

To configure such visitor pinning (origin server persistence) simply specify **os_persist** setting in *website* section(s) of the configuration file. Please do not combine this setting with **leastconn** or any other non-default load-balancing metrics, as it will send most traffic to one of your origin servers !

If the pinned/requested origin server later becomes unavailable - for example it is disabled after failing a health check, or the request fails when sent (and optionally retried) to the requested origin server, a new origin

---

[9] Sometimes the session state is shared through a database or some other form of centralized persistence mechanism. But most of the time developers would simply use Session API that is available with most common frameworks, Java, PHP etc - where the session is tied to a cookie (JSESSIONID etc) and the session state is kept in-memory and is local to a given server.

server is selected via the regular, simple round-robin selection process and client is notified of the new selection.

In case of such origin server failure the existing session state is lost - requiring re-login, re-population of shopping basket etc. But origin server failure is an infrequent event and such loss of session state is a small price to pay for the resulting simplification of the overall setup and not having to replicate session state across origin servers.

aiCache reports number of such "pinned" requests, served by each origin servers, per 5 second interval. The report is available via all three main methods: CLI, SNMP and Web statistics.

Please note that you might see some fairly interesting scenario when you add an origin server and OS persistence is enabled. What you're likely to see is that even when one or more of origin servers are added to a farm that has OS persistence enabled, these new servers don't appear to be getting much of traffic after the change takes effect (aiCache is restarted). The situation will correct itself over time.

You might find yourself in a situation where you want to disable certain URLs from pinning the requestor to an origin server. To do that, simply create a matching pattern and tag it with **disable_persistence** flag. For example:

```
hostname login.acme.com
....
pattern /password_test.aspx simple 0
disable_persistence
....
```

Please note that when a request matches a pattern with **os_tag** specified, the **os_tag** takes precedence. To better understand this, let's imagine 2 successive requests sent from a browser: one for /a.html and second one for /b.html. The request for /a.html is sent (forwarded by aiCache) to an origin server.

With **os_persist** set, the same origin server would be then chosen to fill the request for /b.html. However, should /b.html have its own **os_tag** set, that os_tag takes precedence and a new origin server, with a matching os_tag, is chosen to fill the request for /b.html.

An alternative way to enable origin server persistence is to use *session cookie* feature of aiCache, described elsewhere in this Guide.

Please note that when an origin server be chosen due to use of **os_tag**, users will not be pinned to such origin server, even if **os_persist** is specified for the matching website. In other words, such patterns will effectively have their **disable_persistence** flag set by aiCache.

Assuring OS persistence in mixed HTTP/HTTPS setups.

Lets imagine news.acme.com is accessed via both HTTP and HTTPS protocol. You need to have OS persistence setup so that after a user is persisted to a an origin server, the same origin server is used for both

HTTP and HTTPS request. For example, after accessing **http://news.acme.com/**, user is redirected to **https://news.acme.com/loginform.html**, then user's login name and password are posted to **https://news.acme.com/login_verify.php** - which upon successful login shall redirect user to **http://news.acme.com/home.php**. You need to ensure that all of these URLs are obtained from the same origin server (same OS IP).

There're two different ways to assure that and we shall describe them here.

The easiest way to assure that is to have aiCache terminate the HTTPS traffic while forwarding the requests to origin servers over HTTP. This will assure that session stickiness persists for both HTTP and HTTPS requests.

Another way is to setup 2 different accelerated sites in aiCache and use aiCache's **sub_hostname** , **match_http_only** and **match_https_only** website settings. For example:

```
...
listen http * 80
listen https * 443 news.cert news.key AES256-SHA:RC4-MD5
...


website

##################################### "Plain" site
hostname plain_acme
cname news.acme.com
match_http_only
os_persist
sub_hostname news.acme.com
...
...
origin 1.1.1.1 80
origin 1.1.1.2 80
origin 1.1.1.3 80

######################################## Secure site
hostname secure_acme
cname news.acme.com
match_https_only
os_persist
sub_hostname news.acme.com
use_os_https
...
...
origin_https 1.1.1.1 443
origin_https 1.1.1.2 443
origin_https 1.1.1.3 443
```

Notice how we setup two accelerated sites, one configured for HTTP traffic only and the other one configured for HTTPS traffic only. We made sure we specify origin servers in the same IP address order.

Please note that you can also use **match_http_only** and **match_https_only** settings in pattern sections, effectively making patterns protocol specific.

For convenience reasons we also gave these site make-believe names of *plain_acme* and *secure_acme* . This way you can see these two sites as two separate and easy to distinguish entities in aiCache reporting screens and can manage them separately from each other. To make sure aiCache doesn't send these fake names as values of Host HTTP header, we have configured aiCache to replace the Host header with news.acme.com.

With this simple technique we have assured that OS persistence will work in the fashion that we require.

Origin Server tagging - selecting origin servers based on request's URL.

You might have a need to be able to select origin servers based on request's URL. For example, you might need to make sure that requests containing **login.jsp** are only sent to origin servers 1.1.1.1, 2.2.2.2, 3.3.3.3, while requests containing **search.jsp** need to go to origin servers 4.4.4.4 and 5.5.5.5.

Possible reasons for us of this technique include server partitioning (aka freshly minted *sharding*) - where different user accounts reside on and are served by, different origin servers and/or database servers. Or you may need to configure all POST requests to go to "write" servers, while read requests are directed against "read" servers. Another example: you can send all users with premium accounts to a one farm of servers, while relegating less fortunate visitors to lesser servers and so on.

Different example might include a situation when you're adding new functionality to your site, to be served by a new server farm (and may be, completely different framework) and you want to make sure that requests are seamlessly and properly routed out across your existing and new servers.

To get such request-driven-origin-server-selection to work, aiCache allows to specify *origin server tag* as an optional pattern attribute. You can also tag your origin servers with *origin server tags*. Now, when request matches a pattern and that patterns specifies an origin server tag, aiCache make sure request are only sent to origin servers that have matching tags.

You configure pattern's origin server tag via **os_tag** directive in pattern section of the configuration file. Each pattern can have a single origin server tag defined. For example, we define a simple, 0 TTL pattern containing .AAA and request that matching patterns are sent to origin servers that are tagged with 5:

```
pattern .AAA simple 0
os_tag 5
```

You configure origin server's os tag by adding the tag number as last parameter, following origin server's IP address and port number. For example, here we define 3 different origin servers, all tagged with 5:

```
origin 1.1.1.1 80 5
origin 1.1.1.2 80 5
```

```
origin 1.1.1.3 80 5
```

You can also configure aiCache to send only a fraction of matching traffic to tagged OS servers, using **decimate_os_tag** directive. When specified, matching requests are to be filled, in decimated fashion, only from origin servers that have matching origin server tag. Think of it as a way to bleed a small, controlled portion of traffic to the matching OS.

```
pattern .AAA simple 0
os_tag 5
decimate_os_tag 10 # Send only 10% of traffic to the tagged OS.
```

For example, when set to 10, every tenth request (10%) will be filled from an OS with matching OS Tag, all other matching requests will be filled from "regular" OS.

Please note that in order to tag origin servers, you must provide origin server port number, even if it is the default HTTP port number of 80. If you don't, then the tag number can be mistaken for port number and your setup will fail to work. Tag value must be under 254.

The patterns can specify any TTL, including 0 . In other words the responses don't have to be cacheable. Again, you have full power of patterns, both simple and regexp at your disposal so you're only limited by your imagination.

Please note that os_tag of 100 has a special meaning - it is used to mark an OS server or servers as servers of "last-resort" . See next section for more information on this feature.

Additionally, should an origin server be chosen due to use of **os_tag**, users will not be pinned to such origin server, even if **os_persist** is specified for the matching website. In other words, such patterns will effectively have their **disable_persistence** flag set by aiCache.

Origin Server of last-resort.

Some site resort to having a "disaster-recovery" read-only version of their sites maintained (for example, some sites use periodic, controlled, web-crawling of main sites), so that in case of a catastrophic failure with their main hosting infrastructure, user traffic could be served off such content replica.

To configure such origins of last resort (OSLR), simply create one or more of origin servers with os_tag of 100. When aiCache fails to fill a request from regular origin servers and, in case of cacheable request, no stale

version of cached content is available on aiCache servers, it will attempt a fill again an origin server of last-resort.

Please note that when website health checks are configured, aiCache will perform regular health checks against such origin servers of last-resort, so they must be able to properly respond to the health checks, otherwise aicache will fail these origin servers just as it would any regular origin server failing a health check.

Detecting a failed LR OS in timely fashion is beneficial, as it allows aiCache to deal with this condition in proper fashion, as opposed to trying to direct requests at a bad server.

Origin servers defined with os_tag of 100 are only used as OS of last resort and are not used to deliver any traffic under normal operating conditions.

When configuring aiCache with OSLR, make sure you also configure "regular" origin servers. If you only define OSLR, without defining any regular OS, aiCache will start and/or restart normally, but the behavior will be unpredictable.

Cookie-driven Caching Control.

Quite often, a Web page's content is affected based on whether a cookie is present/set in web request. This mostly happens when a *registered* user logs in into a web site. Now content of a web page might be different for such logged-in user, as opposed to other, "anonymous" visitors. For example, a personal greeting can be shown on top, some other content might be affected by user's preferences etc. In other words, while a given page looks the same for all the anonymous users, it might look different for those who log in.

With Cookie-driven freshness control aiCache allows you to have your cake and eat it too. You can cache a page for most visitors (such as anonymous users in the example above), or serve it from origin servers every time for other visitors (such as registered/logged-in ones in the example above), all based on whether a cookie is present in the request.

Configure a matching pattern as usual, specifying a non-0 TTL pattern for some URLs. The only difference is specifying an additional parameter in the pattern section: **0ttl_cookie** (that's a number '0'). When this cookie matches a cookie set in client request, the TTL for the document will be set (overridden) to 0. For example:

```
0ttl_cookie userid
```

For anonymous users, that are not logged in and thus don't have that cookie set, the page will be cached, with all the regular benefits of caching. Please note that you can specify multiple **0ttl_cookie** settings, presence of any of them in request will result in matching pattern's TTL being reset to 0.

For example on a blog site, most users are anonymous browsers of blogs. They will enjoy fast response times and origin servers will certainly appreciate corresponding reduction of traffic. Yet, as soon as blogger logs in, certain URLs can be made non-cacheable, so that the blogger will see "EDIT", "POST" links and such. All we need to configure such setup is to figure what cookie gets set when a user logs in and configure that cookie as a **0ttl_cookie** for certain patterns.

In some cases, you'd like to look not for a certain, static cookie name but for a certain pattern in the request's Cookie header. For example, Drupal might use semi-random session cookie names, so that you cannot use **0ttl_cookie** setting. But if you know that all of the cookie names or values match certain pattern, you can then configure aiCache to look for such patterns in request's Cookie headers by using **0ttl_cookie_pat** setting. For example:

```
0ttl_cookie_pat SESS.+=  # Match any cookie name that starts with SESS
```

The opposite of **0ttl_cookie** is **cache_cookie** pattern-level setting. When set, it indicates that request is to be considered cacheable only when the specified cookie is present. For example:

```
cache_cookie jscapable
```

Here's how you could use this feature. A large number of mobile devices (cell phones, PDAs etc) are capable of Javascript, but have it disabled at the factory. So even when you know requesting device type, make and model, you can never be quite sure if the requesting device is enabled for Javascript.

Yet whether or not a device supports Javascript is frequently the difference between being able to cache content and not being able to do that. For example, JS-enabled devices can render the ads client-side, so you don't have to run the ad rendering logic server-side. As a result, the page content for such JS-enabled can be cached.

In order to find out if a device is truly JS-enabled, you can imbed a small JS scriplet in your responses. The scriplet can create a cookie - let's call it "**jsenabled**". Now each and every request from JS-enabled devices will carry **jsenabled** cookie in it. When such cookie is present, you can return cached content, while when the cookie is absent, the request is declared non-cacheable and regular non-cacheable request processing logic takes place.

In server side code, you can tailor the content output so that the code delivers cacheable or non-cacheable versions of content, by relying on aicache **httpheader** header (remember these are the headers that are always sent to origin servers when aiCache is requesting cacheable response).

Such use of **cache_cookie** is not limited to accommodating of mobile devices, feel free to use it for any other purpose you can think of.

Content-driven Caching Control.

You might have a need to control whether a not a web page is cacheable, based on page's (response) content. For example, **www.acmenews.com/breakingnews.aspx** web page is normally cached for 10 seconds, unless editorial team decides to publish a survey (poll) on that page, in which case you cannot cache the page for as long as the poll is active. As soon as poll is removed from the page, you can restore the caching back to 10 second.

aiCache offers support for this kind of scenarios via so called "Content Driven Caching" or CDC for short. Here's how it works.

First, you define a cacheable pattern, just as usual. Then you add one or more of **cdc_pattern** pattern-level settings under the matching pattern, specifying regular expression match strings. With these specified, aiCache will analyze response bodies, looking to match the response body to a defined **cdc_pattern**.

Should a match be found, aiCache temporarily overrides TTL for the matching response and sets it to 0. Effectively, the matching web page is declared non-cacheable. Periodically, aiCache will attempt to match the page's content again and should no match be found, the page will have its TTL restored back to the one specified by the pattern. You can control how frequently such-rechecking is done by setting **cdc_interval** pattern level setting, it defaults to 5 seconds.

```
website www.acmenews.com
...
pattern breakingnews simple 10
cdc_pattern acme\spoll
cdc_pattern acme\ssurvey
```

In the example above, we match response's body (content) to see if contains "acme poll" or "acme survey" in it. Should a match be found, aiCache will declare the page non-cacheable and no longer serve it out of cache. Sometimes, you might find it easier to match for auxiliary content URLs instead. For example, you might know that every time a poll is published on a page, the poll's Javascript is included into the page so you can look for that Javascript URL instead. For example:

```
cdc_pattern userpoll.js
cdc_pattern usercomment.js
```

Matching for such JS "includes" might be less resource intensive, as they are often located at the very beginning of the page's HTML, so aiCache can find the match faster.

As long as you know the likely location of the **cdc_pattern** in the response body, you can give aiCache another hint, via **cdc_bytes** setting (specified under the same pattern). When specified, aiCache will only analyze first **cdc_bytes** bytes of response, as opposed to the whole response body, for possible match to **cdc_pattern**. This can save significant amount of overhead, especially when response bodies are fairly large.

Let's consider another situation. Acmenews's editorial team might decide to publish polls or enable comments, in any of the following pages: **usnews.jsp, worlnews.jsp, marketnews.asp** and **cenews.asp**. As you can see, all of these URLs have a common "news.jsp" component to them, so you can create a single pattern to cover all of them:

```
website www.acmenews.com
...
pattern news.jsp simple 10
cdc_pattern userpoll.js
cdc_pattern usercomment.js
```

Now all of the mentioned pages will have their content analyzed for CDC-overrides, independently of each other. In other words, **usnews.jsp** might end up in CDC-override state, while **worldnews.jsp** is still served from cache.

The handling of page with the CDC-overridden TTL is no different from the way 0TTL requests are normally handled by aicache. Specifically, all of the cookies are sent both to OS and back to the requesting client.

When page is in "regular", cacheable state, aiCache only attempts to match the response body against the CDC patterns when the page is refreshed, so we recommend you keep TTL for such pages low enough, so aiCache can detect the change in the page's content quickly enough. As mentioned earlier, when in "CDC-override" state, the matches are performed every **cdc_interval** seconds - every 5 seconds by default.

When aiCache needs to perform content matching for CDC patterns, it requests the response in plain, non-compressed form. This way no CPU cycles need to be spent by origin servers to compress the response and by aiCache to un-compress it, before matching could be performed. After such analysis and before being sent to the requesting client(s), the response will be compressed by aiCache, in accordance with on-the-fly compression settings and client browser indicating support for compression.

When dealing with large volumes of traffic, enabling **cdc_pattern** processing might increase system load. We suggest increasing number of workers to better utilize all of the available CPU cores. For example, having 8 CPU cores and only 2 workers, the aiCache might be constrained due to low number of workers, so you might consider increasing it to match number of available cores.

Content-driven request fallback or retry control.

You might have a need to safeguard against intermittent origin server failures where a bad response body is occasionally sent in response to a cacheable request. The only indication of bad body is something in the body itself (as opposed to connection error, wrong response size or a bad response code – aiCache guards against those via different mechanisms, described elsewhere in this Guide).

In this case you may use **fb_pattern** setting. One or more of these regular expression match patterns can be specified per pattern. Upon obtaining of a response, aiCache will match response body to each defined **fb_pattern** and should a match be found, aiCache will attempt to fallback to previous, stale copy of cached response, assuming such stale cached response is available.

As long as you know the likely location of the **fb_pattern** in the response body, you can give aiCache another hint, via **fb_bytes** setting (specified under the same pattern). When specified, aiCache will only analyze first **fb_bytes** bytes of response, as opposed to the whole response body, for possible match to **fb_pattern**. This can save significant amount of overhead, especially when response bodies are fairly large.

When aiCache needs to perform content matching for FB patterns, it requests the response in plain, non-compressed form. This way no CPU cycles need to be spent by origin servers to compress the response and by aiCache to un-compress it, before matching could be performed. After such analysis and before being sent to

the requesting client(s), the response will be compressed by aiCache, in accordance with on-the-fly compression settings and client browser indicating support for compression.

When dealing with large volumes of traffic, enabling **fb_pattern** processing might increase system load. We suggest increasing number of workers to better utilize all of the available CPU cores. For example, having 8 CPU cores and only 2 workers, the aiCache might be constrained due to low number of workers, so you might consider increasing it to match number of available cores.

Please note that the same basic mechanism of matching of response bodies against FB patterns can be used to determine when a **non-cacheable** response is bad and request a re-try. The only change would be declaring the URL pattern as 0 TTL.

URL-triggered Cache Freshness Control.

Sometimes, you won't be able to use cookie-driven cache freshness control. It can happen when a session cookie is established even before user logs in - so using such session cookie as cache busting indicator is not possible.

Yet another scenario could involve an e-commerce site - where a user can place an item into a shopping basket, without logging in first. Again, you might not be able to use a presence of a cookie in the request as cache busting indicator.

aiCache allows to account for these scenarios with a ***URL-triggered*** cache busting setup. You effectively tell aiCache: after a user visits certain URLs, I want to disable caching for some of otherwise cacheable URLs. For example, after **basketAdd.jsp** is visited, you want to disallow caching for product pages: **showProduct.jsp**, as these will now have display of content of shopping basket in them.

You can configure one or more of such cache-busting-triggering URLs via **0ttl_url** settings at website level. The setting takes a single parameter, that is used to perform partial matching against request URL. For example:

```
0ttl_url  basketAdd.jsp
0ttl_url  login.jsp
```

Now, when any of these links are visited, aiCache sends back a cookie: **aicache0ttlcookie,** set to **yes**. You can change the cookie's name to a different name if you like, by using **0ttl_url_cookie** setting, at website level.

After aiCache sends the cache busting cookie back to the client, the processing logic is similar to regular cookie-driven freshness control. You configure a matching pattern as usual, specifying a non-0 TTL pattern for some URLs. The only difference is an specifying an additional parameter in the pattern section: **0ttl_cookie** (that's a number '0'). When this cookie matches a cookie set in client request, the TTL for the document will be set (overridden) to 0. So this is what you'd add under related pattern, assuming you have not changed the default cookie name of **aicache0ttlcookie**

```
pattern showProduct.jsp simple 120
0ttl_cookie aicache0ttlcookie
```

Now for those users that have not visited any of the triggering URLs and thus don't have that cookie set, the pages are cached. Users that have visited any of the triggering URLs, will have caching disabled for any of the patterns that have matching **0ttl_cookie** setting specified.

Allowing Cookie pass-through for cacheable responses.

By default, aiCache doesn't allow any Set-Cookie headers to be stored in cached responses in order to safeguard from potential sharing of private data. So when a cacheable response, coming from origin servers, contains one or more Set-Cookie HTTP headers, these are always filtered out and it is the "sanitized" response header that is cached. Again, it is done on purpose as cookies are frequently used to "personalize" responses, might act as pointers to HTTP(S) session information and/or directly store private user data.

However, if a particular setup requires some cookies to be allowed to pass from origin server and into the *cached* response, you can enable it via **pass_cookie** setting, under proper URL matching pattern. You configure a matching pattern as usual, specifying a non-0 TTL pattern for some URLs. Then you provide an additional parameter in the pattern section: **pass_cookie**.

```
pass_cookie  lastvisit
```

Now, for cacheable responses, the matching cookie ("*lastvisit"* in this example) will be allowed to pass into the cached response. Multiple **pass_cookie** directives can be specified per pattern.

Please note that 0-TTL responses (non-cacheable responses) always pass all of the cookies (if any are set) from origin servers back to the requesting browser, so **pass_cookie** directive is only applicable to ***cacheable*** responses.

If you have to use this setting, please make sure not to enable caching of private user data by mistake. For example, allowing caching and forwarding/sharing of session ID or user ID cookies is a very bad idea.

Signatures of cached responses.

By default, aiCache uses hostname and URL of request, possibly modified by removing some parameters or discarding the complete query string, and one of "p0","p1","g0","g1" suffixes, as a signature for cached responses. However, in addition to these 3 basic parts of any cached responses signature, more optional components could be added - including a Cookie (**sig_cookie**), User-Agent string (**sig_ua**) or a rewritten User-Agent string, value of an arbitrary request's header (**sig_header)** and last, request's Accept-Language header.

To reiterate - every cached response has a signature. The signature will always have at least 3 parts to it, separated by a single white space:

- hostname

- URL : path+ optional (possibly modified) query

- "p0" or "p1" for plain, non-compressed responses and "g0" or "g1" for gzip-compressed responses.

You can see the cached response signatures in clear, enclosed within angle brackets **(>signature<**), when you run any of CLI "inventory" commands. In order to affect cached responses (for example, expire them), you need to specify either an exact signature of a cached response or a matching pattern. You can obtain exact signature by copying and pasting the string within the angle brackets, as output by any of the CLI *inventory* commands.

For example, **/home.html**, when cached in non-compressed form, for an accelerated domain of aaa.bbb.com, could have a complete signature of (as displayed by \ CLI inventory command):

```
>aaa.bbb.com /home.html p1<
```

As you can see, aiCache's default behavior is to store separate versions of cached responses for HTTP1.1 and HTTP1.0 requests (the p1 vs p0 and g1 vs g0 signature components). However, in some scenarios you

might know for a fact that there are no differences in responses from origin servers, across both HTTP versions. In such case you might find it beneficial to tell aiCache to store unified cached responses, where same cached response, plain or cached, is used for both HTTP1.1 and HTTP1.0 request. To accomplish this, please set **unified_cache** at global level of the aiCache config file. The setting is a flag and requires no value.

```
unified_cache.
```

Doing so can significantly decrease size of RAM used by cached responses (as we don't have to store 2 versions of same content), help increase cache hit ratio (as both request types are now fed the same response) and decrease traffic to origin servers.

By default, the hostname part of the signature comes from **website** setting value, so that no matter the hostname value of the request's Host header, as long as it matches the **website** setting value or any of possible **cname** values, the cached responses will all have the same signature, for the same URL and thusly only one copy of the shared response exists.

For example, with the following configuration:

```
website www.acme.com
cname a.acme.com
cname b.acme.com
```

requests for www.acme.com/1x1.gif, a.acme.com/1x1.gif and b.acme.com/1x1.gif will be all cached as a single cached response. This is the default behavior and mostly likely is how you'd configure most cached websites.

If you instead desire these 3 URLs to be stored as 3 distinct and separate entries, you can specify **sig_req_host** setting at website level. It is a flag and requires no value. Note that such setup will result in higher memory consumption.

Unifying cached content for different websites.

You might have a setup when a number of different sites are cached via aiCache, for example a.com, b.com and c.com. Yet some of the content on these sites is identical and you'd like to cache(store) only a single copy of it, effectively sharing it across different sites.

For example, let's assume that you want to share URLs that contain /css and /images prefix. To accomplish this, we simply tell aiCache to use a different signature prefix – instead of matching website's hostname, we configure a different prefix via pattern-level **sig_hostname** setting. For example:

```
website a.com

pattern /css simple 1d
sig_hostname css_content

website b.com

pattern /css simple 1d
sig_hostname css_content
```

```
website c.com

pattern /css simple 1d
sig_hostname css_content
```

Now, when a request is made for a.com/css/main.css, b.com/css/main.css or c.com/css/main.css, same cached response will be returned, saving the overhead of having to maintain 3 different copies of the same content.

As usual, make sure that the URLs are cacheable and are not website-specific in any way.

Adding a Cookie value to signature of cacheable responses.

By default, aiCache uses hostname and URL of request, possibly modified by removing some parameters or discarding the complete query string, and a "p0","p1","g0","g1" , as a signature of (pointer to) cached responses. However, some sites might serve different cacheable content in response to requests for same URLs, depending on a value of a cookie present in the request.

For example, if a cookie called "**connection_type**" is set, different responses might be sent in response to request for home page (such as index.html) when cookie is set to "**high**", "**medium**" or "**low**". Or a cookie called "**language**" might be set to "**en**", "**fr**" etc. To enable caching and sharing of such responses, aiCache allows you to use this cookie and its value as part of cache signature. Simply specify, in proper website or pattern section, following setting:

```
sig_cookie  connection_type
```

Now, for cacheable responses, the matching cookie (**connection_type** in this example) and its value are added to the response's signature and proper version of cached response is served in response to requests with different value of the cookie set.

When a response to such requests must be obtained (*first fill* or *refresh* of stale cached response), the appropriate cookie and its value are passed to origin server. This is in contrast to normal handling of *cacheable* requests, when no cookies are allowed to pass from users to origin servers. Such blocking of cookies for cacheable responses is done on purpose, as cookies are frequently used to "personalize" responses, might act as pointers to HTTP session information and/or directly store private user data.

Multiple **sig_cookie** can be configured per pattern or website. You can also combine **sig_ua** (see next section), **sig_language** and **sig_cookie** settings - in which case both the selected cookie value(s), Accept-Language and User-Agent headers are used as part of signature.

Please note that as usual, pattern-level setting of **sig_cookie** overrides (supersedes) website-level setting of the same name, if both are set. When **sig_cookie** is not detected in cacheable request, the request cache signature will look as if no **sig_cookie** was specified.

When using this feature, please make sure not to enable caching of private user data by mistake. For example, allowing caching of session ID or user ID cookies is a very bad idea**.**

Adding User-Agent request header to signature of cacheable responses.

By default, aiCache uses hostname and URL of request, possibly sanitized by removing some parameters or discarding the complete query string, and a "p0","p1","g0","g1" , as a signature of (pointer to) cached responses. However, some sites might serve different cacheable content in response to requests for same URLs, depending on a value of a User Agent HTTP header present in the request. As you most likely know already, User Agent HTTP header identifies browser's make and model.

For example, a site serving mobile clients might serve responses whose formatting (content) depends on exact mobile device/browser. To accommodate for such clients, while allowing for caching of responses, we must use User-Agent information as part of cached response signature. So when a URL named "news.html" is accessed by 3 different mobile devices, we shall have 3 different responses cached - each containing mobile device's User Agent string.

To enable such behavior, please specify **sig_ua** setting in website section of the configuration file. All of the cacheable requests willnow have User Agent information appended to their signatures. As using this feature has potential to significantly increase the size of response cache (due to multiple versions of same URLs getting cached), use it only when necessary.

You can also combine **sig_ua** and **sig_cookie** (see previous section) settings - in which case both the selected cookie value *and* User Agent string are used as part of signature. In this case it is the User Agent string that becomes the signature's suffix.

Adding reduced/rewritten User-Agent request header to signature of cacheable responses.

This feature is only available in mobile-enabled edition of aiCache.

It is most desirable to accommodate for different mobile devices in a fashion that doesn't require changing the URLs - so that no matter what mobile device is being used, news page is always accessed as **/news.html**, sports section is always **/sports.html** and so on. An alternative solution where URLs change, say by prefixing every URL with a device type, to accommodate for the device type is clearly less elegant.

Related to the previously discussed feature, this one allows you to rewrite/reduce User-Agent strings to a smaller subset and use the rewritten/reduced values as part of cacheable response signature.

The problem this feature addresses has to do with a great variety of mobile devices currently available. Every firmware revision, different mobile providers/carriers/markets all result in a different User-Agent strings sent by mobile device, making it a challenge to accommodate for all of these devices. Yet at the same time the great variety of mobile devices on the market can be reduced to less than a dozen of distinctly different devices (device families). So Blackberries can be grouped into may be 2 different sets, Android devices into its own set and so on, based on capabilities, support for CSS, Javascript and available screen sizes.

aiCache allows you to accomplish just that. You specify rewrite rules for User-Agent strings. The rewritten/reduced User-Agent strings are then used as part of signature of cached responses.

The same "compressed" User-Agent string is also forwarded as **X-UA-Rewrite** header in requests sent to origin servers. The origin servers (server-side code) can then programmatically access and act on this header value, with a goal of modifying responses to accommodate for mobile device differences - for example resorting to Javascript-free versions of content for devices that don't support Javascript, resizing the images to accommodate for different screen sizes and so on. Of course the server-side code can also tailor the responses based on the actual value of User-Agent string (aiCache never modifies it, it is forwarded verbatim from requesting device to the origin servers) - but then the server-side code has to accommodate for a much larger variety of devices.

Reducing the large number of different User-Agent strings to a much smaller subset also has a positive impact on *caching* of responses - allowing to achieve much higher cache hit ratios and to proportionally reduce the traffic and demands on origin server infrastructure (Web Servers, DB servers etc). It also allows to greatly simplify the logic required to handle the variety of mobile devices available on the market today.

To configure the User-Agent rewrite/reduction, use **ua_sig_rewr** setting(s) in website section of the configuration file. Each of these must have two parameters: the matching pattern and the rewrite string, very similar to how we rewrite the URLs themselves (see elsewhere in this manual).For example:

```
    ua_sig_rewr     .*BlackBerry8.*    berry8
```

reduces all UA strings that contain *BlackBerry8* in it to *berry8* - reducing many different possible UA strings to just one. You can reduce other UA strings in similar fashion by adding more of **ua_sig_rewr** directives.

The **ua_sig_rewr** takes optional third parameter: **0ttl_ua.** Requests with matching User-Agent headers have their TTL reset to 0, when request's URL match patterns that have same **0ttl_ua** flag set.

The use and purpose of this feature are best explained with an example. Let's imagine you have a web site for mobile clients. On that website you have a page: **/news.html** . The page shows breaking news and also serves some ads. When requesting browser (mobile device) is Javascript enabled, you want the *device* to obtain and render the ads via device-side (client-side) Javascript. Yet if mobile browser is incapable of Javascript, you need to render the page server-side and then it cannot be cached.

To accomplish this, we create a number of **ua_sig_rewr** settings, matching the multitude of devices we intend to deal with. We assume that all of iPhone and Blackberries are capable of Javascript:

```
    ua_sig_rewr     .*BlackBerry8.*    berry8
    ua_sig_rewr     .*iPhone.*    iphone
```

Next we declare that any other devices are not Javascript capable, so we want to disable caching for all other device types. We accomplish this by adding **0ttl_ua** to the end of the **ua_sig_rewr** string like so:

```
ua_sig_rewr     .+    non_js 0ttl_ua
```

Now we define a simple pattern for /news.html page and provide 0ttl_ua setting under that pattern's definition:

```
pattern /news.html 60
0ttl_ua
```

Now Blackberry and iPhone customers are served cached version of the page, while all other mobile clients are not allowed to receive cached version of the very same **/news.html** page. Instead, these two are forwarded to the origin servers and have their ads rendered by server-side code. Of course, the example implies that **/news.html** is rendered by the server-side code in a fashion that makes it possible for client-side Javascript ads rendering for Iphone and RIM Blackberry devices, while rendering the ads server-side for ”lesser" devices that do not support Javascript.

A bit more on User-Agent matching patterns: please note that you must use "\s" for white spaces - if you want to match any white spaces in the match string. The replacement string can use back-references to match groups in the match string, using **"\1", "\2"** syntax to reference first, second matched groups and so on.

To assist with pattern testing, aiCache distribution comes with **pattest** command line tool. Here's an example of **pattest** in action:

```
./pattest -p '.*Blackberry\s7.*' -s 'aaaBlackberry 7xxx' -r 'berry7'
[Success]: Rewrote original string:
>aaaBlackberry 7xxx<
 To new string:
>berry7<
 Using pattern:
>.*Blackberry\s7.*<
 And replacement string:
>berry7<
```

When using **pattest**, we recommend enclosing all three of the parameters in single quotes, to prevent possible expansion/interpretation of the special symbols, if any, by shell.

You can also request all of UA rewrites/reductions to be logged into error log file, via **log_rewrite** global setting. Clearly, it is not recommended to enable such logging on a heavy-traffic production site, use it to fine tune the rewriting and then turn it off.

When **ua_sig_rewr** settings are configured for a website, aiCache uses the rewritten User Agent string as part of cached response signature - you will be able to see it clearly and in plain view when running any of the CLI inventory commands (**i, sit, sir, sif, sis**).

Forwarding User-Agent header to origin servers, for cacheable requests.

By default, aiCache strips out cacheable request’s User-Agent (UA) when forwarding such requests to origin servers (for first-fill or refresh). The cacheable response should not depend on the value of UA header,

plus stripping it out saves valuable bandwidth and relieves OS from having to process this header. Should the response depend on the UA, please see the previous chapters on how to configure use of (possibly rewritten) UA in cache signatures.

If you desire to still forward cacheable request's UA to origin server, you can set **forward_ua** flag at website level of the configuration file.

Adding Accept-Language request header to signature of cacheable responses.

By default, aiCache uses hostname and URL of request, possibly modified by removing some parameters or discarding the complete query string, and a "p0","p1","g0","g1" , as a signature of (pointer to) cached responses. However, some sites might need to serve different cacheable content in response to requests for same URLs, depending on a value of a Accept-Language HTTP header present in the request. Accept-Language HTTP header identifies browser's preference for language/locale in which it would prefer to see the response.

Most multi-lingual sites have different sub-domains or URL naming schemes to serve their content in different languages. For example, www.acme.com might have www-fre.acme.com, www-spa.acme.com sub-domains to serve its content in French and Spanish languages. Or, it might have URL naming convention where all of the French content lives under www.acme.com/french/ URL. If that is how your site is setup, you don't need to use the feature we're about to describe.

Yet some sites might use browsers Accept-Language header as an indication of user's preference for language. So when Accept-Language header is set to: "us-en,en" , the site is served in English, "ca-fre;fre" is served in French and so on - while preserving the same URL scheme. In other words, English speakers will see index.html come back in English, while French speakers will see the same URL come back in French. Clearly, if we were to cache such responses, we must be able to discriminate them based on value of Accept-Language header. In other words, "index.html" response in English should be different from the same "index.html" response in French. aiCache accomplished that by allowing you to add the value of Accept-Language header to the signatures of cached responses.

You set it up by specifying **sig_language** in website or pattern section of the configuration file. The setting is a flag and doesn't require any parameters. With this setting in effect, you can observe, via CLI inventory commands, that the response signatures, in fact, have the value of Accept-Language header embedded in them.

Please note that depending on the user's browser and their locale settings, there might be some variation in the values of Accept-Language header, leading to multiplication of cached responses and reduction of cache hit ratios, so use this setting only when necessary.

Adding value of arbitrary request header to signature of cacheable responses.

You can configure aiCache to add a value of an arbitrary request header to the cache signature. To configure, set **sig_header** at website or pattern level. The setting requires a single value: the name of the header whose value you wish to use in the signature. For example:

```
sig_header custom-header
```

As usual, pattern's **sig_header** value, when set, overrides that of website. With this setting in effect, you can observe, via CLI inventory commands, that the response signatures, in fact, have the value of specified header embedded in them. Only one such value is allowed – when multiple are set, only the last value takes effect.

When detected in inbound request, aiCache forwards this header and its value to the origin servers, when refreshing or obtaining first fill for cacheable requests.

As you may have noticed, there appears some overlap between this signature setting and **sig_ua** and **sig_language .** But use of the **sig_header** allows *additional* signature component, even when **sig_ua** and/or **sig_language** are set.

Response-driven Cache Invalidation. [cluster/peer enabled]

This feature is best explained by example. Let's say you have a message board web site where you cache both discussion threads and forum fronts. Yet at the same time, when a new message is added to a thread, you want to have aiCache expire the cached content of respective discussion thread right-away, not waiting for cached content to expire when its TTL runs out. This way the newly added message is seen by visitors as soon as possible and the posters are not confused when the message they've just posted doesn't show up in the thread right away, as they certainly expect it to.

aiCache once again comes to the rescue by allowing you have your cake and eat it too. All you have to do is to send a header called **X-expireURL** in the response to a post, identifying a URL pattern for aiCache to expire. For example, assuming an on-line forum that uses a popular forum software:

```
X-expireURL:  /acme-bb/forumdisplay.php?f=123
```

would expire particular forum page, while

```
X-expireURL:  /acme-bb/showthread.php?t=123456
```

would expire a particular thread page . You can send multiple headers with that name to expire multiple cached responses. Resorting back to the message board example, you might want to expire both the sub-forum page and particular thread – it is up to you.

Do not specify host name or "http://" prefix in **X-expireURL** header, follow the format in the example above.

Java, PHP et al, all have APIs to easily insert such headers into responses. However, in case of Java you might discover that you cannot have multiple header values with the same name (in case you want to expire more than one URL pattern). In this case a little cheating gets the job done: call these headers **X-expir1URL, X-expir2URL** and so on - replacing the "e" after "r" with a number.

Please note that the provided value (the string after **X-expireURL:)** is treated as a pattern. If you need to make sure you only expire a single response, please terminate the pattern with a **\s** (space) - which is always present after the URL, before the "p1", "g1", "p0" and "g0" portion of the signature. In general, being able to specify an expiration pattern, as opposed to exact URL, allows you to expire content with more ease - for example you can expire all of the pages (in a case of paginated URL) with a single pattern.

If you have any *peers* defined, aiCache communicates with each of them, resending the same expire pattern command(s). This way if you have more than one aiCache server accelerating this web site, all of them expire this content simultaneously. For more information please see this section.

You can disable response-driven content expiration at website or pattern level by specifying **ignore_resp_expire** setting at website or pattern level respectively. When this flag is set, aiCache ignores the **X-expireURL** header in responses.

Session-driven content caching. [cluster/peer enabled]

Again we provide an example to best explain applicability and purpose of this feature. Let's say you have a news website - www.acmetimes.com. The site requires users to register and sign-in to view its content. The users that are not logged in can only view the home page.

aiCache allows you to cache such site's content in a fashion that enforces the registration/login policy as described above: the logged in users will be able to see content from cache, while those users that are not logged in, will not be allowed to view cached content and instead, will be subject to your site's handling of such users (normally it means the users are asked to register and sign in to view the content - but again, exact mechanism is completely up to you, aiCache simply forwards requests to origin servers).

Here's how to configure such caching. First, you need to find out what cookie is set when users are successfully logged in. For example, the cookie could be named **sessionid** or **jsessionid**. You also need to know what URL(s) set these cookies. Normally, it would be the URL that lets users to login in. Let's assume the URL is **login.php**.

To identify the session cookie that is used, set **session_cookie** website-level setting:

```
session_cookie jsessionid
```

Your setup might be such that session cookie could be present in either request's Cookie header or embedded into the request's URL. To tell aiCache to attempt obtaining the session cookie value from the URLs, you can set website-level flag setting **session_cookie_url.** When set, aiCache still tries to obtain the cookie

value from request's Cookie header and only when it is unsuccessful, an attempt is made to obtain it from the request's URL.

To specify URLs that are allowed to set session cookies, set **sets_session_cookie** pattern-level flag in a matching pattern. For example:

```
pattern  /login.php exact 0
request_type both
sets_session_cookie
```

Likewise, you may specify URLs that expire session cookies, by setting **expires_session_cookie** pattern-level flag in a matching pattern. This way, when one of such URLs are visited, aiCache will forcefully expire the session cookie by modifying origin server response. A dummy valued session cookie is inserted, set to expire well in the past. For example:

```
pattern  /logout.php exact 0
request_type both
expires_session_cookie
```

Next, you need to define set of patterns, allowing caching for site's content - but specifying that in order for content to be cached and served by aiCache, a valid session cookie is present. To accomplish this, setup the patterns in usual fashion, while providing **session_cookie_required** pattern-level flag.

For example, if you site's content, that requires users to be logged in, has a prefix of **/content/** you can configure the following:

```
pattern /paidcontent/ simple 1m
session_cookie_required
```

You're specifying that any URL that matches **/paidcontent/** is cached for 1 minute, but is to be served to users only when their requests carry a valid session cookie. If no valid session cookie is detected in requests, aiCache simply forwards such requests to origin servers, verbatim, where regular processing takes place. Such processing normally prompts users to register with the site etc, in order to view the content they are interested in.

Alternatively, you can allow caching in both cases:

- When request carries a valid session cookie (i.e for logged-in users)
- When request doesn't have a session cookie (i.e for anonymous users)

by specifying **cache_anonymous** at the respective pattern level like so:

```
pattern /paidcontent/ simple 1m
session_cookie_required
```

```
cache_anonymous
```

With such configuration in effect, aiCache will store 2 different versions of cached responses for the same URL – one for for logged-in users and the other one for anonymous users. To make sure the cached responses don't collide, signature of the former is modified to include trailing "S" character. You can observe such signatures via CLI inventory commands.

aiCache actively monitors responses that match URL marked as **sets_session_cookie.** When a session cookie (Set-Cookie) header is detected in such matching response, aiCache memorizes, in its RAM, that session cookie's value. It also communicates it to all of its peers, so that all aiCache servers in the same cluster are made aware of the new session cookie.

Now, for requests to URLs that match patterns with **session_cookie_required** flag, aiCache checks to see if a valid session cookie is present. If and when it is present, aiCache allows caching of responses to take place, following regular "cacheable request" logic.

If a valid session cookie is not present, the request's TTL is set to 0 - resulting in aiCache bypassing the cache and sending the request directly to an origin server. In other word, the regular "non-cacheable request" logic takes place. Code on origin servers can then prompt users to login or register to gain access to the protected content.

aiCache reports number of entries in its session tracking table via Web, CLI and SNMP interfaces. Periodically, special cleaner logic is executed, removing expired or inactive session entries. Session is considered to be inactive when there were no references to it within 1 hour. aiCache reports the number of sessions that were removed from session table during previous cleaner run via CLI and SNMP interfaces. Each aiCache server in a cluster expires its session cookies independently.

To configure session duration, set **session_duration** global setting to desired session duration in seconds. It defaults to 3600 seconds (1 hour).

Since session state is tracked in via RAM structures and is not persisted to more permanent storage, aiCache loses the session information when *cold-restarted*. Consider using *on-the-fly* reloads instead – see elsewhere in this Guide for more information.

aiCache can also be configured to always forward client requests to the same origin server that originally set the session cookie, by setting **session_cookie_persist** website-level setting. It is a flag, requires no value and is turned off by default. You'd normally use this setting when your origin servers don't have session state replication mechanism in place and you want to pin the client to the same origin server that has created and is maintaining the server-side client state information.

Please note that since we are caching the responses with session cookie set, the responses must be cacheable. Specifically, they should not be tailored/personalized for particular logged in user. Instead, any such personalization should be moved to client-side Javascript logic, as described in Appendix to this manual.

As an extreme case, you can opt to cache content at per-user level - by using **sig_cookie** setting. Set it to the same value as **session_cookie** .

aiCache's support for session cookie is robust and vigorous - it actively monitors session cookie values as returned by particular URLs on origin servers and matches all inbound cookie value to the ones that were served by origin servers.

If your requirements are more relaxed - to where you just want to make sure a cookie with a certain name is present in a request, no matter the value, for the request to be declared cacheable, you can use **cache_cookie** setting instead.

Handling and storing of compressed (gzipped) and plain responses.

Plain (non-compressed) and **gzip'd** (compressed) cacheable same-URL responses are stored as separate entities. In other words, each is a considered to be a separate cached response, with its own refresh and expiration time. For example a gzipped response to a home page request could be obtained at a different time than a plain response for the same URL. Both do, however, share the same TTL.

Additionally, HTTP/1.1 and HTTP/1.0 are also stored separately - see dedicated section on treatment of HTTP/1.0 requests for more information.

aiCache obtains both in a "lazy", on-demand fashion, separately from each other. When a request for a gzip'd version comes in, it is forwarded to an origin server, a response is obtained and cached. No attempt is made to uncompress the gzip'd version, just in case a plain version is needed soon. Instead, when a plain version is requested, a new request is made to the origin servers and returned plain response is cached separately from the gzipp'd version of the response that was obtained previously.

Sometimes you might see responses for the same URL stored in both plain and gzip'd format. Such situation is normally indicative of scripted, non-browser requests being received for such URLs. Some AJAX application might also prefer to receive non-compressed responses.

You can see what responses, plain and compressed, you have in the cache by using inventory CLI commands - see CLI section for more information. The cached responses would have "p1" or "g1" added following the URL, to indicate plain or compressed HTTP/1.1 responses, respectively. Replace "1" with "0" for signatures of HTTP/1.0 responses.

You can disable compression at pattern level by specifying **disable_gzip** setting (it is a flag, no value required). aiCache will not request responses in compressed form and will not compress responses on-the-fly.

On-the-fly Response Compression.

aiCache has ability to compress Web content *on-the-fly*. While it was rare to have this capability in a web server 8-9 years ago, nowadays most web servers support it out of the box. So it is up you to decide *where* you want to compress the responses – right on the origin servers or by aiCache. If you feel that origin servers are already taxed out and could use a break, let aiCache handle it. Otherwise we recommend compressing it at origin server – this way aiCache can cache (fit) more responses in the same amount of RAM, due to much

smaller response size[10]. Also, with responses coming compressed from origin servers, aiCache won't have to spend CPU cycles compressing responses on the fly.

If you decide to let aiCache do the compression, you do not have to modify you origin of web servers in any way or install any software. *Any* web server can have its content compressed, as aiCache is *not* a web server plug-in.

In order to enable on-the-fly compression in aiCache, you have to set **min_gzip_size** for a website where you want to enable this feature:

```
min_gzip_size 2000
```

and aiCache will then compress-on-the-fly responses that have Content-Length (response size minus the header size) of 2000 bytes and more and have Content-Type: of **text/html, text/css and appliction/x-javascript.** No other content types are compressed by default. Compression of HTML, CSS and Javascript responses is known to work across all of the browser types without any issues. Compressing smaller responses (less than 2000 bytes in size) is likely to be counterproductive, as it increases load on the aiCache server without providing any significant reduction in size of responses.

You can also configure aiCache to compress Json and XML responses. Compression of these response types is disabled by default and can be enabled by providing **compress_json** and **compress_xml** directives in website or global sections of the configuration file. After enabling compression, please test carefully, as compressing these response types can cause problems with some browsers.

The responses are only compressed when triggering request indicates support for gzip compression via **Accept-Encoding: gzip** request header.

Please note that on-the-fly compression happens within worker thread, as opposed to getting delegated to another thread. aiCache uses special logic to assure fairness to all connected clients: if a large response body needs to be compressed, aiCache will compress a portion of it, process data for other outstanding clients, return back and compress some more and so on. This way a large response, requiring compression, doesn't unfairly stall other clients.

You can disable compression at pattern level by specifying **disable_gzip** setting (it is a flag, no value required). aiCache will not request responses in compressed form and will not compress responses on-the-fly.

Compression and IE6 (Internet Explorer v6).

aiCache never compresses JS, CSS, JSON or XML responses when replying to requests from IE 6 and IE 5 browsers, as there are known issues with how these browser types handle compressed responses of certain types. Only HTML responses are compressed for these browser types.

---

[10] Significantly large text-based responses, such as HTML, CSS, JS, JSON, XML might compress 5:1 and better, a very significant benefit indeed. Your BW utilization drops, you save money and the end-users, on slower connections, will see responses coming much faster. And the whole Internet, too, is a better place for it. So please do compress your content !

Here's how this logic works. Upon obtaining the request, if URL contains **.css, .js** or **.xml** and browser type is IE 6 or IE 5, compression is disabled and everything is safe. If that's how your website is setup, you don't have anything to worry about, aiCache has you covered and you can skip the rest of this sub-section.

However, sometimes the response content type cannot be easily deduced from the URL and is not known till the actual response is obtained. For example, a request for URL **/node/12345** can return a CSS or JS response, although there is nothing in the URL indicating possibility of such outcome.

Upon obtaining the response, aiCache will be execute IE6 safeguard logic again and if response content type is anything but HTML and the requesting browser is IE 6 or IE5, compression is disabled as well.

You can opt to turn off all and any compression for IE 6 and IE 5 browsers by setting **disable_gzip_ie56** global option. The setting is a flag and requires no value.

When you know that a particular pattern is safe for IE5 and IE6 compression, you can set **enable_gzip_ie56** setting at pattern level. The setting is a flag and requires no value. You will obtain most reliable setup by setting **disable_gzip_ie56** global option and then qualifying particular patterns as safe for IE5 and IE6 compression by using **enable_gzip_ie56** setting at pattern level. It does require a bit of work, but might offer significant benefits in case of large HTML responses - these are best when sent in compressed form (both getting to the clients faster and lowering your bandwidth bills).

As a result of aiCache IE6 safety logic, you might see (via CLI inventory commands) cached responses whose signature includes "ie56" component and be interested as to where this signature component has come from. Don't be alarmed, this is expected behavior.

Tell-tale signs of IE6-related content-compression problems are IE6 users reporting seeing gibberish in their browser windows , seeing content download popup windows when simply browsing an affected site, content formatting issues (due to corrupt CSS) or Javascript errors (due to corrupt JS).

There shall a worldwide celebration when last of IE6 browsers leaves this planet, but for now, unfortunately we still must support these outdated and quirky browser types. aiCache is there, to assist you in this less than pleasant endeavor.

You can disable compression at pattern level by specifying **disable_gzip** setting (it is a flag, no value required). aiCache will not request responses in compressed form and will not compress responses on-the-fly.

Forwarding cache control headers as received from origin servers.

Normally, aiCache overrides and overwrites all and any HTTP cache control headers, as received from origin servers, with its own. Some of such headers are "Cache-Control", "Pragma", "Expires". If you desire to instead have aiCache not do that and *forward* these headers verbatim, as received from origin servers, please use **forward_os_cache_headers** setting, at website level. For example:

```
website
hostname news.acme.com
forward_os_cache_headers
....
```

Handling of certain response headers with body-less responses.

When a response, as received from origin server, has no body, aiCache strips both Content-Type and Content-Encoding headers from such responses, if any are present, before responses are sent to client. This is done to avoid confusing certain clients that might expect a response body upon seeing such headers. Additionally, not sending these headers under such conditions also saves precious bandwidth.

Normally, when a body-less response is to be delivered to clients, aiCache doesn't insert **Content-Length: 0** header, as responses without such header are assumed to be body-less by most clients. Alternatively, you can configure aiCache to explicitly insert **Content-Length: 0** header into body-less responses by specifying **send_0_cl** flag (requires no value) at website level.

Handling of POST requests with Expect header.

You might see POST requests that carry "Expect: 100-continue" header. Such header normally is used by client to see if a server is willing to accept a large POST body. Only upon seeing "HTTP/1.1 100 Continue" response will client continue with the request and send the actual POST body.

Upon seeing such "Expect: 100-continue" header, aiCache will respond back with "HTTP/1.1 100 Continue" response. It will then collect complete response and forward it to an origin server. aiCache doesn't not forward the "Expect: 100-continue" header to origin servers, freeing them from having to deal with this extra logic.

aiCache also logs and reports the number of requests with "Expect" header, at both global and website levels, via CLI, Web and SNMP reporting.

De-chunking of request and response bodies.

In order to safeguard origin servers from bogus/malformed requests, aiCache will always de-chunk *request* bodies, when Transfer-Encoding: Chunked is specified in incoming requests. Only when request body can be properly de-chunked, request is processed. Should request require a trip to an origin server, aiCache will forward de-chunked request body, sparing de-chunking cycles on origin servers.

Likewise, aiCache always de-chunks chunked *response* bodies - verifying their validity before forwarding the responses to clients. It is the de-chunked response bodies that get sent back to clients.

Cache size management via cache cleaner process.

As aiCache stores all cached responses in server RAM, it is important to optimize the RAM usage so that more responses can be stored and overall memory management overhead could be reduced. We will now describe the techniques used by aiCache to accomplish this goal.

Cache size management via cache cleaner process.

In order to keep size of response cache under control, aiCache implements cache cleaner logic of 2 different types: *soft cache cleaner* and *hard cache cleaner*. Both run periodically looking for cached responses that have not been accessed in a while. These documents are typically ones that were requested at some point a while back but were not requested since then again. Remember that aiCache uses on-demand cache refreshing algorithm and does not try to refresh a stale cached document unless a new request comes in for this particular document.

Soft cleaner removes response data but it preserves an entry in the response cache - so later on, you can still see those response cache entries and associated statistics.

Hard cleaner removes response data and the corresponding entry in the response cache - so all and any statistics about such responses is lost. Compared to soft cleaning out of cached response, hard cleaner frees up a bit more memory.

Cleaning out such "unwanted" cached responses releases memory and allows for caching of more content. It is only beneficial for the sites with lots and lots of cacheable content. While it can lower aiCache's memory footprint, properly sizing the amount of RAM in your aiCache server is still the best way to go. To reiterate the state-of-the-art in that area: as of 2007-2008 you can have up to 64GB of RAM in better servers and that number has grown all the way to 256GB in 2009. Sometimes, as they say, there's no replacement for displacement - so do size up the RAM in your servers properly to allow for most optimal caching of content !

To adjust the periodicity of soft cleaner runs set **cache_cleaner_interval** to desired interval, in seconds. aiCache uses a reasonable default, so you don't have to set it. Experiment with different settings if you detect memory overutilization .

To adjust the periodicity of hard cleaner runs set **hard_cache_cleaner_interval** to desired interval, in seconds. By default, hard cleaner is set to run once per hour. To disable hard cleaner, set this setting to 0.

Normally, CC cleans out cached responses irrespectively of whether or not the cached responses are still fresh. If you want to disable cleaning out of cached **fresh** responses , set **cc_obey_ttl** flag at website level.

By default, CC cleans out responses that have not been accessed in **cache_cleaner_interval** seconds. You can change that by setting **cc_inactivity_interval** setting at website level. The cache cleaner continues to run every **cache_cleaner_interval** seconds.

Occasionally you might want to disable cache cleaner altogether, for a given website. Do accomplish that, set **cc_disable** setting at website level.

Here's how to decide between cache cleaner types: you'd always want to enable soft cleaner. Depending on how active your site is, reasonable settings for **cache_cleaner_interval** might vary from 1m (60) for incredibly busy sites with wide range of cacheable URLs to may be 1hr (3600) and above if the URL set is more confined.

You don't have to use hard cleaner unless your URL set is stressing your memory footprint. Otherwise, if you have a large number of cacheable "one-timer" URLs - that get cached but are rarely accessed again and the cache size keep growing, putting a stress on the system, you can enable hard cache cleaner.

Cache size management via Cache-by-Path Feature.

Let's say your web site is fairly large and has significant number of various cacheable URLs. After deploying aiCache, most of the content from your web servers will end up in the aiCache's response cache as users request it from your web site. aiCache is designed to keep such cacheable responses in memory (RAM) and never tries[11] to save cached documents to secondary storage, such as hard drives.

aiCache uses request's URL as a pointer to (a signature/key of) cached copy of the Web content. Typically we would want to use as much information as possible in the URL to act as such "signature" or pointer to this cached object. For example let's assume that our web site has just published a breaking news article and it is available under the following URL: **www.acmenews.com/stories.dll?articleid=12344**.

In order to cache this document we would need to refer to it by the whole URL string, as shown above. Assuming that we have about a thousand active stories on our Web Site, we would end up with about a thousand cached Web documents, all with unique signatures of

www.acmenews.com/stories.dll?articleid=NNNNN, representing these news articles, which is absolutely fine and is just the way the aiCache was designed to operate.

---

[11] It might happen on a resource-starved system when OS forcefully pages out an application (aiCache in our case) to secondary storage to free RAM space for another application. Swapping is to be avoided at all cost - not just for aiCache but any other application as well. This, amongst other reasons, is why it is important to have ample RAM available.

Now, let's consider different example. In today's world of Internet it is quite common for one web site to provide a link to content on a different site. Let's assume that a number of external web sites point to our web site's homepage. In order for us to know which of those Web Sites has referred a user to our Web Site, those referring sites would normally provide a "referrer ID" of some form.

For example, the web site "**www.acmebusinesspartner.com**" might point (via HTTP href) to

"**www.acmenews.com/breakingnews.html**" via the following link:

"www.acmenews.com/breakingnews.html?partnerid=partner1"

The "partnerid" parameter does not affect appearance (content) of the resulting Web page in way. It is provided simply to end-up in the log file or to be analyzed by client-side Javascript code, so that at the end of day we know how many users were referred to our web site by our partner sites. [12]

Normally, we would use the whole string above,

www.acmenews.com/breakingnews.html?partnerid=partner1

as a signature for the cached copy of breakingnews.html. Let's assume that there are hundreds of sites out there that point to our Web Site. This would lead to hundreds of different URLs pointing to the same document and would force us to populate cache of our aiCache server with hundreds of copies of the same Web response, polluting the cache unnecessarily with exact bit-for-bit copies of the same content – clearly rather wasteful situation in respect to utilization of RAM.

The example might get even more extreme in certain other cases. For example, some web sites append random strings of characters as a parameter to static web pages or JavaScript files, as a way to obtain certain functionality[13]. However this parameter does not have any impact on the web document itself and is essentially ignored by the web server(s). Once again, if we use the complete URL string as a signature for the cached copy of the web documents, we would pollute response cache with possibly thousands of copies of the same response and adversely affect performance.

Imagine a web site with a hundred thousand subscribers that appends subscriber ID as a parameter to a static HTML or JavaSript file.[14] If we follow our regular routine and store each resulting Web document as a separate entry in the aiCache's cache (even though all of these have absolutely identical content), we might simply run out of RAM space on our servers. Let alone that caching responses under signatures containing such a random string would effectively make such responses non-cacheable.

---

[12] This information can be used to reward our partners for driving visitors to our site or for other purposes that are outside the scope of this guide.

[13] Colorfully named "cache-busting" is one such functionality. Another one is to append parameters to JavaScript and/or HTML files for "client-side" processing, to be used by JavaScript.

[14] So that it becomes available in JavaScript, for example.

aiCache addresses this problem of cached content explosion with a feature that allows designating certain URLs as cacheable "by-path-only". In other words these URLs will have their entire query part stripped to obtain signature that point to resulting (cached) Web document. For example

www.acmenews.com/breakingnews.html?partnerid=partner1

becomes **www.acmenews.com/breakingnews.html** which then would be used as signature for the cached copy of this page. No matter how many sites refer to us in this fashion, we only store a single copy of this web document in our cache. We save significant amount of RAM space, yet we retain the required functionality, as the referrer information still can be processed client-side and ends up in the aiCache log files should server-side log crunching be required.

Resulting resource preservation can be quite significant. Just as explained above we might be able to cut the number of objects we have to store (and manage too) in aiCache's cache by many orders of magnitude.

Configure this functionality by specifying **ignore_query** for particular pattern:

```
pattern html simple 10m  ignore_query
```

Ignoring the URL's query string only affects the cached response's *signature* and has no effect on anything else. Specifically, when a response needs to be obtained, the whole, complete original URL, as received from the requesting browser - including query string, is sent to origin servers.

Cache size management via query parameter busting.

Similar to the feature described above, you can also keep some of the parameters in the query portion of the URL, while removing others. Such modification only affect the cached response's *signature* and have no effect on anything else. Specifically, when a response needs to be obtained, the whole, complete URL - including query string - with all of the parameters, as received from user browser , is sent to origin servers.

Again, we remove parameters from query string to optimize performance and reduce waste that would occur otherwise. Let's consider this URL:

www.acmenews.com/showstory.asp?storyid=1234&partnerid=partner1

We must use query string in addition to the URL path to be able to cache proper responses - so we can not use the cache-by-path feature, as it ignores the query string entirely. Yet once again we have a parameter added to the query string that doesn't affect the response.

aiCache to the rescue ! We can ignore the **partnerid** parameter, while keeping the **storyid** in the signature. To accomplish that, create a pattern that matches this url (something like **showstory.asp** might be all you need) and specify:

```
ignoreparam partnerid
```

in the parameter section. Now incoming request for:

www.acmenews.com/showstory.asp?storyid=1234&partnerid=partner1

has its signature transformed to

www.acmenews.com/showstory.asp?storyid=1234

and that is what is used as cached response's signature. The complete original request string is still forwarded to the origin servers when we need to perform a first fill or a refresh, so it doesn't break any logic on origin servers - as that logic most likely cares about each and every parameter in the request string, even those we might want to discard from the signature.

You can ignore one parameter, a few or all of them, although ignoring all of them is best accomplished via cache-by-path feature.

Quite frequently web sites use a technique called "cache-busting" . It mostly boils down to adding a parameter whose value is a random string. It is aimed at making sure no HTTP caches between a visitor and the target web site can cache content. When such cache-busting parameters are used in URLs of cacheable responses, you're advised to configure aiCache it to remove such cache-busting parameter from URL signature. Figure out what the cache busting parameter name is and add it as **ignoreparam** setting under the proper URL patterns.

Normally, aiCache uses exact matching when matching parameters to **ignoreparam.** However you can configure it to do partial matching via **param_partial_match** setting under matching pattern. It has potential to speed up lookup and removal of **ignoreparams.** You can also use this technique to remove parameters that have semi-random names, but with a common sub-string to them. For example, to get rid of *bustXXX*, *bustYYY* and other parameters that contain "*bust*" in them, you can specify one ignoreparam "*bust*" and set **param_partial_match** under the matching pattern.

Cache size management via ignore case feature.

Normally cached response signatures are case sensitive. So when you cache responses to 2 requests:

acmenews.com/search.jsp?query=Vacation and

acmenews.com/search.jsp?query=vacation

the responses are stored as 2 different responses, even though the actual response bodies are most likely identical. If that's the case, you can tell aiCache to ignore case, website-wide or per pattern, via **ignore_case** setting When configured so, aiCache stores only one response for both of these requests. To accomplish that, aiCache converts signatures to lower case - you can see that by executing CLI inventory command.

If you cache search results, you most likely would benefit from enabling this feature. Another example might include caching responses to financial stock quote requests - as users might enter a stock symbol in upper or lower case. With responses to both inputs being the same, you'd want to employ this feature to reduce memory utilization.

Caching of responses to POST requests.

aiCache allows to cache POST requests just the same way as GETs, although you must be extra careful not to allow caching of private information.

When caching GET requests, the response's signature is formed from original request's *path and query strings.* However, with POST requests, query string might not be present in request's header. For some POST requests it is instead embedded into the body of request - such requests have POST bodies that are URL-encoded (have **Content-Type: application/x-www-form-urlencoded** header specified) . Other POST requests might have portion of the query in the URL and some of it in the body, XML-formatted bodies, plain text bodies, serialized Java POJO bodies and so on .

As cached response signatures are used as pointers into cache repository, generating such pointers takes more time as signatures grow larger. As POST bodies can grow very large, aiCache imposes a limit to the max body size that can be used as a signature. The limit is configurable via **max_post_sig_size** *global* parameter and is set by default to 256 bytes.

You would not encounter any problems as long as POST body simply contains a few URL-Encoded variables. But if you're trying to cache responses to SOAP requests and things of that nature, you might want to enforce the size limit as explained above. And of course you need to make sure that whatever part of the POST's body is used for cache signature, is large enough to make sure that such signatures are unique for different requests.

For a POST request to be cacheable, the following criteria must be satisfied:

- Request's URL must match a pattern that has **request_type** set to **post** or **both**
- The matching pattern's TTL must be non-zero
- The POST's body, if any, must be under **max_post_sig_size** limit
- Cache-busting Cookie, if any are configured, must not be present in the request
- Authorization header must be absent

Similar to handling of cacheable GET requests, cacheable URL-encoded POST requests can have one or more of their parameters removed from their cache signature via **ignore_param** directive. You'd use such parameter busting in order to control the size of response cache. See "Cache size management via Cache- By-Path Feature" and "Cache size management via query parameter busting". You cannot request removal of parameters from non-URL-encoded POST requests, as there's no parameters to discern.

When response's signature includes CR (carriage return) and/or LF (line feed) characters, you might have difficulties specifying such signatures at CLI prompt for the purpose of **dump** command. To diagnose such responses, aiCache lets you use CLI **inventory** command - when a *single* response is matched to the provided hostname and pattern, it is written out to a file, similar to what **dump** command does. So effectively you'd rely on a partial match to locate and write out the cached response of interest.

You can also configure aiCache not to use any of the POST request's body in the signature of the cached response by specifying a **sig_ignore_body** setting at website level or under matching pattern.

Here's a technique you should be aware of: even when request's payload is in the request's body, possibly XML formatted and/or encrypted, you might still be able to get your Dev team to change the code to where they append a meaningful, human-readable signature for the response right to the URL, as an extra parameter. This way you can use it a signature for the cached response, w/o having to include potentially very large request body right into the signature (and you accomplish that via **sig_ignore_body** directive).

Such extra parameter is most likely to be ignored by the server side code, as it doesn't know and/or look for it, yet it can be then used by aiCache as a signature of the response, allowing for faster processing and better troubleshooting capabilities.

aiCache response preload (pre-fetch) feature.

A website might rely on a number of API calls to other websites/service providers for functionality such as Ad calls, analytics etc. Such calls might be expensive in terms of amount of time they take to execute and as a result, they might slow down the pages on your website that rely on these calls.

aiCache allows you to configure a set of such slower URLs for preload, where aiCache pre-fetches and actively maintain a queue of fresh responses to such slower calls, in anticipation that these responses might be soon requested. When they are requested, instead of going to the remote site to obtain the response, aiCache, virtually instantaneously, serves pre-fetched response.

By tailoring pre-fetch parameters, you can fine tune preload so that most of such responses are in fact pre-fetched. To assist with it, aiCache collects and reports a comprehensive set of pre-fetch statistics.

One must not confuse the preload functionality with caching of responses. While caching is only applicable to shared, cacheable responses, preload logic only acts on non-cacheable responses (we also refer to these as 0TTL throughout this manual).

You may specify one or more of **prefetch_url** settings in the website section(s) of aiCache configuration file. This setting requires 2 mandatory parameters: the URL itself, how many of preloaded responses for that URL you want aiCache to maintain and optional third parameter, indicating whether you want to request the preload responses in compressed form.

For example:

```
prefetch_url  /slowGetAdCall?page=home&area=top 100 gzip
prefetch_url  /slowGetAdCall?page=home&area=bottom 100 gzip
prefetch_url  /slowGetAdCall?page=home&area=rightrail 100 gzip
```

specifies that you want to preload responses to URL of **/slowGetAdCall?page=home&area=top**, ask aiCache to maintain a queue of 100 of such responses and you want aiCache try to obtain gzipped responses for that URL. Total of 3 different preload URLs are specified - each requesting 100 responses to be preloaded.

Every second, aiCache analyzes the queues for all three URL and if any of the 3 queues contain less than 100 responses, aiCache requests more responses to maintain the queues at 100 responses.

When a request comes for any of these URLs, aiCache removes a matching response from appropriate queue and sends to the requesting browser. Again, the next second aiCache will request a response to top off the matching queue.

Global, per-Website Web statistics screens contain information on the efficacy of pre-load: ratio of responses served from preload queues to all 0TTL responses. You'd want that ratio to be as close to 100% as possible, to maximize the benefits of response preload. To get there, adjust the number of preloaded responses.

If you get that number too high, it is possible that some preloaded responses might get stale before a client has a chance to request it. To deal with this scenario, you can specify the max age of preloaded responses that is allowed to be served to the clients, via **max_pref_fresh_time** parameter in website section, it defaults to 10 minutes.

aiCache reports the queue length statistic for each URL you configured for preload via Web and CLI interfaces. Preload counters are also available via SNMP. Please note that occasionally you might observe number of preloaded response exceeding the number you specified in the configuration file. It might happen when responses are slow to obtain and is a harmless condition.

During request processing, when no preloaded response is available, aiCache requests a response via regular logic, completely seamlessly to the requesting client. Responses that were served from preload queues are also indicated as such in the access log files.

You can also specify arbitrary HTTP headers to be added to preload requests via **prefetch_http_header**. Each preload URL can have its own set of HTTP headers (more than one such additional header can be specified for each configured preload URL). Each **prefetch_http_header** directive takes two required parameters: header name and header value. Header value follows header name and extends till the end of the line. For example:

```
prefetch_url  /slowGetAdCall?page=home&area=top 100 gzip
prefetch http header User-Agent MyCustomUser Agent V.10
prefetch_http_header X-Custom-Header Test Value

prefetch_url  /slowGetAdCall?page=sports&area=top 100 gzip
prefetch_http_header User-Agent MyCustomUser Agent V.22
prefetch_http_header X-Custom-Header Another Value
```

And lastly, you can configure aiCache to close origin server connection after obtaining preload responses via **prefetch_conn_close** directive. This setting overrides whatever origin server keep alive setting you might have configured in that it closes the connection used to obtain the preload response - while other OS connections might be reused after obtained regular, non-preload responses.

aiCache request retry logic.

By default, aiCache tries to obtain a valid response from origin servers up to 3 times - unless a request is a POST, in which case only one attempt is made. This behavior can be overridden at website or pattern level by specifying **no_retry** setting in respective section - in which case only a single attempt is made for all requests.

aiCache can be configured to retry POST requests - by specifying **retry_post** setting at website or pattern level. You must understand the consequences of retrying POST requests before enabling this feature. While desirable in some cases, in others it might lead to doubling (and tripling and so on ...) of purchase orders, credit card charges, message board postings etc. So please exercise caution with this option.

aiCache enforces time limit on how long an origin server can take to provide a response, it defaults to 10 seconds. You can set it to a different value at pattern, website or global level, in that order of precedence (where pattern's setting overrides website's which in turn override's global setting), by using **max_os_resp_time** setting (in seconds). Do not set it to 1 second as unpredictable behavior might ensue, it must be 2 seconds or more.

If a response cannot be obtained within this amount of time, aiCache might retry it up to total of 3 attempts or fail it right away, depending on **no_retry** setting (see above).

Be aware that you might encounter problems when certain, mostly POST requests are configured for retries. For example, you might end up resending user-registration request - resulting in error response, as registration took place after the first request. Good understanding of site's logic is required to prevent things like these from happening. To be safe, be careful enabling retries for POST requests and increase the **max_os_resp_time** setting for the slower requests - giving them more time to complete so that aiCache doesn't end up generating (premature) retries.

In case of requests, aiCache will use "origin server of last resort" - if any are provided, after exhausting retries against "regular" origin servers. Such LR OS are identified by assigning **os_tag** of 100.

Forcing retry (refresh) by response header or response size check.

aiCache can be configured to *force retries* of cacheable requests against origin servers by returning a special response header: **X-NoCache** . You can set it to any value, it is ignored. Another way to indicate a bad response to aiCache and force retry is to configure minimum and maximum acceptable response size via **retry_min_resp_size** and **retry_max_resp_size** directives in the pattern configuration section.

Here's a quick example where this feature can be useful: if you have intermittent errors originating randomly from origin servers, where a "normal-looking" 200 response is returned, yet you can determine if the response is unacceptable and should be retried based on responses size, in hope that you can obtain a good response on such subsequent retries.

Please note existance of **no_retry_min_resp_size** and **no_retry_max_resp_size** - when response size fails these conditions, no attempts are made to retry the request and the obtained response is fed back to the client, but is not cached.

Be aware that aiCache will compare whatever is the response size as reported by origin server. So when response comes back compressed (gzipped), its size might be much smaller, compared to uncompressed version of same. So you might need to adjust your **retry_min_resp_size** and **retry_max_resp_size** settings accordingly, to deal with compressed responses.

Alternatively, if limitations of API/etc force code on origin servers to return 200-response code, yet you want to somehow indicate to aiCache that response should be retried and should not be cached (even if

matching pattern is configured otherwise). In this case you can programmatically insert the **X-NoCache** response header and aiCache will re-try the request up to 3 times. If it still fails, aiCache sends the obtained response back to the requesting clients, complete with **X-NoCache** header - but the response is not cached.

In other words, you can effectively override pattern-specified TTL by returning such response header.

For cacheable responses, should a previously cached stale copy of content be available and **fallback** setting being set for matching website, such *previous stale cached response* is returned in response when X-**NoCache** header is specified or response size limits are violated.

Should aiCache be forced to return a response that carried **X-NoCache** header from an origin server, this header is forwarded by aiCache in its own response , so that you can use it in client-side logic or other downstream logic/infrastructure. **Expires** header is also inserted and set well in the past to indicate to downstream recipients that the response should not be cached.

Please note that retrying can be turned off by specifying a **no_retry** setting at website or pattern level.

Alternatively, you can force retry by matching *content* of the response body against a set of patterns. Please see chapter titled "Content-driven request fallback or retry control" for more information on this feature.

Modification/insertion of HTTP Via header.

Normally, aiCache indicates its presence in the HTTP response path (the fact of aiCache being an intermediary) by modification or insertion of HTTP Via header. aiCache identifies itself under the name that you specify via **server_name** global setting, default value being ***aicache6.***

You can configure aiCache to not indicate its presence via **disable_via** website-level setting.

Additionally, you can configure aiCache to indicate it's presence to origin servers, by inserting/modifying Via header in the requests sent to origin servers, by setting **send_os_via** website-level setting.

Adding HTTP response headers.

You can configure aiCache to add custom response headers to responses via pattern-level **resp_header** parameter. For example:

```
pattern /blah simple 10
resp_header X-Header Some Header Value
resp_header Another-X-Header A different Header Value
```

Dealing with malformed 3xx responses from Origin Servers.

Occasionally, you might observe malformed 3xx (such as 302, 301) responses being sent by origin servers, where a non-zero Content-Length is indicated in the response header, but the actual body is missing. Normally, such malformed response would time-out in aiCache, as aiCache expects, in vain, to receive the declared response body which never comes.

The best way to fix this is to properly configure origin servers to not indicate presence of a response body when there's none. However, in a pinch, you can tell aiCache to not expect 3xx responses to ever have a body by setting **zero_body_300** flag at website level, in the aiCache configuration file.

Likewise, you might find yourself in a situation where origin servers are sending a response body in 302 and 301 responses. Such response bodies are unnecessary, as they are never even seen by users. Additionally, you in heavy traffic setups, sending such extra bytes to the requesting browsers is wasteful of bandwidth and other resources. You can configure aiCache to disregard such bodies, if any, by setting **zero_body_300** flag at website level, in the aiCache configuration file.

Diagnosing request and response cookies.

Let us provide a helpful reminder about HTTP requests and responses as they relate to cookie handling. Responses *set* cookies via Set-Cookie response headers. Response can carry multiple Set-Cookie headers in order to set multiple cookies. Cookies can have values and a number of attribute – such as expiration date, domain they are applicable to and so on. Here's an example snippet of a response header:

```
Set-Cookie: dLTHID=AD7631B846DA143B0C92789729E2; Path=/; Domain=.acme.com
Set-Cookie: userid=213452345234; Path=/; Domain=.acme.com
Set-Cookie: attribute=account=paid;
```

As you can see, a total of three different cookies are being set via this response. Notice that in the last Set-Cookie header only a cookie named *attribute* is being set, with the value being "account=paid". In other words, *account* is not a name of a cookie - but rather a part of *attribute* cookie's value.

While a single Set-Cookie can, in theory, set multiple cookies, you are not likely to see this arrangement in practice. Instead, each cookie that is being set, will likely be set via a separate Set-Cookie header.

After a cookie is set via a response, client browsers will carry such cookie and its value, when appropriate, in the request header, via Cookie request header. Continuing with the example above, the request header might contain the following:

```
Cookie: dLTHID=AD7631B846DA143B0C92789729E2
Cookie: userid=213452345234; attribute=account=paid
```

As you can see, all 3 cookies that were set via Set-Cookie response header, are now sent, by the browser, via request's Cookie header. Notice how first Cookie header only specifies a single cookie, while the second Cookie header carries 2 different cookies via a single Cookie header. Browsers might combine a whole number of different cookies in a single Cookie header and it is a fairly common occurrence.

In order to assist you in diagnosing of both request and response cookies, you can set global level flags of **debug_request_cookie, debug_response_cookie** or **debug_cookie .** When so configured, aiCache will log out, to the error log file, names and value of both request and response cookies, as they are being processed.

For example, when you specify a **0ttl_cookie** , aiCache will analyze if an incoming request contains such cookie, and you will see appropriate diagnostics output. Ditto for **sig_cookie, sets_session_cookie** etc.

Diagnosing bad responses.

aiCache logs, in a throttled fashion, information on failed responses. There could be a multitude of reasons as to why a response could not obtained. aiCache assigns numeric codes to help you diagnose just what went wrong. The code is written out in entries in both **error** and **access** log files.

Error Code	Explanation
1	Generic connection error - ai could not connect to selected origin servers, an unexpected connection close or reset has occured.
2	Timed out waiting for response - origin server did not serve response back in allotted amount of time.
3	No origin servers were available (for example, all OS have failed the health check)
4	Response size was below min or above max allowed resp size (when either limit is configured)
5	Response was marked as chunked, but de-chunking has failed.
6	A non-200 or non-301 status response was received for a cacheable request and retry_non200 flag is set
7	X-nocache header has been received in response. Such header is an indication of a bad response

aiCache doesn't write more than 120 entries per minute, to error log file, to prevent a run-away consumption of disk space.

Disallowing downstream caching of responses.

aiCache can be configured to disallow downstream caching of responses while still caching them at aiCache itself. Reasons for such configuration might vary, but the main idea behind such caching is to make sure clients always come back to aiCache to obtain response, possibly a cached response from aiCache's response cache.

For example, if you deploy response-driven content expiration (see a separate chapter later in this document) so that posting a new message to a discussion thread expires that thread within aiCache's response cache, you'd want to make sure that after cached response is expired and refreshed by aiCache, client does not use a locally cached copy and instead, re-requests it from aiCache again. For such patterns, specify *negative* TTL.

For example, to cache response at aiCache for 10 minutes while disallowing downstream caching, set TTL to -600 (negative 600).

You'd almost never use negative TTLs for common auxiliary content, such as images, JS and CSS files - these you want to be cached on client side and intermediate proxies (if any), to speed up loading of pages and reduce load on aiCache servers and your uplink infrastructure (links, routers, firewalls, load balancers etc).

Also, allowing for aggressive downstream caching is likely to have a positive financial effect through reduction of bandwidth utilization. And it reduces overall load on the Internets, as fewer bytes need to be shuffled around.

By default, when responding to requests that match patterns with negative TTLs, aiCache sends **Expires** header, set well in the past, to disallow any downstream caching of the responses.

In addition, you can configure aiCache to also add "**Cache-Control: no-cache**" header to such responses, by setting **send_cc_no_cache** setting at website level. By default, sending of this header is turned off to reduce the amount of data that needs to be sent back to the client.

Forwarding Client IP address information to origin servers.

As aiCache front-ends all of client traffic, the origin servers see all requests coming from aiCache's IP address . aiCache still collects and logs (if so configured) *true* client IP addresses in the access log file.

However if your setup requires code on origin servers to have programmatic access to *true* client IP information, you can configure this on per-website basis by providing a **forward_clip** (forward **Cl**ient **IP**) setting. Normally, the client IPs are forwarded as **X-aiCache-CLIP** header, but you can change that by providing a different header name as part of **forward_clip** directive. For example this configuration:

```
hostname www.acmenews.com
forward_clip X-Client-IP
```

results in aiCache forwarding client IP as **X-Client-IP: 1.2.3.4** HTTP header. The value can be now programmatically accessed by server-side code and acted on.

Please note that by default, aiCache attempts to minimize modification of request headers for performance reasons, so do not use this setting (it is turned off by default) unless you need it. Depending on your configuration, your load balancers (if any) might already be configured to forward true client IPs in the headers.

Forwarding Origin Server IP and port number to clients.

As aiCache front-ends all of client traffic, the clients see all responses as coming back from aiCache's IP address. In case you need to know what origin server was used to obtain a particular response, you can configure aiCache to forward origin server IP and port number as an HTTP header.

To enable this feature, please set **x_os_header** global setting to the desired name. For example this setting:

```
x_os_header  X-aiCache-OS
```

results in aiCache forwarding origin server IP and port number as **X-aiCache-OS** response HTTP header, for example:

```
X-aiCache-OS: 1.2.3.4:80
```

The value can be used for diagnostics purposes and/or accessed programmatically by client-side Javascript code.

Parsing out forwarded Client IP from request header.

In certain network setups, where aiCache receives requests not directly from requesting clients, but through some intermediary, such a load balancer, a proxy or another aiCache, the IP address that requests come from might be that of the intermediary, not of the requesting client.

In this scenario aiCache will log in its access log file the IP address of the intermediary, not that of the client. However, if you want to record the true client IP and can configure your intermediary to forward the true client IP as an HTTP header (aiCache can certainly be configured to do so as described above), you can tell aiCache to parse our such forwarded client IP and log it.

To configure aiCache for this processing, you need to specify **hdr_clip** at global scope of the configuration file, setting to the name of the HTTP header that carries client IP. For example:

```
hdr_clip X-Forwarded-For
```

aiCache will then parse out such forwarded client IP and log it in access log file, as client IP field. When no such processing is requested or no client IP is available in request header, a dash "-" is logged instead.

Please note that aiCache must have access to true client IP information for DOS countermeasures to work.

When multiple requests come over the same client connection (a Keep-Alive connection), aiCache obtains the client IP upon first request and reuses that value when logging the other requests it receives over the same client connection, as client IP is most likely to stay unchanged throughout the lifespan of a single Keep-Alive client connection.

However, you can force aiCache to re-obtain (re-parse-out) forwarded Client IP for each and every request by setting **refresh_hdr_clip** flag at global scope of the configuration file. Examples when you need to have this setting set is when an intermediary can send requests from multiple clients over the same connection to aiCache (aiCache itself does that when configured for OS Keep-Alive connection).

Please note you can cheat a bit and use this directive to log an arbitrary request header, in lieu of using it for its intended purpose, by specifying header's name via **hdr_clip** - should you have a need to log a request header.

Forwarding response's TTL value to clients.

To enable this feature, please set **x_ttl_header** global setting to the desired name. For example this setting:

```
x_os_header  X-aiCache-TTL
```

results in aiCache forwarding response's TTL (in seconds) as **X-aiCache-TTL** response HTTP header, for example:

```
X-aiCache-TTL: 600
```

Value of 0 is returned for non-cacheable responses. The value can be used for diagnostics purposes and/or accessed programmatically by client-side Javascript code. The TTL values returned via this header are static, not the countdown type.

The *client_linger* and *os_linger* settings.

These have to do with what happens to TCP/IP connections after they are closed by aiCache. When no SO_LINGER option is set, TCP/IP connection proceeds down regular, fairly lengthy yet orderly route. This is the way you want to have it , unless you're running an extremely busy server with very high connection rate.

When running such high-connection rate websites, you will discover that at any point in time you might have thousands upon thousands of connections in TIME_WAIT state. Now such connections do not translate to any extra load on aiCache, yet should they create a problem for your setup, you can try reducing number of such connection by turning on **client_linger** and **os_linger** options.

With these options set, the TCP/IP connection close takes a shortcut -instead of an orderly termination, a TCP/IP reset is sent instead and connection is disposed of immediately, without going through TIME_WAIT state. You must test this before enabling it in production setting. Some client browsers might not appreciate getting such TCP/IP resets none too much.

However, this should be much safer with origin server connections - as by the time aiCache issues a reset, it has obtained a complete response from origin servers. In addition, it is not only aiCache that will show reduced number of connections in TIME_WAIT state, origin servers will also see similar reduction. Yet again, please test before enabling it in production.

Alternatively and/or in addition , you can explore setting Linux's own TIME_WAIT interval to a lower value (some heavy traffic sites set these it to as low as 1 sec):

```
echo 5 > /proc/sys/net/ipv4/tcp_fin_timeout
```

Dealing with empty HTTP Host headers.

As aiCache is capable of accelerating multiple websites off a single instance, you can see that not having Host header provided in the request, leads to a dilemma: just which of many websites accelerated by this instance of aiCache is the request for ?

When request doesn't have Host header specified, aiCache can take *one* of the following actions:

- When **default_host** global level setting is set, its value is assigned to the request's Host header
- When **enforce_host** global level setting is not specified in the configuration file or is set to "**on**", aiCache enforces host header (which is the default setting). As a result a 409 "website not recognized" response is returned to the requesting client, as aiCache cannot match the request with an empty request hostname to any of the defined websites.
- When **enforce_host** global level setting is set to "**off**" , the hostname of the first defined website is assigned to the request's Host header. Subsequently, the request will be matched to the first defined website.

Dealing with HTTP Host headers that cannot be matched to defined websites.

By default, aiCache tries to match request's HTTP header value to a configured website. It attempts such match based on the value of website's **hostname**, any of its **cnames** and/or **wildcard** settings. If no such match could be obtained, a 409 "website not recognized" error response is returned to the requesting client.

If your setup is such that you want to send all of requests to a certain website, no matter the actual value of request's Host header, you can configure it via **ignore_host** global setting. In this case, if no match can be found during regular processing, the request is assumed to be for the first defined website (as configured in the configuration file). You can accomplish similar effect by defining a **wildcard** of "." (period) for website that you want to make "default" website.

Note that this is somewhat different from dealing with requests that have no Host header specified at all (see previous section on how aiCache deals with such requests) in that request's host header is not modified.

Rewriting of HTTP/1.0 requests to HTTP/1.1.

You can configure aiCache to rewrite, in situ, HTTP request's "minor" version to 1 - so that HTTP/1.0 requests are forwarded as HTTP/1.1 requests to origin servers. It is done to speed up processing of responses, as

origin servers are more likely not to use connection close as an indication of complete response, in case of HTTP/1.1 requests. It boils down to sparing an unnecessary system call.

To configure this behavior, you can set **keep_http10** setting to **off** at website level: Be aware that such overwrite much cause problems so test it thoroughly before putting it into production.

```
keep_http_10 off
```

On dealing with HTTP/1.0 clients and/or proxies.

You might still encounter requests from HTTP 1.0 clients or proxies. Such requests might differ from much more common HTTP 1.1 requests in the following ways:

- HTTP Host header might be absent. aiCache's handling of such requests is described on previous page.
- Size of response body might be indicated via connection close, as opposed to using much more common way of indicating the same via Content-Length or Transfer-Encoding: Chunked HTTP headers.
- Support for compression might be indicated via **Accept-Encoding: gzip** header, but in reality the client (such as a Web browser ,corporate or ISP proxy) cannot deal with compressed responses.
- Support for connection Keep-Alive might be indicated via **Connection: Keep-Alive** header, but in reality the client (such as a Web browser or a corporate or ISP proxy) cannot deal with Keep-Alive connections.

Use of connection close by Origin Servers.

To assist you with diagnosing of possible issues related to origin servers using connection close or half close to indicate size of response in lieu of providing the same via Content-Length, a warning is a printed in error log file every time such request or response is detected. Clearly, as origin servers are likely to be under your control, you should configure them not to use this technique.

Compression of HTTP/1.0 responses.

By default, aiCache will compress responses to HTTP/1.0 requests, if requesting HTTP/1.0 client (browser or proxy) indicates support for compression. However in some situations such indication might be misleading. It might work for some HTTP/1.0 clients, but not others - so when a cacheable response is gzipp'd as a result of HTTP/1.0 request that indicates support for gzip, and later is fed to a different HTTP/1.0 request, indicating

same support for gzip, it might not work either or both times . You can disable compression of responses to all and any HTTP1.0 requests by setting **enable_http10_gzip to off**, in global section of the configuration file.

```
enable_http10_gzip off
```

Keep-Alive for HTTP/1.0 connections.

By default, aiCache allows connection Keep-Alive for HTTP/1.0 connections , if requesting HTTP/1.0 client (browser or proxy) indicates support for Keep-Alive. However, such indication might be misleading. It might work for some HTTP/1.0 clients, but not others. So to be safe, aiCache doesn't allow it.

If you want to override this behavior, you can set **enable_http10_keepalive** option to **off** at global level of configuration file.

```
enable_http10_keepalive off
```

Reporting number of HTTP/1.0 requests.

As a helpful hint as to the volume of HTTP/1.0 requests seen by aiCache, it is reported at both global and website level, via Web, CLI and SNMP interfaces.

Storing different versions of cached responses for HTTP/1.1 and HTTP/1.0 clients.

To accommodate for HTTP/1.1 and HTTP/1.0 clients (Web browsers , corporate and ISP proxies etc), aiCache might store up to 4 different versions of responses for any given URL:

- Compressed responses for HTTP/1.1 clients. Cache signature of such responses includes "g1". Please note that aiCache always de-chunks chunked responses from origin servers, so that clients don't have to do it.

- Plain (not compressed) responses for HTTP/1.1 clients. Cache signature of such responses includes "p1". Please note that aiCache always de-chunks chunked responses from origin servers, so that clients don't have to do it.

- Plain (not compressed), responses for HTTP/1.0 clients. Cache signature of such responses includes "p0". aiCache never solicits chunked response for HTTP/1.0 requests and properly configured origin servers should never send chunked responses in response to HTTP/1.0 request. Likewise, aiCache doesn't solicit, by default, compressed responses for HTTP/1.0 requests and properly configured origin servers should never send compressed responses in response to HTTP/1.0 request that doesn't indicate support for compression.

- Compressed responses for HTTP/1.0 clients - only when enable_http10_gzip global option is specified (be warned that enabling it has a potential to cause problems!). Cache signature of such responses includes "g0". aiCache never solicits chunked response for HTTP/1.0 requests and properly configured origin servers should never send chunked responses in response to HTTP/1.0 request.

  Under normal conditions and for most site, you're likely to see mostly "g1" responses in aiCache and these also are most likely see the bulk of requests.

  Please also note that well behaved origin server should never reply with chunked responses to HTTP1.0 requests.

  Should you ever discover origin servers responding incorrectly to HTTP1.0 requests, you can set aiCache to overwrite HTTP versions to HTTP1.1 by setting:

```
keep_http10 off
```

Configuring additional HTTP headers for HTTP/1.1 and HTTP/1.0 requests.

To accommodate for differences in HTTP/1.1 and HTTP/1.0 clients, aiCache allows to configure additional, trailing HTTP headers to be added to cacheable requests destined to origin servers (as you might recall we send such requests for first fill and refresh requests).

These headers are configured per website via **httpheader** (for HTTP/1.1 requests) and **httpheader0** (for HTTP/1.0 requests) directives in website sections of the configuration file. Multiple directive of both types can be provided, each becoming an additional HTTP header.

These settings are optional and it is up to you to provide these if you want to , for example, affect origin server responses in a certain way, etc.

Redirecting for 404 and 500+ response codes.

aiCache allows you to redirect to a custom location when 404 (Document Not Found) response is received from an origin server in response to a client request. The idea is to send users to a more user-friendly page that your typical "Not Found" default page. To accomplish such redirection, set 404_redirect_location in website section. For example:

```
404_redirect_location  http://acmenews.com/404grace.html
```

aiCache also allows to redirect to a custom location when 500+ (all kinds of nasty error responses, like "internal server error") response is received from an origin server in response to a client request or aiCache has generated such 500+ error response internally (for example, when no origin servers are available). The idea is to send users to a more user-friendly page that your typical "Internal Server Error" default page. To accomplish such redirection, set 500_redirect_location in website section. For example:

```
500_redirect_location  http://acmenews.com/500grace.html
```

Both of these redirects are *website*-wide. If you want to apply more granular control over error-driven redirection, you can use **redirect_4xx** and **redirect_5xx** *pattern* level settings. For example:

```
pattern / simple 10
redirect_4xx  http://acmenews.com/4xxgrace.html
redirect_5xx  http://acmenews.com/5xxgrace.html
```

You can use these settings when you want to provide redirects that are specific to the request URL. For example, you might know that your message board system can sometimes take a while to post a new message. As a result, such posting requests can occasionally timeout awaiting response from origin server. So you want to let users know to be patient and not try to repost the message right away, instead come back and check in a minute or so. So you might craft a special page explaining this and specify:

```
pattern messagepost.jsp simple 0
request_type both
redirect_5xx  http://acmenews.com/bePatientAndDontRepost.html
```

Serving/injecting file system content.

As you already know, aiCache allows for very flexible setups - where you can have an accelerated website that serves some content from certain origin servers (os tagging), while optionally going to a different set of origin servers for other content . Now you are about to find out that aiCache can also *inject* file system content into accelerated sites.

You can configure aiCache to serve files off a local file system for certain request URLs - instead of taking the regular route of trying to obtain such responses from origin servers. All you need to do is to specify website-level or pattern-level **file_doc_root** setting, with pattern-level setting, when set, overriding the website-level setting (please note that global-level setting of file_doc_root has a very different meaning) . You then set pattern-level **filesystem** flag. For example:

```
website
hostname www.acme.com
file_doc_root /var/www.acme.com


....
pattern /static 30m
filesystem
```

With this configuration in effect, aiCache will attempt to obtain responses to matching requests by serving matching files in a local directory of **/var/www.acme.com .** For example, an incoming request for :

http://www.acme.com/static/js/main.js

will result in aiCache attempting to return back content of a file located at

/var/www.acme.com/static/js/main.js

Likewise, request for :

http://www.acme.com/static/images/news.png

will result in aiCache attempting to return back content of a file located at

/var/www.acme.com/static/images/news.png

As you can see, aiCache simply appends the URL *path* of the request to the directory specified by **file_doc_root** setting and then it attempts to read in the file at that location. Should a file be found at the

specified path, it's content is returned as result. If a file cannot opened (non-existing file or file with permissions issues), a 404 "Not Found" response is returned instead.

aiCache will cache the content as per specified pattern TTL. Please note that for aiCache to serve content off file system, the pattern must specify non-zero TTL. Requiring caching of file-based responses is done to avoid excessive file system IO.

The URL *query*, if any, is ignored when constructing the file path, but it is still both used as part of response cache signature and logged.

aiCache has simple logic to establish Content-Type response header for such file system content. It understands and sets proper Content-Type for files with the following extensions: **.htm, .html, .css, .js, .xml, .flv, .gif, .jpg, .png, .tiff.**

You can also specify the Content-Type value right at the pattern level by setting **file_content_type,** for example:

```
website
hostname www.acme.com
file_doc_root /var/www.acme.com


....
pattern /static/blah 30m
filesystem
file_content_type text/blah
```

aiCache will optionally compress the content of the file, before returning it to the requesting agent, using the regular content-compression logic (see dedicated chapter for more information on aiCache on-the-fly compression).

The static file injection could be used for a variety of reasons. For example, you might need to serve certain content in response to certain URLs, but you don't have access to the origin servers to install the required files there. Or simply don't have time or desire to propagate a number of files across a number of origin servers. No matter the reason, there's an easy way to have aiCache serve these files for you.

Remember that aiCache, when running, assumes identity of a non-root user (as specified by **username** global-level setting). Make sure the **file_doc_root** directory and all files in it are accessible to the aiCache user - typically requiring "rx" flags set on all of the directories and sub-directories and "r" flags set on all of the files you want to serve to users.

Configuring HTTPS.

Introduction.

You can configure aiCache to perform HTTPS traffic encryption. When acting in this fashion, aiCache maintains HTTPS communications between itself and clients, while optionally forwarding requests in clear (HTTP) to origin servers. This way you get to have your cake and eat it too: you protect the information while in transit, yet you relieve your origin servers from having to deal with HTTPS overhead. Of course, you can also have a more traditional configuration when aiCache accesses origin servers over HTTPS.

With reasonably fast hardware dedicated to aiCache servers, you can expect very high HTTPS session establishment and bulk encryption rates from aiCache. For example , 8 threaded aiCache running on 8-core Intel Nehalem server, can accomplish over 15,000 RSA-1024 key signs/sec, 25,000 key verify/sec. Same configuration is capable of driving around 1.5Gbps of traffic in 3DES or AES-256 encryption mode, certainly more than adequate numbers for even higher volumes of HTTPS traffic.

aiCache requires *OpenSSL* shared library (*libssl* and *libcrypto*) to be installed on server. As of late 2009, the current OpenSSL version is 0.9.8. Chances are these libraries are already installed on your server, as part of standard Linux install. If you don't know where these reside, you can look for them via:

```
find / -name 'libssl*.so'
```

Should these libraries be found in non-standard location, you should add that directory to the LD_LIBRARY_PATH before starting aiCache. Alternatively you can copy these or symlink to these from a more standard libraries location, such as **/usr/local/lib** .

aiCache doesn't carry within itself or distribute any of OpenSSL source or binary code. However, should you need to acquaint yourself with OpenSSL license, it can be found at the following location: http://www.openssl.org/source/license.html .

Please also note that we also offer a non-HTTPS aiCache binary - you can use it when you don't require HTTPS support and don't have OpenSSL libraries available on the aiCache servers.

In this Guide, we use HTTPS and SSL interchangeably, to refer to protected/encrypted connections – as opposed to HTTP connections, that happen without any encryption applied to the data.

Obtaining an HTTPS certificate.

While covering just how HTTPS operates in establishing trusted communications between a client browser and a Web server, is well outside of scope of this manual, we shall provide a very brief introduction on the subject.

For a client browser to trust a web site, the site must have applied for and acquired, a digital ID of sorts - a so called HTTPS certificate. Much like your local government agency can issue you some form of ID that you can then present to others to prove your identity, so can a trusted Internet "agency" of sort issue a digital ID certificate to your web site.

To obtain such digital certificate, you need to generate a request and file it with an ***established*** Certificate Authority, such as Verisign, Thawte, GeoTrust, Network Solutions etc. The request generation can be accomplished using your own installation of OpenSSL toolkit - it comes with tools necessary for certificate generation.

A modest fee is normally levied for HTTPS certificate issuance.

Upon receiving of signed certificate from Certificate Authority, you tell aiCache where to find it and for what web site to present it to user browsers. All common web browsers come preconfigured with Certificate Authority information. In other words, when aiCache presents to the web browser, the certificate you've received from an ***established*** Certificate Authority, browser will know that certificate is signed by an ***established*** and ***trusted*** authority and will allow establishment of encrypted HTTPS connection, along with displaying of proper *trusted HTTPS indicator* to the user.

As part of establishing of HTTPS connection, another piece of information will be required - a file with your web site's ***private key.*** Through some key exchange magic - using your site's ***private key*** (which you never disclose or share with anybody) and your site's signed ***certificate*** (which, within itself, contains your site's ***public key)***, aiCache and browser will choose a ***secret encryption key*** that will be then used to encrypt the actual data, for certain duration of time and certain amount of data. Periodically, browser and aiCache might decide to re-negotiate a new ***secret encryption key,*** to make the communication even more secure.

Here's an excellent on-line resource explaining how to generate both ***private key*** and a ***certificate request***: http://sial.org/howto/openssl/csr/ .

Self-signed HTTPS certificate.

Some sites might find acceptable to deal with inconvenience of ***self-signed certificate*** or you might want to configure one for testing purposes. With such ***self-signed certificates,*** instead of asking an ***established*** and ***inherently trusted Certificate Authority*** to issue your site an official ID, you issue your own.

When aiCache (or any other web server) presents such self-signed certificate to your site's visitors, the user's browser will immediately alert that the certificate is not signed by a known certificate authority. A dialog will be presented, letting users to accept your self-signed certificate and continue the access to your site or to abandon the connection.

Most users on the Internet will opt to abandon the connection, so presenting such self-signed certificate is not a good idea for a public site. However, for an Intranet site, you might notify your employees that they should accept the self-signed certificate and you can certainly use such self-signed certificate for testing purposes.

To create a self-signed certificate, just like with a request for an official ***certificate***, you start by creating a private key. For example, assuming your server has an OpenSSL toolkit installed, run:

```
openssl genrsa -out host.key 1024
```

Now you will have a private key, with key length of 1024, stored in a file called *host.key*. Please note that key length of 1024 is considered to offer adequate protection, but do not use any shorter keys for production sites.

Next, we generate a self-signed certificate:

```
openssl req -new -x509 -nodes -sha1 -days 365 -key host.key > host.cert
```

When prompted, provide required information. Should take you about a minute. Should you make a mistake, simply re-run the **openssl req** command. Similarly, should you corrupt, misplace or remove the private key file, you can regenerate that as well, using **openssl genrsa** command - but then you'd need to regenerate the self-signed certificate as well.

That is it, we're done: our private key is in ***host.key*** file and our self-signed certificate is in ***host.cert*** file. Now all you need to do is to configure aiCache HTTPS ***listen port,*** pointing to both the private key file and the certificate file, along with a cipher list. We specify the following, in aiCache configuration file's global section:

```
listen https * 443 host.cert host.key AES256-SHA:RC4-MD5
```

Now, when a client browser makes an HTTPS connection to port 443 on any IP on aiCache server, aiCache will present the self-signed certificate to the browser. Should user then accept the certificate through the certificate dialog, an HTTPS connection will be established.

Making sense of certificate file content.

The content of certificate file is normally not human-readable due to being encoded. Two different encodings are commonly used for certificate file: PEM (that's what aiCache requires) and DER (it can be converted to PEM, but we advise getting PEM-encoded certificates).

Looking at PEM-encoded certificate, it is impossible to tell what's inside. Thankfully, there're a number of sites that you can use to decode content of a certificate file. Here's a good one you can use: http://www.sslshopper.com/certificate-decoder.html . OpenSSL toolkit can also be used, via command line interface, to decode a certificate file – google for instructions. You might need to decipher the certificate if you're not sure what's inside – as in who is the certificate for, what CA has issued it, what is the expiration date etc.

It is absolutely safe to let an external site to decode your certificate file – as it is, after all, a file your site freely shares with each and every visitor. However, we most strongly advise to never share your private key for any reasons, always keep it protected. Anyone who possesses your site's private key could then potentially impersonate your website.

Configuring multiple HTTPS web sites.

If you must configure multiple HTTPS web site at a single aiCache instance, each such HTTPS site will typically require its own certificate, properly matching each site's name (say login.acme.com, store.acme.com) to what you provided as CNAME during certificate issuance. Alternatively, if the domains that require certificates have a common suffix (say login.acme.com, store.acme.com are both "subdomains" of acme.com), you can opt for so called wildcard certificate instead. Such wildcard certificate would cover all of possible subdomains and might prove to be more convenient to obtain and manage, when compared to obtaining separate certificates for each subdomain.

Each certificate will then require its own ***listen*** directive in aiCache configuration file. Each listen directive would also require its own IP address. For example:

```
listen https 1.1.1.1 443 acme.cert acme.key AES256-SHA:RC4-MD5login.acme.com
listen https 1.1.1.2 443 check.cert check.key AES256-SHA:RC4-MD5 checkout.acme.com
```

The reason for this is as follows: when an HTTPS connection is made to certain IP:port endpoint, aiCache must present a valid HTTPS certificate, matching the requested web site. So as long as we want to use standard HTTPS port of 443, the only way we can make a different end point and present a proper certificate, is to use a different IP address.

Using non-standard port number, anything else than 443, is possible, but then end users would have to type the special port as part of URL. That is likely to be too much hassle for a public web site. Decision is up to you.

You can use a different port method if you have a load balancing device in front of aiCache, that can map ports - but such device, in turn, might need to use additional IP address to distinguish between multiple HTTPS websites, so you simply push the problem upstream.

Chained HTTPS/SSL Certificates.

In certain cases, a bit more work is required to configure aiCache SSL certificate. You might use a Certificate Authority (CA) that sends you a *number* of certificates, in *addition* to the regular server certificate. For example, let's assume you received the following certificates:

- acme.cert (this file contain your server certificate)
- TrustedSecureAuth.cert (contains certificate of the issuing CA)
- UberTrust.cert (contains certificate of the issuing CA)

As usual, you can first configure just the server certificate you received, to be used by aiCache

```
listen https 1.1.1.1 443 acme.cert acme.key AES256-SHA:RC4-MD5 login.acme.com
```

Test the site in HTTPS mode (browse to it using **https://** prefix, as opposed to **http://**), using a number of different browsers. If you do not get "unknown" or "un-trusted" certificate warning, you're all set and no further action is required.

However, if you receive the warnings about unknown certificate authority, you'd need to create a so called *chain* certificate. We simply concatenate all 3 certificate into one file. **The file name must contain "chain" substring, for aiCache to recognize it as chain certificate**.

For example, on you Linux server, run the following commands, to create a chain certificate file called **acme.chain.cert** :

```
cat acme.cert > acme.chain.cert
cat TrustedSecureAuth.cert >> acme.chain.cert
cat UberTrust.cert >> acme.chain.cert
```

Now you configure aiCache to use the chain certificate file via familiar:

```
listen https 1.1.1.1 443 acme.chain.cert acme.key AES256-SHA:RC4-MD5 login.acme.com
```

At this point, restart the aiCache with the new configuration and retest you site in HTTPS mode, using a number of different browsers. It should work without you getting the certificate warnings. In an unlikely event you still get a warning, you can recreate the chain certificate file - but this time, try changing the order of the last 2 components:

```
cat acme.cert > acme.chain.cert
cat UberTrust.cert >> acme.chain.cert
cat TrustedSecureAuth.cert >> acme.chain.cert
```

The reason for use of chain certificate files is as follows: most client browsers come pre-configured with a number of root CAs they inherently trust. If you use a certificate issued by one of these CAs, you are not likely to need a chain certificate. Newer browsers tend to come with a larger list, encompassing more CA.

However, if you receive a certificate from a different, perfectly legitimate CA, that is not pre-configured into your browsers, then aiCache must present to the browsers, both your site's certificate ***and*** certificate for the CA's itself ! You can see this chain can get even longer, requiring possible additional certificates - but the chain must always come to a certificate issued by a CA that your browser trusts. Greatly simplified, you can imagine the following series of actions taking place, when a client attempts HTTPS connection to aiCache server:

Client: requests https://secure.acme.com

aiCache: hello there, here's my certificate, it is signed by UberSuperTrust Inc

Client: hmmm, never heard of it ! Do you have a certificate for UberSuperTrust Inc itself ?

aiCache: sure, here it is . This one is signed by CertsAreUs Inc !

Client: hmmm, never hear of CertsAreUs ! Do you have a certificate for CertsAreUs Inc itself ?

aiCache: sure, here it is . This one is signed by a well-known Root CA !

Client: it sure is. Since I can verify the complete certificate chain, I am now sure that I am indeed, connected to secure.acme.com!

About HTTPS/SSL Cipher List.

You **must** provide a cipher list, when specifying aiCache HTTPS listen port. No default value exists. You may opt to start with cipher list of very strong ciphers, like AES256-SHA:RC4-MD5, as shown in example below:

```
listen https 1.1.1.1 443 acme.chain.cert acme.key AES256-SHA:RC4-MD5 login.acme.com
```

aiCache doesn't provide a default value in order to protect your site and its visitors by letting them negotiate weak ciphers – unknown to you, the site's administrator.

While AES256-SHA is presented by aiCache as the default cipher and will be selected as such by majority of newer browsers (IE, Firefox, Chrome, Opera, Safari, etc), you might get a fair bit of traffic from older browsers that don't support such strong ciphers. In order for these clients to connect you must specify an alternative, more "relaxed" cipher, which, in our case, is specified as RC4-MD5. If you don't, such outdated browsers will not be able to connect to aicache over HTTPS/SSL. Their HTTPS connections will be abruptly reset and fairly cryptic messages thrown at unsuspecting visitors. For debug purposes, you might want to enabled cipher list of "ALL" – this way, no matter what the client browser suggests as a cipher, aiCache will accept it. This might lead to some very weak ciphers being choses.

Of course, feel free to tailor cipher list to your liking/requirements, but make sure to test the settings before applying them in production. WireShark is an excellent tool that can capture and display the intricacies of HTTPS/SSL protocol in all its detail, should you encounter a tough nut to crack.

Disabling SSLV2.

You can configure aiCache to disable SSLV2. This protocol has well published vulnerabilities and you might consider disabling it, as a good practice or for security compliance reasons. To do so, set global-level flag setting: **disable_SSLV2 .**

HTTPS/SSL error counters.

aiCache collects and reports both client-side and OS-side SSL init errors. These normally happen when aiCache cannot negotiate an SSL connection with clients or servers. Large number of such errors could mean misconfigured cipher list or wrong certificate. Counters are reported via CLI, SNMP and Web interfaces.

Limiting websites to HTTPS traffic only.

By default, with both HTTP and HTTPS enabled, aiCache allows access to all of the configured websites via both HTTP and HTTPS protocol. When accelerating multiple websites, you might need to limit some of them to HTTPS traffic only. To accomplish that, set **https_only** flag in appropriate website section of the aiCache configuration file. You can also specify **https_only_redirect** setting, specifying redirect location - aiCache will then send a redirect response when HTTP request comes for HTTPS-only website.

```
website secure.acme.com
https_only
https_only_redirect https://www.acme.com
```

Likewise, you can limit websites to HTTP traffic only:

```
website www.acme.com
http_only
http_only_redirect http://www.acme.com
```

And lastly, when using **rewrites**, you can make them applicable to HTTPS or HTTP only, see **rewrite** section for more information.

Origin Server traffic with aiCache in HTTPS mode.

By default, aiCache doesn't encrypt traffic to origin servers. Even when an HTTPS request is received and as aiCache encrypts the traffic between itself and client browser, the traffic to origin servers is sent just as usual - in HTTP, plain mode. So origin servers don't need to support HTTPS, be configured for it or waste CPU cycles performing encryption.

Most sites will find this arrangement very convenient. With aiCache and origin servers located in the same datacenter, there's no security compromise as unencrypted traffic between aiCache and origin server is confined to trusted network segment and cannot be observed outside of such segment.

Alternatively, you can configure aiCache in a more traditional fashion, to use HTTPS to obtain responses from origin servers, in response to HTTPS requests. To do so, specify **use_os_https** setting in website sections:

```
hostname secure.acme.com
...
...
use_os_https
# use_os_https AES-256:SHA secure.client.cert secure.client.key
...
```

This setting takes up to 3 optional parameters: cipher list, client certificate file and private key file. You don't have to provide any of the three. If you do need to provide a certificate file, you then must provide a cipher list and a private key file. You can generate self-signed certificate and key, as per instructions in this chapter.

When **use_os_https** setting is set for a website, the website must have at least one origin server tagged as *HTTPS origin server*. To accomplish this, use **origin_https** setting, instead of **origin:**

```
hostname secure.acme.com
...
...
use_os_https AES-256:SHA secure.client.cert secure.client.key

origin 1.1.1.1 80
origin_https 1.1.1.2 443
origin_https 1.1.1.3 443
```

When used with HTTPS OS traffic, aiCache supports all the regular features, including OS keep-alive and OS tagging.

In some setups you might decide to secure communication between aiCache and origin server, using client-certificate feature of HTTPS. Effectively, you'd tell origin servers to check for client certificate and only if client certificate, as presented to OS by aiCache, matches certain criteria, you'd allow the request through. Alternatively, you might accomplish some of the same functionality using a simple IP filter, limiting traffic to OS only to a set of IP addresses.

Please note that aiCache doesn't forward the true client certificate, as seen in client requests (nor would it even have the ability to do so). Instead, it presents the optional certificate that you configure via **use_os_https** setting.

If you know that certain auxiliary content (such as images, JS, CSS) in HTTPS pages can be obtained from your origin servers via HTTP, you can configure matching patterns with **fill_over_http** flag. This will reduce the HTTPS load on origin servers and likely improve you site's performance. For example:

```
pattern .jpg simple 1d
fill_over_http
pattern .css simple 1d
fill_over_http
pattern .js simple 1d
fill_over_http
```

To understand this feature better, imagine a secure web page, for example **https://secure.acme.com/account.php**. Imagine that the page has 3-4 CSS and JS files referenced in it and 10 or so of assorted reference, all of these being referenced from under **https://secure.acme.com/** . Normally, aiCache would retrieve these auxiliary elements via HTTPS. However, if origin servers are configured so that the same elements can be retrieved via **http://secure.acme.com**, then you can use the **fill_over_http** setting under matching patterns. Now aiCache will use HTTP, not HTTPS, to retrieve these elements from origin server(s).

In order to help origin servers to identify HTTPS requests, aiCache forwards a special header name and value to origin servers. The application code can then programmatically see if the header is set and act accordingly. For example, let's say there's some functionality that you want to have exposed only when requests came over HTTPS and this is where you'd use this feature. Likewise, you might set up the OS servers

themselves to deny requests unless such header is present, effectively disallowing non-HTTPS access to such sites.

The header name defaults to X-AI-CONN and can be set via **https_hdr** global directive. The header value defaults to "https" and can be change via **https_val** global directive. If you must trust certain functionality to HTTPS connections only, we recommend you to change header name, value or both, to your own values, for additional security.

HTTPS request logging.

When logging requests in aiCache's own, extended format, aiCache marks HTTPS requests with capital "S" (secure), HTTP requests are market with capital "P" (plain). The letter is the last symbol in the access log line.

HTTP-to-HTTPS and HTTPS-to-HTTP redirection.

You might have a need to make sure that a site, a section of one or just a single URL, are always accessed via HTTPS or HTTP only. For example, while allowing browsing of an e-commerce site via HTTP, you want to make sure the user login page, basket and checkout pages are only available over HTTPS.

Likewise, when user is viewing a product description page, you don't want to burden your servers and/or end user computer with unnecessary overhead of HTTPS – no need to heat up our planet any more than we have to.

aiCache provides a number of accommodations for these requirements:

- Declaring a website as HTTP or HTTPS only with **http_only** or **https_only** flags. This is a "hard drop" configuration. aiCache will silently drop non-conforming request. You are not likely to want such harsh treatment for your visitors - so, optionally, you can tell aiCache to, instead of silent dropping of non-conforming requests, redirect users to a certain, *static* URL, via **http_only_redirect** and **https_only_redirect** parameters. Rather rigid method.

- Declaring a website *twice* as HTTP or HTTPS only via **match_http_only** and **match_https_only** flags. Effectively you create 2 different **hostname** sections with the same hostname and provide different set of rules for each. Please note that you must have same **hostname** defined twice, as otherwise, non-conforming connections will not be matched to a website. You'd use this method when you have widely diverging configurations for HTTP and HTTPS versions of the site – so you store each in its own section.

- Making patterns HTTP or HTTPS specific with **match_http_only** and **match_https_only** flags. You keep a single **hostname** section and accomplish what you want at pattern level

- Configuring HTTP or HTTPS-specific rewrites (deprecated, still described elsewhere in this guide).

Most of the time you want to be nice to your audience and provide grace redirects, as opposed to outright dropping of non-conforming requests. For example, if you only expect login.aspx to come in as HTTPS, you don't want to drop requests for that URL that come in over HTTP. Instead, you want to redirect user to HTTPS version.

Likewise, if a request comes in, over HTTPS, for catalog.aspx, but you want catalog functionality to serve of HTTP only, you don't really want to drop such request, but instead redirect it to the HTTP version of the site.

You'd use method number two when you want to have widely diverging configurations for HTTP and HTTPS versions of the site. Using method #2, this is what your configuration might look like:

```
website

hostname store.acme.com
match_http_only
…

pattern login.aspx simple 0
rewrite (.+) https://store.acme.com\1

# HTTP specific-settings
…


hostname store.acme.com
match_https_only
…

pattern catalog.aspx simple 0
rewrite (.+) http://store.acme.com\1

# HTTPS specific-settings
```

You'd use method number three when HTTP and HTTPS versions of the site have largely the same configuration. Using method #3, this is what your configuration might look like:

```
website

hostname store.acme.com
…

pattern catalog.aspx simple 0
match_https_only
rewrite (.+) http://store.acme.com\1


pattern login.aspx simple 0
match_http_only
rewrite (.+) https://store.acme.com\1
```

...

aiCache Plug-in Support.

Introduction.

Starting with version 6 of aiCache, you can provide your own logic to execute when requests enter aiCache. You can have such logic executed for all requests, requests for certain website or requests matching certain patterns.

You code has access to most of request's information, including URI, body , Agent, Referrer, Cookies, Client IP and other information. Of course, some of these might not be present in every request.

You plugin code, after analyzing the request, can tell aiCache to:

- drop the request
- redirect the request to a provided location
- rewrite the request URL
- rewrite the request body
- modify request's TTL
- finish execution of plug-ins, so that any other plug-ins that yet to fire, are skipped
- log a certain message as a result of executing plugin code

A good example use of aiCache plugin functionality could be enforcing access token verification logic - looking for certain "access-token" parameter to be present in certain requests and programmatically validating the value of such "access-token" parameter.

aiCache allows you to have, for each plugin, a set of three function: init, exec and destroy. Only the exec function is mandatory, the init and destroy functions are optional. In the init function you can allocate a plugin-specific data structure. aiCache will then pass a pointer to that data structure to plug-n execution function. Same pointer will be passed to plug-in destroy function, if one is provided.

You can specify a number of plugin exec functions to be executed, one after another, in pre-determined order. Unless a prior plugin requests dropping of request, a redirect or specifies a complete response body, all of the plugins will be executed. Another way to break out of the plugin execution early is to return a special flag in plugin response status.

Sample C code, outlining basic plugin functionality, is provided within aiCache distribution file. We also provide a simple shell scripts that compiles plugin code into a shared module (Linux .so file).

aiCache employs a naming convention aimed at simplifying configuration chores. You specify plugin exec function by name, for example ***test_token***. aiCache then infers from this name, the names of optional *init* and *destroy* plugin functions, by adding ***_init*** and ***_destroy*** to the specified exec function name. In our example, aiCache plugin expected init function's name shall become ***test_token_init.*** Likewise, the expected destroy function name will be ***test_token_destroy.***

Defining plugins.

As you shall discover, writing plug-ins is a very straightforward exercise, requiring only minimal C coding skills and adhering to a few basic principles.

aiCache will attempt to locate and initialize plugin init function only when you provide init string, in the plugin declaration directive. These are to be placed in global section of aiCache configuration file. For example:

```
req_plugin_define demo_pl_func ./demo_pl_func.so aaa=1,bbb=2
```

The declaration directive is called **req_plugin_define** . It takes up to 3 parameters: a required exec function name (**demo_pl_func** in this example), module file name where this function is to be located **(./demo_pl_func.so**) and optional third parameter, an initialization string you want to pass to plugin init function (**aaa=1,bbb=2**). aiCache doesn't attempt to interpret the init string in any way and it is up to plugin developer just what and how is placed within the string.

The names of exec function and the plugin module file don't have to match each other. We recommend that you specify either full absolute or a relative path in the plugin file name, otherwise aiCache might have difficulty opening the plugin module file.

When and if the third parameter - the optional init string, is specified, will aiCache attempt to locate the init function. The init string will be passed as (const char *) to the init function. Init function is supposed to return a generic pointer (void *). This pointer will be passed to the plugin exec function every time aiCache needs to run the plugin exec function, as the second parameter, in addition to the request data.

Should init string be provided in the **req_plugin_define** directive, a valid plugin init function *must* exist within specified module file. Lack of the init function in this case is considered a configuration error and aiCache will refuse to start.

aiCache will always attempt to locate optional plugin destroy function within specified module file. Failure to locate a destroy function is not considered an error condition - aiCache will advise about lack of destroy function but will start up normally.

You can declare an arbitrary number of plugins, spread across a number of module files, or combined within a single file, it is up to you. aiCache will run usual series of tests to locate and optionally initialize defined plugin functions.

```
req_plugin_define demo_pl_func ./demo_pl_func.so aaa=1,bbb=2
req_plugin_define access_token ./access.so SomeInitValue
req_plugin_define message_plugin ./ message_plugin.so level:high
```

Attaching plugins.

After you define a number of plugin functions, as described in the previous section, you can then tell aiCache which plugin exec functions to fire when. You can configure some or all of the defined plugin exec functions to fire at global, website or pattern level.

Pattern level exec functions take precedence over website-level plugins, which in turn take precedence over global-level plugin. You attach plugin exec functions via **req_plugin_exec** directive. This directive can be placed in global, website or pattern sections. Same function could be used in different sections.

For example:

```
req_plugin_exec access_token
req_plugin_exec message_plugin
```

Each of the specified plugin exec functions must have been previously defined via **req_plugin_define** directive.

The order in which attach plugin functions is important - the functions are executed in the provided order. Normally, all of the specified plugins will be executed, unless one of the plugins returns a **PL_RET_F_BREAK** flag in its response status or instructs aiCache to drop or redirect the request, in which case aiCache doesn't execute the remaining plugin exec function.

Coding plugins.

A plugin header file is provided in **plugin/plugin.h** file within aiCache distribution file. We include content of this file here, for your convenience:

```
/*
 * plugin.h
 *
 * (C) 2001-2010 aiCache LTD aicace.com .
 *
 */

#ifndef PLUGIN_H_
#define PLUGIN_H_


#define AI_PL_V1 1

typedef struct pl_req_t {
   int version;
   void * pl_req_data;
} pl_req_t ;

typedef struct pl_ret_t {
```

```
      int version;
      void * pl_ret_data;
    } pl_ret_t ;


    /*
     * This is prototype for request-handling plugin function.
     * It takes a pointer to pl_req_t and returns a pointer to
     * pl_ret_t.
     *
     * It must dynamically allocate, using simple malloc or
     * calloc functions, pl_ret_t structure and any of the required
     * (char *) return parameters. Do not use alloca(). Do not return
     * pointers to static data.
     * Remember that malloc() doesn't initialize allocated memory.
     * Either manually set unused fields to NULL or use calloc().
     *
     * AI will free up the returned values after processing the return.
     *
     */
    /*
     * Optional tnit function receives a pointer to string of init data, returns
     * back an opaque data pointer that we then pass to both exec and
     * destroy functions
     */
    typedef void * (* pl_req_init_func_t) (const char *init_str);
    /*
     * Mandatory plugin exec function - gets passed req data and optional opaque plugin
     * specific data, created in plugin init function
     */
    typedef pl_ret_t * (* pl_req_exec_func_t) (pl_req_t *message, void *pl_data);
    /*
     * Optional plugin destroy function (cleanup etc) Gets passed optional opaque plugin
     * specific data, created in plugin init function
     */
    typedef void (* pl_req_destroy_func_t) (void *pl_data);

    /*
     * Complete description of a plugin
     */
    typedef struct pl_info_struct {

      char *name; // Name of exec function, init and destroy have _init and _destroy
suffix
      char *mod_name; // filename of share module file (something.so)
      char *init_str; // Init string... for example "key:val;key:val...", format and
value up to plugin
      void *pl_data;  // Optional, returned by optional init function
      int version;  //  Plugin data format versioning
      pl_req_init_func_t    init_func;
      pl_req_exec_func_t    exec_func;
      pl_req_destroy_func_t destroy_func;

      // Some stats
      unsigned long n_times_called;
      unsigned long n_times_dropped;
      unsigned long n_times_redirected;
```

```
    unsigned long n_times_url_rewr;
    unsigned long n_times_body_rewr;
    unsigned long n_times_resp_provided;
    unsigned long n_times_ttl_set;

} pl_info_t;
/*
 * This is how we pass incoming request to  plugins.
 * All char * pointers and ttl value are constant. Do
 * not modify content they point at or their values
 */
typedef struct pl_req_v1 {

    const char * method; //  request method string GET POST etc
    const char * version; //  HTTP version: HTTP1.0 HTTP1.1
    const char * protocol; //  HTTP or HTTPS
    const char * uri; //  request string: path + optional query
    const char * body; // when one is present
    const char * clip; // Client IP
    const char * agent;
    const char * ctype; // Request's Content-Type
    const char * referer;
    const char * cookie;
    const char * auth; // value of authorization header
    const char * host; // value of request's host header
    const char * al; // Accept-Language
    int ttl; // TTL as determined by aiCache - changes to this field are ignored

} pl req v1 t;



/*
 * This is is how plugin returns its result.
 * Exact action is indicated by OR'ng a number of flags.
 * To learn more about this mechanism:
 * http://en.wikipedia.org/wiki/Bit_field
 *
 */
typedef struct pl_ret_v1 {

    int action; // ORd collection of PL_RET_ flags
    int ttl;    // if you want to set a new TTL, use this field
    char * uri;
    char * req_body;
    char * redirect;
    int    resp_code;
    char * resp_body;
    char * sig_suffix; // when set, is added to the cache signature
    char * log_message; // A way for plugin to ask ai to log into error log
   char * exp_pat; // A way for plugin to ask to expire matching cached responses
   int os_tag; // If you want to influence os_tag selection
   int sets_session_cookie; // see this Guide for explanation of session_cookies
   int expires_session_cookie; // see above
   int session_cookie_required; // see above
} pl_ret_v1_t;
```

```
#define PL_RET_F_NA           (1 << 0) // Plugin took no action
#define PL_RET_F_DROP         (1 << 1) // drop the request
#define PL_RET_F_REDIRECT     (1 << 2) // redirect, char *redirect must be set
#define PL_RET_F_URI_REWR     (1 << 3) // URI rewritten, char *uri must be set
#define PL_RET_F_BODY_REWR    (1 << 4) // Body rewritten,char *body must be set
#define PL_RET_F_BREAK        (1 << 5) // Dont fire any following plugins
#define PL_RET_F_TTL          (1 << 6) // TTL value is provided
#define PL_RET_F_RESP_BODY    (1 << 7) // complete response body provided
#define PL_RET_F_OS_TAG       (1 << 8) // os_tag is set



#define PL_RET_F_WRONG_V      (1 << 31) // Data version mismatch

#endif /* PLUGIN_H_ */
```

The plugin exec function receives 2 input parameters and is supposed to return a pointer to a return data structure:

```
/*
 * Mandatory plugin exec function - gets passed req data and optional opaque plugin
 * specific data, created in plugin init function
 */
typedef pl_ret_t * (* pl_req_exec_func_t) (pl_req_t *message, void *pl_data);
```

Within **pl_req_t *message** parameter, is contained a version indicator and a pointer to actual request data:

```
/*
 * This is how we pass incoming request to  plugins.
 * All char * pointers and ttl value are constant. Do
 * not modify content they point at or their values
 */
typedef struct pl_req_v1 {

  const char * method; //  request method string GET POST etc
  const char * version; //  HTTP version: HTTP1.0 HTTP1.1
  const char * protocol; //  HTTP or HTTPS
  const char * uri; //  request string: path + optional query
  const char * body; // when one is present
  const char * clip; // Client IP
  const char * agent;
  const char * ctype; // Request's Content-Type
  const char * referer;
  const char * cookie;
  const char * auth; // value of authorization header
  const char * host; // value of request's host header
  const char * al; // Accept-Language
  int ttl; // TTL as determined by aiCache - changes to this field are ignored

} pl_req_v1_t;
```

As you can see, all of the parameters are const char pointers, meaning you should not modify them in any way and/or attempt to free them.

The return data structure is

```
/*
 * This is is how plugin returns its result.
 * Exact action is indicated by OR'ng a number of flags.
 * To learn more about this mechanism:
 * http://en.wikipedia.org/wiki/Bit_field
 *
 */
typedef struct pl_ret_v1 {

  int action; // ORd collection of PL_RET_ flags
  int ttl;    // if you want to set a new TTL, use this field
  char * uri;
  char * req_body;
  char * redirect;
  int    resp_code;
  char * resp_body;
  char * sig_suffix; // when set, is added to the cache signature
  char * log_message; // A way for plugin to ask ai to log into error log
 char * exp_pat; // A way for plugin to ask to expire matching cached responses
  int os_tag; // If you want to influence os_tag selection
} pl_ret_v1_t;
```

As you can see, you can specify an action to take, by ORing together a number of action flags, specify complete rewritten request URI, request body, provide a message for aiCache to log, provide an OS Tag for aiCache to use, etc. You can also return a complete response (header and body combined in a string buffer pointed to by **resp_body**), along with response code.

And of course, you must allocate the memory for the return data structure itself.

! ● When compiling plugins, make sure to use matching plugin.h header file from the aicache distribution tar file. The header file is located in the plugin subdirectory. Compiling pluging while using outdated plugin.h header file can lead to unpredictable behavior – core dumps etc.

! ● Should you need to return a string buffer or a set of same, via char pointers in the response data structure, allocate the space use malloc() or calloc() C functions (but not alloca()). Do not pass pointers to static data, as aiCache will attempt to free() all of the non-NULL pointers after receiving the response data from your plugin. Should you want to use static data, strdup() it first.

! ● Remember that malloc() doesn't initialize allocated memory. Either manually set unused fields to NULL or use calloc(). Should you attempt to pass a return structure with uninitialized, garbage-filled fields back to aiCache, it will lead to memory corruption and aiCache shutdown.

Take a look at the provided **plugin_demo.c** C source file to see how to use some of the return values.

```
   pl_ret_t * demo_pl_func(pl_req_t * plreq, void *pl_data) {


      /* Allocate memory for return data container */
      pl_ret_t * plret = calloc(1, sizeof(pl_ret_t));

      /* Set return type to version 1 */
      plret->version = AI_PL_V1;

      /* Allocate memory for  version-1-specific return structure */
      plret->pl_ret_data = calloc(1, sizeof(pl_ret_v1_t));

      pl_ret_v1_t *pl_resp_data = plret->pl_ret_data;

      pl_req_v1_t *pl_req_data = plreq->pl_req_data;


      /* Make sure we're passed what we expect, which is V1 data format */
      if (plreq->version != AI_PL_V1) {
            pl_resp_data->action |= PL_RET_F_WRONG_V;
            return plret;
      }


      printf("[PLUGIN PLUGIN PLUGIN] URI: %s\n", pl_req_data->uri);

      pl_resp_data->log_message = strdup("Hello there ....");

      /*
       * At this point we have allocated response structure.
       */
   // if (pl_req_data->uri && strstr(pl_req_data->uri,"/")) {
   //       pl_resp_data->action |= PL_RET_F_DROP;
   //       return plret;
   // }

   // if (pl_req_data->uri && strstr(pl_req_data->uri,"/")) {
   //       pl_resp_data->action |= PL_RET_F_URI_REWR;
   //       pl_resp_data->uri = strdup("/violation.html");
   //       return plret;
   // }

   // if (pl_req_data->uri && strstr(pl_req_data->uri,"/")) {
   //       pl_resp_data->action |= PL_RET_F_REDIRECT;
   //       pl_resp_data->redirect = strdup("http://www.aaabbb.com");
   //       return plret;
   // }

   // if (pl_req_data->uri && strstr(pl_req_data->uri,"/")) {
   //       pl_resp_data->action |= PL_RET_F_RESP_BODY;
   //       pl_resp_data->resp_body = strdup("HTTP/1.1 200 OK\r\nBlah:
yadayada\r\n\r\n");
   //       return plret;
   // }

      /*
       * If we get here, we should allow request to pass through unchanged
       */
```

```
    pl_resp_data->action |= PL_RET_F_NA;

    /*
     * Send PL_RET_F_BREAK break out of plugin list
     */

//  pl_resp_data->action |= PL_RET_F_BREAK;
    return plret;

}
```

Please note that plugin can affect signature of cached responses by specifying its own signature suffix. The **sig_suffix** value set in the plugin response structure will be added to the cache signature .

Plugin can optionally request expiration of cached responses by setting **exp_pat** value. It will be treated as a regular expression and all matching entries will be expired. Note that only entries cached for request's website will be expired (only applicable when multiple websites are accelerated). Number of expired responses will be logged in aiCache error log file.

Should you require some sort of initialization functionality, you should code it in a plugin init function. Allocation of some shared data structure is a common use requiring an init function. aiCache expects init function to return an opaque void pointer to some sort of plugin-specific data. The same pointer shall be passed to plugin exec function, as second parameter (see example code above).

Here's a very simple init function - returning a pointer to static data buffer, filled with "INIT INIT INIT":

```
void * demo_pl_func_init(const char *str) {
    return "INIT INIT INIT";
}
```

aiCache runs the optional module init functions before any of the exec functions are called. When plugin init functions are ran, the STDOUT is still open, so you can use simple printf() C functions to print out init diagnostic messages or log them to some form of file output.

The same opaque void pointer returned by the plugin init function is passed to optional plugin destroy function, should one be found within the module file. You'd normally perform some form of data cleanup in the destroy function.

```
void demo_pl_func_destroy(void *pl_data) {
}
```

Compiling plugins.

A simple plugin compilation script is provided for your convenience, showing how to compile a single plugin source file into a single plugin module (Linux .so file):

```
gcc -Wall -fPIC -c demo_plugin.c
gcc -shared -Wl,-soname,demo_pl_func.so -o demo_pl_func.so demo_plugin.o
```

You can modify the script to compile a number of source files, possibly combining all of the plugin functions into a single plugin file. Or, you can place each set of plugin functions, into its own plugin file. Should you end up with a number of plugins to maintain, a makefile could be in order.

! When compiling plugins, make sure to use matching plugin.h header file from the aicache distribution tar file. The header file is located in the plugin subdirectory. Compiling pluging while using outdated plugin.h header file can lead to unpredictable behavior – core dumps etc.

Plugin statistics.

aiCache internally tracks how many times each plugin has been executed. It also tracks other actions taken by plugins, you don't have to code any such stat gathering logic into your plugin code. The stats are available via Web, CLI and SNMP stats. Here's a set of counters output in the SNMP stat file (please note that plugin statistics start at sub-index of .1000.) :

```
######## Plugin-level stats #########

.1.3.6.1.4.1.9999.1.1000.0.0 string demo_pl_func # exec function name
.1.3.6.1.4.1.9999.1.1000.0.1 integer 9 #number of plugin execs
.1.3.6.1.4.1.9999.1.1000.0.2 integer 0 #number of plugin drops
.1.3.6.1.4.1.9999.1.1000.0.3 integer 0 #number of plugin redirects
.1.3.6.1.4.1.9999.1.1000.0.4 integer 0 #number of plugin URL rewrites
.1.3.6.1.4.1.9999.1.1000.0.5 integer 0 #number of plugin body rewrites
.1.3.6.1.4.1.9999.1.1000.0.6 integer 0 #number of plugin provided resp
.1.3.6.1.4.1.9999.1.1000.0.7 integer 0 #number of plugin TTL sets
```

As usual, you can configure your SNMP monitoring SW to collect, graph and alert on any of these SNMP counters.

Access and Error Log Functionality.

Introduction.

Logging access information is one of the least exciting yet important aspects for many web sites. Some web sites are driven by advertising revenues and have to provide usage statistics based on (or verified by) the information that is available in the log files – as opposed to outsourcing such analysis to external Web Analytics provider.

Some websites perform significant post processing on the log files for various data mining, troubleshooting and other purposes. Thus it is crucial to collect all the relevant information in the log files.

Simplifying and speeding up logging routines and overall log-related workflow is also important, especially for larger sites.

aiCache provides **advanced logging capabilities**:

- It collects much more information in its log files compared to a typical web server.
- It can suppress logging for non-essential requests, that too often end up polluting and increasing size of log file, while no adding no value.
- It allows to decimate, by a configurable factor, access log files - ideal for sites with heavy traffic, where you still want to collect a small sample of traffic for troubleshooting purposes, but cannot afford to log every request.
- You can opt not to process/collect/log **User-Agent** header value, resulting in marked decrease of log file size.
- You can opt not to process/collect/log **Referer** header value, resulting in marked decrease of log file size.
- It allows for on-demand or size-based instantaneous log file rotation, with no downtime or user impact.
- In addition to traditional access logging, it can collect and log a rich set of run-time statistics. Every few seconds these are generated and written out to statistics log file. aiCache reports RPS, response times, number of outstanding client and origin server connections, response cache size and other crucial system statistics.

Configuring logging

All you need to do to configure logging is to point to location of log directory and optionally, provide names for *access* and *error* log files in case you don' t like the default names of, well, *access* and *error*. You may also specify if you want to have health check requests logged as well.

```
logdirectory /var/log/aiCache
log_healthcheck
accesslog access
errorlog error
```

Size-based access log file rotation

aiCache provides automated, on-the-fly, size-based access log file rotation. This feature is typically used to cut down size of the access log files to some reasonable value, as it might be easier to deal with 10 files of 100MB each, as opposed to a single file of 1GB. To enable, simply set:

```
max_log_file_size 100000000
```

Be careful with those zeroes – the size is specified in bytes ! When **max_log_file_size** is set, aiCache rotates present log file upon it reaching the maximum size. The renamed access file has a timestamp added to the name of the file. The size-based rotation is a zero-downtime operation and happens automagically.

On-demand log file rotation via CLI or USR1 signal.

To execute on-the-fly, on-demand log file rotation, send SIGUSR1 to aiCache's process ID at time of your choosing - manually or using a script driven by Linux cron facility. For example:

```
pkill -USR1 aicache
```

Simply add this command to be scheduled by Linux *cron* facility at time that is convenient for you.

aiCache's log file rotation – size-based or on demand, is a zero-overhead and zero-downtime operation, so feel free to do it on schedule that is convenient for you, there's no performance impact. Please note that aiCache rotates log files *after* the rotation is requested, upon receiving first log-enabled request.

You can also tell aiCache to rotate the log files from the CLI, using *rotatelog* command (see the CLI chapter for more details). Log file rotation is not cluster-wide, it is only the aiCache instance that the command is directed at, that executes the rotation.

Access log file formats.

aiCache supports 2 different log file formats: “apache” and “extended”. Apache log format is the same as default format used by the popular Open Source web server – so that any existing log processing tool can be used, without any changes, on aiCache log file.

Extended format is an extension of Apache format – it simply appends a number of fields (fields from 10 to 21, in the table below). These extra fields include: origin server response time, whether the response came from cache (cache-hit) or origin server (cache-miss), TTL, compression flag (gzip'd or not), origin server (both IP and port) that was used to obtain the response, unique connection “ID” etc.

Let's now explain log file format. Access log file is a text file where each line contains information about a single request/response against your Web Site. Each line consists of number of fields, separated by whitespaces. Whenever there’s a field that might have whitespaces within it, such fields are enclosed within double quotes to remove any ambiguity.

When a field doesn’t have a value, it is logged as “-“ (dash). For example, responses served from cache don’t have an associated origin web server (as one doesn't need to be accessed to serve cached responses) – so both OS IP and OS port number are set to “-“ .

Please note that collecting and logging out of certain request headers, such as *cookie*, *user-agent* and *referrer* are likely to significantly increase the size of the log files, so collect these only when you must.

Request’s cookie header is not logged by default. To enable logging of cookie header, please set **log_cookie** setting at website or pattern level. The setting is a flag and requires no value.

The cookie value is always logged as the last field of the access log file line. It is enclosed in double quotes. Should double quotes be used within the cookie header itself, they are passed verbatim to the log file and are not escaped. Be aware of that if you plan to do parsing of the cookie values and/or quotes matching.

Lines in the aicache log file are limited to maximum of 4K characters in length.

Let's describe the fields collected in aiCache's log files:

#	Meaning	Example
1	Client IP (numeric, never a DNS name), followed by optional parsed-out-forwarded Client-IP or "-" when not available or not configured to parse out forwarded client IP.	1.2.3.4 2.2.2.2
2	Not used, always set to '-'	'-'[15]
3	Not used, always set to '-'	'-'
4	Date/time of request in format [DD/MON/YYYY:HH:MM:SS GMTOFFSET]	[17/Dec/2008:16:10:18 -0500]
5	Complete request string, enclosed in ""	"GET / HTTP/1.1"
6	Response HTTP status.	200
7	Response Size (bytes) followed by Request size (as sent to origin server, requests satisfied from cache have it set to "0").	12345 234
8	Referrer string, enclosed in ""	"http://a.b.c/from.html"
9	Browser string, enclosed in ""	"Magic Web browser 2.0"
10	Request 's Host: value	www.acmenews.com
11	Request's TTL value, in seconds, 0 for non-cacheable requests	120
12	Cache hit flag: 1 for cache hit, 0 for miss or 0 TTL	1
13	Request fill time , milliseconds - time from obtaining complete request to obtaining complete response	10
14	Request send time , seconds - time from obtaining complete response to writing of last byte of response to the client.[16]	3
15	Response "Gzipped" Flag: set when response was served in compressed form.	1

[15] this one and one below are artifacts from the long-gone days of "identd" - identity daemon and were never actually used.

[16] It is only an approximation as there's no way to know when that last byte was actually *received* by the client.

16	Response Gzipped On-the-Fly Flag: set when a response was compressed on-the-fly by aiCache and served compressed to the client	1
17	Origin Server IP:PortNumber. Set to "-:0" for responses served from cache.	1.2.3.4:80
18	Connection number (unique for ea new connection, growing from 1 at start).	13452345234
19	# of keep-alive client requests served by this connection, followed by number of keep-alive origin server requests, followed by preload-flag (1: response was obtained from a preload queue, 0: response was obtained normally).	12:5:0
20	Client File Descriptor Number, followed by OS FD .	234:235
22	OS Index as detected in request, -1 when none requested	8
23	Number of attempts it took to obtain a response for this request. Is "1" for most requests. When this field is set to 10, it indicates request that failed response size enforcement check (response size was smaller or large than mix and max limits configured)	1
24	"S" for HTTPS request, "P" for HTTP request	S
25	Bad response error code. 0 for no error. See dedicated section on bad response diagnostic.	0
26	Request's Cookie header, when cookie logging is turned on via log_cookie website or pattern-level flag. When cookie header is absent or cookie logging is turned off, a dash "-" is written out .	"uname=Mr.Pink" "-"

For performance reasons aiCache never resolves numeric IP, client's or origin servers', to DNS names.[17]

[17] Doing it is very time and resource consuming. Sites that are interested in this information usually obtain it during post-processing of log files.

As you can see a number of extra fields are added to the access log files when collecting "extended" access information. These extra fields are very helpful in assisting with optimum configuration and troubleshooting of aiCache instance.

The *error log* file contains information, in a free flowing form, for certain kinds of errors, warnings, notices and assorted messages, but predominantly uses the same format as *access log* file. Each line is time-stamped in the same format as in access log file. You can use information in the error log file to help with troubleshooting and similar purposes.

While access log files can be configured so that each website is logged to its own, dedicated access log file (**logging dedicated** setting), error log file is always shared across all of the websites.

Selective Log suppression

aiCache provides another important feature: log suppression. As it was mentioned before, busy web sites often have to deal with very large log files. For some of such web sites the situation is so bad that they have to rotate log files every hour or so. Otherwise the log files grow to completely unmanageable size. It means that in order to offload log file from web servers they have to spend hours to just transfer the log files to log processing systems.

Yet if we take a look inside of a typical Web log file, we would see that large percentage of it is of no interest to us. Assuming client-side analytics is not used and server-side log crunched is employed instead, typical web sites only care about impressions or user hits against points of interest on their website, such as homepage, section fronts, feature sections, promos, polls, registrations and contests and other important content. Once again very rarely do companies care about how many times 1x1 pixel image files or JavaScript files, or content style sheets were downloaded from their site. Yet this information invariably ends up in the log files and increases log file sizes many-fold.

aiCache improves this situation by allowing you to specify what requests should be excluded from the log files. To enable this feature, simply set "**no_log**" for matching patterns. This feature is a godsend for busy web sites. We strongly advise you to use this feature to simplify and speed up your log file processing. *Have it habitually turned on for .js, .css and images***.**

Another way to control the size of the log files is turn off collection and processing of either (or both of) User-Agent or Referrer HTTP headers. To accomplish this, please set **drop_user_agent** and **drop_referrer** settings in global section of the configuration file.

Log decimation.

aiCache provides another important feature: log decimation. It differs from selective log suppression - instead of completely turning off logging, it lets you to decimate it by a configurable factor. So you still get some request imprints in the log file, while reducing their number by a configurable factor. You might want to log *a sample of* requests for troubleshooting and other purposes.

To configure, set pattern or website-specific **decimate_log** setting. For example, when set to 100, every 100th request is logged, when set to 355, every 355th and so on. Each website can have its own **decimate_log** setting.

In order to override website-level decimation setting, you can set a different one at pattern level. If you decide to log *every* request matching a pattern when website-level decimation is turned, set **decimate_log** to -1 at pattern level:

```
decimate_log -1
```

Logging requests sent to origin servers.

If you need to always log requests that are sent to origin servers (such as first fill or refresh of a cacheable request, or a 0 TTL request), specify **log_os_hit** in pattern section. This setting overrides any other logging settings, including decimation or **no_log**. For example:

```
pattern expensiveApiCall.php simple 10
log_os_hit
```

Logging slow responses from origin server.

If you need to always log requests that were sent to origin servers and took longer than certain amount of time to fill, specify **log_osrt** in pattern section. This setting overrides any other logging settings, including decimation or **no_log** . For example, to log requests that took more than 1000 ms (1 sec) to fill:

```
pattern canTakeLongTime.aspx simple 10
log_osrt 1000
```

Error logging logic when under duress.

In unlikely event of a DOS attack against your site or some other error condition that causes abnormally high volume of invalid, bogus or malicious requests and/or connections, aiCache doesn't lose its cool. It uses special logic to make sure such flood of mischief traffic is handled with flying colors. In addition, error messages are decimated so as to not overflow the file system: aiCache logs no more than 100 error messages per minute.

Stayin' alive...

An all too common web site failure is running out of space in log partition. It can happen due to a failure to rotate and offload log files, unexpected spike in traffic and similar conditions.

A normal reaction to such out-of-space condition is for web server to abort serving requests (exit) and hope that some kind of monitoring system catches such down condition, someone notices the alert, logs in and rectifies the problem. All the while the users are receiving error message. Clearly less than desirable situation.

If it is more important to you to continue serving user requests than it is to make sure each user request is logged, you can instruct aiCache to ignore write failures via **stay_up_on_write_error** global setting.

Front-ending with aiCache, the additional benefits.

With aiCache front-ending client requests – receiving and processing them before they ever get to your origin web servers and other components of your infrastructure, aiCache is capable of providing even more benefits in addition to caching of responses.

Off-loading of TCP/IP processing, request/response delivery and enforcing time, size and sanity limits on requests.

Due to extremely efficient, non-blocking multiplexed IO model of aiCache, it is ideally suited to front-ending user traffic , while deflecting a number of different DOS-type attacks with little or no overhead.

aiCache enforces *time-limits*: a client is supposed to provide a complete, well-formed request header within certain amount of time – it is configurable via **max_client_idle_time** setting . Likewise, after providing request header, client is then supposed to provide a complete, well-formed request body (a complete request) within certain amount of time – it is configurable via **max_client_body_time** setting.

It helps to protect against potential DOS attack where a large number of idling connections are being opened to a web site in hope of overwhelming its capacity to sustain large number of open connections/TCP sessions.

aiCache enhances concept of protection by providing additional security features. As you might know, sometimes hackers try to attack web sites by feeding them URLs configured in certain way, most often to cause buffer overflows with resulting execution of carefully constructed code. When such malformed requests or URLs are discovered, websites are often left at the mercy of their Web software providers to issue quick patch to address the problem. Often there's no way to prevent this dangerous URLs from hitting the web servers, as there is nothing in front of the victimized web servers to filter out these dangerous URLs.

To protect against such malicious/malformed requests, you can specify maximum HTTP header size and body size, via **max_header_size** (defaults to 8KB) and **max_body_size (**defaults to 1MB), both must exceed server MTU to be enforceable. You can also use URL blocking (see below) for DOS protection.

You can also enforce URL length limit at website or pattern level by using **max_url_length** parameter in the respective section. Pattern setting overrides website-level setting, if any .

To assist in troubleshooting of malicious/malformed requests, you can configure aiCache to write out such requests to /tmp directory via **dump_bad_req** setting. Beware of privacy concerns, should there be a private user data in the requests. No more than 120 bad reqs are written out per minute. No more than 4KB of body data is written out - to safeguard aiCache server from exhausting disk space. Consider periodic cleaning out of such written out bad requests when you enable **dump_bad_req** setting.

You can tell aiCache to only accept HTTP GET, POST and HEAD requests, by setting **get_post_head_only** setting at website level.

You can configure aiCache to not send an error response whenever a bad or blocked request is detected and instead, immediately drop the client connection (via TCP reset), so that no error response is sent to the client. Such behavior is configured via **silent_400** setting, in global or website section(s) of the configuration file.

Please note that with certain bad/malformed requests , such requests can not even be matched to a configured website, so you must set **silent_400** at global level if silent dropping of connections is the desired effect.

Request blocking and basic redirection.

You can also explicitly specify list of patterns that are *blocked* outright by the aiCache. To block a pattern, simply define it in the configuration file and add "**block**" directive below the pattern definition. For example:

```
pattern exploited.html simple 0
block
```

You can also configure aiCache to block or drop POST requests. To block or drop a pattern, simply define it in the configuration file and add "**post_block**" or "**post_drop"** directive below the pattern definition. For example:

```
pattern noPostsAllowed.html simple 0
post block

....
pattern PostProhibited.html simple 0
post_drop
```

You can also opt to *redirect* requests matching certain pattern. Use **redirect_location** directive below the pattern definition. For example:

```
pattern nolongerhere.html simple 0
redirect_location http://aaa.bbb.ccc/newlocation.html
```

Typically you'd want to provide a complete URL for the new location, including the **http://** prefix.

Normally aiCache issued a “302” redirect. If 301 redirect is desired instead, specify redirect_location_301 instead:

```
pattern nolongerhere.html simple 0
redirect_location_301 http://aaa.bbb.ccc/newlocation.html
```

Please be also aware of the *URL rewrite* and *rewrite-redirect* features of aiCache, described elsewhere in this document.

Filtering of request bodies (BMR patterns).

You can also define a list of regular expression patterns to apply to *request bodies* (when such request bodies are available). Should content of request body match one of such body-match-reject (BMR) patterns, aiCache will drop the matching request.

You can use this functionality to filter out common threats such as SQL injection or any other body patterns that you might find troublesome.

The BMR patterns are defined via **bmr_pattern** directive . Arbitrary number of these could be placed in global, website or pattern section of the aiCache configuration file. For example:

```
bmr_pattern update\s+
bmr_pattern delete\s+
bmr_pattern insert\s
```

Flexible request decimation.

You can configure aiCache to apply flexible request decimation logic. For example, should your site be bombarded by certain requests that you don't mind replying to under normal (low load) condition, yet would like to decimate when traffic starts to spike, you can use request decimation feature.

In its most basic form, you simply tell aiCache to allow a portion of matching traffic to be allowed through via **decimate_req,** no matter the load:

```
pattern notalwaysthere.asp simple 0
decimate_req 10 # Only let through 1/10 of the traffic
```

In more evolved setup, you can configure aiCache to apply decimation when load (pattern or website-level RPS) exceed certain threshold:

```
pattern notalwaysthere.asp simple 0
decimate_req 10 # Only let through 1/10 of the traffic
```

```
decimate_req_pat_rps 100 # Only decimate when this pattern sees more than 100 RPS
```

Or, testing against website-level RPS:

```
pattern notalwaysthere.asp simple 0
decimate_req 10 # Only let through 1/10 of the traffic
decimate_req_ws_rps 400 # Only decimate when this website sees more than 400 RPS
```

When activating pattern-level decimation in this load-driven fashion, aiCache will turn the decimation on for 60 secons, before re-testing for the load condition and possibly turning decimation off. You can change this interval by setting **decimate_req_interval** pattern-level setting. You can think of this interval as a hysteresis feature.

```
pattern notalwaysthere.asp simple 0
decimate_req 10 # Only let through 1/10 of the traffic
decimate_req_ws_rps 400 # Only decimate when this website sees more than 400 RPS
decimate_req_interval 600 # decimate for 10 mins before retesting the load condition
```

Note that when requests are decimated in such fashion, they are silently and instantly dropped, to reduce the load on aiCache. No grace responses and/or redirection of any sort is sent back to the requesting clients.

Also note that aiCache automation activates pattern-level stat gathering when you enable request decimation.

The number of decimated requests is reported via usual means – the Web, CLI and SNMP stats.

Operating Accelerated Websites in Fallback Mode.

Sometimes origin server infrastructure might be in a state when it is unable to serve any requests. A number of reasons could lead to such state - DB or file store failure are some examples. Normally, without aiCache front-ending the requests, the website would be in a hard down situation - not serving anything at all or serving all kinds of errors to the end users.

With aiCache you can save the day by going into "forced fallback" mode and have aiCache serve available cached content so that the site retains some or most of its content and functionality. Please see a dedicated Fallback mode chapter for more information on it.

Operating Accelerated Websites in HC Fail Mode.

Sometimes you might want to force all of the traffic to go to origin servers instead of aiCache servers, for example when you need to perform some maintenance. It is easiest to accomplish when you have a load

balancer - such as HAProxy, F5, A10, Netscaler, Alteons etc, load balancing between aiCache servers, while having origin servers as "warm" standbys. So under normal conditions, it is aiCache server(s) that receive all traffic, but when aiCache is not available, traffic starts flowing directly to origin servers.

To accomplish this in most flexible fashion, you can enable content-matching health checks (HC) on the load balancers. Direct the HC to pull aiCache's own website-statistics Web page - for example (we cite default URL, yours might be different if you specified a **stat_url** setting)

/accelstattext?www.acmenews.com

Configure the load balancers' HC to match for "hcgood". When website is operated normally by aiCache, this string is always present in the aiCache's Website statistics page.

To force aiCache to fail the LB's health check, you use CLI **hcfail** command. When issued against a website, it forces removal of "hcgood" from aiCache's Website stats page - which can be detected by load balancer's HC probe, leading to LB disabling aiCache and starting to forward the traffic to origin servers instead. Please note that **hcfail** CLI command is cluster aware.

[Deprecated] User-Agent based redirection.

This feature is only available in mobile-enabled version of aiCache. Deprecated starting with version 6.267, use much more flexible UA-tagging feature instead.

You can configure aiCache to redirect requests based on request's User-Agent information, acting on the type of browser used to access the site. This is most commonly used to redirect mobile users to mobile version of the site.

For example, when a request to **www.acmenews.com** is a made from a device running Android operating system, you can assume it is being made from a mobile device and want to redirect such request to **mobile.acmenews.com .** aiCache provides for two different methods to accommodate for this.

Please note that while you can have some kind of application logic on origin servers determine if a redirection must be issued based on User-Agent string, doing so will require forwarding requests to origin servers for that determination to be made, effectively negating some of the caching benefits. Not only that, maintaining and executing the redirection logic on origin server has a price tag and maintenance overhead.

Having aiCache decide when to redirect solves all of these issues and is the preferred way to go about redirecting mobile users.

Website or pattern level redirection, inverted logic.

You configure aiCache to *allow* certain User-Agent strings *in*, while telling it to redirect other requests with any other User Agent strings to a website of your choosing, optionally forwarding the original URL along. After a request is allowed in this fashion (as opposed to being redirected), normal processing logic applies.

The requests without User Agent string specified are always processed as usual, no UA-based redirection can happen here as there's no UA string to process.

The basic idea is configure aiCache to process requests from known PC-based browsers, such as IE, Firefox, Safari, Opera, Chrome - running on Windows NT, XP, Mac OSX, Linux etc, while redirecting all other User Agents to a different, mobile site. So you'd provide a list of a dozen or so of common OS versions - these are always present within UA string, have aiCache match against this very short list, as opposed to providing a list of exceptions (mobile UA strings) . The list of mobile UA strings is very long and is growing by the day, so maintenance is bound to be quite a chore, this is why it is sometimes better to implement the redirection logic "the other way around".

At the same time, you can specify UA match strings that are to be redirected.

To configure such behavior you need to specify one or more of **ua_pattern** at website or pattern level (when specified, pattern level settings take precedence). These are patterns that are matched against incoming request's User-Agent string and matching requests are processed by aiCache, just as usual, while those that don't match are redirected. For example:

```
ua_pattern Windows\sXP keep
ua_pattern Windows\sNT keep
ua_pattern iPhone redirect
```

To specify the location (host) to which non-matching User Agents are redirected, you may specify **ua_redirect_host :**

```
ua_redirect_host mobile.acmenews.com
```

Please note that when a redirection needs to happen, all of the requests are redirected to the same host, specified as **ua_redirect_host.** Alternatively, you can add optional rewrite-from and rewrite-to parameters to **ua_redirect** like so:

```
ua_pattern iPhone redirect originalurl.html acme.com/newurl.aspx
ua_pattern Android redirect story=(\d+) android.acme.com/article.php?id=\1
```

As you can see, such UA-driven request-redirection is very flexible, as it allows you to rewrite original URL using regular expression symbol. Please note that redirects are always issued in "absolute" fashion, with leading **http://** prefix .

Please note that you specify optional **fallthrough** flag in *pattern-level* **ua_pattern** settings, as long as **ua_pattern** is qualified as **keep:**

```
ua_pattern Windows\sXP keep fallthrough
ua_pattern Windows\sNT keep fallthrough
```

```
ua_pattern iPhone redirect
```

With **fallthrough** specified, aiCache will continue matching the request's URL to remaining URL patterns. Effectively, having this setting lets you tell aiCache "If UA matches, *keep* the request and *continue* URL matching to remaining URL patterns". You might it very helpful to streamline complex mobile/desktop UA-driven URL rewriting/redirection.

The **fallthrough** setting cannot be specified in *website-level* **ua_pattern.**

By default, when no URL rewrites are specified in **ua_pattern** settings, aiCache redirects to the original request URL, unchanged, on **ua_redirect_host** host. You can configure it not to forward the original URL (redirecting to just home page instead) by specifying **ua_redirect_host_only** setting. It is a flag and requires no value

```
ua_redirect_host_only
```

Alternatively, you can configure aiCache to redirect requests (assuming UA doesn't match any of the **ua_pattern** with its own URL rewrite string) , to a different URL by specifying **ua_redirect_url** setting:

```
ua_redirect_url /news.aspx
```

And still, you can also rewrite the original URL (assuming UA doesn't match any of the **ua_pattern** with its own URL rewrite string) and use the rewritten value by specifying **ua_redirect_rewrite** :

```
ua_redirect_rewrite .+ /news.aspx
```

Just like the regular URL rewriting, **ua_redirect_rewrite** takes two parameters: the regexp match and the rewrite string. As usual, you can use back-references in the replacement string.

To simplify and speed up processing of requests - determining if they should be processed by aiCache or redirected to a mobile site, aiCache issues a special cookie, with default name of **xaikeeperua,** whenever a request is accepted or redirected. This way the following requests from the same client browser will carry this cookie and aiCache can more quickly establish whether request needs to be handled by aiCache, as opposed to getting redirected.

To optionally modify default value of this cookie, set **ua_keep_cookie** directive at website level, followed by cookie name (it defaults to **xaikeeperua**). When User Agent string is present and is matched of one of the **ua_keep_pattern**, the cookie to set to "yes" - indicating that we ***keep*** this request with aiCache instance. It is set to "no" otherwise - when the redirect response is issued. Effectively when cookie is present and is set to "no", aiCache assumes the request is coming from a mobile client and should be redirected to a mobile site (assuming one is configure). Alternatively, when cookie is present and is set to "yes", aiCache assumes the request is coming not from a mobile client and as such, should not be redirected to mobile site, but ***kept*** at this aiCache instance.

To disable such cookie issuance and processing, set **ua_keep_cookie** to **disable.**

Please note that when you enable UA-driven redirection, it happens ***first, it takes precedence over "plain" URL rewriting.*** In other words, for *plain* URL rewrite to happen, the request has to "stay" within aiCache, not get redirected to a different site as a result of UA-driven redirection.

As a safety trigger of sorts, you can instruct aiCache to *keep* requests whose User-Agent headers are shorter than certain length. To configure this, set **ua_keep_length** value at website level.

Pattern-level redirection, regular logic.

This method is different from previously described method in that it is pattern-specific (cannot be used at website level) and uses "*redirect* if UA matches" logic. To accomplish such redirection, create a pattern that matches the URL you want to redirect and add a **ua_redirect** directive below it. For example:

```
pattern ^/$ regexp 1m
# home page: "/" .  Cache it for 1 minute but ...
ua_redirect Windows\sCE http://mobilece.acmenews.com
# redirect for Windows CE devices
ua_redirect iPhone http://mobileiphone.acmenews.com/homepage.asp
# and iPhone
ua_redirect Android  http://mobileandroid.acmenews.com/index.jsp
# and Android
```

Please note that you can specify a full path to a URL as second parameter for **ua_redirect** setting, see example above.

You'd use the **ua_redirect** setting only when you want to redirect to different websites, based on device type - as shown above. If all you need to do is to send all mobile user agents to a certain site, irrespective of device type, please use **ua_pattern** setting, as described in previous sub-section.

You can use both methods, if appropriate. Please also be aware that there're literally hundreds upon hundreds of known Mobile device types, with new devices appearing every week, so we recommend to accomplish UA-driven redirects with help of **ua_pattern** mechanism, as opposed to **ua_redirect**.

Please note that when you enable UA-driven redirection, it happens ***first, it takes precedence over "plain" URL rewriting.*** In other words, for *plain* URL rewrite to happen, the request has to "stay" within aiCache, not get redirected to a different site as a result of UA-driven redirection.

Pattern-level redirection exclusion based on client IP address.

In some cases you might require not executing the UA-based redirection logic when requests come from certain IP ranges. For example, a website might be programmatically accessed from code outside of site's control, that is sending a User-Agent string value that is unknown and/or doesn't match any of the **ua_pattern** settings you have configured.

In such cases, with UA-driven redirection configured and request's User-Agent header not matching any of the **ua_pattern** settings, the request will be redirected to mobile URLs you have specified, even though requests are not coming from mobile agents, but are programmatically generated instead.

You can stop such unintended redirection, if you know requesting parties' source IP addresses and/or IP address ranges. Configure these via **keep_clip (Keep Client IP)** settings at website level. For example:

```
website
hostname www.acme.com
...
...
keep_clip 1.2.3.4
keep_clip 204.22.56.0/24
keep_clip 212.90.34.0/24
keep_clip 190.133.144.234/192
```

As you can see, one specify either a complete IP address in decimal notation **x.x.x.x** (all four octets provided, no "**/netmask**") or a whole network IP address range via standard **x.x.x.x/netmask** notation. Note that all four octets have to be always specified, use zeroes if needed.

Please note that for this method to work, aiCache must have access to the true client IP. If aiCache's are receiving user requests through a NAT'ing intermediary, you'd need to configure it to forward true client IP addresses via an HTTP header and configure aiCache to parse it out via **hdr_clip CLIP_HDR_NAME global** setting, for example:

```
hdr_clip X-ClientIP
```

Cookie-driven redirection.

This feature is only available in mobile-enabled version of aiCache.

You can configure aiCache to redirect requests based on request's cookie information. This is most commonly used to redirect users to localized versions of site, based on criteria of your choosing. For example, based on user preferences, you might decide to send the visitor to a different version of the site – such as international site in a different language, etc.

For example, when a request to **www.acmenews.com** a cookie called ***edition*** with a value of ***pacrim*** you might want to redirect such request to **pacrim.acmenews.com .**

Please note that while you can have some kind of application logic on origin servers determine if a redirection must be issued based on a cookie value, doing so will require forwarding requests to origin servers for that determination to be made, effectively negating some of the caching benefits. Not only that, maintaining and executing the redirection logic on origin server has a price tag and maintenance overhead.

Having aiCache decide when to redirect solves all of these issues and is the preferred way to go about redirecting users based on a cookie value.

To configure cookie-driven redirection, one of the following three pattern-level settings could be used:

```
pattern …..
cookie_pattern COOKIE_NAME COOKIE_PATTERN keep
cookie_pattern COOKIE_NAME COOKIE_PATTERN keep fallthrough
cookie_pattern COOKIE_NAME COOKIE_PATTERN redirect  FROM_URL_PATTERN TO_URL_PATTERN
```

The first setting:

```
cookie_pattern COOKIE_NAME COOKIE_PATTERN keep
```

configures aiCache to keep matching requests for further processing. Pattern matching is stopped when a match is obtained and no further pattern matching will be attempted.

The second setting:

```
cookie_pattern COOKIE_NAME COOKIE_PATTERN keep fallthrough
```

configures aiCache to keep matching requests for further processing. Pattern matching is continued with the next pattern in the configuration file when a match is obtained , so you can configure more processing to happen on the request, based on actions configured in subsequent patterns. The **fallthrough** flag could be used to accommodate for logic shortcuts that could be difficult/cumbersome to configure otherwise.

The third setting:

```
cookie_pattern COOKIE_NAME COOKIE_PATTERN redirect FROM_URL_PATTERN TO_URL_PATTERN
```

configures aiCache to redirect the request by rewriting request URL using the **FROM_URL_PATTERN TO_URL_PATTERN** . It only happens when aiCache sees that request has a cookie with **COOKIE_NAME** present and the cookie value matches the **COOKIE_PATTERN** setting. All of the pattern settings take full regexp patterns as values, so the rewritten URL can include (some of) the original URL, through use of regexp grouping and back referencing techniques.

Some examples:

```
pattern …..
cookie_pattern edition usdesktop keep fallthrough # let regular logic apply
cookie_pattern edition eudesktop redirect (.+) eu.acmenews.com\1
cookie_pattern edition asiamobile redirect (.+) mobile.asia.acmenews.com\1
```

Please note that cookie-based redirection happens before user-agent driven redirection, if both are configured for a matching pattern. You can see it might get pretty interesting when you need to match for both, so use good judgment to avoid ending up with too complex of configuration. Sometimes you might decide to send certain URLs to origin servers, via 0 TTL patterns, that might apply a more evolved logic to redirection decision making. These URLs would then set cookies that aiCache later can use to issue its own redirection.

The global, website and pattern-level cookie redirection stats are reported by aiCache as usual via a number of interfaces. You can enable debugging of rewrites via **log_rewrite** global or website level setting.

Cookie-driven OS tagging.

This feature is only available in mobile-enabled version of aiCache.

You can configure aiCache to select an origin server with a specific tag, based on requests' cookie value.

For example, when a request to **www.acmenews.com** a cookie called ***edition*** with a value of ***pacrim*** you might want to fill such requests from origin servers with tag of 88. Clearly, you must have specified one or more of origin servers with matching tags (88 in this example), see elsewhere in this guide about OS tags.

```
cookie_pattern COOKIE_NAME COOKIE_PATTERN ostag NNN
```

An example cookie-driven OS tagging (selection) setting could be configured like so:

```
pattern ...
cookie_pattern edition pacrim ostag 88
```

When no such cookie is detected in request or its value cannot be matched, no OS tag is assigned to the request.Should pattern itself specify an OS tag, cookie-driven tag takes precedence.

Managing Keep-Alive Connections.

Client-Side Keep-Alive connections.

Normally aiCache does not force connection close after serving a response to a client, allowing clients to send more requests over already-established connection. This is knows as connection persistence or connection Keep-Alive feature. It has a potential to speed up user access to your web site by amortizing TCP/IP connection establishing overhead over many client requests.

Instead of closing client connection after serving a response, aiCache indicates to clients, via Keep-Alive HTTP header, its Keep-Alive preferences as to for how long the connection could be kept open by the client. However not all clients (read: browsers) may obey this hint. In order to prevent abnormally high number of open, yet mostly idle, client connections having to be maintained by aiCache and server's operating system, you can configure aiCache to drop idle Keep-Alive connections if there was no client input on particular connection for more than **maxkeepalivetime *seconds***. Default value: 10 secs.

You can limit maximum number of requests allowed to be served over a single Keep-Alive client connection - use **maxkeepalivereq** directive in global and/or website sections of the configuration file. Default value: 20 requests.

aiCache reports number of keep-alive requests served over each keep-alive client connection in its access log file. If you're kind of person that likes to fine tune stuff, try setting different increasing **maxkeepalivereq** values till you the reported number of served keep-alive requests stops growing. Doing so will speed up the loading of the page for your visitors.

Sometimes you know that a certain request URL results on in a response that is unlikely to be followed by additional Keep-Alive requests from the clients. In this case letting connection to go into Keep-Alive state is a straight waste of resources - as it is never used again. For such URLs you can set **conn_close** setting in the matching pattern section.

You can see gauge the effectiveness of client Keep-Alive connections by observing average number of client requests per client connection - it is reported both in Global and Website sections of Web self-refreshing monitoring pages. The higher the number, the better the experience for your visitors, however, larger number might mean more open connections on your aiCache serve, so you might need to adjust aiCache's client Keep-Alive settings to find a middle ground.

Server-Side Keep-Alive connections.

aiCache is capable of maintaining Keep-Alive connections to origin servers. Similar to client-side Keep-Alive connections, having OS-side Keep Alive connection allows you to speed up request processing time by amortizing TCP/IP connection establishing overhead over many requests. It is less beneficial however, compared to client-side Keep-Alives, as aiCache and origin servers are frequently located not only in the same hosting facility/datacenter, but are attached to the same switch - with latency in microseconds, not tens or hundred(s) of milliseconds as often is the case with client-side connections. However, if the origin servers are

hosted in a geographically remote Datacenter, a significant latency away, consider using OS Keep-Alive, it will make responses much faster.

Please note that you *should test* Origin Server Keep-Alive feature before turning it on for production use. You must ensure that origin web servers support Keep-Alive connection (most do) and also configure it to persist for the reasonably long time and number of request, to realize maximum benefits.

To specify maximum per-origin-server number of open Keep-Alive connections to maintain, set it via:

```
max_os_ka_connections NNN
```

To specify maximum number of requests to be allowed per Keep-Alive origin server connection, set it via:

```
max_os_ka_req MMM
```

Please note that you might see a very long response time in case of conditional keep-alive requests to some origin servers. To fix this issue, simply set **max_os_ka_req** to 0, at global or website level:

```
max_os_ka_req 0
```

Most origin web servers are configured to only use a Keep-Alive connection for a few seconds, before discarding it and requiring opening of a new connection . The reasons for this include logic "resets" - to catch and stop any memory leaks in the application code and reducing number of processes and open connections that operating system needs to maintain on origin server. You might want to fine-tune aiCache by setting **max_os_ka_time** parameter in global section match origin web server setting .

Please note that these OS Keep-Alive settings can be specified at global or website level, latter overriding global values.

You can see judge the effectiveness of Origin Server Keep-Alive connections by observing average number of OS requests per OS connection - it is reported both in Global and Website sections of Web self-refreshing monitoring pages. The higher the number, the faster it is to obtain a response from Origin Server, as it reduces (amortizes) overhead of TCP connection handshake. Again, the savings are most noticeable when aiCache and Origin Servers are located in different Datacenters.

While not directly related to OS Keep-Alive (or lack of thereof), we also recommend enabling SO_LINGER option for origin server connection, by specifying this option and value in the website section:

```
 os_linger 0
```

Look elsewhere in this guide for explanation of **os_linger** option.

Please also note that in order to enable OS HTTPS keep-alive connections, you must set both **max_os_ka_connections (**global level) and **enable_https_os_ka** (website-level) settings. Test thoroughly before enabling **enable_https_os_ka** in production setup.

```
server
....
max_os_ka_connections 10
```

```
website
hostname www.acme.com
....
enable_https_os_ka
```

Due to significant overhead of establishing a new HTTPS connection to origin server (the SSL handshake, selection of session key etc), we recommend to enable enable_https_os_ka setting in high traffic setups. Configure origin server to allow for extended length and extended number of requests for such HTTPS connections. Such reuse of HTTPS connections is likely to have positive effect throughout your environment.

aiCache doesn't use origin server keep-alive connections for POST requests, even when OS KA is enabled as described above. The reason for this as follows: any re-use of a Keep-Alive connection, client or server-side, for second or subsequent requests, is not guaranteed to succeed and it can fail in most unpredictable fashion. This doesn't create a problem for GET requests, as aiCache retries GET requests. However, automatically retrying POST requests is very dangerous - as it might lead to doubling and tripling and so on, of blog posts, on-line orders and credit card charges and so on. So in order to take this uncertainty out of processing of POST requests, aiCache always uses a nice, fresh origin server connection for POST requests.

Command Line Interface

aiCache provides advanced Command Line Interface (CLI). It is discussed in greater detail in Administration section of this guide.

To enable CLI you need to provide CLI password, hostname (or IP address) and port number for the CLI Server. This information is configured in the main configuration file:

```
admin_ip *
admin_port 2233
admin_password secret
```

We recommend changing the default password of "*secret*". To secure CLI from remote access you can either set proper firewall rules (chances are in your existing setup you are adequately protected from external access to the CLI interface without having to do anything at all) or only accept incoming connections on *localhost* interface of 127.0.0.1 . In latter case you will need to connect to the CLI from the aiCache host itself, connecting from any other system will not be possible.

You can connect to the CLI interface using any common telnet program, by specifying IP address and port number of the CLI :

```
telnet 127.0.0.1 2233
```

We have much more to say about CLI interface in a dedicated chapter later in this Guide.

Self-refreshing Web Monitor: statistics and pending requests.

aiCache provides simple self-refreshing statistics Web pages - for both global and per-website statistics. These are enabled by default and can be accessed at the following URLs:

http://a.b.c.d/accelstattext?global

http://a.b.c.d/accelstattable **(we recommend to start at this URL!)**

http://a.b.c.d/accelstattext?all

http://a.b.c.d/accelstattext?hostname

where **a.b.c.d** is *numeric* IP of particular aiCache instance. The reason for these URL to be accessible via numeric instance IP is so that the URLs cannot be found under your website's URL structure. Also, directly accessing an aiCache instance via IP address might be required to assure you're requesting the statistics from a particular node. For example, www.acme.com could be DNS-mapped to 3 different IP addresses, each aiCache

instance of its own and unless you request particular aiCache instance by its IP address you wouldn't know which instance you're accessing.

However, should you so require, you can "fold" the stat URL "under" URL structure of your site. To do so, create matching patterns for both **stat_url** and **table_stat_url** settings (see below) and mark them with **stat_req** flag (requires no value). For example:

```
…
website
hostname acme.com

…
pattern accelstattext simple 0
stat_req

pattern accelstattable simple 0
stat_req

…
```

Yet another way to access these stat pages is via dedicated hostname(s). Please see next section for more information.

Statistics pages auto-refresh every 2-4 seconds and show most important statistics collected by aiCache - global or per-website. These URLs offer a read-only view of the aiCache statistics, with no ability to change, affect or modify anything within aiCache instance.

http://a.b.c.d/accelstattext?global displays global aiCache statistics, **http://a.b.c.d/accelstattable** and **http://a.b.c.d/accelstattext?all** provide summary pages, table formatted and plain text, displaying abbreviated statistics of all of the accelerated web sites while **http://a.b.c.d/accelstattext?hostname** allows you to "zoom into" particular website's statistics in greater details.

If, due to security concerns, you must limit access to these pages, you can change the **accelstattext** and **accelstattable**, via **stat_url** and **table_stat_url** settings in the global configuration section. If you so desire, you can set **stat_url** and **table_stat_url** to random strings and then no one will be able to access the Web monitoring page. For some setups - especially ones with firewalls and load balancers front-ending and load balancing traffic to aiCache servers, you might be protected from direct outside access to aiCache's IP - check with your Network team.

Consider having these URLs (**http://a.b.c.d/accelstattable** is especially good place to start at) open in a browser whenever you're monitoring a stress test or are watching a traffic spike unfold, it will tell you much you need to know about the status of the site. However, for real-time monitoring and alerting, we recommend you consider the built-in Alerting feature of aiCache - please see a dedicated chapter later in this Guide.

Reported via **accelstattable** self-refreshing web statistic page are:

- R/Sec: requests per second
- RTime: average response time
- OrigS: configured, heathy and admin_disabled origin servers, separated by ":" (colon)

- CConn: number of open client connections
- OConn: number of open origin server connections
- OCR %: overall caching ratio (cacheable request, served fresh to all requests)
- SCR %: specific caching ratio (cacheable request, served fresh to cacheable requests)
- CEntr: number of cached responses
- OutBW: output bandwidth in KB/sec
- BRR: bad request per second and responses per second, separated by ":" (colon)

As a helpful reminder about that different fields represent, hints are shown during mouse-hover over the table column headers (the first, gray line of headers).

Please note that Firefox retains (remembers) your "scroll" position, while IE resets it on each request.

Attached below are screenshots of example aiCache statistics reporting pages. Please note that as product evolves, there might be additional information added to these screens. You should also consider setting up monitoring of aiCache instances via SNMP, as explained in the SNMP chapter , along with enabling of aiCache's automated alerting feature.

aiCache also provides simple self-refreshing Web pages displaying pending requests - both global and per-website. It is enabled by default and can be access by the following URLs:

http://a.b.c.d/accelpendtext?global

http://a.b.c.d/accelpendtext?hostname

where **a.b.c.d** is *numeric* IP of particular aiCache instance. Pending requests pages auto-refresh every 4 seconds , show requests currently awaiting responses from origin servers and can assist with troubleshooting of slower responses.

If, due to privacy concerns, you want to limit access to these pages, you can change the **accelpendtext**, via **pend_url** setting in the global configuration section.

Please note that as product evolves, information displayed on self-refreshing monitor pages might change without further notice, as we strive to add ever evolving set of helpful instrumentation data to these pages.

The hostname-specific (website-specific) pages provide information specific to that site. Most important difference will be showing of origin servers and their associated statistics.

You can configure aiCache to collect and report pattern-level statistics by specifying **collect_pattern_stat** at website level. The collected pattern stats can be accessed via Web and SNMP interfaces. Here's an example of the data collected.

```
    Pattern: >\.gif<, TTL: 10 sec, OST: 0
        #Req: 118, #CHits: 90, #CMiss: 28, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.jpg<, TTL: 10 sec, OST: 0
        #Req: 79, #CHits: 48, #CMiss: 31, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.png<, TTL: 10 sec, OST: 0
        #Req: 8, #CHits: 5, #CMiss: 3, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.ad<, TTL: 10 sec, OST: 0
        Label: Ad calls
        #Req: 0, #CHits: 0, #CMiss: 0, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.js<, TTL: 10 sec, OST: 0
```

This information provides very granular, real-time view into traffic to a given website. Consider examining this information closely when you need to troubleshoot an issue. Assuming your patterns are setup to finely dissect and group inbound requests, you should be able to quickly establish which URLs are getting hit, are slow to response or are receiving errors from origin servers.

You might find it convenient to label some of the patterns via **label** setting. Such labels will be displayed in both self-refreshing web stat screen and made available via SNMP.

Accessing statistics web pages via dedicated hostname.

You'd normally request these self-refreshing stats web pages by accessing a running aiCache instance via its IP address, for example : **1.1.1.1/accelstattable** or alternatively, by setting up matching patterns with **stat_req** flag set (see previous section for more on this).

aiCache will fail to match the hostname (1.1.1.1 in this case) to any known website and will then attempt to match the request URL to special request URLs, including web stats URLs. However, if you desire to access these stats URLs via an alpha-numeric hostname, you can do so by specifying **stats_host_prefix** in *global* section of aicache configuration file. For example, if you specify:

```
    stats_host_prefix aistats
```

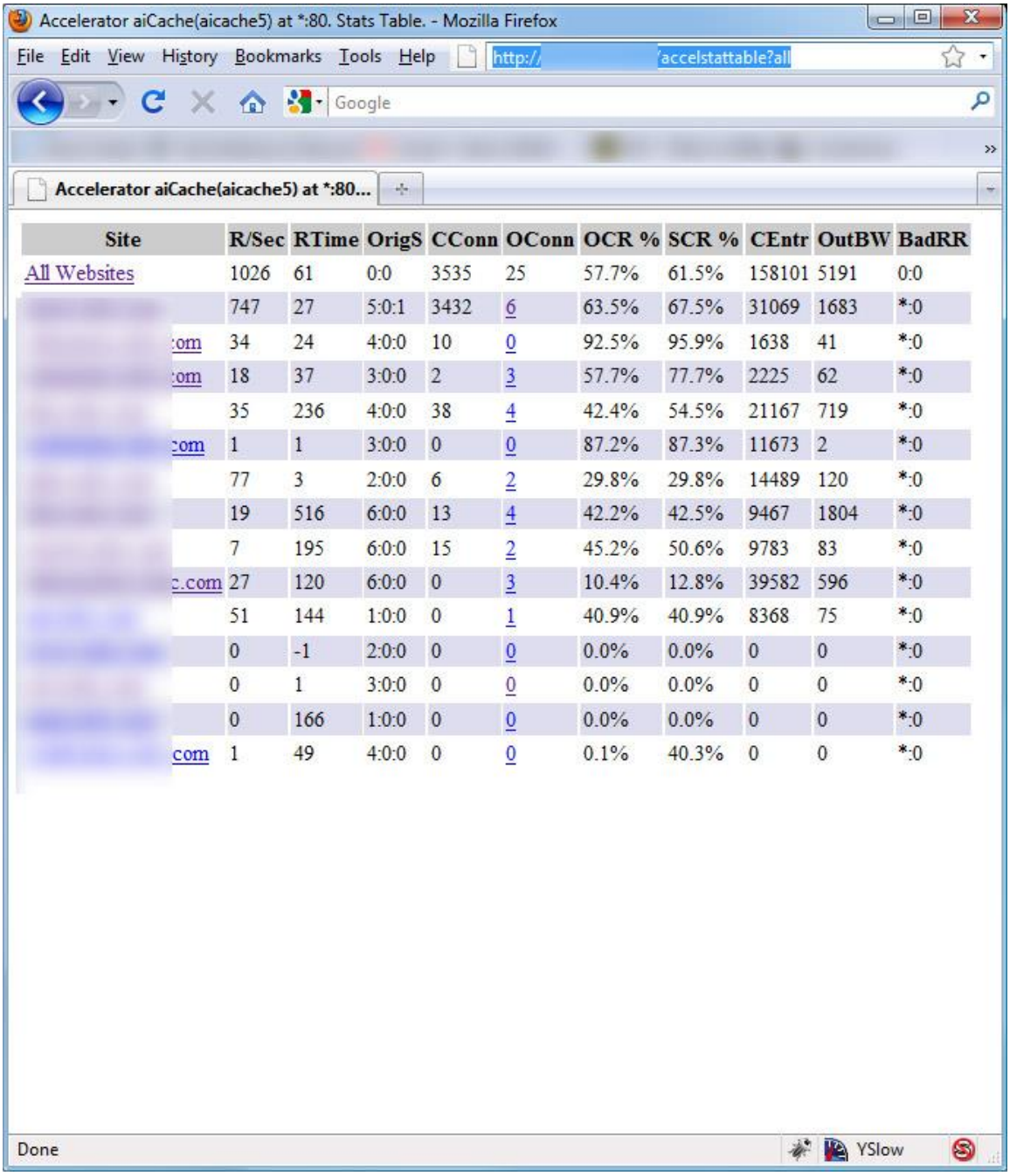
you'd then be able to access aiCache stats pages by mapping any name that *starts* with **aistats** to this particular aiCache instance IP, for example **aistats.aicache01.acme.com , aistats.aicache02.acme.com, aistats1.acme.com** and so on. Again, you'd need to setup proper name to IP mapping, via DNS or through modification of a local hosts file. Please note that this same technique could be used for hand-crafted special requests, such as expiration etc, if you desire accessing these via a hostname as opposed to IP.

You can also specify a special **admin** listen port type. For example:

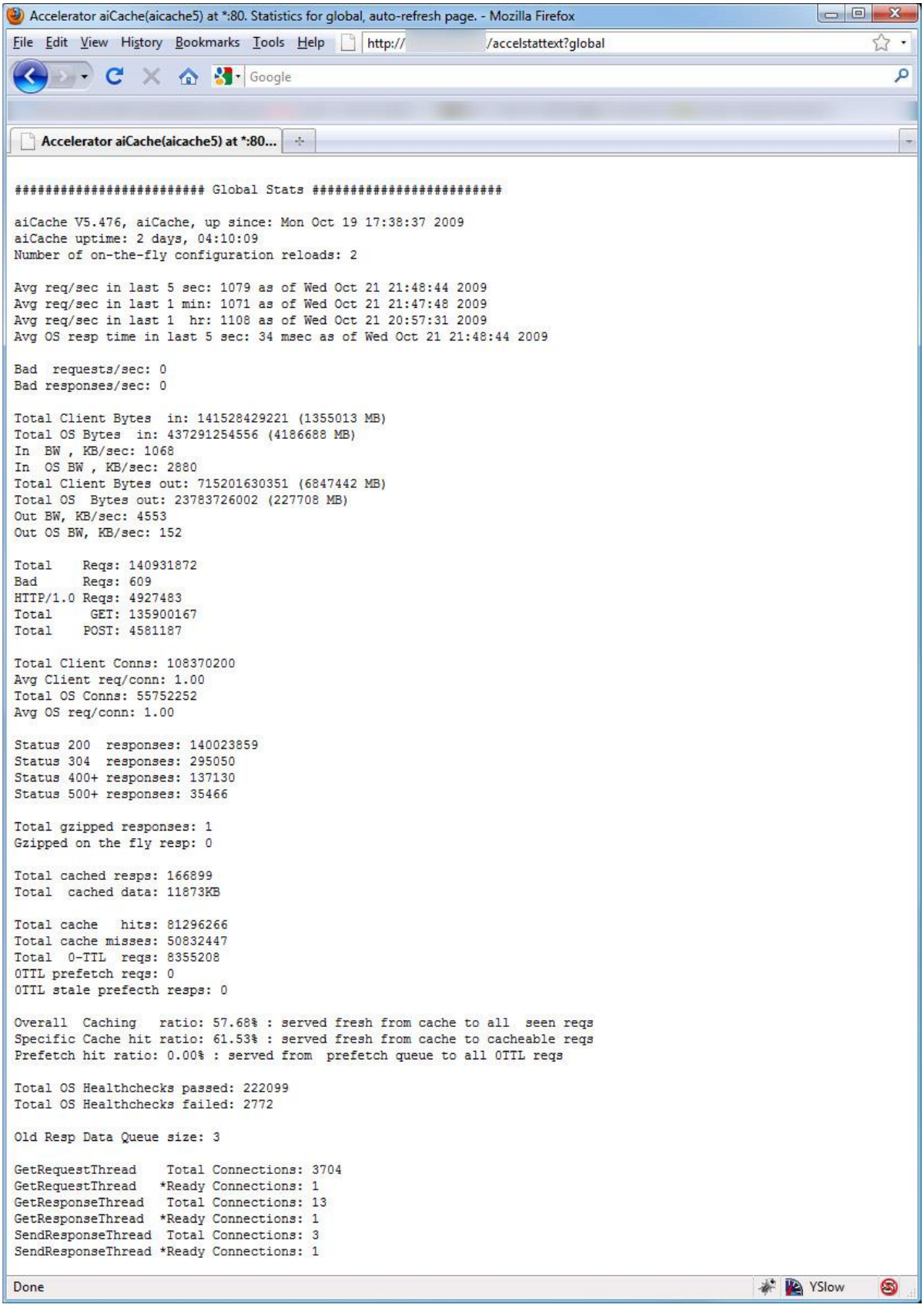
```
    listen admin * 5001
    listen admin 192.1.1.1 1234
```

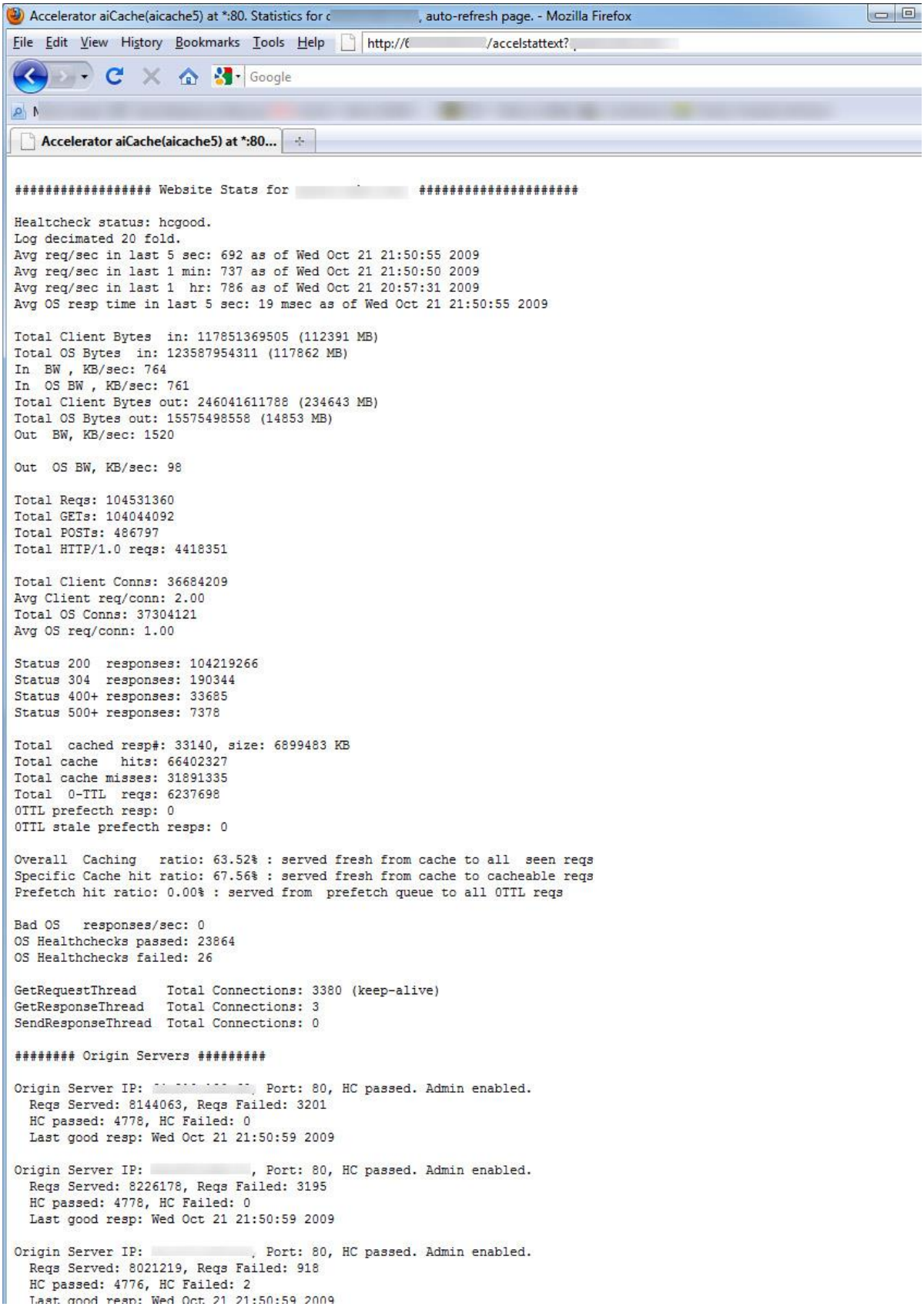
The admin type ports only respond to special type requests– including *webstats*, *peer* requests and such. Admin ports do not match to any defined websites and can do not serve "regular" webpages/traffic. They exist only to serve special traffic and might come in handy in special occasions when you want to create such *special* listen ports that are *separate* from *regular* ports.

And now we present some example Web monitoring/statistics web pages as provided by aiCache. Please note that as new capabilities are added to the product, the appearance of the web pages might change.



Site	R/Sec	RTime	OrigS	CConn	OConn	OCR %	SCR %	CEntr	OutBW	BadRR
All Websites	1026	61	0:0	3535	25	57.7%	61.5%	158101	5191	0:0
	747	27	5:0:1	3432	6	63.5%	67.5%	31069	1683	*:0
:om	34	24	4:0:0	10	0	92.5%	95.9%	1638	41	*:0
:om	18	37	3:0:0	2	3	57.7%	77.7%	2225	62	*:0
	35	236	4:0:0	38	4	42.4%	54.5%	21167	719	*:0
:om	1	1	3:0:0	0	0	87.2%	87.3%	11673	2	*:0
	77	3	2:0:0	6	2	29.8%	29.8%	14489	120	*:0
	19	516	6:0:0	13	4	42.2%	42.5%	9467	1804	*:0
	7	195	6:0:0	15	2	45.2%	50.6%	9783	83	*:0
c.com	27	120	6:0:0	0	3	10.4%	12.8%	39582	596	*:0
	51	144	1:0:0	0	1	40.9%	40.9%	8368	75	*:0
	0	-1	2:0:0	0	0	0.0%	0.0%	0	0	*:0
	0	1	3:0:0	0	0	0.0%	0.0%	0	0	*:0
	0	166	1:0:0	0	0	0.0%	0.0%	0	0	*:0
com	1	49	4:0:0	0	0	0.1%	40.3%	0	0	*:0





Simple pattern-driven content expiration page.

By popular demand, aiCache provides a simple, built-in web page that could be used to enter a pattern into a web form and force cluster-wide (peer-aware) expiration of matching content. Contrast this with having to login into CLI to perform the same function via aiCache's command line interface.

To help you safeguard this page so that unauthorized personnel could not force content expiration, you must set the URL to one of your liking and share it only with proper personnel. No default value is provided and as a result, the page is not enabled by default. The setting is called **exp_page_prefix** and is set at global level. For example:

```
exp_page_prefix secretpatternpage
```

To access this page, point a browser to page's URL. As with most peer-commands, you need to access this page via one of aiCache's IP addresses, for example:

http://1.2.3.4/secretpatternpage

The web page contains a number of web forms , a form per defined website. Each form contains a single text field – this is where you enter a pattern that you wish to expire. After clicking on "Expire" button, the request is sent to aiCache server. It will then process it first internally and fan it out across all of its peers, as defined in its configuration file.

A brief response page indicating aiCache receiving of the request is shown upon submission of the request. No indication is provided as to the number of entries that were expired as a result – neither on the local aiCache instance, nor on any of defined peers. However, this information is reported, as usual, in the aiCache error log file. This approach is chosen on purpose, as in clustered setup, one cannot be sure just what aiCache nodes might end up containing a given response.

Feel free to copy the source of this pattern expiration Web page into your site's maintenance Wiki or like, spicing it up to your liking in process. By hosting the page in such manner, you can enforce additional authentication if so desired.

5-second statistics snap files.

When so configured, aiCache collects and saves both global and per-website statistics. This information is saved, every 5 seconds, to dedicated statistics log files. To enable this feature, specify "**logstats**" directive in global and/or website sections.

The files are called **stats-global** and **stats-*website_name*** and are located in the log directory. Each line in this files contains a string similar to this:

[Wed Feb 25 21:20:16 2006] 0 R/S 0 OKB/S 0 IKB/S 0 CC# 0 OC# 53.95 OCR 21.27 SCR -1 ART 0 BRq/S 0 BRp/S 283 CR# 3663 KB CSz 3452345 BIB 9348593584 BOB 9348593584 TBIB 23239348593584 TBOB 3459348593584

The fields are:

1. [timestamp]: self explanatory
2. RPS: avg number of RPS in prev 5 seconds
3. Output BW, KB/sec - client traffic only
4. Input BW, KB/sec - client traffic only
5. total_client_conns: total number of open client connections, including idle Keep-Alive connections
6. total_os_conns: total number of open and active origin server connection, excluding idle OS Keep-Alive
7. OCR: overall caching ratio, %. Ratio of responses served fresh out of cache to all responses
8. SCR: specific caching ratio, %. Ratio of responses served fresh out of cache to all cacheable responses
9. os_avg_resp_time: average origin server response time in that interval, milliseconds
10. bad_RPS: number of bad request per sec (only in global stats file)
11. os fails/sec: number of failed responses from origin server, per second
12. total_cached_resp: total number of cached responses
13. total_cached_responses_size: total size of data in cached responses, KB.
14. (BIB) total bytes in (from clients and origin servers), since previous reading (interval value)
15. (BOB) total bytes out (toward clients and origin servers), since previous reading (interval value)
16. (TBIB) total bytes in (from clients and origin servers), accumulated since aiCache startup
17. (TBOB) total bytes out (toward clients and origin servers), accumulated since aiCache startup

This information is invaluable if you need to quickly see what traffic volume, response time and other critical aiCache counters were at a particular point in time, without having to crunch through gigabytes worth of log files (and some of this information is not even available in the log files, no matter how much crunching you do).

SNMP Monitoring.

aiCache provides SNMP read-only access to very rich set of global, website and pattern-level statistics. To accomplish this it relies on *pass integration* to the most popular open-source SNMP package : Net-SNMP. This guide assumes some very basic SNMP knowledge. You can pick up most of what you need to know via a quick Google search for "SNMP tutorial" or start here:

http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

After configuring SNMP as we describe below (and it is very easy to do), you can use a number of commercial and open-source SNMP-capable packages to monitor, chart and alert on all kinds of counters that aiCache collects.

Chances are your Linux server already has Net-SNMP installed as part of basic OS install. If not, you need to install the package. Please follow the instructions for your particular distribution. On Debian and Ubuntu Linux distributions, for example, you can use:

```
apt-get install snmpd
```

After installing the SNMP package, you need to configure snmpd server to *pass* access to aiCache's set of SNMP OIDs by adding following 2 lines: a *view and a pass* directives, that you need to add to **/etc/snmp/snmpd.conf** file.

```
view systemonly included .1.3.6.1.4.1.9999
pass .1.3.6.1.4.1.9999 /path_to_aicache_install/snmp.pl
```

where **path_to_aiCache_install** is the location of your aiCache installation. For example, if you installed aiCache in the default directory called: **/usr/local/aicache**, then your *pass* line will be:

```
pass .1.3.6.1.4.1.9999 /usr/local/aicache/snmp.pl
```

We are telling snmpd server to pass access to all SNMP variables with OIDs that start with **.1.3.6.1.4.1.9999** to program called /**usr/local/aicache/snmp.pl**

These two lines can be added in the "Pass through control." section of the **snmpd.conf** file - just page through the file and you will quickly find that section, it looks like this:

```
###############################################################################
# Pass through control.
#

# Usage:
#   pass MIBOID EXEC-COMMAND
#
# This will pass total control of the mib underneath the MIBOID
# portion of the mib to the EXEC-COMMAND.
#
# Note:  You'll have to change the path of the passtest script to your
# source directory or install it in the given location.
#
# Example:  (see the script for details)
#           (commented out here since it requires that you place the
#           script in the right location. (its not installed by default))
view systemonly included .1.3.6.1.4.1.9999
pass .1.3.6.1.4.1.9999 /usr/local/aicache/snmp.pl
```

Next you need to set a community string in the snmpd.conf file - in the example below it is set to *public*.

```
####
# First, map the community name (COMMUNITY) into a security name
# (local and mynetwork, depending on where the request is coming
# from):

#       sec.name  source          community
#com2sec paranoid  default         public
com2sec readonly  default         public
```

You may also need to modify the **snmp.pl** file (it is a part of aiCache distribution), pointing it to the exact location of **snmp.stat** file. For example:

```
# Point me to snmp.stat file, full path please.
#
$statfile = "/var/log/aicache/snmp.stat";
```

After setting both the **pass** directive and **community** string as outlined above, you need to restart snmpd daemon:

```
/etc/init.d/snmpd restart
```

At this point you can fire up aiCache, let it run for a few seconds (so that **snmp.stat** file is generated) and test the entire SNMP integration by running **snmpget** command (it is part of snmp package) to retrieve and display one of aiCache SNMP OIDs.

As an example, we request value of aiCache's "total number of requests" OID: **.1.3.6.1.4.1.9999.1.1.1,** to be read (**snmpget**) from SNMP agent (snmpd acts as an *agent*) on **locahost** using V2c of SNMP protocol. We also specify community string of **public.** Enter the following command at the server's command prompt:

```
snmpget -v 2c -c public localhost .1.3.6.1.4.1.9999.1.1.1
```

If everything is configured properly, you shall see a response that looks like this:

```
SNMPv2-SMI::enterprises.9999.1.1.1 = INTEGER: 23452343
```

As you can see, a value of **23452343** was returned in response to an SNMP query. Should **snmpget** return an error message, double check that you have configured the **pass** directive, have set a valid **community** string and **snmpd** is running on the server.

Here's how the SNMP integration works : aiCache periodically (every few seconds, but you can configure how frequently) snaps all of its counters and saves them into a file called **snmp.stat** This file resides in **log_directory** (same directory where your access and error log files are). The **snmp.pl** program processes that file to extract the value of a particular variable when requested so by **smnpd** - which, in turn, does it in response to a SNMP request from SNMP software that you use .

Please note that due to certain deficiencies in Net-SNMP, integer counters are limited to 32bits only. All but busiest sites will not have any issues with that, but in case of heavy web traffic, some SNMP counter might overflow. For example, the number of requests collected by aiCache is a 64bit unsigned long integer - a number with no practical limit on how big it can get. Yet it is accessed via SNMP as 32bit variable and is thusly limited to about 4,000,000,000. Serving 1000 RPS, aiCache instance will overflow the 32bit in 4,000,000 seconds or ~46 days. To remediate, you might want to restart aiCache once a week - it is entirely up to you.

The **snmp.pl** has a built-in logic that monitors the age of the **snmp.stat** file . When it detects that **snmp.stat** file has gone stale, meaning aiCache is not running for whatever reason, it returns error responses to SNMP *get* requests sent by your SNMP monitoring SW. So you can configure your SNMP monitors to alert on such errors.

After enabling aiCache SNMP integration, you can configure your SNMP monitoring/charting software to read, display, chart and alert on the variables that are of interest to you. Examples of such "more interesting" variables include number of requests, RPS values, origin server response times, number of open client and origin server connections etc.

If you're interested in collecting and reporting of pattern-level statistics and making them available over SNMP, you need to configure **collect_pattern_stat** setting for the websites of interest. The setting is a flag and requires no value.

Please note that more SNMP variables/counters can be added at a later point, to check for that simply look at the **snmp.stat** file in your **log directory**. New OID are added in a way that does not affect OIDs of existing variables, so whatever you configured in your SNMP monitoring software won't have to be changed.

Global OID prefix.

By default, aiCache assigns OID prefix of .1.3.6.1.4.1.9999.1 to all of the defined OIDs. You can change it to your own value by configuring global-level setting of **oid.** For example:

```
server
oid .1.2.3.4.5.9999
```

Website-specific OIDs.

By default, aiCache auto-assigns OIDs to website-specific values. It does it so that the first defined website gets OID prefix of "**oid**.1.", second one gets "**oid**.2." and so on, where **oid** is the global OID prefix (defaults to .1.3.6.1.4.1.9999.1, can be set via global setting "oid"). It works in most cases, but should you re-arrange ordering of the websites inside of the configuration file by adding, removing or moving around defined websites, the website-specific OIDs do change as a result –potentially requiring reconfiguration of your SNMP polling software.

Alternatively, you can tell aiCache to hardware website-specific OID prefixes by configuring website-level setting of "**oid_idx**". When so configured, the corresponding website gets OID prefix of "**oid.oid_idx"** and will retain said prefix even if website configuration changes – websites added, removed or re-arranged. To avoid confusion, should you desire to use **oid_idx** setting, we recommend assigning unique **oid_idx** to each defined website. For example:

```
server
oid .1.2.3.4.5.9999

…
…
website
hostname www.acme.com
oid_idx 555
…
hostname images.acme.com
oid_idx 556
```

To see complete and up-to-date set of available SNMP OIDs please look inside of **snmp.stat** file generated by your instance of aiCache. Here is an *example* set of SNMP variables, complete with SNMP OIDs and their meaning. Simply select those you have interest in and configure your SNMP software to retrieve them.

Example list of aiCache SNMP OIDs.

```
############################## Global Stats ######################
.1.3.6.1.4.1.9999.1.1.0 string global #stat type
.1.3.6.1.4.1.9999.1.1.1 string aiCache5 #accelerator name
.1.3.6.1.4.1.9999.1.1.2 timeticks 381500 #uptime in ticks
.1.3.6.1.4.1.9999.1.1.3 integer 11 #n_reqs
.1.3.6.1.4.1.9999.1.1.4 integer 194 #n_bad_reqs
.1.3.6.1.4.1.9999.1.1.5 integer 11 #n_gets
.1.3.6.1.4.1.9999.1.1.6 integer 0 #n_posts
.1.3.6.1.4.1.9999.1.1.7 integer 0 #n_200_resp
.1.3.6.1.4.1.9999.1.1.8 integer 0 #n_304_resp
.1.3.6.1.4.1.9999.1.1.9 integer 194 #n_400_resp
.1.3.6.1.4.1.9999.1.1.10 integer 0 #n_500_resp
.1.3.6.1.4.1.9999.1.1.11 integer 0 #N of cached responses
.1.3.6.1.4.1.9999.1.1.12 integer 0 #size of cached responses, KB
.1.3.6.1.4.1.9999.1.1.13 integer 0 #n_cache_hits
.1.3.6.1.4.1.9999.1.1.14 integer 0 #n_cache_misses
.1.3.6.1.4.1.9999.1.1.15 integer 0 #n_0ttl_hits
.1.3.6.1.4.1.9999.1.1.16 integer 0 #CCR
.1.3.6.1.4.1.9999.1.1.17 integer 0 #OCR
.1.3.6.1.4.1.9999.1.1.18 integer 2376 #bytes_recv
.1.3.6.1.4.1.9999.1.1.19 integer 0 #In BW KB/sec
.1.3.6.1.4.1.9999.1.1.20 integer 36354 #bytes_sent
.1.3.6.1.4.1.9999.1.1.21 integer 0 #Out BW KB/sec
.1.3.6.1.4.1.9999.1.1.22 integer 0 #n_os_hc_pass
.1.3.6.1.4.1.9999.1.1.23 integer 0 #n_os_hc_fails
.1.3.6.1.4.1.9999.1.1.24 integer 0 #bad_RPS
.1.3.6.1.4.1.9999.1.1.25 integer 0 #bad_resp/sec
.1.3.6.1.4.1.9999.1.1.26 integer 0 #5sec RPS
.1.3.6.1.4.1.9999.1.1.27 integer 0 #1min RPS
.1.3.6.1.4.1.9999.1.1.29 integer -1 #5sec OS Resp Time
.1.3.6.1.4.1.9999.1.1.30 integer 0 #old RDT Len
.1.3.6.1.4.1.9999.1.1.31 integer 0 #total client conn
.1.3.6.1.4.1.9999.1.1.32 integer 0 #get_req_tconn
.1.3.6.1.4.1.9999.1.1.33 integer 0 #get_req_ready
.1.3.6.1.4.1.9999.1.1.34 integer 0 #get req ready
.1.3.6.1.4.1.9999.1.1.35 integer 0 #get_resp_ready
.1.3.6.1.4.1.9999.1.1.36 integer 0 #send_resp_tconn
.1.3.6.1.4.1.9999.1.1.37 integer 0 #send_resp_ready
.1.3.6.1.4.1.9999.1.1.38 integer 0 #n_gzipped_responses
.1.3.6.1.4.1.9999.1.1.39 integer 0 #n_os_resp_gzipped_by_us

######## Plugin-level stats #########


.1.3.6.1.4.1.9999.1.1000.0.0 string demo_pl_func # exec function name
.1.3.6.1.4.1.9999.1.1000.0.1 integer 1245 #number of plugin execs
.1.3.6.1.4.1.9999.1.1000.0.2 integer 0 #number of plugin drops
.1.3.6.1.4.1.9999.1.1000.0.3 integer 0 #number of plugin redirects
.1.3.6.1.4.1.9999.1.1000.0.4 integer 0 #number of plugin URL rewrites
.1.3.6.1.4.1.9999.1.1000.0.5 integer 0 #number of plugin body rewrites
.1.3.6.1.4.1.9999.1.1000.0.6 integer 0 #number of plugin provided resp
.1.3.6.1.4.1.9999.1.1000.0.7 integer 0 #number of plugin TTL sets
```

```
############################## Website Stats ######################
.1.3.6.1.4.1.9999.1.2.0 string aaa.bbb.com #name
.1.3.6.1.4.1.9999.1.2.1 integer 11 #n_reqs
.1.3.6.1.4.1.9999.1.2.2 integer 11 #n_gets
.1.3.6.1.4.1.9999.1.2.3 integer 0 #n_posts
.1.3.6.1.4.1.9999.1.2.4 integer 0 #n_200_resp
.1.3.6.1.4.1.9999.1.2.5 integer 0 #n_304_resp
.1.3.6.1.4.1.9999.1.2.6 integer 0 #n_400_resp
.1.3.6.1.4.1.9999.1.2.7 integer 0 #n_500_resp
.1.3.6.1.4.1.9999.1.2.8 integer 0 #N of cached responses
.1.3.6.1.4.1.9999.1.2.9 integer 0 #size of cached responses, KB
.1.3.6.1.4.1.9999.1.2.10 integer 0 #n_cache_hits
.1.3.6.1.4.1.9999.1.2.11 integer 0 #n_cache_misses
.1.3.6.1.4.1.9999.1.2.12 integer 0 #n_0ttl_hits
.1.3.6.1.4.1.9999.1.2.13 integer 0 #CCR
.1.3.6.1.4.1.9999.1.2.14 integer 0 #OCR
.1.3.6.1.4.1.9999.1.2.15 integer 2376 #bytes_recv
.1.3.6.1.4.1.9999.1.2.16 integer 0 #In BW KB/sec
.1.3.6.1.4.1.9999.1.2.17 integer 1628 #bytes_sent
.1.3.6.1.4.1.9999.1.2.18 integer 0 #Out BW KB/sec
.1.3.6.1.4.1.9999.1.2.19 integer 0 #n_os_hc_pass
.1.3.6.1.4.1.9999.1.2.20 integer 0 #n_os_hc_fails
.1.3.6.1.4.1.9999.1.2.21 integer 0 #bad_resp/sec
.1.3.6.1.4.1.9999.1.2.22 integer 0 #prev 5sec RPS
.1.3.6.1.4.1.9999.1.2.23 integer 0 #prev 1min RPS
.1.3.6.1.4.1.9999.1.2.25 integer -1 #prev 5sec OS Resp Time
.1.3.6.1.4.1.9999.1.2.26 integer 0 #website in forced fallback mode

############################## Origin Servers ######################

######## Origin Servers #########


.1.3.6.1.4.1.9999.1.2.100.0.0 string  127.0.0.1 #os IP
.1.3.6.1.4.1.9999.1.2.100.0.1 integer 8888 #os port
.1.3.6.1.4.1.9999.1.2.100.0.2 integer 0 #os disabled?
.1.3.6.1.4.1.9999.1.2.100.0.3 integer 0 #os reqs served
.1.3.6.1.4.1.9999.1.2.100.0.4 integer 0 #os reqs failed
.1.3.6.1.4.1.9999.1.2.100.0.5 integer 0 #os hc passed
.1.3.6.1.4.1.9999.1.2.100.0.6 integer 0 #os hc failed

.1.3.6.1.4.1.9999.1.2.100.1.0 string  127.0.0.1 #os IP
.1.3.6.1.4.1.9999.1.2.100.1.1 integer 8889 #os port
.1.3.6.1.4.1.9999.1.2.100.1.2 integer 0 #os disabled?
.1.3.6.1.4.1.9999.1.2.100.1.3 integer 0 #os reqs served
.1.3.6.1.4.1.9999.1.2.100.1.4 integer 0 #os reqs failed
.1.3.6.1.4.1.9999.1.2.100.1.5 integer 0 #os hc passed
.1.3.6.1.4.1.9999.1.2.100.1.6 integer 0 #os hc failed

.1.3.6.1.4.1.9999.1.2.100.2.0 string  127.0.0.1 #os IP
.1.3.6.1.4.1.9999.1.2.100.2.1 integer 8890 #os port
.1.3.6.1.4.1.9999.1.2.100.2.2 integer 0 #os disabled?
.1.3.6.1.4.1.9999.1.2.100.2.3 integer 0 #os reqs served
.1.3.6.1.4.1.9999.1.2.100.2.4 integer 0 #os reqs failed
.1.3.6.1.4.1.9999.1.2.100.2.5 integer 0 #os hc passed
.1.3.6.1.4.1.9999.1.2.100.2.6 integer 0 #os hc failed
```

```
######## Plugin-level stats #########


.1.3.6.1.4.1.9999.1.1000.0.0 string demo_pl_func # exec function name
.1.3.6.1.4.1.9999.1.1000.0.1 integer 1245 #number of plugin execs
.1.3.6.1.4.1.9999.1.1000.0.2 integer 0 #number of plugin drops
.1.3.6.1.4.1.9999.1.1000.0.3 integer 0 #number of plugin redirects
.1.3.6.1.4.1.9999.1.1000.0.4 integer 0 #number of plugin URL rewrites
.1.3.6.1.4.1.9999.1.1000.0.5 integer 0 #number of plugin body rewrites
.1.3.6.1.4.1.9999.1.1000.0.6 integer 0 #number of plugin provided resp
.1.3.6.1.4.1.9999.1.1000.0.7 integer 0 #number of plugin TTL sets
```

Clustering aiCache: United We Stand ...

Most web site deploy aiCache in multiples, for a number of reasons. Even if your site's traffic doesn't warrant multiple aiCache servers, you're still advised to have at least two, so that one can go down or be taken down for maintenance, without affecting the site.

With more than one aiCache server deployed, it makes sense to manage such collection of servers as a *cluster* - so that certain commands are applied cluster-wide, without having to apply them on each server separately. Most noticeably, when we expire content on-demand, manually via CLI or Expire-Header response-driven expiration, place sites into fallback mode, HC-fail mode or disable alerting, we want to make sure these commands execute on all of the aiCache servers at once. Otherwise, one has to login to each and every aiCache and re-run these commands, rather tedious and error prone procedure.

To support this type of behavior, aiCache uses notion of a *peer*. When one or more peers are defined, aiCache notifies all defined peers about cluster-aware commands as entered from CLI . Additionally, aiCache servers communicate via peer commands as required to accomplish assorted other chores.

To enable and define peers, simply add one or more of "**peer peer_IP [optional peer_PORT#]"** directives in the global section of the configuration file. For example the following snipped defines 3 peers, all running on port 81:

```
peer 1.2.3.4 81
peer 1.2.3.5 81
peer 1.2.3.6 81
```

The port setting is optional, it defaults to port 80. So the following snippet defines same 3 peers running on standard port 80:

```
peer 1.2.3.4
peer 1.2.3.5
peer 1.2.3.6
```

To communicate *peer-aware* commands to peers, aiCache sends a specially crafted HTTP request to each configured peer. Peer request is identified by a special URI prefix with default value of "/**xaicachepeer**". You can change it via **peer_prefix** configuration setting in the global section:

```
peer_prefix  supersecretpeerprefix
```

We highly recommend changing it to your own, secret, string to safeguard peer commands. Clearly all of the peers must have **peer_prefix** setting set to the same value. Please make sure **peer_prefix** and **stat_url** settings are set to different values so that one is not a prefix of the other.

With peers enabled, aiCache servers communicate with each other upon receiving of peer-aware CLI commands. Proper messages are shown and/or logged on both the sending aiCache instance and receiving aiCache instance(s). The command, command's parameters, the sending aiCache server and the receiving aiCache server(s) are all identified.

Please note that you can also disable peer-capable commands from communicating with the defined aiCache peers, if you want to run a such command locally as opposed to running it cluster-wide (which is the default behavior). To disable/enable peer communication, use **peer** CLI command (see CLI chapter for more on this and other CLI commands).

The commands received from peers are not retransmitted, it is the sender that communicates to each defined peer via a one-to-many communication. You can have aiCache identify its own IP address as one of peers, in case you want to maintain the same, bit-for-bit identical aiCache configuration file on a number of aiCache servers. Please see the dedicated CLI chapter for more information on cluster-aware CLI commands.

Somewhat related to the subject of clustering of aiCache servers is discussion on how the user traffic is directed to these servers. You can load balance the user traffic amongst all of the available aiCache servers, using a number of techniques: dynamic, short-TTL multiple DNS A-records, some form of load balancing in front of aiCache (F5, A10, Netscaler, HAProxy etc are some examples of commercial and open-source load balancers). Such setup might be referred to as "all-hot", as in "all aiCache servers serving traffic simultaneously".

This is the opposite of "hot-cold" - where only one aiCache server serves traffic, while other servers are in cold-standby mode, ready to take over should the primary fail. A variation of this scheme is having some aiCache servers be *primary* for some domains, which acting as *stand-by* for other domains.

Simplifying configuration management in distributed setups.

When managing a number of aiCache servers, you might find yourself first modifying, testing and applying new configuration files on a preferred, "master" aiCache server and then performing frequent configuration file(s) copying from such *master* aiCache server to other aiCache servers. To simplify and automate this chore, such *master* aiCache server can be configured to serve files from its local file system in response to special HTTP requests.

For example, you can configure aiCache to serve files from a directory called **/usr/local/aicachefiles**, when request URL start with a certain prefix, by specifying **file_req_prefix** and **file_doc_root** global-level settings :

```
file_req_prefix /xaigetfile
file_doc_root /usr/local/aicachefiles
```

Let's imagine there's a file called **aicache.cfg.tar** in **/usr/local/aicachefiles** folder . To request that file from aiCache master server running on a system with IP address of 1.2.3.4 , you can request the following URL:

http://1.2.3.4/xaigetfile/aicache.cfg.tar

Such requests can be programmatically scripted, to be executed from "slave" servers via a **wget** command. For example:

```
$ wget http://1.2.3.4/xaigetfile/aicache.cfg.tar
```

And here's how you can wire this up. First, setup the master aiCache server to tar up it's configuration file(s) upon successful configuration reload, by specifying global-level setting of **reload_run_cmd**. You can create a simple script that creates a tar archive of the configuration file(s) and copies the resulting file to **file_doc_root** location. If there's only one configuration file (no aiCache **include** directives used), there's no need to use tar and you can just copy that single configuration file around. If you configuration files contain sensitive information, you can encrypt their content or rely on security of **file_req_prefix** setting.

Now, on the slave servers, you can run a daemon-like script, that periodically pulls this file from the master aiCache server, via a simple **wget** . Upon receiving of the file, the script can compare the MD5 checksum of the downloaded file to the prior checksum. Should checksums differ, the script can untar the downloaded file so that proper configuration files end up in proper location and the force aiCache configuration reload by creating a reload watch file (you can specify one using **reload_watch_file** global-level setting).

This way you simply modify and apply the new configuration on one aiCache server and then the changes are automatically and rapidly propagated to all of the other aiCache servers you might have, without having to login into each and every one and manually applying the new configuration to each server. We recommend to setup such configuration polling to run every 10 seconds or so, assuring very little incremental load on the master server and expeditious propagation of the new configuration.

Here's an example configuration download script. Feel free to use it as a starting point for your own setup. Carefully review and modify it to fit your setup. Don't forget that the *master* aiCache server needs to be configured to serve the requested file(s) . You setup might be very simple so that you can allow direct access to the main configuration file - in which case you don't need to have any post-reload scripts on master server. If you do have a number of configuration files, you will likely need a simple post-reload script to tar these up and to copy them to **file_doc_root** .

```
#!/bin/bash

#
#
#
# URL to pull the configuration file(s) from master aiCache.
# Make sure you setup matching file_req_prefix and file_doc_root settings
# on master aiCache server, in global settings section.
#

URL=192.168.168.8/xaigetfile/

#
# The actual name of configuration file(s) (bundle) that is to be requested
# from master aiCache. If multiple  files are to be transferred,
# you can use tar archive
#

DFILE=accel.cfg
#DFILE=config.tar

#
# Script persists the MD5 of the previous download in this file
#

MD5FILE=/tmp/md5file

#
# Reload watch file. Make sure to specify matching reload_watch_file
# aiCache global-level setting.
#
RELOAD_WATCH_FILE=/usr/local/aicache/reload

# Config directory, might need to change to match your installation
AI_CFG_DIR=/usr/local/aicache


#
# Sleep interval
#

SLEEP=10

#
#  Work dir
#
WORKDIR=/tmp

#
#
#

cd $WORKDIR
```

```
while (true); do

rm -f $DFILE
wget $URL$DFILE

if [ -f $DFILE ]; then

echo "Obtained $DFILE"

omd=`cat $MD5FILE`
md=`md5sum $DFILE | cut -f1 -d" "`

echo "Comparing >$omd< to >$md<"

if [ "$omd" = "$md" ]; then

 echo "No configuration changes detected."

else

 echo "Configuration changes detected. Applying new configuration"
#
# Here you can either copy the new configuration file to the aiCache
# configuration directory, or untar the downloaded configuration file
# bundle to proper location.
#
# After doing that, we trigger aiCache reload by creating reload watch
# file. You can also email the results of the reload (success or failure)
# - these are written by aiCache when you have specified reload_success_file
# and/or reload_fail_file settings
#

 cp $DFILE $AI_CFG_DIR
 touch $RELOAD_WATCH_FILE


fi # EO MD5 compare

echo -e $md > $MD5FILE

fi # DL file test


sleep $SLEEP;
done;
```

You can run this script at system boot up or manually, using **nohup** option to make sure script keeps running even after you log out.

To assist you in ascertaining what version of configuration file is in effect on different aicache servers, you can "tag" configuration file by setting **cfg_version** global level setting. It takes a single value, which is then reported via Web and CLI statistics display. For example:

```
cfg_version 03102010-004
```

Don't forget to modify this setting every time you make changes to the configuration file - similar to how DNS zone files are managed. You can use some sort of naming convention - such as specifying the data and time of configuration change, along with a version number for that date. Exact format is completely up to you.

Please note that for requested file to be delivered, the request's Host header must not match to any known accelerated site. You can either use master aiCache's server IP address or "fake" it via host-file trick.

Please also note that when serving files in this fashion, aiCache doesn't cache the content of the file - it is read anew every time, every request. So do not use this feature to inject static content into accelerated websites, see "Serving/Injecting Static Content" chapter on how to do it properly instead.

Make sure the file and directory permissions are set to allow read access by aiCache username.

Building HA Clustered aiCache setup with VRRP.

In this section we will describe the de-facto golden standard of highly available, fully redundant hot-hot aiCache setup, using VRRP protocol. VRRP stands for Virtual Router Redundancy protocol. While VRRP is frequently used with routers and firewalls, here we consider VRRP's applicability to server setups.

The basic idea here is to have 2 aiCache servers that back each other up, so that when one server sustains an outage or needs to be taken down for maintenance, the remaining server seamlessly and expeditiously takes over the traffic from the fallen comrade.

Now, such setups are normally done in a **hot-cold** fashion, where second server is not handling any traffic, unless and until the first server fails. As you can imagine, it is rather wasteful to have a server doing nothing, most of the time - so the setup we describe is **hot-hot**, where both servers are serving traffic while backing each other up.

Prerequisites.

The setup requires 2 servers running aiCache. Both servers need to be connected to common network segment (VLAN). For servers to be able to handle any network traffic at all, they have to be attached to a network and have their own, so called *physical* (as opposed to virtual) IP addresses. Our example uses aiCache servers called #1 and #2, attached to common segment 1.1.1/24 (1.1.1.255 is the broadcast address). Server #1 has physical (regular, as opposed to VRRP *virtual IP, or VIP)* IP of 1.1.1.1, while server #2 has IP of 1.1.1.2.

To accomplish HA network configuration, we shall use Open Source implementation of VRRP protocol. Unless already installed on your servers, you need to install two packages: KEEPALIVED and IPVSADM. Obtain and install recent, production releases of both packages, on both aiCache servers.

We shall explain VRRP and its capabilities to the extent required for our example and in the fashion sufficient for the same. As result, we shall barely scratch the surface of what VRRP is capable of and our explanation of VRRP's inner workings is greatly simplified.

When running clustered HA aiCache instances, you must make sure all of the nodes have the same aiCache configuration, so that each node can accelerate all of the sites that you want to make highly available. To ease the management of cluster, please make aiCache aware of each other via **peer** settings.

VRRP introduction.

Greatly simplified, VRRP is all about high availability and it breaks down like this. What VRRP makes highly available are so called **Virtual IP**, or **VIP**. In other words, through magic of VRRP and proper setup, you can configure virtual IP addresses that are always up. To configure a highly available VIP, we need to have more than 1 server that is capable of claiming the VIP, so that a failure of a single server doesn't take the VIP down. All of the servers (clearly, at least 2 are required for HA setup) capable of claiming a given VIP, belong to the same group - called VRRP *instance*.

In our example we cover 2 VIPs: 1.1.1.20 and 1.1.1.30. Each belongs to its own VRRP instance. A server can participate in a number of VRRP instances.

For servers to participate in VRRP, they must be running proper VRRP software and have proper VRRP configuration. Through VRRP software, VRRP participants constantly exchange assorted messages, including heartbeats, over IP multicast. This way participants always are aware of each other and can have elections - electing master for a given VIP. Master claims the VIP and all of the traffic destined to that VIP now flows to the master server.

To make sure only one server claims any given VIP at any given point in time and it happens in deterministic fashion, servers are normally configured with different *priorities* set for each VIP. The server with higher priority wins and claims the VIP. A VRRP server can be a master for and claim a set of VIPs, while acting as one of backup servers for a set of different VIPs.

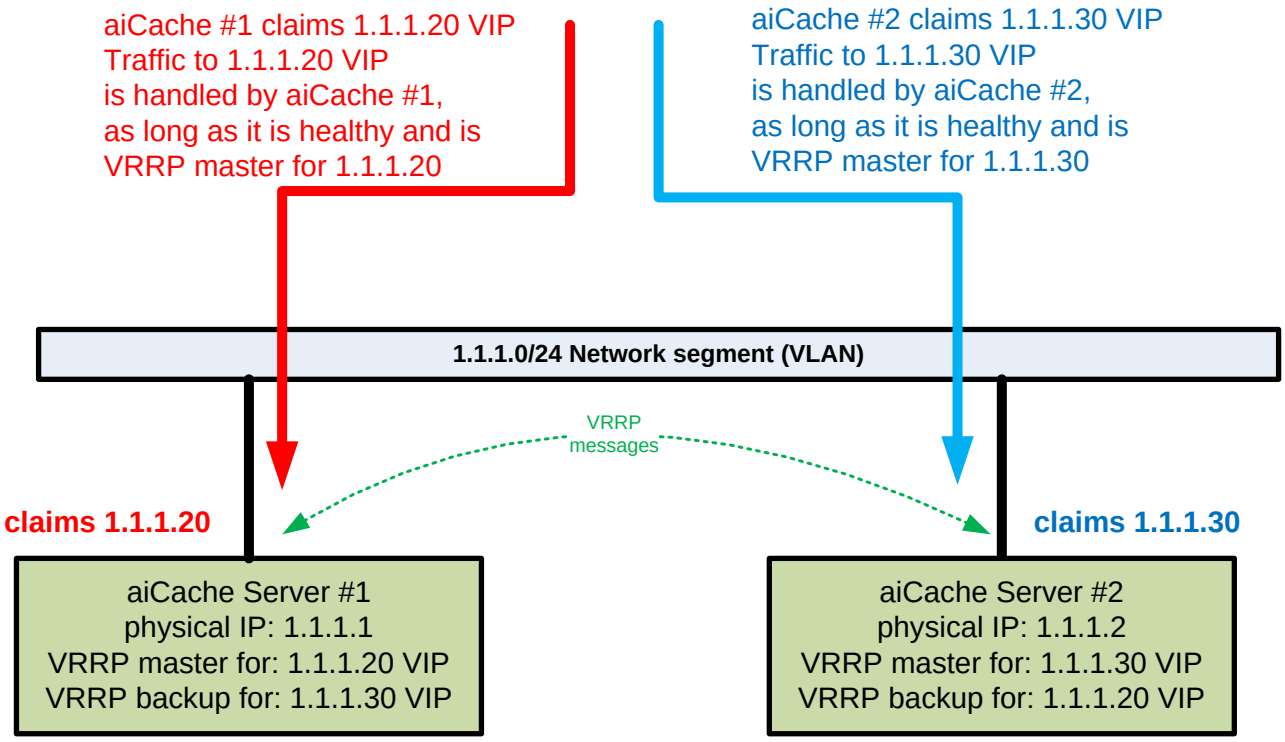
Should the current VIP master sustain an outage and disappear from the network, so will the VRRP heartbeat messages from this server. The other members of the same VRRP instance detect that fact and after a brief pause (1-3 seconds, depending on the configuration) hold another election, selecting a next server to claim the VIP from the fallen server. In our case, since there're only 2 servers, it is the other, surviving, server that will claim the VIP.

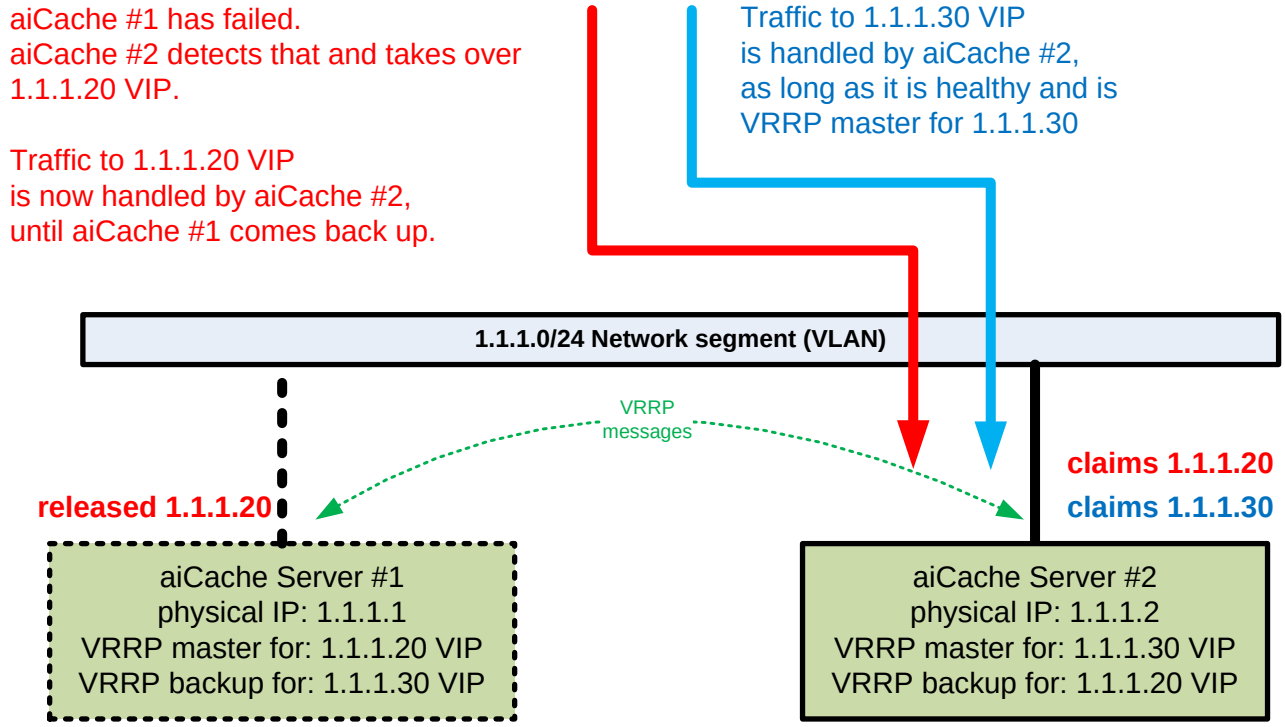
With the VIP claimed by server #2, now we must make sure that the network switch/router knows that traffic for the VIP should now be sent out the network port that server #2 is attached to. To make sure it happens as soon as possible (as opposed to after a lengthy ARP expiration routine), server #2 issues a gratuitous ARP message - which is received by the switch it is attached to and leads to immediate port resassignment. VRRP VIP MAC addresses belong to a special MAC address block, so a given VIP always has the same MAC address, even as it travels from one server to another.

Should server #1 come back up, it starts sending heartbeat messages again. As a result, another VIP election takes place and server #1, having higher priority, takes back the VIP. The *take back* is nearly instantaneous, as opposed to *take over*. Alternatively, VRRP can be configured so that the VIP take back is not automatic, but requires operator intervention for it to happen. If server #2 has no issues handling traffic for both VIPs, you might elect to keep the VIP on server #2, even after server #1 comes online.

The diagram on the next page depicts an example VRRP/aiCache setup that we discuss in this section. Two aiCache servers each have a single interface, "eth0", on 1.1.1/24 subnet (VLAN). The servers might be single or dual-attached (NIC teaming). Both servers can be attached to the same or different switches/routes. Exact setup is up to you, but better setups assure additional resiliency by having redundant switches.

The top section of the diagram shows normal operating conditions, with both servers up. The bottom section displays one of the failure scenarios, with server #1 sustaining some form of failure and disappearing off the network, leading to second server taking over of VIP. Example failures include loss of power, network connection etc.





Assuring hot-hot setup.

In our example, 2 aicache servers serve two different VIP addresses: 1.1.1.20 and 1.1.1.30. Under normal operating conditions, with both servers up, VIP of .20 is claimed by aiCache server #1, while VIP of .30 is claimed by aiCache server #2.

For setup to be truly hot-hot, we need to make sure both VIPs receive traffic. How can we go about it ? Two basic ways to accomplish this exist: DNS magic and domain-splitting.

We can setup multiple A records for the same domain. For example www.acmenews.com can have two DNS "A" records: one pointing to 1.1.1.20 and the second one pointing to 1.1.1.30. Mission accomplished. Now both VIPs will receive HTTP traffic for our domain, as web browser load balance requests to both VIPs.

Domain splitting is a different method where we can point www.acmenews.com to 1.1.1.20, while pointing media.acmenews.com to 1.1.1.30. This way both VIPs server traffic, albeit for different domains.

You'd need to decide which of the 2 methods is most appropriate for your setup. Let's say you need to serve 1.5 Gbps of peak traffic for www.acmenews.com . Sustaining this volume of traffic (and it assumes you do have that much of egress capacity to the Internet) while having servers attached to the network at 1Gbps, clearly requires more than 1 server pumping data for www.acmenews.com at the same time. So you'd need to use multiple A records methods in this scenario. On a side note, one can argue that to have truly redundant setup in this case, one needs at least 3 servers, so that with 1 server down, you still have enough network capacity to serve 1.5 Gbps.

On the other hand, if your traffic levels are more moderate, you might resort to domain splitting instead. For example, if www.acmenews.com peaks at 300 Mbps and media.acmenews.com peaks at 450 Mbps, you can point www.acmenews.com to 1.1.1.20 VIP and point media.acmenews.com to 1.1.1.30 VIP.

Now, under normal operating conditions, with both servers up, traffic is split between both servers. Should a server fail, the surviving server can take over the VIP and the VIP's traffic from the fallen server and serve both domains, peaking at 750 Mbps, still under its maximum capacity of 1 Gbps.

Example VRRP configuration.

Let's review example VRRP configuration settings for both servers. Examples imply that both servers have their physical IP pre-configured. Each server has 2 VRRP instances configured, one for each VIP. As you can see, the configuration files are nearly identical, with the only differences being the *state* and *priority* VRRP settings.

The keepalived VRRP configuration is setup through a single file: **/etc/keepalived/keepalived.conf**

Here's content of **/etc/keepalived/keepalived.conf** file on server #1:

```
vrrp_instance VIP1 {

  state MASTER
  interface eth0
  virtual_router_id 1

  priority 100
  authentication {
    auth_type PASS
    auth_pass secret
  }
  virtual_ipaddress {
    1.1.1.20 brd 1.1.1.255 dev eth0
  }
}


vrrp_instance VIP2 {

  state BACKUP
  interface eth0
  virtual_router_id 2

  priority 80

  authentication {
    auth_type PASS
    auth_pass secret
  }

  virtual_ipaddress {
    1.1.1.30 brd 1.1.1.255 dev eth0
  }

}
```

Now content of **/etc/keepalived/keepalived.conf** file on server #2:

```
vrrp_instance VIP1 {

  state BACKUP
  interface eth0
  virtual_router_id 1

  priority 80
  authentication {
    auth_type PASS
    auth_pass secret
  }
  virtual_ipaddress {
    1.1.1.20 brd 1.1.1.255 dev eth0
  }
}
```

```
vrrp_instance VIP2 {

  state MASTER
  interface eth0
  virtual_router_id 2
  priority 100

  authentication {
    auth_type PASS
    auth_pass secret
  }

  virtual_ipaddress {
    1.1.1.30 brd 1.1.1.255 dev eth0
  }
}
```

Basic VRRP commands, operation and troubleshooting.

To start VRRP keepalived daemon, run the following on both aiCache nodes:

```
/sbin/service keepalived start
```

To see what VIPs are claimed by what node, run the following on both aiCache nodes:

```
/sbin/ip addr show
```

To test VRRP operations, you can simulate server failure by stopping keepalived service on a node and observe the VIP being taken over by the surviving node.

To see how long it takes for a VIP to be taken over, have a continuous ping going against the VIP from a third server. You will see a loss of some (1-3) packets as VIP is taken over and taken back. Consider executing physical network disconnection test and carefully measure the time it takes for VIP to come back. Likewise, study and measure what happens when the server is reconnected back to the network.

Here's another operating routine worth describing. Let's say you need to add more RAM to server #1, which requires a shutdown. Clearly, you can just power off server #1 and few seconds later the server #2 will take over the VIP and restore the service.

However, there's a better way to go about. Reduce VRRP priority on server #1 and watch the VIP travel to server #2 with no downtime at all.

Conclusion.

We've shown an example clustered and highly-available aiCache setup, complete with an introduction to VRRP protocol, diagrams and actual VRRP configuration files. The information we have provided, while brief, is fully sufficient to setup and configure state-of-art highly available clustered configuration.

There's a wealth of available VRRP information on the Internet that you could study to obtain a deeper understanding of VRRP. Some of the more interesting topics to consider:

- VRRP checks. You can run pre-built checks or create your own. Based on the result, you can lower or increase the VRRP priority for a given VIP, to have even more resilient setup.
- Automated alerting. VRRP can send an email every time VIP travels from one node to another.

Executing aiCache commands via web requests.

As peer commands are effectively regular HTTP requests, you might, in a pinch, use this to execute aiCache commands via a simple web request. All you need to know is how to build the proper command URL, copy the string into a browser and send the request to proper aiCache instance(s).

Peer commands must start with **peer_prefix.** Default value: **/xaicachepeer** . **We highly recommend changing it to your own, secret, string to safeguard peer commands.** This prefix serves dual purpose: it is an indication to the receiving aiCache instance that it is receiving a special *command* HTTP request and it is also a security measure, a password of sorts. One has to know **peer_prefix** value to have aiCache execute the received request.

Following the prefix is one letter command code. Following the command code is the web site name that the command is for. The web site name is followed by a "?" (question sign). Following the question sign, there might be optional parameters. Parameters depend on the peer command.

Here's an example URL for aiCache "expire pattern" command. The command code is "p". We're executing it for acmenews.com, asking to expire all of the content that matches "breakingnews". We're sending the command to aiCache server running on IP address of 1.2.3.4.

http://1.2.3.4/xaicachepeerpacmenews.com?breakingnews

aiCache logs all peer commands it receives, along with command execution results in its error log file. aiCache doesn't send any response to peer commands, the connection is silently closed. You can change this behaviour by setting server-level flag of **send_peer_response.** When set, a small HTML body will be sent in response to peer requests, containing string of "Peer request received".

We most strongly advise setting the **peer_prefix** to a different value, to safeguard your installation from unauthorized peer commands.

Denial-of-Service Attack Protection.

Introduction to DOS Attacks.

DOS is an attack aimed at reducing ability of a web site to offer service to legitimate users, by generating and directing, at the said web site, specially crafted and typically overwhelming, volume of traffic.

Some of the examples of such DOS traffic might include:

- Fake requests for establishment of new TCP/IP connections (SYN floods etc). These are best defended against "in hardware" - for example by uplink firewalls.
- Oversized, malformed or specially crafted Web requests, sometimes aimed at exploited vulnerabilities of assorted infrastructure components - routers, switches, firewalls, web, application and DB servers.
- Legitimate looking requests, generated at such a high volume that they strain infrastructure capacity to the breaking point, again, at different levels - uplinks, routers, switches, firewalls, web, application and DB servers.

Rather frequently, the perpetrators of such DOS attack might have under their control a whole number of well distributed load-generation computers all over the Internet. Such computers are mostly personal computers of un-suspecting users, that have been infected with viruses that places these infected computers under complete control of attackers - turning them into bots, able to do whatever their masters demand of them, from stealing keystrokes to generating thousands of requests against a victim website.

First Level Of Defense: malformed request protection, URL blocking, BMR patterns and *no-replacement-for-displacement*.

It is aiCache's unrivaled ability to detect and deal with malformed and oversized requests that constitutes first line of defense. Again, due to aiCache's non-blocking multiplexed IO model, it is capable of dealing with tens of thousands of requests and connections, without overloading origin servers.

Only after a valid, legitimate request is obtained and it requires a response from origin servers, does aiCache forward such requests to the origin servers, effectively blocking and dissipating malicious requests. You already know how to enforce assorted time and size sanity checks on requests and block certain patterns outright. As a quick reminder, you can configure aiCache to send back error responses or drop the connections silently, whenever such malformed/oversized or blocked requests are discovered.

You can also configure aiCache to inspect request bodies, looking for suspicious patterns, using aiCache's BMR (body-match-reject) patterns, as described earlier in the manual.

Additionally, a request for anything that is cached by aiCache, doesn't even require a request to origin server and due to aiCache's non-blocking multiplexed IO model, it is capable of dealing with tens of thousands of requests and connections, without overloading origin servers.

Second Level of Defense: IP blocking.

Frequently, as DOS attacks unfold, you will discover that you're being bombarded with DOS traffic from certain number of bots on the Internet. From you networking team or by processing log files, you might be able to discover certain IP addresses and/or ranges of IP addresses that seem to be generating the bulk of the load and would like the ability to block such IPs from accessing your website. aiCache also offers its own, evolved reporting of such abuser IP addresses (see next section).

You might be able to request such IP blocking to be instituted on your uplink routers or firewalls (which might require cooperation from your Networking team), but aiCache is also capable of doing the same.

To configure such IP addresses, you can provide a list in the aiCache configuration file, feed such IP ranges at runtime via CLI interface - one by one, or load a whole list from a file.

To configure IP addresses to block in the configuration file, provide one or more of **block_ip** global directives, for example:

```
block_ip 1.2.3.4
block_ip 45.222.13.0/24
block_ip 10.0.0.0/8
```

As you can see, the parameter could be a simple IP in a "dot" notation or a whole IP range/class: a valid IP address (all 4 octets must be present), followed by optional bitmask.

Upon seeing any traffic from matching IP addresses, aiCache can either silently and instantly drop the request and TCP/IP connection, or redirect offending requestor to a "sorry" page. The idea behind it to minimize the amount of work your infrastructure needs to do when under attack. So, instead of sending back a lengthy response, you might want to consider either dropping the connection outright or sending back a very small redirect to a "sorry, your IP address is temporarily blocked" page.

If your setup requires matching any IP, you can specify it as 0.0.0.0.

To configure silent drop, specify **silent_block_ip** global setting. It is a flag and requires no parameters.

To configure redirection to a sorry page, specify **block_ip_sorry_url** global setting, for example:

```
block_ip_sorry_url  http://acme.com/ipblockedsorrypage.html
```

When aiCache has to deny a request due to client IP address matching one of blocked IP addresses/ranges, it sends a redirect to the specified page. If one is not specified, the niceties are dispensed with, aiCache silently

and instantly drops the connection. It is up to you to craft the page. We recommend to keep it short to preserve the bandwidth and other resources.

You might have a setup where you might be receiving lots of requests from certain "trusted" systems, that you don't want to ever block or apply any other DOS protection to. To specify such trusted IP addresses or ranges, please specify them via **allow_ip** global configuration setting. If your setup requires matching any IP, you can specify it as 0.0.0.0.

For example:

```
allow_ip 1.2.3.4
allow_ip 45.222.13.0/24
allow_ip 10.0.0.0/8
```

To configure additional block ranges via CLI, use CLI **blip** command. Without any parameters, **blip** simply displays configured blocked IP addresses (ranges) - both the address and number of blocks that have been applied so far for this particular IP/range.

To add another range, use blip like this:

```
blip 2.2.2.2/16 on
```

To enable an IP (range), change **on** to **off**. For example:

```
blip 2.2.2.2/16 off
```

Sometime, you might have a file containing a list of IP addresses/ranges that you need to block or unblock. Clearly, you'd rather avoid having to specify each and every of those separately, via **blip** command. Worry not and use CLI **fblip** command (file-based blip).

To specify a list of IP addresses/ranges to block, using an input file, use **fblip** like this:

```
fblip block.txt on
```

To specify a list of IP addresses/ranges to block, using an input file, use **fblip** like this:

```
fblip block.txt off.
```

Please note that both **blip on|off** and **fblip filename on|off** are peer-aware CLI commands, so you don't have to re-issue these at each and every aiCache instance, but only do it once at any of the nodes and the change will be automatically propagated to all nodes of aiCache cluster.

Both the total and per/sec numbers of requests and/or connections that were blocked due to IP matching a defined blocked IP address or range, are displayed/reported via CLI, SNMP and Web interfaces. You can also choose to be alerted when number of blocks/sec is over a set limit by providing **alert_blocked_ip_sec** global setting, for example:

```
alert_blocked_ip_sec 20
```

When aiCache has to drop an incoming request due to its source IP being blocked, it uses special logic not to overwhelm the log files with the "blocked" messages, so no more than 120 messages are logged per minute.

The third level of defense: intelligent request throttling.

Sometimes, you won't be able to tell if a DOS attack can be attributed to a range of IP addresses/ranges that you could easily block, rather it seems to be coming from a whole number of systems/bots, all over the Internet and you're going desperate. The site just cannot stay up under the onslaught of DOS traffic.

Time to deploy another level of protection - intelligent traffic throttling. When enabled, this protection mode allows any given source IP to make only a certain amount of requests per certain amount of time - both, of course, configurable.

For example, you know that on your website, a typical webpage consists of up to 10 elements that are served from your site: the main HTML, 2 CSS, 1 JS file and 4-5 images. With this in mind, you can configure aiCache to allow no more than 10 requests every 20 seconds. This way the legitimate users won't be inconvenienced too much - they can still open a page every 20 seconds - so the all important profile page can be modified, news be read, video played etc, but the protection will certainly decimate the DOS traffic - and save the day !

To configure this mode, you need to specify optional **block_ip_interval (**default value 10 seconds) : the amount of time that up to **block_ip_max_req (**default value: 20 requests) are allowed to pass through. For example:

```
block_ip_interval 20
block_ip_max_req 10
```

The settings above allow up to 10 requests from every single source IP address every 20 seconds. Just as with regular IP blocking, you can also specify a "we're sorry" page:

```
block_ip_sorry_url  http://acme.com/ipblockedsorrypage.html
```

When aiCache has to deny a request due to throttling rate being exceeded, it sends a redirect to the specified page. If one is not specified, the niceties are dispensed with, aiCache silently and instantly drops the connection. It is up to you to craft the page. We recommend to keep it short to preserve the bandwidth and other resources.

Normally, DOS bots do their best to swamp your site with DOS requests. But what the bots don't know, is going to hurt them: the more traffic in excess of allowed limit a bot tries to generate, the easier it is for aiCache to make the determination that the traffic is in fact coming from a bot . So aiCache will block the offending IP address for a much longer period of time (yes, you guessed it - the penalty time is configurable too).

Here's how it works: if aiCache has to block more than **block_ip_punish_factor** (default value: 5) times the amount of **block_ip_max_req** requests, it will disable the traffic from the offending IP for **block_ip_punish_time** (default: 600 seconds) seconds. Both parameters are optional. For example:

```
block_ip_punish_factor 5
```

```
block_ip_punish_time 1200
```

specifies that if number of requests are clocked at 5 times or more of the allowed limit, the offending IP is to be hard-blocked for 20 minutes of time (20*60 = 1200 sec).

There two different ways to turn on this level of defense: you can issue either a CLI command: **clipt** or let aiCache to auto-active it. Please note that **clipt** CLI command is peer-aware.

```
> clipt on
> clipt off
```

You can specify **clip_throttle** global level setting in the configuration file - in which case aiCache will start up with the CLIP throttling enabled. Make sure to white-list any of the internal IPs before turning CLIP protection on.

You can also tell aiCache to auto-activate the IP throttling by configuring the following server-level settings: **auto_throttle_cps** and **auto_throttle_cps_interval**. Effectively, you're telling aiCache: activate throttling when number of client connections per second exceeds the **auto_throttle_cps** , activate it for **auto_throttle_cps_interval** seconds (defaults to 600 seconds). After this interval passed, aiCache will re-test the CPS against the provided limit and should the CPS drop below the threshold, the throttling will be disabled.

Both enabling and disabling of the throttling will be logged in the aiCache error log file. When you configure aiCache to auto-throttle in this fashion, beware that issuing manual **clipt** command might override presently set protection state.

When auto-arming CLIPT mode in this fashion, aiCache does not communicate to its peers, the idea being that they, in turn, can auto-activate or de-activate this protection on their own.

Both the total and per/sec numbers of requests and/or connections that were throttled due to intelligent request throttling, is displayed/reported via CLI, SNMP and Web interfaces. You can also elect to be alerted when number of blocks/sec is over a limit by providing **alert_ip_throttled_sec** global setting, for example:

```
alert_ip_throttled_sec 20
```

When aiCache has to drop an incoming request due to its source IP being throttled, it uses special logic not to overwhelm the log files with the "throttled" messages, no more than 120 messages are logged per minute.

The fourth level of defense: Reverse Turing Access Token Control (RTATC).

Introduction to RTATC.

As most of DOS attacks are driven by load generation binaries/scripts executed by bots, you might be able to weed these off by challenging the requesting side to what is known as a reverse-Turing test (http://en.wikipedia.org/wiki/Reverse_Turing_test) .

When the requesting side can prove that there's in fact an operator (a human being) requesting access to your website, you would allow the request to pass through.

aiCache offers complete and very flexible RTATC implementation. Here's a very brief summary of how it works: when RTATC mode is on, aiCache challenges the requesting side to a reverse Turing test. Should the response to the test indicate presence of an operator, an Access Token, good for configurable amount of time, is issued to the requesting browser. The requesting browser send the AT in all subsequent requests and aiCache allows these requests to pass through based on the presence of a valid AT.

Should requesting side fail 5 consecutive challenges, its access is blocked entirely, for configurable amount of time. Should an invalid AT be repeatedly sent by a requesting side, again, its access is blocked entirely, for configurable amount of time.

Lets now go into detailed description of the configuration.

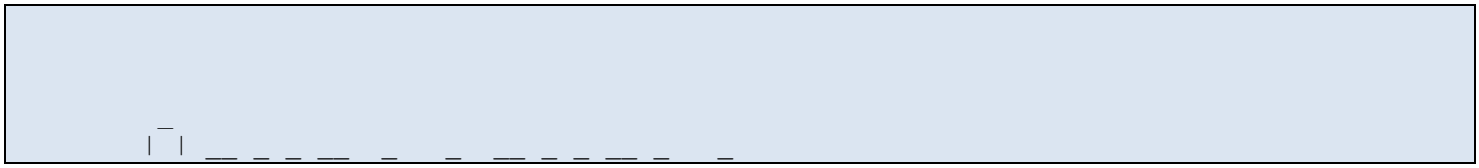
Flexible Challenges.

You tell aiCache the name of a directory that contains a set of challenge files. Each file is named after the proper response to the challenge. This provides ultimate flexibility in just what can be in the challenge file.

For example, you can have a file named "July" with the following content: "What is the summer month that starts with J that follows the first summer month that starts with the same letter?".

Another example: a file named "elephant", with the following challenge "What has four legs and is always willing to travel ?".

Now, admittedly these are tough challenges and you can certainly come up with even harder ones. But we recommend you instead resort to CAPTCHA challenges. Enter the ASCII art CAPTCHA.



```
      _   | |/ _` | '_ \| | | |/ _` | '__| | | |
     | |_| | (_| | | | | |_| | (_| | |  | |_| |
      \___/ \__,_|_| |_|\__,_|\__,_|_|   \__, |
                                          |___/
```

Can you recognize the word "January" in the ASCII art above ?

You can use a program called "figlet" (http://www.figlet.org/) - which you can download and install on aiCache server or a different computer of your choosing, to generate the ASCII art challenges.

The challenge page consists of 2 parts: a configurable prefix and the actual content of the challenge file, inserted as <PRE> (preformatted) HTML .

The configurable prefix comes from a file, whose location you specify via required **atc_challenge_prefix_file** setting, for example:

```
atc_challenge_prefix_file apf.txt
```

For example, you can place the following HTML snippet into this file:

```
<P> We're sorry, but due to present situation we require you to enter the name of a
month you see below into the text box and click submit, in order to gain access to our
site.</P>
```

Please note that you're not required to place any of common HTML opening tags into that file, such as <HTML>, <BODY> , <TITLE> etc, aiCache takes care of that. Nor are any of customary closing tags required.

You can then use ***figlet*** to generate 12 challenges, in font (it can get quite fancy in a hurry) of your choosing and store these files as ***January***, ***February***, etc under the directory you specify via required **atc_challenge_dir** setting, for example:

```
atc_challenge_dir /usr/local/aicache/atc_challenges
```

Now aiCache will randomly select one of the challenges and present them to the requesting side when in RTATC mode. You can certainly opt for different challenges - say not limited to the names of the months of the year, but have a list of say thousand common dictionary words and produce ASCII art CAPTCHA out of them, using ***figlet*** and a simple script. It is up to you, there's no limit as to what can be in the challenge file.

When RTATC mode is on (you can turn it on via CLI), the requests without a valid ATC are subjected to the challenge. When a proper response to the challenge is detected, an Access Token is issued, as a cookie. You can configure the cookie's name via optional **atc_cookie_name** global setting, for example:

```
atc_cookie_name  xaiatccookie
```

The default ATC cookie name is **xaiatcookie**.

The cookie is issued to the requesting browser with expiration set to *session*. Internally, aiCache marks the issued cookie for expiration within a configurable amount of time, set via optional **atc_ttl** setting (defaults to 30 minutes, specified in seconds).

The cookie is a one way cryptographic function of Client IP, Token expiration time and a special secret. You configure the secret via optional **atc_salt** setting. While it does have certain default value, we recommend you change to your own alphanumeric string of 6-10 characters.

Upon successful completion of the challenge, the requestor is redirected to a URL that you configure via required **atc_welcome_url** setting, for example:

```
atc_welcome_url http://acme.com/DOSChallengeOK.html
```

It is your responsibility to assure you configure a valid page. The content is entirely up to you, but in that page you can thank the user for taking the time to complete the challenge, explain the reasons why they were subjected to it and advise to go the link they intended to go to originally (simply providing a link to the home page will suffice in most cases).

When aiCache receives a wrong response to the challenge, the requesting side is given up to 4 more opportunities to complete the challenge. A different challenge file will be selected, randomly. Should all attempts fail, aiCache will block the requestor's IP address for configurable amount of time. You configure this punishment interval via optional **atc_fail_punish_time** setting (default value 1800 seconds or 30 minutes, value must be provided in seconds).

The same punishment applies when aiCache detects repeated attempts to pass an invalid access token.

The challenge verification URL is configurable via optional **atc_submit_url** setting, that has a certain default value. It is up to you if you want to change it.

Turning RTATC mode on/off.

The RTATC mode is off by default. You would normally engage it only when under DOS attack and you do so via **atct** CLI command. For example:

> **atct on** : turns the RTAC mode on

> **atct off** : turns it off

> **atct** : shows the status of the mode

The atct command is peer-aware, you only need to execute it one node of the cluster.

If you want to be able to turn the RTATC protection mode on, you must take care to specify all required RTATC settings before starting up aicache, as explained above.

RTATC Statistics Reporting.

Most crucial RTATC statistics are reported via CLI, Web and SNMP interfaces. You can see number of times challenges were issued, met or failed, along with number of time good and bad access tokens were received. These are only reported when the site is in RTATC or Intelligent Throttling mode, to preserve screen real estate.

RTATC and Intelligent throttling.

When you activate RTATC mode, the intelligent throttling is activated automatically, as best DOS protection is attained when both protection modes are on. When turning RTATC mode off, please turn off intelligent throttling separately, via **clipt off** CLI command.

Blocking and intelligent traffic throttling in NAT'd setups.

Sometimes aiCache is deployed behind a NAT'ing (Network-Address-Translation) load balancer or other intermediary networking device. As a result, all Web requests that aiCache sees, will appear to come from the IP address of the NAT'ing device. But, blocking and intelligent throttling both rely on knowing of the true client IP address to operate, you say, so what do we do now ?

Fortunately, almost all intermediaries are capable of forwarding true client IP as an HTTP header. In other words, the NAT'ing devices can modify the inbound requests to include an additional request header. Typically you get to specify what that header is called. Sometimes it is called "**X-Forwarded-For**" .

No matter the name, all you have to do for the DOS protections magic to continue working in a NAT'd setup is to specify what header is called via hdr_clip global setting, for example:

```
hdr_clip X-Forwarded-For
```

Assisting with DOS forensics.

aiCache can help you determine the offending IPs in case of DOS attack. Of course, you can manually go through the log files and tab the requests by IP addresses - but doing that in when under a massive DOS attack is less than a pleasant exercise.

With Intelligent Throttling in effect, aiCache automatically produces the list of heaviest hitters - the source IPs that drive 10-times or more, of **block_ip_max_req**. The file is refreshed every **block_ip_interval** interval and is called /tmp/clip_offenders.txt . Each line of that file contains the observed number of hits, the source IP of the offending requestor and the hard-drop flag. The hard drop flag is set to "1" to indicate that this IP is being punished . You can easily sort the file by number of requests to obtain the top offenders.

Client IP dependency when in IT or RTATC DOS protection modes.

Please note that both protection modes take into account the source IP of the requesting client. As explained above, the source IP must be made available for these protection modes to function. There're a number of reasons for this decision.

While having a potential to inconvenience a very small percentage of users (say when an IP address of a corporate proxy ends up being blocked, effectively temporarily disabling all of the users behind that particular proxy), it is a small price to pay when you want to restore the service to the majority of your users.

You can also white-list the IP addresses of well-known large-scale proxies (such as AOL etc) via **allow_ip** configuration setting.

Best defense is layered defense.

As you can see, aiCache offers a very potent and flexible defense against a number of common DOS attacks. However, some types of attacks are best when deflected before they ever reach aiCache servers. For example, SYN Flood DOS attacks are best handled by dedicated hardware - firewalls and load balancers. Some of the better devices are capable of sustaining millions of SYNs per second, offering SYN cookie mechanism to effectively and nearly effortlessly weed off bad connections!

Such devices make sure that the requesting TCP/IP connection is fully established and legitimate, before ever forwarding it downstream - to aiCache and/or origin servers. These devices typically accomplish this feat all in hardware, with extremely low overhead - something that is not typically attainable in general purpose servers.

So our message is: if you have such devices in your possession, do build a layered defense - enabling the DOS protection features at different levels of your infrastructure. We do also recommend reducing the **tcp_synack_retries** Linux kernel setting to "1" when under attack, enabling TCP/IP cookies and increasing the backlog of connections in SYNC_RECV state:

```
echo 1 > /proc/sys/net/ipv4/tcp_syncookies
echo 1 > /proc/sys/net/ipv4/tcp_synack_retries
echo 32000  > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

You can also consider running Linux firewall : iptables or ipchains, on aiCache servers, if you want some extra control over IP and TCP/IP traffic, before it ever hits aiCache servers. Explaining these products is outside of scope of this manual.

DOS attack detection.

One of the tougher things about DOS attacks is actually detecting them. aiCache helps by alerting on a whole range of crucial Web traffic statistics, in real time. So when you see that volume of requests have spiked,

for no good reason or number of bad requests or bad responses are jumping up, you will certainly know, right away, that something is not right. Again, do enable aiCache real-time traffic alerting on some of these counters.

So now that you know that something is not right, what is it ? Is it a genuine DOS or is it because a link to one of your pages is now live on one of most popular link aggregators ? Or is it because your DB is having a problem and through some unforeseen chain of events and may be less than robust JS code, that results in an endless request loops being executed on thousands of computers all over the internet, all aimed at your site ?

Unfortunately, ability to obtain the exact cause often comes only with experience, as it is more art than science. Having monitors on every critical components is a very important tool to have in this troubleshooting exercise !

Having established that you're in fact under a DOS, do deploy the countermeasures right away. Start with intelligent throttling - that should drop DOS volume very significantly. Follow up with RTATC if you still see active and well distributed bot traffic . Monitor DOS defense stats to see how many connections are being dropped and throttled , stats on ATC challenges etc.

If you can identify the offending source IPs, try having them blocked at aiCache and upstream of your setup - may be at your ISP level.

Reporting and Aggregation of aiCache statistics via HTTP PUT method.

As you already know, you can access an extremely rich set of aiCache statistics via SNMP. This the best way to go about accessing, collecting and reporting on counters of interest. But sometimes, you might deploy aiCache instance(s) in a way that precludes, by design or through no fault of your own, SNMP access to some/all of the aiCache instances you want to monitor.

In such a situation, you can configure aiCache to report its statics in a *push* fashion, as opposed to you *pulling it from aiCache* via SNMP access. To accomplish this, you need to configure the following global settings:

aggr_host - aiCache will send the complete set of statistics to this aggregation host

aggr_url - via an HTTP/1.0 POST to this aggregation URL

aggr_port - (optional, defaults to 80) on this aggregation port

Both **aggr_host** and **aggr_url** must be specified for such reporting to take effect.

When configured so, aiCache HTTP POSTs the full content of it's SNMP statistics file to the specified URL on the specified aggregation host and port, every few seconds. You can then deploy some custom logic, written in language of your choosing (PHP, Java, .Net etc) on the aggregation host, that would process the body of the aggregation POST request, to parse out and process the information that is of interest to you. Processing might include persisting, reporting, alerting, charting and other logic that is of value to you.

You can also deploy this method to aggregate statistical reporting from a number of aiCache instances, no matter where they are deployed. The only requirement is for the instances to be able to reach the aggregation host via over the specified port.

For example, let's assume you've deployed a number of aiCache servers in a geographically distributed fashion, with some hosted in the Cloud and others hosted in your Datacenters or Colo facilities. Via the aggregation reporting method, you can aggregate and report the total number of requests per second, ingress and egress bandwidth, have it broken down by region - for example, East Coast Requests vs West Coast and so on.

Automated Monitoring and Alerting.

aiCache collects and reports on an extremely rich set of statistics and instrumentation information via CLI, SNMP and Web interfaces. You're advised to use these facilities as part of regular monitoring and troubleshooting arsenal.

For example, SNMP is especially well suited for charting and reporting functions, including historical (trend) analysis. Some SNMP packages also support evolved alerting functionality, alerting when SNMP OIDs go out of configured ranges.

However, to simplify matters aiCache also provides a built-in automated alerting capability. A number of most critical parameters (counters) can be setup for monitoring and when aiCache detects that the values are out of configured bounds (so called out-of-spec condition), alert emails are generated and sent to the configured email addresses.

The alerting is done in a fashion that prevents *alert email floods*, a common mishap that is known to overflow many a mailbox and to cause apathy, loss of attention and sleep deprivation of IT staff . No more than one alert is generated per minute, no matter how many error conditions are detected. When multiple error conditions are detected, these are aggregated in alerts. No "back-to-normal" alerts are generated, such "OK" condition is indicated by lack of new alerts. So at 2am you don't have to try to read pager/phone screen to see if it is yet another down alert or back-to-normal alert - the silence and few extra minutes of sleep are what is most welcome in ungodly hours like these.

Alerts can be *global*, *website or pattern-specific*. Each type can have its own alert email address defined. The *global* alerts contain global out-of-spec conditions and include (are a superset of) all of website alerts that were generated at the same time.

Let's explain this with an example. Assuming aiCache accelerates *news.acme.com*, *video.acme.com* and *boards.acme.com*, ***global*** alerts (when enabled) are sent out whenever a problem is detected with *any* of these sites . Each of the defined (accelerated) websites can have their own alert email defined that is used to alert when that *specific* website is in out-of-spec condition. So you can alert Helpdesk when *any* of the sites are affected, yet problems with *boards.acme.com* are reported to *boards* team and so on. This way you can make sure you don't disturb wrong teams or, in shared hosting setup, don't share uptime information with other customers.

aiCache relies on its host server's ability to deliver outbound emails so make sure you configure your email delivery system properly. In most cases it means you need to setup/configure mail relay after making sure your mail daemon (sendmail, postfix etc) is installed and operational. aiCache sends email via 2 stage process.

First, whenever an alert condition is detected, as alert file is generated and spooled out to **alert_dir** location (defaults to **/usr/local/aicache/alerts**).

Next, an external monitoring program - **alert.pl**, supplied with aiCache, that periodically looks at the content of the **alert_dir** (every 10 seconds), detects that a new alert file is now available in the **alert_dir**

directory, processes it and sends an email out, via *sendmail* Linux binary. After email is sent out, the alert file is removed.

Please note that for email to be sent out, in addition to configuring your sendmail (mail relay), you also must make sure that alert.pl is continuously running. You can provide for that via starting up the alert.pl during system boot (/etc/rc2.d script or similar). Good old **nohup /usr/local/aicache/alerp.pl &** might also work in a pinch, but it won't survive system reboot.

alert.pl generates diagnostic output as it runs - it is up to you to spool it to a file if you so desire, or to ignore it by redirecting the output to **/dev/null**. Feel free to tailor/modify the script to suit your needs, you can modify it to alert (page, turn on a siren or strobe lights etc), run an arbitrary action of your choosing and so on.

Test your email setup by logging to aiCache host server and generating a simple test email message to a known good address and make sure email gets to its destination. When you want to deliver to a list of email addresses you can use distribution lists or aliases - for example, you can edit the **/etc/aliases** file and run **newaliases** afterwards, to have locally managed aliases.

To enable global or website specific alerts you need to provide **alert_email** setting and at least one alert *condition* setting in global and/or website section of the configuration file. The *global* condition settings are a superset of *website* settings as they include some alert conditions that are not available for use in *website* alerting.

Please also note that for alerting to work you also need to make sure statistics logging is enabled (**logstat** must be set).

Here's the list of *global* conditions that you can alert on, along with example values:

- **alert_bad_req_sec 20**
- **alert_max_cache_entries 10000**
- **alert_client_conn_max 30000**
- **alert_client_conn_min 5**
- **alert_os_conn_max 20**
- **alert_bad_resp_sec 2**
- **alert_req_sec_max 2500**
- **alert_req_sec_min 10**
- **alert_os_rt 200**
- **alert_ip_blocked_sec**

- **alert_ip_throttled_sec**

Here's the list of *website* conditions that you can alert on, along with example values:

- **alert_max_cache_entries 10000**
- **alert_client_conn_max 30000**
- **alert_client_conn_min 5**
- **alert_os_conn_max 20**
- **alert_bad_resp_sec 2**
- **alert_max_bad_os 2**
- **alert_req_sec_max 2500**
- **alert_req_sec_min 10**
- **alert_os_rt 200**

Here's the list of *pattern* conditions that you can alert on, along with example values:

- **alert_req_sec_max 2500**
- **alert_req_sec_min 10**
- **alert_os_rt 200**

In addition to these conditions, *global* and *website* alerts are generated whenever a disabled origin server is detected. As you recall, origin servers are disabled by aiCache when they fail to pass a health check. If no health checking is configured for a website, this condition is not tested for and not reported, so make sure you setup/enable health check monitoring if you want to be alerted on state of origin servers.

The names of condition settings are self-explanatory, but here are some quick suggestions:

- Monitor origin server response times by configuring **alert alert_os_rt**, this way if they ever slow down, you are alerted.
- Monitoring number of cached responses by configuring **alert_max_cache_entries** to alert when the number of cached responses grows past set limit. This might indicate you needing to enable parameter busting or ignoring query string in its entirety for some URLs.

- Setup monitors for origin server failures by configuring **alert_bad_resp_sec**, abnormally high number typically indicate serious problems with origin servers, App or DB server and other backend infrastructure . Setup **alert_os_conn_max** to monitor for growing number of origin server connections, this typically indicates a similar problem with origin infrastructure being unable to satisfy requests in timely fashion.

- Monitor number of RPS by configuring **alert alert_req_sec_max** and **alert_req_sec_min.** Abnormally low and high numbers might indicate a problem or an attention-worthy condition (i.e where did all the users go ? do we have problems with uplink connectivity ? a story going viral !?).

- Monitor number of bad requests per sec by configuring **alert_bad_req_sec** . High numbers might mean you're under DOS attack and are being bombarded with malformed requests and/or bogus connections .

- Monitor number of origin servers that failed origin server checks via **alert_max_bad_os**. By default, a single bad origin server causes an alert to fire (in other words, **alert_max_bad_os** defaults to 0**)**.

- Likewise and for the same reason, pay attention to number of open client connections. Larger than usual number of these might also indicate a traffic spike or a website's code malfunction (server side or client side).

- You can also choose to be alerted when a DOS condition is detected: exceeding number of blocked or throttled requests or connections. See dedicated DOS chapter for more information on this feature.

It is rather hard to come up with reasonable defaults that work for any site, but you shouldn't have problems deciding what should constitute an out-of-spec condition for *your* particular setup.

You might want to disable website alerting before performing maintenance on the origin servers and similar type of activities that might cause alerts to fire. To accomplish this you can use CLI **alert** command - turning the alerting off before performing the maintenance and restoring after the maintenance is over.

Look Ma, I can now sleep at night: as a special gift to overworked and underpaid IT crowd, you can disable alerting between hours of midnight and 7am server *local time* by specifying **alert_humane** directive in global and/or website sections. This also might make sense if the traffic to your web sites drops to really low numbers during off-hours as it might cause some monitors to alert on out-of-specs conditions : low RPS, low client connections etc.

Alert suppression via alert_exclude_pat.

You can opt to suppress email alerts via alert exclusion patterns. These are regular expression patterns that could be specified at global and/or website level (latter taking precedence, as usual). More than one could be specified. When any of these are specified, the current local date/time stamp is compared to the patterns. If a match is found, no email alert is sent.

Here's an example date/time stamp format: "**Tue Jan 1 08:54:08 2001"**. Please note that there could be two spaces between the name of the month and the day of the month. You can get quite creative with the exclusion patterns – specifying very granular or very loose matching patterns. As always, full power of regular expressions is at your fingertips. A simple pattern of "Sat|Mon" would suppress alert emails on Sat and Sun.

```
server
…
alert_exclude_pat Sat
alert_exclude_pat Sun
…
website
…
alert_exclude_pat \s0\d:  # date/time stamp matches 00:xx:xx to 09:xx:xx
```

When an alert email is suppressed, an error log entry is added to that effect.

Expiration Email alerts.

aiCache can be configured to automatically email messages about content expiration. Such content expiration can be caused by CLI or peer *expire* commands. To automatically email the expiration notifications, setup **adm_email** in global section of the configuration file, for example:

```
admin_email webteam@acmesupport.com
```

aiCache aggregates the messages so that the emails are only sent once every minute and only when there were actual expiration commands issued/received in the said interval.

Just as with regular email alerts, aiCache writes out a file called **NNN.admin.alert** in **alert_dir** (defaults to **/usr/local/aicache/alerts),** where NNN is a time stamp in seconds since epoch. The expiration messages are aggregated across all of the configured websites into a single message.

An external program shall then take care of periodically combing through the directory and emailing the files. The file format is in the following form:

```
EMAIL_ADDRESS
EMAIL SUBJECT
EMAIL_BODY ….
…..
….
```

A simple Perl email script, called **alert.pl**, is provided within aiCache distribution and could be used to accomplish this function. Feel free to modify the file to suit your needs and/or create your own email driver based on alert.pl. Make sure the script is always running so that emails go out as soon as they are created.

And lastly, make sure to configure the server so that emails can go out. Normally all that is needed is to specify SMTP smart host.

Using aiCache for 0-overhead error reporting, logging and alerting function.

You might have a need for a method to allow for reporting of assorted error conditions. For example, your web site might offer videos, breaking news and other APIs that are invoked by AJAX or some other form of client-side Javascript. When these APIs fail to respond, you would like to have some way to report the error condition so that it can be recognized, logged and acted upon.

Normally, accomplishing this requires writing custom server-side code that would receive the error notification and execute some form of reporting, logging and alerting function. However, outside of having to write such code, you might face another danger when following down this path: when in error state, your site will be receiving large number of error message and that in itself, can cause further deterioration of service which would result in even more of error conditions being reported. In just few quick seconds you whole site just might melt down.

aiCache offers an easy way to accomplish the required functionality, without requiring any custom code to be written or any code to be executed by your web, application or database servers. It is aiCache that receives, logs and alerts on error condition, without ever forwarding any of the traffic to the rest of your infrastructure.

Reporting using a dedicated error reporting website.

To accomplish this, setup a dummy website in aiCache, for example **error.acmenews.com** . Identify the site as dummy website via **dummy_website** website-level configuration directive. You don't need to specify any origin server for websites defined as dummy, as no requests ever are sent to origin servers for such websites.

Setup the site to alert on when number of requests exceed the desired threshold - for example set **alert_req_sec_max** to alert when this website receives more than 2 requests per second.

Setup the client side code to execute a simple HTTP GET against this website, passing arbitrary URL and parameters, as part of the request. For example, to report a problem with your News API, you might request **http://error.acme.com/app=newsapi&errorcode=123&client=premier .** It is up to you just what parameters you want to pass via such a request.

Now, when an error condition is detected, a request is made to the specified URL. aiCache receives the request and logs it. When number of such requests exceed the configurable threshold, an alert is sent out. So there you have it: without writing any of the server side logic and without placing any extra load on your infrastructure, you now can receive, log and alert on error conditions - all courtesy of aiCache.

In the way of responding to requests to such dummy websites, aiCache can either send a redirect - to a location you can specify via optional paratemer **to dummy_website** directive, or if one is not specified, with a an empty "200" response.

```
hostname dummy.acme.com
dummy_website http://www.acme.com/we_are_sorry.html
```

It doesn't have to be only the client-side logic that sends such error-reporting requests. You can use method of your choosing to execute such requests - including server-side code, custom script etc. You can also redirect clients to the error-reporting URL in case of a problem that you discover when executing some logic. Again, you're only limited by your imagination in how this functionality can be deployed.

You can train your support personnel to look at the aiCache log file for such error reporting web sites, when the error condition is reported and alerted by aiCache, to see the additional information that is embedded inside of the error reporting URLs (such as application, error code and client information in the example above).

To assist in rapidly detecting the error condition, aiCache lights up the respective web site name in red, in its self-refreshing web monitor screen, when it detect that RPS for a dummy website exceeds the set limit. This is in addition to regular email alerting on the error condition - which should be configured as per instructions in Automated Alerting and Monitoring section of this manual. Specifically, you must provide the email address to send the alert to, at either or both **global** and **website** levels.

Error reporting using *bad_response* pattern flag.

You can set a **bad_response** pattern flag. When aiCache matches a request to such pattern, the response is reported as bad response (same as say 5xx response code) - so you can monitor for such bad responses, log and alert on them.

Here's an example explaining how you can use this feature. Some sites have dedicated pages setup for common errors - such as PNF (page-not-found) and similar. So let's imagine there is such a dedicated error page called 404error.html . The page is requested/redirected to in response to assorted error conditions, as detected by assorted server and client side code.

By setting **bad_response** flag for matching pattern, you will configure aiCache to report and optionally, alert on, whenever this page is requested, providing for 0-cost monitoring and alerting of this error condition.

Please note that setting of **bad_response** flag doesn't affect anything else in terms of request/response handling logic, its sole purpose is for reporting and optional alerting of matching traffic. In other words, aiCache will serve the responses to such requests in a regular way, contacting origin servers when necessary and caching responses when appropriate.

Error reporting using pattern-level alerting.

And yet another way to setup alerting - this time, via pattern-level alerting. You can defined a number of patterns for a website. For each of the patterns, you can set **alert_max_req_sec** to a desired value. Also, set **send_200_resp** flag for matching pattern.

Now, as matching requests come in, aiCache will serve empty 200 responses in response to such requests. Origin servers are never contacted to obtain responses. aiCache also logs and calculates RPS for each pattern. When the calculated RPS exceeds **alert_max_req_sec** , aiCache will alert as usual.

Here's an example use. Setup **errors.acme.com** website, specify a dummy origin server (no requests are ever sent to it, we specify an OS just to satisfy aiCache configuration sanity checks).

Setup a number of patterns like so:

```
hostname errors.acme.com
alert_email operations@acme.com
....
....

pattern newserror simple 0
send_200_resp
alert_max_req_sec 5

pattern videoerror simple 0
send_200_resp
alert_max_req_sec 10
```

Now, if there're more than 5 RPS to **errors.acme.com/newserror** URL, aiCache will alert. Likewise, if there're more than 10 RPS to **errors.acme.com/videoerror** URL, aiCache will alert. This way your Ops team know that average RPS to these URLs exceed configured threshold and there's some kind of widespread issue that you need to react to.

Arbitrary information can be added to the URL, for example:

errors.acme.com/newserror?error=api_timeout

errors.acme.com/videoerror?errorcode=stream_timeout

aiCache will log the complete request URL in the log file, so you can examine the log entries to better identify what is happening on your site.

These error-reporting URLs can be coded into your client-side JS code or Flash video players etc, to be called whenever client-side code detects an error. This way your client side can report issues back to base, so to speak. By tuning **alert_max_req_sec** setting, you can configure aiCache to filter spurious errors, while alerting on more widespread issues.

Advanced performance tuning.

aiCache is a finely tuned HTTP engine. However, for those that want to squeeze every possibly ounce of performance out of their installations, we offer some advanced settings that we explain in this section.

Deferred TCP Accept Option.

By default, aiCache is notified about a new connection as soon as first TCP/IP packet, the so called SYN, is received. However, the actual data won't start flowing till two more TCP/IP packets are exchanged (this is how TCP/IP works) - meaning that aiCache has to wait a bit before it can obtain the request and send a response back.

You can change this behavior and instruct the operating system to notify aiCache only after a connection is fully established and actual data is available. To configure such behavior, specify **defer_accept** global option. It takes a single numeric parameter: the amount of time, in seconds, that operating system will wait for TCP/IP connection to get fully established, after receiving first packet (aka SYN). 2 or 3 seconds is a safe setting to use:

```
defer_accept 3
```

By specifying this setting, you're asking the Operating System to handle more of the overhead of the connection establishment, slightly reducing the load on aiCache.

Administration

Starting up and shutting down.

Before starting up.

As with any network daemon, in case of a busy web site, you might want to increase the number of allowed file descriptors via **ulimit**:

```
ulimit -n 64000
```

We've chosen limit of 64000 file descriptors, you can choose a different number - but on a really busy site you'd want to be in the 64000+ area. Should aiCache reach the limit on number of file descriptors, it will write diagnostic message to error log file and exit. To see what the limit of file descriptors is set to, you can execute

```
ulimit -n
```

You can also use "**num_files NNNNN**" configuration directive in the global section to tell aiCache to raise it's open file limit.

Due to IP protocol limitation on port# (16 bit port#), one cannot have more than 64K open connections against a single IP address. aiCache is certainly capable of handling much more than that - but you need to make sure a number of IP *aliases* are configured on the system and requests are fanned out across all of them. Exact mechanism to achieve the fan out varies, but it easiest to accomplish if you have a load balancer or via multiple DNS A records etc.

Another limit on max number of open connections might come from a *system-wide* limit. To see what it is set to and change it, read/write from **/proc/sys/fs/file-max** file:

```
# cat /proc/sys/fs/file-max
128000
# echo 512000 > /proc/sys/fs/file-max
# cat /proc/sys/fs/file-max
512000
```

To assist in unlikely event of program crash, we recommend you enable core dump collection via:

```
ulimit -c 1000000000
```

This allows generation of core dump up to ~1GB in size , should the aiCache ever crash. The core file can then be analyzed to determine and fix the root cause. If you need to send a core file to aiCache for analysis, don't despair - it might look fairly large on a file system, but it compresses almost 10-fold !

You can place these 2 commands into a custom shell startup script or add them to your profile.

Some sites choose to modify some of TCP/IP parameters - including send/receive buffer sizes and various timeout settings. You can certainly do so at your discretion, such changes are transparent to aiCache. Here's an example set of assorted network-level Linux kernel settings along with possible values. Chances are you won't need to change of the stock kernel settings, unless you're running an extremely busy website.

```
net.core.rmem_max=16777216
net.core.wmem_max=16777216
net.ipv4.tcp_rmem=4096 87380 16777216
net.ipv4.tcp_wmem=4096 65536 16777216

net.ipv4.tcp_fin_timeout = 3
net.ipv4.tcp_no_metrics_save=1
net.core.somaxconn = 262144

net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_max_orphans = 262144
net.ipv4.tcp_max_syn_backlog = 262144
net.ipv4.tcp_synack_retries = 2

net.core.netdev_max_backlog = 30000
net.ipv4.ip_local_port_range = 1024 65536
```

Starting up

To start up the aiCache server you need to execute the aiCache binary while providing at least one *required parameter* to the binary (parameter values are for example only, yours might be different): location of the configuration file (required): **-f site.cfg .** For example:

```
aicache  -f site.cfg
```

When running "regular", non-cloud instance of aiCache, you will require a license file and must specify its location like so: **-l 34134132.lic .** For example:

```
aicache  -f site.cfg -l 34134132.lic
```

Full list of parameters include:

```
-f <config_file_name>  filename of the config-file,  always required
 -l <license_file_name>  filename of the license file, required for non-Cloud
instances, when -c is not specified
-c <cloud_name>  cloud name, use "aws" for Amazon.
 -t         test-parse and print the configuration file,  exit.
```

```
  -D          stay in foreground, don't detach, preserve STDOUT and STDERR
  -v          show program version and exit
  -h          show this help screen and exit
  -s          (strict) generate strict warnings for AUTH headers in cacheable responses
```

When running aiCache.com's own Amazon AWS aiCache instance or other cloud-enabled instances, license file is not required. The aiCache binary is pre-enabled for a number of Cloud Providers and when started in supported clouds, requires no license - but you must specify **-c <cloud_name>** option. As with most cloud images, you will be billed per unit of time and consumed bandwidth, of actual usage of aiCache. As rates are subject to change, make sure to check the applicable rates, terms and conditions with your Cloud Provider.

"-t" option is there to assist you in unlikely case of an error in the configuration file. When "-t" is specified, aiCache parses provided configuration file line by line, printing it out as it goes along. It also prints out its "interpretation" of what it sees in each line in the configuration file. Once again, configuration file is case sensitive, some parameters are written using underscores to separate parts of the composite parameter names and some composite parameters do not use underscores.

If you specify what you think is a valid option yet aiCache doesn't seem to understand it, use the "-t" option and see what aiCache outputs when it comes across that parameter in the configuration file.

The easiest way to create your own config file is to copy and modify an example configuration file, provided with distribution. So let's assume you copy **example.cfg** to **acme.cfg .** We can now modify the **acme.cfg** file to configure your particular instance.

By default aiCache accepts inbound connections over any of its IP addresses (if more than one is defined), on standard HTTP port 80 - so unless your setup is different, you don't have to change that.

You must start aiCache as **root** (superuser). Shortly after startup, before any incoming connections are accepted, aiCache switches to identity of **username** and **groupname** - as configured in the configuration file. Common examples include running as a dedicated web user - such as (recommended) **aicache/aicache**.

You can use any valid **username** and **groupname** that are defined on the host server where aiCache is installed. Check to see if you server already has a suitable user and group configured. aiCache installation script creates user name and group name of **aicache** for you.

Relinquishing root privileges and lowering access level to that of a dedicated user/group is a common paradigm used in Unix/Linux world, to limit possible damage should running program binary every be breached.

As aiCache starts up, some messages are printed out to the console. Unless "-D" option is specified, aiCache disconnects from controlling console after the start up (becomes a Linux "daemon"), so you can safely logout without causing aiCache to exit. All error messages after that point will be logged to error log file as defined in the configuration file.

During startup aiCache performs extensive check of most configuration parameters, any problems are reported immediately to the console (STDOUT**)** and aiCache aborts**.** It will not start until you fix all and any configuration problems that it complains about.

Access and error log files are also opened by aiCache during the startup phase. These files are opened in “append” mode and are not overwritten during the startup. When multiple websites are accelerated, you can

configure aiCache to log requests to a *combined* access log file or *dedicated* access log files, one per accelerated web site - see Logging chapter for more information.

If global-level setting **pid_file** is specified, aiCache will write out its PID (Unix process ID) to the specified file . Make sure aiCache user can write to that file. For example:

```
pid_file /var/run/acme.aicache.pid
```

Shutting down.

Preferred way to shutdown running instance of aiCache is to send a SIGHUP to **aicache** process:

```
pkill -HUP aicache
```

or issue CLI **shutdown** command. These are the only ways that guarantee *graceful* shutdown of the aiCache. While no significant harm would be done by *forcefully* killing running aiCache instance, you might inconvenience those clients that are receiving responses at the moment when you run kill command. Executing graceful shutdown avoids that problem: aiCache stops accepting incoming connections, but the ones that are established at the moment of the graceful shutdown are allowed to complete for the duration of configurable grace interval (defaults to 5 second).

Assuming you use some kind of load-balancing mechanism and a highly available setup, the fact that this aiCache instance doesn't accept incoming connections is detected and user requests are redirected to another instance of aiCache, so there's no user impact whatsoever during such graceful aiCache shutdown.

You may also consider configuring clustered VRRP-enhanced aiCache setup, a true hallmark of high availability. Please refer to dedicated chapter on particulars of such setup.

Stopping aiCache also stops all of the built-in auxiliary aiCache services: CLI, SNMP and Web Monitor. No additional actions are required to stop aiCache. aiCache logs shutdown requests in the error log file.

License files.

Unless you intend to run aiCache as a pre-configured and pre-approved Cloud image or in a demo mode, you will need to obtain, from aiCache or one of its resellers, a valid, production aiCache license file. Production license files are issued in two different modes:

- Tying the license file to a valid network interface on a particular server. Such license files have a **.lic** extension.
- Tying the license file to a particular list if allowed domains. Such license files have a **.dom** extension.

The former, interface-bound licenses, can only be used on a ***particular*** host, but allow accelerating a number of different domains. No enforcement is made as to what domains or ***cnames*** could be defined in such mode. You would not be able to transfer the license file to a different host and will require contacting aiCache to obtain a new license in such scenario. In other word, such license ties you to a server, but doesn't tie you to any particular set of websites and you can choose and change the said set at any point.

The latter, website-bound licenses, can only be used on any server(host), subject to agreement between you, as a customer and aiCache, but each defined website and/or ***cname*** must match allowed list of websites, as specified in the license file. You can move the license file to a server of your choosing, including one in a private or unsupported Cloud, etc. In other word, such license ties you to a particular set of websites – as defined in the license, but doesn't tie you to any particular server – as long you're not violating aiCache licensing terms.

When running under such license, aiCache disallows use of **wildcard** matching of hostnames and use of **ignore_hostname** setting.

Making Changes to Configuration Files.

In order to make changes to configuration files, simply edit related configuration file(s) with editor of your choosing. You can apply the new configuration *on-the-fly, without any interruption of service* by issuing CLI "reload" command or by creating a reload watch file.

```
> reload
```

Alternatively you can stop and restart the aiCache. Please note that there will be momentary (a few seconds) interruption of service if you choose to stop and restart aiCache.

If you decide to apply new configuration on-the-fly, without stopping and restarting aiCache, you need to be aware of the following:

- The procedure doesn't affect any connected or connecting users. There's no service interruption of any sort.

- You cannot change aiCache HTTP, CLI server port or IP address on-the-fly, these changes require stop/start sequence.
- aiCache runs exhaustive checks on the new configuration and it will not take effect unless it passes all checks. Monitor aiCache log file to see what's happening. If any errors are discovered in the new configuration file, the previous configuration stays in effect.
- The new configuration file takes effect upon receiving of the first inbound HTTP request after the change was requested.
- You must modify the original configuration file if you want to apply the configuration changes on-the-fly. Do consider making a backup copy of the configuration file before applying any changes.
- Applying new configuration file resets most of aiCache statistics. You can use this technique to your advantage when you want to reset the counters - simply reload the original configuration file

Forcing configuration reload via watch file.

You can tell aiCache to reload it's configuration files by creating a reload watch file. For this to work, you need to specify **reload_watch_file** global level setting. For example:

```
reload_watch_file /usr/local/aicache/reload
```

Now, after modifying configuration file, you tell aiCache to load the new configuration by creating the said file (there needs to be no content). For example:

```
$ touch /usr/local/aicache/reload
```

Observe aiCache error log file. A second or two after you created the watch file, aiCache will attempt to load the new configuration - the attempt will be logged in the error log file.

Collecting reload output messages.

You can tell aiCache to write out the results of configuration reload attempt, pass or fail, into a file, by specifying **reload_success_file** and **reload_fail_file** global level settings. You may want to email the reload results to operations team or copy the files into a directory.

For example:

```
reload_success_file /var/run/aicache_reload_success
reload_fail_file /var/run/aicache_reload_fail
```

Executing post-reload actions (deprecated).

You can tell aiCache to execute arbitrary scripts upon successful or failed configuration reload, by specifying **reload_run_cmd** and **reload_fail_run_cmd** global level settings. For example, you may want to email the reload results to operations team or tar up the new configuration files for distribution to other aiCache servers (please see elsewhere in this Guide for more on running of distributed/clustered aiCache setups). Here's an example:

```
reload_run_cmd /usr/local/aicache/aiacache_reload_success.sh
reload_fail_run_cmd /usr/local/aicache/aiacache_reload_failed.sh
```

Please note this feature is deprecated. We recommend using **reload_success_file** and **reload_fail_file** global level settings instead combined with an external watchdog script.

Recommended configuration modification procedure.

If you intend to stop and start aiCache with the new/modified configuration file, we recommend to check the new (modified) config file for correctness before stopping current aicache process (one that still is using the previous version of now changed config file) and trying to start aicache with the new config file. If any problems are discovered, aiCache will refuse to start and you might find yourself scrambling to figure out what to do. Here's how you check the new config file for correctness in safe way:

You don't have to *restart* aiCache to test (freshly modified) config file for syntax correctness. Even when an instance of aiCache is currently running on this system, you can run *another* aiCache command - complete with license and configuration file parameters, so see if config file passes the syntax check. *After* parsing the config file, such "second" instance of aiCache detects that another one is already running and harmlessly exits - but you will know at that point that config file is good. Now that is you know the new configuration is good, you can stop currently running instance and restart it using the new configuration file.

If high-availability is a must for your Web Site, you should resort to applying configuration changes on-the-fly or restart aiCache only during off hours or allowed maintenance window. If you have a number of aiCache servers with Web traffic shared amongst them, you might be able to restart aiCache at will, one at a time, without causing any interruption of service . Again, consider using *graceful shutdown* to limit impact on your users.

We strongly recommend making backup copies of configuration files before editing them. For very complicated Web setups use of a Version Control System, such as Subversion, CVS etc, is highly recommended. This allows you to keep track of config file changes, along with ability to fallback to configuration files that are known to work. At the very least use the ages-old tactic of copying the known good configuration file to a *.bak* file, before editing it.

If you desire to break up your aiCache configuration into a number of smaller configuration files, instead of having one large configuration file, you can do so via one or more of **include file_name** directives. It can be placed anywhere within configuration file. Upon encountering **include** directive, aiCache opens the file that it references and continues parsing process on the content of that file. The included file can, in turn, provide a number of **include** directives of its own.

We suggest you use **include** directive to break up configuration so that each accelerated file is stored in its own configuration file, or perhaps, store patterns in their own sections etc. aiCache allows pretty deep nesting of **include** directive, but it does enforce a sanity limit of 10 on just how many levels deep can include files be nested.

Health, Statistics and Performance monitoring of aiCache.

Basics of health monitoring

You already know about CLI, SNMP and Web based ways to monitor aiCache. SNMP and Web provide read-only access to a number of crucial performance and statistics-related parameters of aiCache.

CLI provides for additional ways to peek even deeper inside of the aiCache, as described later in CLI subsection of Administration section.

We have previously discussed Health Monitoring of origin servers - periodic attempts to retrieve predefined URL from origin web servers and do content matching on responses. aiCache logs any errors discovered during such health checks to the error log file, so you might want to monitor those. Origin server failures to pass health checks are also reported via SNMP, Web and CLI interfaces.

If you use custom Web monitoring agents, you can set them up to periodically retrieve some URLs from aiCache, with a content match if possible. Should such retrieval fail, you will know there's a problem.

You may also configure your SNMP monitoring software to watch some of aiCache's vital counters and alert when those exceed some thresholds or no responses are coming back.

And lastly you can monitor the "usual suspects" - CPU, memory and disk utilization on aiCache hosts, using monitoring setup/software of your choosing.

Here're some sure tell-tale signs of more common problems:

- increasing counts of pending client and origin server connections (available via SNMP, CLI and Web page). This normally happens when origin servers have difficulties responding to user requests
- another indication of the same problem - increasing average origin server response times (available via SNMP, CLI and Web page).
- sudden decrease or increase in number of requests per unit of time. (RPS available via SNMP, CLI and Web page).

Performance and Statistics Information

Here we list some of performance and statistics counters as provided by aiCache, along with explanation on how you might use it. Many more counters are available - see the SNMP statistics file, Web statistic page or output of "statistics" CLI command.

Information	**What does it mean and why is it important.**	**Available via**
Overall Cache Hit Ratio, accumulated (OCR)	Percentage of requests served from the aiCache's cache to total number of **all** requests, since latest startup. This number depends on how much cacheable content a given web site has.	SNMP, Web, CLI
Specific Cache Hit Ratio	Percentage of requests served from the aiCache's cache to total number of **cacheable** requests. Well configured aiCache will have SCHR in 95%+ range, depending on site's specific.	SNMP, Web, CLI
Total number of document in cache (# of cached objects).	Shows how many documents are in aiCache's cache. Typically this number should stabilize at some level, depending on particular site.	SNMP, Web, CLI
Total number of responses.	Total number of responses served since startup.	SNMP, Web, CLI
Total number of cache hits.	Total number of requests satisfied from cache.	SNMP, Web, CLI
Total number of cache misses.	Total number of responses that had to satisfied from origin server(s), as Web documents were not available in cache (either not available at all or available in fresh form).	SNMP, Web, CLI
Number of outstanding (open) client connections.	Indicates how many client connections are currently registered with aiCache. This number depends on how many people are browsing the site at the moment	SNMP, Web, CLI
Number of total active (open) client connections.	Indicates how many requests are currently being served (sending request or receiving response).	SNMP, Web, CLI
Number of total active (open) origin server connections.	Indicates how many requests are currently being obtained from origin servers. This number normally would stabilize around a certain level.	SNMP, Web, CLI
Requests per second.	Most commonly used indication of traffic against aiCache. Every few seconds, it is recalculated and updated automatically by aiCache. Depending on the	SNMP, Web, CLI

	Web Site, low numbers during typically active hours indicate some problem with the site (where did the users go?)	
Detailed information about cached objects.	For each cached object (Web document), displays it's age in the cache, size, return code, number of times this document was served and whether this URL is logged. Also shows total size of cached objects in memory.	CLI

You can configure aiCache to collect and report pattern-level statistics by specifying **collect_pattern_stat** at website level. The collected pattern stats can be accessed via Web and SNMP interfaces. Here's an example of the data collected.

```
Pattern: >\.gif<, TTL: 10 sec, OST: 0
    #Req: 118, #CHits: 90, #CMiss: 28, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
Pattern: >\.jpg<, TTL: 10 sec, OST: 0
    #Req: 79, #CHits: 48, #CMiss: 31, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
Pattern: >\.png<, TTL: 10 sec, OST: 0
    #Req: 8, #CHits: 5, #CMiss: 3, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
Pattern: >\.ad<, TTL: 10 sec, OST: 0
    #Req: 0, #CHits: 0, #CMiss: 0, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
Pattern: >\.js<, TTL: 10 sec, OST: 0
```

Operating Websites in Fallback Mode.

Sometimes origin server infrastructure might be in a state when it is unable to serve any requests. A number of reasons could lead to such state - DB or file store failures, deployment of faulty code are some examples. Normally, without aiCache front-ending the requests, the website would be in a hard down situation - not serving anything at all or serving all kinds of errors to the end users. [18] And even that is not the worst that can happen: frequently meltdowns of one website might lead to downtimes for all others that share networking HW (as in an entire Datacenter) with the culprit - taking firewalls and/or load balancers down is one such way of spreading the misery around.

[18] And it is expected that some users will simply give up on such website, find a different one serving their needs and never come back.

With aiCache you can save the day by going into "forced fallback" mode. When operating in such mode, aiCache behaves as follows:

- requests for cacheable content are instantly satisfied by (possibly stale) cached responses whenever possible, w/o attempting refresh against origin servers
- no requests are allowed to go to origin servers
- non-cacheable requests and requests for cacheable content that doesn't exist in the cache are served instant 503 responses by aiCache, again, no attempt is made to go to origin servers. Remember that aiCache allows you to redirect to a friendlier page in case of 500+ errors.
- health checks, if any are configured, continue to be generated and sent to origin servers
- any cached content cleaning out is suspended, for as long as website is in fallback mode

As you can see the basic idea behind fallback state is to make sure visitors receive some content back, even if it is stale, and to shield the origin infrastructure from any user traffic so that you could restore the service faster.

To put an accelerated website into fallback mode or to restore normal operation is to use CLI "**fallback**" command :

```
fallback hostname on|off
```

For example:

```
fallback www.acmenews.com on
fallback www.acmenews.com off
```

Please note that when aiCache detects a complete failure of website's origin servers (such as when all of them fail health checks and/or are administratively disabled), it starts behaving the same way as if the site was in fallback mode: stale responses are served if possible, no traffic is sent to origin server, but health checks are still generated and sent to origin servers.

When an accelerated website is in fallback mode, it is indicated as such via a warning in Web monitoring page (an (F) is displayed to the right of website name), output of CLI (**config** and **statistics** commands) and an SNMP OID (please see **snmp.stat** file for exact OID).

Every 100th client connection (request) that is served 503 response due to request's website being in fallback mode, is logged in error log file with a message like:

```
[Sun Jan 18 08:59:21 2008 [notice] GetResponseThread:  www.acmenews.com  in fallback
state, 503 served.
```

The logging of this error message is reduced by 99% in order not to overwhelm the error log file with these messages.

Please note that *website* forced fallback mode is different from *pattern-specific* fallback configuration setting (see Patterns section of this documents).

The differences are as follows:

- when in forced fallback mode, aiCache does not try to reach origin servers, so either a (possibly stale) cached response or an error response is served right away, as opposed to, in case of pattern-specific fallback, optionally (re)trying a number of requests against origin servers before falling back to old cached response. Such re-tries can potentially take a significant amount of time and cause heavy queuing up of client connections on load balancers, firewalls and aiCache servers. When such queuing up occurs all bets are off and consequences might be most dire. [19]

- when in forced fallback mode no user requests are sent to origin servers until fallback mode if turned off. This helps with service restoration, as constant flood of user requests to origin servers might complicate recovery efforts.

Soft Fallback Mode.

Related to *forced fallback mode* (see above), this mode can be *automatically* enforced by aiCache. All you need to do to enable it is to specify a website-level setting **soft_fallback_eps** . The setting takes a single numeric value – maximum allowed website errors per second. Should aiCache detect that website's Origin Servers are generating more than said number of errors per second, the website is automatically put into soft fallback mode and a message is logged in the error log file. Such analysis takes place every 5 seconds.

Just like with forced fallback mode, when placed into soft fallback mode, aiCache does not try to reach origin servers, so either a (possibly stale) cached response or an error response is served right away, as opposed to, in case of pattern-specific fallback, optionally (re)trying a number of requests against origin servers before falling back to old cached response.

Unlike forced fallback mode , aiCache automatically resets the soft fallback mode every 120 seconds. You can change the duration via website-level setting of **soft_fallback_duration**, by specifying a single numeric value, in seconds.

When an accelerated website is in soft fallback mode, it is indicated as such via a warning in Web monitoring page (an (S) is displayed to the right of website name), output of CLI (**config** and **statistics** commands) and an SNMP OID (please see **snmp.stat** file for exact OID).

Every 100th client connection (request) that is served 503 response due to request's website being in fallback mode, is logged in error log file with a message like:

```
    [Sun Jan 18 08:59:21 2008 [notice] GetResponseThread:  www.acmenews.com  in fallback
state, 503 served.
```

The logging of this error message is reduced by 99% in order not to overwhelm the error log file with these messages.

---

[19] Total infrastructure meltdown is the most likely outcome - frequently taking down all and any websites and services (inbound and outbound) served by the same firewall and network/load balancing infrastructure.

The Command Line interface

Introduction.

You will find CLI to be the most powerful way to monitor, fine-tune and control aiCache. It allows for additional functionality that is not available via either SNMP or Web interface.

In order to use CLI you may need first to configure it as described in the "Configuration" section: provide IP address to listen on (defaults to ANY), port number (defaults to 2233) and password **(**default password is "secret"**)**. If defaults are reasonable, no changes are required. Most network setups provide natural protection for the access to CLI, where users on the Internet would not be able to connect to it, such connection are only allowed from within a trusted network (such as DMZ or other non-public networks). If such protection is not in place by default, you may configure CLI to listen only on localhost interface: 127.0.0.1. In this case one can access CLI only from aiCache host server itself.

CLI interface is started automatically as soon as you start aiCache.

Most commonly used, harmless CLI commands have handy one or two letter shortcuts, which you can see via "?" (*help*) command. More disruptive and potentially dangerous CLI commands do not have shortcut and have to be typed in full, as a precaution measure. The command line interface constantly evolves and new features are added to it. Please use *help* command to discover exact set of functionality available to you.

A number of CLI commands require hostname as first parameter. To avoid having to type hostname in case only one website is defined, you can use "." (dot) as substitute for the first defined hostname.

In some cases you want to specify "all defined hostnames", for example when examining inventory of cached responses . To specify "all defined hostnames" you can use "*" - asterisk.

Hostname parameter can also be a partial match string - for example "acme" matches any hostname (website) that contains "acme", for example: www.acme.com, news.acme.com etc. Please note that hostname matching is "simple", not regular-expression based, as opposed to URL patterns (see below) that are always regexp based. In most cases it doesn't make much of a difference.

A number of CLI commands require a regular expression pattern as second parameter. When you need a wildcard (match all) pattern, use "." (dot).

And lastly, to repeat previous CLI command without having to re-type it, enter "." (dot) as command at the CLI prompt.

Cached Response Signature.

In order to specify a particular cached response, as required for a number of CLI commands, you must provide exact cached response signature - same as reported by inventory CLI command. The URL signature consists of following parts:

1. hostname - always present

2. URL (path + possibly modified/removed query parameters) - always present
3. "p0" or "p1" for plain and "g0" and "g1" for gzipped (compressed) cached responses, HTTP/1.0 and 1.1 respectively
4. User Agent string (optional, possibly rewritten, present only when **sig_ua** is set)
5. **sig_cookie** (s) and their value (present only when a *signature cookie* is configured for the matching pattern).
6. Accept-Language request header value (present only when **sig_language** is specified)

The components of cached response signature are separated by spaces. To simplify entering of the signature, first lookup cached response of interest using CLI "**inventory**" command and copy/paste the string within "><" symbols. Here's an example signature, as you'd see it reported by "**inventory**" command:

```
>www.acmenews.com  /breakingnews.html p1<
```

Logging into CLI.

Assuming you have aiCache running, use your favorite telnet client to open connection to CLI. By default, aiCache accepts CLI connections using TCP/IP port 2233, over any of the defined IP addresses.

For example, to connect to aiCache CLI on a system that is named *aicache01*, type :

```
telnet aicache01 2233
```

You are greeted with CLI authentication prompt asking you to authenticate:

```
"enter password> "
```

Enter the password as configured in the configuration file (defaults to "*secret*").

Please note that the password is echoed in clear as you type it in. Upon successful login you are presented with a prompt consisting of aiCache server name, IP address and port number - as configured in the configuration file. It is done so that you could distinguish between different instances of aiCache. Countless IT mishaps happen when administrators think they are managing system **A** when, in reality, they are logged in to system **B** instead, so do consider using different names for different instances of aiCache. Having different instances have different names might also help with troubleshooting, as aiCache server name is inserted in all response headers as Via HTTP header.

```
"aiCache@1.1.1.1:80 > "
```

Look Ma, I don't get kicked out every 60 seconds : CLI has a generous inactivity timeout of 20 minutes.

help (shortcut: h)

This command shows available CLI commands. Please note that as CLI evolves, new commands might be added so this screen might look slightly different.

```
    1.1 paiCache *:80 >h

    ######################### CLI Commands Help Screen. ##################

    Most commands require hostname as first parameter
    Dot (.) can be used as convenience shortcut for first defined hostname
    [?|h|help] : this help screen
    alert hostname on|off
     [d|dump] : write out cached response to a file
    [e|expire] hostname URL : expires single cached URL
    [ep] hostname regexp : expires cached URLs that match provided pattern
    fallback hostname on|off
    [g|get] hostname URL : displays header of cached URL
    hcfail hostname on|off
    [i|inventory] hostname REGEXP : shows cached URLs matching the REGEXP
    [rl|rotate] : execute access log file(s) rotation
    [r|runstat] g|hostname s|m|h : displays global or website's running stats samples
    resetstat [hostname] : reset statistics counter globally or for matching website(s)
    [p|pending] [hostname]: displays  global or website's pending requests (awaiting
response)
    [sif] hostname REGEXP : shows up to 40 cached URLs matching the REGEXP, sorted by # of
fills(refreshes)
    [sit] hostname REGEXP : shows up to 40 cached URLs matching the REGEXP, sorted by
fill(refresh) time
    [sir] hostname REGEXP : shows up to 40 cached URLs matching the REGEXP, sorted by
requests
    [sis] hostname REGEXP : shows up to 40 cached URLs matching the REGEXP, sorted by
response size
    [s|stats|statistic] [hostname] : displays global or website's statistics
    shutdown : execute graceful shutdown of server
    quit|exit : closes CLI connection
    . : .(dot) repeats previous command, verbatim

    1.1 paiCache *:80 >
```

alert hostname on|off *[peer enabled]*

This command allows you to disable/enable aiCache alerting for specified website. You'd normally issue this command before performing maintenance on a website, to avoid issuing unnecessary alerts, and restore alerting after maintenance is over.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

blip IPrange on|off *[peer enabled]*

This command turns on or off blocking for provided IP address/range. See DOS protection section for more information.

clipt on|off *[peer enabled]*

This command turns on or off Intelligent Throttling. See DOS protection section for more information.

dump (shortcut: d)

This command dumps (writes out, saves) specified cached response to a file. The file's location is specified in the output. You can use this command to examine content of cached response - both header and body.

```
1.1 paiCache *:80 >d www.acmenews.com /acme/img/2.0/content/services/blogs.gif
Non-compressed body and header dumped to /tmp/dump_plain
Compressed body not found (cleaned by cache cleaner?).
1.1 paiCache *:80 >

1.1 paiCache *:80 >d www.acme missing.html
URL NOT FOUND in response cache: www.acme missing.html
1.1 paiCache *:80 >
```

exit or quit.

This command closes current CLI connection (logs the user out).

expire hostname uri_signature *[peer enabled]*

This command allows you to forcefully expire *single* cached response from the response cache that has the exact provided signature . The expiration is accomplished by resetting time of last refresh to 0. As soon as aiCache sees a new request for this Web document, a new (fresh) copy is requested from an origin server.

To expire a number of responses, use "**ep**" (expire-by-pattern) command below.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

ep hostname regex_pattern *[peer enabled]*

This command allows you to forcefully expire matching cached *responses* from the cache - as in responses that match provided hostname and the pattern. Only responses with matching URL are expired by resetting time of last refresh to 0. This way when the next request for any of these responses comes in, it forces refresh from an origin server.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

fallback hostname on|off *[peer enabled]*

This command allows you to force a website into or out of fallback mode. Please see "Operating Web sites in Fallback mode" for more information on this feature.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

fblip filename on|off *[peer enabled]*

This command turns on or off blocking for provided IP address/range, reading the IP addresses/ranges from the specified file name. See DOS protection section for more information.

hcfail hostname on|off *[peer enabled]*

This command allows you to force a website into "hc fail" state. Please see "Operating Web sites in HC fail mode" for more information on this feature.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

inventory hostname regex_pattern.

This command allows you to view matching cached responses. To view all of the cached responses for a particular hostname, you can use "." (dot) as regexp, as it matches any string. To view cached responses for all of the defined hostnames, use "*" as hostname. Hostname too can be a partial string, you don't have to type it in full.

Matching cached responses are printed out to the CLI console and written out to a file . In case that list is too long and is likely to take a while to scroll through the console, you might want to use "silent" inventory command instead: **si** - it only writes out the list of matched responses to a file, without displaying them at the CLI console.

Should a single match be found, the matched cached response is written out to a file. You can use this technique to assist with troubleshooting of responses with long and/or inconvenient signatures.

osdisable hostname ip_addres[:port] on|off *[peer enabled]*

This command allows to administratively disable an origin server. Port's numeric value needs to be provided if it is not a default port of 80. When as OS is administratively taken down , it stays down and is not allowed to serve any requests till it is re-enabled. However, health checks, if any configured, are still sent to origin servers even when they are in admin-down condition - so you might a server that is both administratively disabled AND has failed a health check. Admin status is displayed via Web interface, SNMP variable and CLI.

If any aiCache *peers* are defined, this command is communicated to each defined peer.

p or pending [hostname].

This command shows all outstanding (pending) cacheable requests - either global (no parameter) or matching specified hostname. This are the cacheable requests that are awaiting responses from origin servers. This command can be used to troubleshoot slow cacheable URLs in real time - these will be frequently seen in the output of this command, awaiting responses from origin servers.

peer [on|off].

This command allows to turn on|off (default: on) peer communications. It can be helpful if you want to execute a peer-capable command on a node, without notifying aiCache peers and having peers execute the same command. When executed without parameters, current status of peer communication is displayed.

The peer setting is specific to a given CLI session and doesn't persist past it. In other words, each logged in CLI client has its own peer setting (default is on) and whatever changes you might've made in previous CLI session won't affect the next one. Being local to a CLI session and transient, the value of this setting is not reported via Web or SNMP.

r or runstat [g|hostname] [r|s|m|h]

This command allows you to view matching running statistics - number of RPS, or average response time, as recorded by aiCache. You can request global (specify "g") or host-specific (specify matching hostname) running stats.

When "s" is specified, you will see 5-sec intervals, "m" - 1min intervals and "h" - 1hr intervals of average RPS data. Each sample has timestamp and value attached to it, freshest data is displayed at the end of output (sorted by time, ascending order). Depending on the interval value, you might see up to few days worth of data, most recent data shown at the bottom of the CLI screen.

When "r" is specified, you will see up to 240 of global or website-level, snapped every 5-sec average response timesEach sample has timestamp and value attached to it, freshest data is displayed at the end of

output (sorted by time, ascending order). Up to 20 minutes of such data will be shown, most recent on the bottom of the CLI screen.

Hint: you can obtain an instant chart of this data by copying and pasting the output into a spreadsheet program (Excel etc).

The ability to obtain such important stats on the fly, in real time, without having to crunch through gigabytes worth of log files, is an important analytics and troubleshooting tool. This data comes instantaneously from aiCache's internal in-memory structures, no log file processing is involved.

resetstat [hostname]

This command resets global (no hostname provided) or website-specific statistics. The hostname, when specified, is used as a wildcard match for defined websites. When no hostname is specified, aiCache resets global stats along with stats for all of the configured websites.

Use this command when you want to reset (set back to zero) the numerous stat counters aicache collects. For example, you make changes to patterns and want to see how the caching ratios are affected. Normally, the cache ratios are calculated (averaged out) over all of the accumulated requests since aiCache's latest restart and you might want to reset the stats so that only the new requests are accounted for.

reload.

This command forces on-the-fly reconfiguration (reloading of the configuration file). Use it to have new configuration to take effect without having to stop and start aicache - which would cause downtime and inconvenience site visitors. Please note that you can add additional listen ports or plugins via a reload, complete restart is required instead.

An alternative way to force configuration reload via reload watch file, as explained elsewhere in this Guide.

Please note that aiCache inherits all of the accumulated stats upon reload. Use "rs" CLI command if you want to reset global or website-specific stats.

rl or rotate_log.

This command forces log file rotation. Existing access, error and statistics log files are renamed, with a time of rotation appended as suffix to the file names. They reside in the same directory where the original access log file(s) are.

There's no downtime required, rotation is performed on-the-fly. You can also accomplish such on-demand rotation by sending **SIGUSR1** to aiCache PID.

si hostname regex_pattern. *(Silent Inventory)*

This command allows you to write out list of matching cached responses to a file called /tmp/inventory.out without writing it out to the CLI screen. This is different from regular **inventory** command that not only writes the list of matching cached responses to a file, but also displays it via CLI. So if the list of matching responses is very long and you'd rather not have it scroll on the CLI screen, but still want to analyze it, you can use the silent inventory **si** command. To view all of the cached responses for a particular hostname, user "." (dot) as regexp.

sif hostname regex_pattern. *(Sorted By Fills)*

This command allows you to view matching cached responses, sorted by number of fills, most-refreshed on top. To view all of the cached responses for a particular hostname, user "." (dot) as regexp. Consider increasing the TTL for responses that are subject to too many refreshes. To view cached responses for all of the defined hostnames, use "*" as hostname.

sit hostname regex_pattern. *(Sorted By fill Time)*

This command allows you to view matching cached responses, sorted by fill time, slowest responses on top. To view all of the cached responses for a particular hostname, user "." (dot) as regexp. Should abnormally slow response time be reported, you might want to investigate the reason for such behavior - you just might discover a SELECT from a table with 10 million rows .... without a WHERE clause (yep, a true case).

To view cached responses for all of the defined hostnames, use "*" as hostname.

sir hostname regex_pattern. *(Sorted By Requests)*

This command allows you to view matching cached responses, sorted by number of requests, most requested on top. To view all of the cached responses for a particular hostname, user "." (dot) as regexp. The most frequent requests should be viewed as prime candidates for caching.

To view cached responses for all of the defined hostnames, use "*" as hostname.

s|stats|statistics [hostname]

This command allows you to view global or host-specific statistic, providing a large set of data. Make sure you see CCR and SCR in 80%+, number of error responses is low, origin server response times are reasonable, origin servers are healthy and their HC fail and general fail count are low, etc.

No other tool provides nearly as much useful statistics as aiCache, instrumentation set is truly remarkable in its breadth and depth. Use CLI “rs” command to reset global or website-specific stats.

shutdown

This command requests a graceful shutdown of accelerator. Incoming connections are disallowed, while the ones that were already established at the moment of shutdown, are allowed to complete, for 5 seconds (exact value is configurable via configuration file).

Recommended aiCache setup.

One of better architectures for you to consider is when you use a load balancer (such as A10, F5, Netscaler, HAProxy+VRRP, etc) to load balance traffic amongst your aiCache servers, while having origin servers act as warm *standby*. So under normal operation conditions it is the aiCache servers that receive all of the traffic, accessing origin servers behind the scenes.

But when you take aiCache servers down for maintenance, then the traffic starts flowing directly to origin servers, assuring zero impact on your clients. When aiCache servers come back online, all of the traffic is sent to aiCache servers. To assure zero downtime, please use graceful shutdown (via **SIGHUP** or CLI **shutdown** command). Clearly you must make sure such maintenance, when origin servers are exposed to client traffic and are not accelerated by aiCache, is performed during time windows with low traffic so that your origin servers can handle it.

Exactly how you configure such setup varies based on HW/SW you use for your load balancers. With F5 you can put both aiCache and origin servers into a pool that is assigned to a VIP. Then you configure aiCache servers to have higher priority than the origin servers and you're all set.

In general, it is highly recommended to have some kind of load balancer distributing the load amongst your aiCache servers (aiCache can load balance origin servers by itself, no need for an additional layer there). While we cannot recommend a particular product to use, common, well known and proven load balancing products include A10, F5, Netscaler, HAProxy etc.

A true hallmark of highly available network setup is a clustered aiCache installation, enhanced with VRRP. Please see a dedicated chapter on particulars of such setup.

Common pitfalls and best practices.

The most painful, yet easiest to avoid or rectify, problem to deal with is allowing, by mistake, **sharing of private (personal) user information**. In the context of Web accelerators, it means that you have allowed caching of a URL that renders such private information.

Identify these URLs (links) on your web site and do not allow them to be cached. aiCache does not allow caching by default and you always have to tell it explicitly what URL are allowed to be cached.

Be careful with web site setups that are glued together with a heavy dosage of JavaScript. These might be tricky in this respect.

Do not underestimate power of caching – even for what seems like very brief moment of time. For example, let’s assume there is an HTML page that is quite popular with users – say a “Latest/Breaking News”

page that is rendered dynamically and is rather expensive to render, requiring access to a number of backend systems, including databases and a file store.

Assuming this page is requested a lot, say 100 times/sec, you can obtain major benefits by caching such page for just few seconds. Such small caching time will still reduce traffic to the backend (including load on application and database servers and your NFS/CIFS file store) **more than 100-fold !**

Do consider using selective log rejection feature. Implement it and watch most of the pain of having to deal with unmanageable log file sizes go away. As simple as that. Turn off collection of User-Agent information and Referrer - unless you must collect these fields.

If your site's having an issue and you must block or redirect set of URLs - do it via aiCache ! It is ideally suited to front-ending user requests due to it's lightweight, non-blocking multiplexed architecture - that is capable of managing tens of thousands of simultaneous connections off a single system, using a single process !

If your Web Site, like many others, appends various parameters (mostly for information tracking purposes) to static HTML or JavaScript URLs, make sure to declare these affected URLs as "cacheable-by-stem-only" or configure it to ignore some of the parameters in the URL (**ignoreparam** directive). You will avoid cache poisoning[20] problem and you will be glad you did.

Let's now discuss security aspects of deploying products such as aiCache. Most Web Sites deploy a firewall in front of their web servers. This firewall is typically configured to allow only for HTTP/HTTPS traffic to pass-through, best shielding the servers from an outside attack. Assuming you install aiCache and enable SNMP and CLI monitoring interfaces, you should make sure that those are not reachable from outside (the Internet). Once again the easiest way to achieve that is to have firewall installed in front of the aiCache.[21]

We assume that you already have a pre-existing firewall of some kind that shields you web servers. In this case you would not have to change anything on the firewall assuming the aiCache takes over the IP addresses of your former web servers. For more on that see Example Deployment scenario at the end of this document.

A true hallmark of highly available network setup is a clustered aiCache installation, enhanced with VRRP. Please see a dedicated chapter on particulars of such setup.

[20] This term referrers to having way too much content stored in the cache when it is not needed, cannot be shared, should not be there or contains many unnecessary replicas of the same object.

[21] Not that aiCache *requires* firewall, we simply assume you have it to begin with, just as sound best practice approach.

Testing and troubleshooting.

Frankly, there is not a whole lot to getting aiCache up and running and doing things just the way you want. Assuming you have basic understanding of your existing architecture you would not have a problem incorporating aiCache into the setup. This manual does not explain how to install operating systems or configure them. You are also supposed to know or have someone who knows, how to modify network settings and perform other basic Web and System administration duties.

After installing and configuring aiCache, but before putting it into production use, you'd definitely want to test it. Select a test system - a common computer running a browser of your choosing. Do the "hosts" file trick on a test system by mapping the site you're accelerating to aiCache's IP. For example, on Windows platform, you'd edit file called: **/windows/system32/drivers/etc/hosts , /etc/hosts** on Linux etc.

Now, as far as this particular test system (one where you modified the *hosts* file) is concerned, **that system and that system only** see your website *through the eyes of aiCache*. No other users - internal to your company or elsewhere on the Internet, are affected yet. Please note that for this trick to work, there must be no proxy servers between your test system and aiCache as such proxies have their own DNS resolution performed (and their own hosts file) - in this case you might need to play with proxy exclusion lists in your browser.

Fire up the aiCache and run a thorough test - different pages/sections of the site, logging in and logging out, posting, searching etc. Test it from both IE, Firefox, Chrome and Opera and any other browsers you can think of. Fine tune TTLs, fix any issues you discover. Use CLI and monitor the access and error log file to verify that aiCache actually caches objects you request as you move along. Use CLI to make sure objects age properly and obey the TTL you have configured for them.

Fire up some *stress tests* - use one of many available stress test tools (Webtool from Microsoft, Apache benchtool and Jmeter are free, amongst others) and generate a representative (meaningful) load against the aiCache. Just make sure the caching ratio is high enough so you're stress-testing the aiCache instance and not the origin servers (or other components of network infrastructure - including firewalls and load balancers).

As you run the stress tests, monitor aiCache servers via self-refreshing Web statistics pages - this way you can see traffic and response time in real time. Tail access log file - this provides even more information on what documents are cached, compressed, served from cache etc.

At this point you might want to make sure no personal information is accidentally shared because of the caching (and resulting sharing) that is now in place. Exactly how you can accomplish this varies and no universal approach exists. If you can, arrange for a number of different users to log in from a few different PCs and visit URLs that are supposed to serve personalized information and make sure each user gets to see his or her information only. If you see a problem here, do not panic.

Let's review a brief example of what might cause this kind of problem.

Our acmenews.com example web site has a URL that displays user's profile (private) information: **http://www.acmenews.com/viewprofile.asp.**[22]

The code behind .asp URL relies on a user ID cookie being present and this is how it knows what information to pull from the user registration database and display to the user.

If by mistake, you allow this URL to be cached - say, by allowing cashing for any URL that matches **.asp**, it will cause a serious problem – that of privacy nature. Here's how it happens. The first time and then every time this URL is refreshed by the aiCache, it duly asks for a fresh copy from an available origin server. origin server will execute .asp file, that will in turn use user id cookie that of the current user and send resulting document (HTML) back to the user's Web browser. On its way back aiCache will cache this HTML response. Everything looks nice and rosy so far.

The problem appears the very next time another user asks for his/her profile to be displayed. aiCache, knowing that the URL is cacheable (because someone has configured it this way) serves a copy from its cache right back to the current user. Thus the problem – **this copy contains personal information of a completely different user**.

Now that you know why there is a problem, it is very easy to prevent it from happening. Simply disallow caching of this URL. Please also note that aiCache never passes any cookies from browsers to the origin servers - so even when wrong is declared cacheable, it won't jeopardize your security setup. Still please do not cache URLs that render private data.

After the tests you should be ready to send actual user traffic to aiCache. Depending on your particular setup it could be as simple as adding aiCache IP to the load balancer configuration or adding aiCache IP as one of the DNS "A" records for your site. In either case some user traffic will start flowing to aiCache. Monitor it for a while to make sure no issues are reported and then you will be ready to "cut-over" all of the user traffic to aiCache. Do it after fine-tuning the configuration to maximize the benefits of caching and making sure the number of cached documents is appropriate for the amount of RAM you have in your system.

You might opt to still keep your origin servers configured as fall-backs on your load balancers etc - *just in case*. Chances are that after running aiCache for few days, you shall see most dramatic reduction of traffic to your origin servers and the rest of your infrastructure - including (not limited to) application servers, database server - the primary, the read replicas etc, file servers and so on. Hopefully you will be able to repurpose/retire a whole number of servers as you won't need them anymore.

You might come across a puzzling situation that requires capturing/seeing both requests - as they are sent to origin servers, and responses, as they come from origin servers. To accomplish that in controlled fashion, you can use **debug_clip** global setting. Set it to an IP address - now every request (complete request) and response (up to 32KB of response) with matching client IP is written out in **/tmp** folder as *debug_request.NNNN* and *debug_response.NNNN*, where NNNN is connection number.

[22] The way this script is implemented is not important – it could alternatively be a servlet or a cgi-bin script.

The Single Server Deployment

It could be that you can only spare a single system to install aiCache on. It might be the same system that you run your web server on. While not recommended for performance and reliability reasons, this single server setup is quite possible.

To install aiCache in this fashion, you would need to make sure aiCache and origin web server do not step over each other when trying to open and listen on the same network port 80 (default HTTP port). One way to accomplish that is to assign a different port number to be used by the origin web server, say port 81, assuming you want to run aiCache on default port of 80. After re-assigning the port on your original (aka origin) web server, you will configure aiCache to use this new port number at this particular origin web server.

Alternatively, you can assign an *alias IP* to the host server and have aiCache use the "new" IP while the origin server uses the old one.

Please refer to administration guide for your particular web server on more information about assigning a different port and/or IP address.

Webby-worthy production setup ideas.

Here's some of the items that you want to address to have a truly bullet-proof setup.

- A true hallmark of highly available network setup is a clustered aiCache installation, enhanced with VRRP. Please see a dedicated chapter on particulars of such setup.
- Front-end Load Balancers (F5, Netscaler, A10, HAProxy etc). Use these to load balance traffic amongst your aiCache servers. If possible, have origin servers configured as fall-backs. With LB in the mix you can have a very resilient setup that allows for zero downtime maintenance of components .
- Internal monitoring: you must monitor your setup. You can use Ganglia, Naggios, Sitescope and similar, to monitor all of the components of your infrastructure - routers, firewalls, aiCache server, origin servers, DB servers etc. Collect and store statistics for charting and trend. We recommend collecting aiCache statistics with SNMP.
- External monitoring: ideally, you'd want to make sure your site is available, functional and reasonably fast when viewed from outside. Keynote is one of the vendors offering external, well distributed monitoring. If you have presence in additional datacenters, you can setup your own distributed, external monitoring.

- Alerting. You can use a plethora of tools to watch your components. Most such tools require some work to get them up and running. aiCache's internal monitoring requires almost no effort - simply specify alert email address and tell what to alert on and you're in business. *You* should know about problem *before* customers call in and complain.

- We won't talk about security, backups etc, as these, while very important, are outside of scope of this Guide and aiCache as a product.

Web site troubleshooting 101.

You're in charge of a complex web site with a multitude of sub-domains, hosting all kind of information: editorial content, search, viewer comments, videos, news feeds, financial stock quotes. It is a thing of beauty, with 20+ APIs of all sorts, few dozen web, application and database servers - all working in concert to drive millions of page views per day.

So your site is humming along quite nicely yet all of a sudden your monitoring screens light up red and site slows down to a crawl. A minute later it is down hard and no responses are getting back to the clients. Where do you start looking to understand what's going on ? What do you do to restore the service ASAP ? How do you make sure it doesn't happen again ?

Fortunately, with aiCache front-ending the traffic problem resolution becomes a fairly easy and straightforward exercise. A *divide'n'conquer* approach is in order - we need to understand what component is ailing and what traffic patterns are prevalent.

This is where you start: pull up the aiCache Web monitor - it will show all of the domains aiCache is accelerating. Look to see which ones are showing highest client/origin server session counts, slowest response times and highest increases in traffic.

aiCache displays all of that information in real time - refreshing it every few seconds ! You shall almost instantly see which site is getting hammered with traffic . aiCache displays avg RPS in last 5 sec, last minute and last hour - so you can quickly see what site saw the highest traffic *jump*.

From this all-websites-overview screen you can also zoom into a particular site or see list of pending origin server requests - this one is likely to provide a list of URLs that are unable to obtain quick responses from origin servers, per website, all in real time again.

You can configure aiCache to collect and report pattern-level statistics by specifying **collect_pattern_stat** at website level. The collected pattern stats can be accessed via Web and SNMP interfaces. Here's an example of the data collected.

```
Pattern: >\.gif<, TTL: 10 sec, OST: 0
    #Req: 118, #CHits: 90, #CMiss: 28, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
```

```
    Pattern: >\.jpg<, TTL: 10 sec, OST: 0
        #Req: 79, #CHits: 48, #CMiss: 31, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.png<, TTL: 10 sec, OST: 0
        #Req: 8, #CHits: 5, #CMiss: 3, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.ad<, TTL: 10 sec, OST: 0
        #Req: 0, #CHits: 0, #CMiss: 0, #OS Err: 0, RPS: 0 req/sec, Avg OSRT: -1 msec
    Pattern: >\.js<, TTL: 10 sec, OST: 0
```

This information provides very granular, real-time view into traffic to a given website. Consider examining this information closely when you need to troubleshoot an issue. Assuming your patterns are setup to finely dissect and group inbound requests, you should be able to quickly establish which URLs are getting hit, are slow to response or are receiving errors from origin servers.

Some setups share many components so that after a spike against one of services/sub-domains, literally in 10-20 seconds the whole site might come to a screeching halt. How do you reconstruct the sequence of events ? Again, aiCache come to the rescue - it collects 5-second snapshots of traffic and stores it both in aiCache memory, where it can be instantly pulled up for each of the accelerated websites, and also in statistics log files - one per each accelerated domain. So you can use "runstat" CLI command or look at the statistics log file to see what domain started seeing elevated traffic levels and/or slower response times from origin servers first.

You can also use CLI's "inventory" commands, the "sorted by fill time, # req, # fills" variety to instantly see the most requested URLs, the slowest URLs to obtain - again, for each and every subdomain.

So you narrow your search down to a failing domain. It is still unclear how to go about fixing it, yet you want to restore the service so that all the other domains/services can start working again. The easiest way to accomplish that is put the ailing sub-domain into *fallback* mode. This way it continues to serve cached and possibly stale content, while completely disengaging from origin servers and letting your other team members concentrate on restoring that service.

For many a busy website, use of *fallback* command has another important benefit - as aiCache now either instantly delivers a cached (possibly stale) response or, equally instantaneously, sends an error message to the client, you stop the uncontrolled growth of session counts on your network/security devices and start dissipating those.

Bottom line: with aiCache in the path of traffic, understanding, troubleshooting and recovering of websites becomes a fairly easy and straightforward exercise. The benefit of understanding traffic flows, in real time, if a significant enough reason to consider inserting aiCache into the path of all web traffic you can think of - while providing a host of additional benefits !

Using aiCache to facilitate website transition.

aiCache's pattern and origin server tagging (**os_tag** directive) is ideally suited for easing the typical chores of website transitions. Many an IT professional shiver at the mere mentioning of this two words: website transition, but it doesn't have to be painful at all. Let's imagine that Acme Inc has its current website of news.acme.com hosted with a legacy provider, FlyByNite WebHosting Inc . The site was developed a while back and is running a custom written CMS that is, quite frankly, growing long in the tooth. It is rather slow, lacking in features, hard to maintain and especially hard to modify. It can go down without much of a warning and doesn't have much of monitoring in place - which results in fair bit of frustration at Acme, as helpless Acme web site crew observes, in horror, another meltdown, unable to do anything to help the situation.

Not content with status quo, Acme leadership team decides to embark onto a difficult journey - a complete redo of the site's CMS, this time going not with a home-grown solution, but adopting one of more popular Open Source CMS. Such endeavors are typically very labor intensive and error prone all-or-nothing approaches. In other words, after spending much time and money, you'd *throw the proverbial switch* and send traffic from old environment to the new one and hope it all works out.

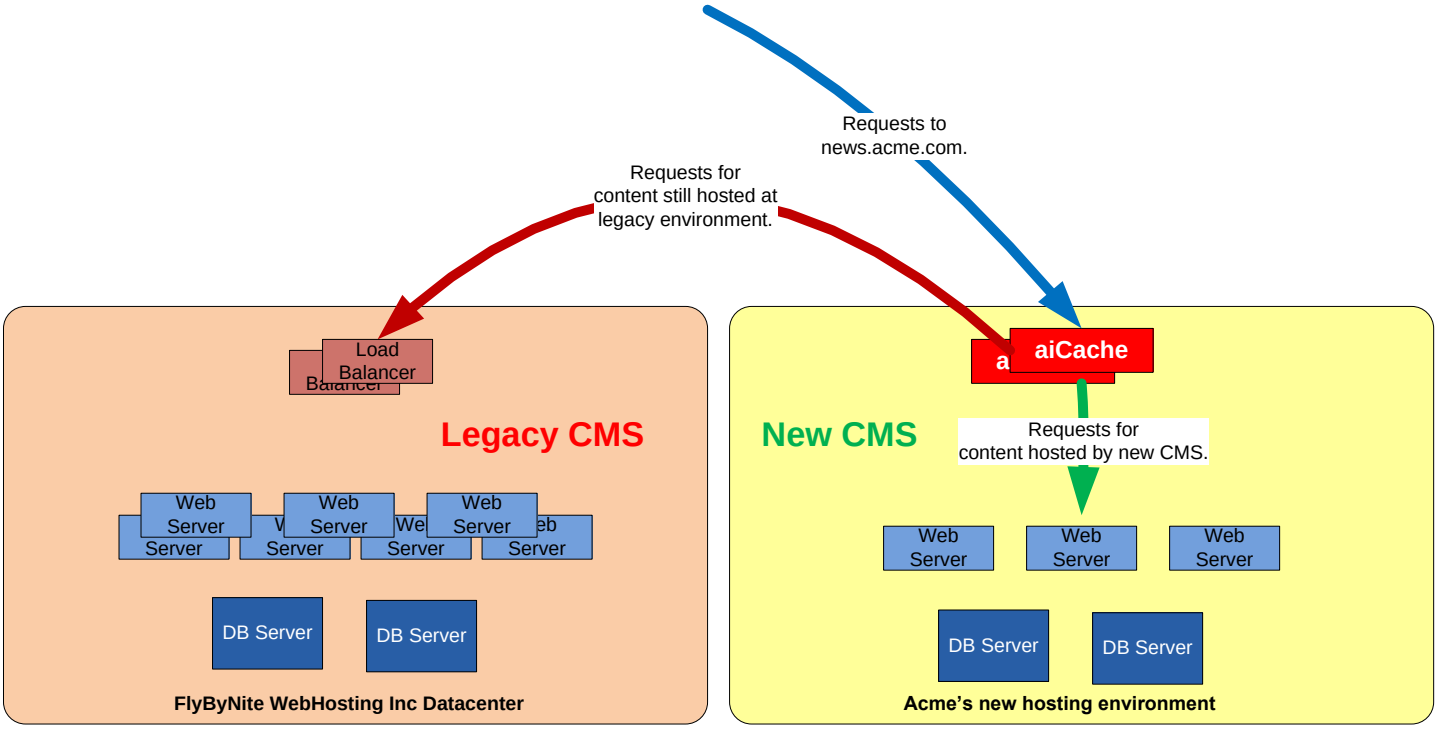
However, with aiCache in the mix it doesn't have to be such a monolithic effort. Instead, you can transition your site section by section, without any impact on your users, having complete control and confidence every step of the way.

Acme decided to start by placing aiCache in front of news.acme.com domain. They can now cache some content, reducing the load on the frail legacy environment, observe traffic and it's vital statistics in real time, alert when things go sour, still keep the site up in case of complete hosting meltdowns, that are growing more frequent at FlyByNite WebHosting Inc's datacenter. The situation is instantly improved and there's now some breathing room for the transition.

Next, Acme sets up their own CMS instance (with most CMS, 2-3 web servers and a DB, ideally in a highly available setup). Now Acme chooses a section that they want to improve first - typically one that gets most traffic. Let's say it is US News and it lives under news.acme.com/us_news . Having setup the section on the new CMS, now we can tell aiCache to fill the requests for news.acme.com/us_news not from legacy, but from the new CMS - by tagging the relevant pattern and adding the new CMS web servers to the configuration.

And with a simple, instantaneous aiCache configuration change, Acme can introduce the new CMS into the traffic path in an incremental, controlled, measurable and fail-safe way. As more and more sections are developed in the new CMS, they too, are configured to be served from the new CMS. Incrementally, section by section, Acme takes over the control and hosting of news.acme.com and after a while the FlyByNite WebHosting Inc is out of the picture .

The picture below depicts the setup, with legacy CMS on the left and the new CMS on the right. You can see basic flows of traffic coming to Acme's hosting environment and aiCache deciding where to fill the content from. A very straightforward setup indeed. We also attach a small snippet of aiCache configuration file that shows basics of using **os_tag** settings to accomplish the desired effect.




```
......
website
hostname news.acme.com

pattern /us_news simple 1m # Cache for 60 seconds
os_tag 2                   # Fill from the new CMS by specifying os_tag of 2

pattern .....              # Patterns for content to be served from old CMS
pattern .....


origin 1.1.1.1 80          # Legacy origin server, no os_tag used!
origin 1.1.1.2 80          # Legacy origin server, no os_tag used!

origin 2.2.2.2 80 2        # New CMS origin server, note the os_tag of 2!
origin 2.2.2.3 80 2        # New CMS origin server, note the os_tag of 2!
```

Using aiCache to address AJAX cross-domain limitations.

aiCache's pattern and origin server tagging (**os_tag** directive) is ideally suited for addressing the well known limitation of AJAX cross-domain scripting. As you may well be aware, the XMLHttpRequest object (aka XMLHTTP object in Internet Explorer) is the centerpiece of today's AJAX-driven web applications. Many a popular site owes its rich functionality and responsive interface to use of AJAX.

However, AJAX has one significant security constraint - one cannot create an AJAX request to a domain different from the one that the JS script itself was downloaded from. For example, if one has AJAX functionality embedded in news.acme.com/ajax.js , the AJAX calls from the said JS file can only address URLs at news.acme.com and not any other domain, or even a subdomain of acme.com . You shall see that this limitation can be easily addressed by using aiCache **os_tag** setting.

Let's imagine that news.acme.com has "newsmaker" page - a mash-up of data about celebrity of the day. The page displays the editorial content about the person, along with user-submitted comments and may be latest RSS-driven news about the celebrity. We also display the search results so that users can see prior content that was published by Acme. The comments, the search results and the latest RSS data are embedded into the page client-side, using AJAX. So as new comments are posted and RSS feeds are updates, the page refreshes itself without user having to reload it.

To obtain this functionality, there's some JS code, downloaded from news.acme.com, that needs to access URLs at comments.acme.com, search.acme.com and rss.acme.com. The three subdomains we mentioned could be hosted in different environments and/or datacenters. Some could be .Net-based, some PHP-based and some developed in Java. Some environments could be white-labeled from an external provider. No matter the technology, you can see that accessing three different subdomains presents a problem. It won't work unless one of the two different technologies is deployed to address it:

- a client side Flash proxy (see http://blog.monstuff.com/archives/000294.html)
- a server-side proxy solution

Flash proxy solution requires Flash support in the user browser. While almost ubiquitous these days, such Flash support is not present in quite a number of mobile devices, including Android-based ones and the latest product from Apple, the iPad. From the looks of it, we can take as a given that this Flash-deficiency is not likely to get rectified any time soon.

Server-side proxy solution doesn't suffer from such limitations and is completely browser and client-agnostic (although, clearly, for any of AJAX to work, client browser must support AJAX). Instead of trying to access the required functionality *directly* on the three different sub-domains, we instead request the functionality from news.acme.com and have aiCache servers to direct (relay) the requests to proper sub-domains. So let's provide a configuration file snippet showing how one can configure aiCache servers that accelerate news.acme.com, to transparently forward requests, based on the request URLs, to different origin servers.

We will assume that requests sent to news.acme.com/search.php?anything should be sent to *search* origin servers. Likewise requests to news.acme.com/comments.do?anything should be sent to *comments* origin servers.

And lastly, requests sent to news.acme.com/rss.apsx?anything should be sent to *rss* origin servers. In all three cases, we will rewrite the Host header to match the origin farm we choose.

```
....
website
hostname news.acme.com

pattern /search.php simple 1m # Cache for 1 minute
os_tag 2 # Send it to search servers
sub_hostname search.acme.com

pattern /comments.do simple 1m # Cache for 1 minute
os_tag 3 # Send it to comments servers
sub_hostname comments.acme.com


pattern /rss.aspx simple 1m # Cache for 1 minute
os_tag 4 # Send it to RSS servers
sub_hostname rss.acme.com


....
origin 1.1.1.1 80 # Regular origin servers for news.acme.com, no os_tag !
origin 1.1.1.2 80 # Regular origin servers for news.acme.com, no os_tag !

origin 2.2.2.2 80 2 # origin servers for search.acme.com, note os_tag of 2 !

origin 3.3.3.3 80 3 # origin servers for comments.acme.com, note os_tag of 3 !

origin 4.4.4.4 80 4 # origin servers for rss.acme.com, note os_tag of 4 !
```

As you can see, with a very straightforward and easy-to-manage configuration setup, we can now joggle, on the fly and completely transparently to the end users, requests amongst 4 different subdomains. All while caching responses, having real-time view of the traffic statistics, along with ability to shield end-users from any issues at the said sub-sites. Please note that such request relaying happens completely transparently to end users, in a sense that it is aiCache that obtains the actual response, no HTTP *redirects* are sent to the requesting browsers (and even if they were sent, such redirect still won't work).

When any of the sub-sites are hosted by external providers, we don't require any involvement from them, to accomplish such setups. Also, with such *hosted-API* arrangements, where charges are based on volume of requests, we can save significant amount of money by caching responses.

Frequently Asked Questions.

What kind of web sites benefit from aiCache the most ?

Short answer - all and any. News sites , community: message boards and blogs, social networking, rich media - on-line video, picture hosting , on-line shopping/e-commerce sites. By no means this is an exhaustive list, but the point is - all of the web sites will benefit from deploying of aiCache. You will spend less on HW, hosting, will have a much faster site and happier visitors, better uptime statistics, be able to handle traffic spikes that used to take it down.

Main benefits are caching and off-loading of client request/response processing away from servers that were never designed to handle it..

We're a busy site that is low on tech staff, can we get help with install and configuration ?

Absolutely, optional professional services engagement is available as part of purchase. We'd have our PS team evaluate your site setup to see how it can best benefit from aiCache, install and configure product in a turn-key fashion. Very little to none upkeep is required after installation, except regular chores of maintaining disk space etc.

What kind of hardware do you recommend for an aiCache server ?

Any 64bit Intel or AMD processor should do. Nowadays all of the 64bit processer are multi-core, meaning they have a number of separate CPUs on one piece of silicon.

A quad core (a single quad core CPU or 2 dual cores) are where you get the most bang-for-your-buck - as aiCache is right-threaded. Only 4 threads are used to do most of processing (and 2 more are only used episodically), so that works most nicely with 4 CPUs. You can be serving thousands of requests per seconds, having tens of thousands of clients connected - yet still it is all served by a single aiCache process's 4 hard working threads.

Getting even more cores (CPUs) will not have nearly as much impact as spending this extra money on extra memory. Sites differ in the amount of cacheable content, but you definitely want to make sure it fits into available RAM - so size the RAM accordingly. 8GB is a good start for most sites.

An aiCache server is capable of completely saturating a gigabit Network interface (and then some). While only truly busy sites ever see this kind of HTTP traffic, be cognizant of it.

As to what vendor to choose, we cannot recommend a particular one, for obvious reasons, but most companies have excellent luck with established vendors - such as HP, Dell, IBM. And still others have no issues rolling their own servers - such as a well known search giant.

Why does aiCache only run on Linux and why 64bit ?

Linux is the only OS that offers the most efficient network IO model - multiplexed EPOLL mechanism. No other systems offer this particular mechanism and/or scale anywhere close to how EPOLL scales. Google for EPOLL and C10K if you want to better understand scalability issues related to high traffic web sites.

aiCache caches all of the cacheable responses in memory (RAM), never on disk (local or NFS etc) - so the more RAM you can have in your server the merrier. 32bit systems are limited to offering around 3GB of RAM space *per process* (even though more than 4GB may be installed). 64bit systems do not have this limitation and there's no practical limit to just much memory might be available to a process. As of 2008 you can buy systems with up to 64GB RAM, in 2009 we expect systems with 256GB RAM to become available. Chances are you won't neat anywhere close to this much RAM, but large sites do benefits from having extra RAM.

Why aiCache when we have Apache and Microsoft IIS ?

aiCache doesn't *replace* the regular web servers - it only *dramatically reduces the number of them* that you need to support your web site. It also dramatically reduces the number of Application and DB servers, file appliances etc.

The content still *originates* at web servers. However regular web servers are ill suited to supporting large number of users and/or connections - most require a dedicated process per connected user, especially for dynamically generated content. This is the main reason the large web sites often grow out of control across the board - in total spend, number of servers, number of staff, amount of datacenter space, power and cooling that is required . And still, even with this many servers plenty of web sites have difficulties staying up.

aiCache offloads most of the load off the rest of your infrastructure (up to 95%+ in some cases) , so that most user requests are served right by aiCache, without ever having to touch your web servers, propagating in turn to your application servers, database servers, filers and any other components of your infrastructure.

Why aiCache when we have [insert a proxy web server here] ?

aiCache gives you complete control over what and how you want to cache - configured via a single source - no changes to code or web servers or your network setup.

It uses RAM to cache response - assuring near 0 time-to-first-byte latency for cached responses. It is extremely light weight and efficient, right-threaded application capable of serving tens of thousands of

connected users off a single server and up to 45 000 RPS, per server, in well-configured setup[23] - that is in excess of 160,000,000 req/hr !

It collects and makes available a wealth of performance data, most in real time, allowing for dramatically lower MTTR when you do have a problem with your web servers or app servers or databases.

It allows you to configure cache-by-path feature, selectively bust query parameters, supports cache-busting cookies, expiry of content by response header, user-agent-based redirection, comprehensive logging control, SNMP integration , a powerful CLI interface etc - too many features to mention. It's developed and supported by people that were at forefront of Internet's busier sites from early 1990-ies.

aiCache does work miracles with HTTP, what about HTTPS ?

aiCache offers full support for HTTPS - see a dedicate HTTPS chapter in aiCache User Guide.

Can we use aiCache to redirect mobile users to a different web site ?

Yes, please use "**ua_redirect**" directive under the proper pattern (normally, one for the home page). Normally, you'd provide a list of patterns that matches most common mobile device "User-Agent" HTTP headers (Windows CE, RIM are some examples) and aiCache takes care of the rest.

[23] assumes higher cache hit ratios, faster server HW, smaller response sizes and sufficient network BW.

Obtaining aiCache support.

Please use www.aiCache.com as central point for all aiCache related information. Pay a special attention to the FAQ section and support section. We also regularly blog about more interesting features in the Blog section of the same web site. In order to be eligible for over the phone support you must purchase a support contract. However even without such support contract you can expect to find a wealth of information in both the FAQ section of our web site and in support section as well. The support section points to a message board system. Chances on the question you have had been already answered in the forum.

You might also email any questions you have about aiCache to support@aiCache.com.

Appendix A: complete list of configuration directives.

For your convenience we have provided complete list of aiCache configuration directives, sorted alpha-numerically .

0ttl_cookie

0ttl_cookie_pat

0ttl_url_cookie

0ttl_ua

0ttl_url

404_redirect_location

500_redirect_location

accesslog

add_body_url_length

admin_ip

admin_password

admin_port

atc_challenge_dir

atc_challenge_prefix_file

atc_cookie_name

atc_fail_punish_time

atc_salt

atc_submit_url

atc_welcome_url

aggr_host

aggr_port

aggr_url

alert_bad_req_sec

alert_blocked_ip_sec

alert_client_conn_max

alert_client_conn_min

alert_dir

alert_exclude_pat

alert_humane

alert_ip_throttled_sec

alert_max_cache_entries

alert_max_bad_os

alert_os_conn_max

alert_bad_resp_sec

alert_req_sec_max

alert_req_sec_min

alert_os_rt

allow_ip

auto_throttle_cps

auto_throttle_cps_interval

bad_response

block

block_ip

block_ip_interval

block_ip_max_req

block_ip_punish_factor

block_ip_punish_time

block_ip_sorry_url

bmr_pattern

cache_anonymous

cache_cleaner_interval

cache_4xx

cache_cookie

count_4xx_as_bad

cc_disable

cc_inactivity_interval

cc_obey_ttl

cdc_bytes

cdc_pattern

cdc_interval

cfg_version

chunked_resp_size_hint

client_linger

client_tcp_no_delay

clip_throttle

cname

cookie_pattern

compress_json

compress_xml

collect_pattern_stat

conn_close

debug_clip

debug_cookie

debug_dns

debug_request_cookie

debug_response_cookie

debug_ws_match

decimate_log

decimate_bad_log

decimate_req

decimate_req_interval

decimate_req_pat_rps

decimate_req_ws_rps

default_host

disable_host_normalization

disable_gzip_ie56

disable_gzip

disable_os_tag_hc

disable_persistence

dns_interval

drop_referrer

drop_user_agent

drop

dummy_website

dump_bad_req

dump_req_resp

enable_gzip_ie56

enable_http10_gzip

enable_http10_keepalive

enforce_host

errorlog

error_stat_ignore

exp_watch_dir

expires_session_cookie

fallback

fallthrough

fb_pattern

fb_bytes

file_doc_root

file_req_prefix

filesystem

forward_clip

forward_os_etag_header

forward_ua

get_post_head_only

groupname

hard_cache_cleaner_interval

hdr_clip

healthcheck

hostname

host_url_rewrite

httpheader

httpheader0

http_only

https_only

if_name

ignoreparam

ignore_case

ignore_host

ignore_ims

ignore_no_cache

ignore_resp_expire

include

keep_clip

label

leastconn

listen

logdir

logging

log_cookie

log_osrt

log_os_hit

log_healthcheck

logstats

logtype

mail_path

match_http_only

match_https_only

match_max_url_length

match_min_url_length

max_body_size

max_post_sig_size

maxclientbodytime

maxclientidletime

max_header_size

maxkeepalivereq

maxkeepalivetime

max_log_file_size

max_resp_size

max_os_ka_conn

max_os_ka_req

max_os_ka_time

max_os_resp_time

max_pref_fresh_time

max_sig_size

max_url_length

min_gzip_size

min_resp_size

no_gzip_ie56

no_retry

no_retry_min_resp_size

no_retry_max_resp_size

num_files

origin

oid

oid_idx

os_linger

os_persist

os_tcp_no_delay

orig_err_resp

param_partial_match

pattern

pass_cookie

peer

peer_prefix

pend_url

pid_file

post_block

post_drop

prefetch_conn_close

prefetch_http_header

prefetch_url

process_etag

redirect_location

redirect_4xx

redirect_5xx

refresh_hdr_clip

refresh_website

reload_fail_file

reload_fail_run_cmd

reload_run_cmd

reload_success_file

reload_watch_file

request_type

req_plugin_define

req_plugin_exec

retry_min_resp_size

retry_max_resp_size

retry_non200

rewrite

rewrite_http_only

rewrite_https_only

send_200_resp

send_cc_no_cache

send_cc_cache

send_os_via

server_ip

server_name

server_port

session_cookie

session_cookie_required

session_cookie_persist

session_cookie_url

session_duration

sets_session_cookie

silent_block_ip

silent_400

shutdown_grace_wait

sig_cookie

sig_header

sig_language

sig_ua

sig_req_host

snmp_stat_interval

soft_fallback_duration

soft_fallback_eps

stat_req

stat_url

stats_host_prefix

stay_up_on_write_error

ttl_scale_factor

ua_keep_length

ua_pattern

ua_redirect_host

ua_redirect_host_only

ua_redirect_rewrite

ua_redirect_url

ua_tag_cookie

ua_tag_file

ua_tag_pattern

us_keep_cookie

ua_redirect

ua_sig_rewr

ua_url_rewrite

unified_cache

username

website

wildcard

work_dir

x_ttl_header

x_os_header

Appendix B: Up and running in 5 minutes or less.

This is a copy of README file that is included with aiCache distribution.

Welcome to aiCache !

Thank you for acquiring or trying out the aiCache Web accelerator.

To find out more about the product, obtain latest updates, download
Admin Guide etc, please visit www.aiCache.com.

In this document, we provide a quick overview of steps required to
get you up and running in no time at all.

Let's get started and in 5 minutes or less you should be able to
accelerate and test your website from a test computer. There's no
impact to actual site, its visitors or anything or anybody else, for that matter,
as you follow the steps below - it is absolutely safe and transparent.

##############
0. You need to be root or use sudo command to finish the install
of aiCache. We've provided a small script: install.sh , that automates
steps 1-3 below. Feel free to execute the script from this directory

or run steps 1-3 by hand.

############
1. You need to create a directory where aiCache binary, configuration and license files will reside. While you can have these in different directories, we recommend following the KISS principle. So, please create a directory called "/usr/local/aicache":

```
    # mkdir -p /usr/local/aicache
```

############
2. You need to create a directory where aiCache log files will reside. "/var/log/aicache" is a good name:

```
    # mkdir -p /var/log/aicache
```

If you run a busy website, please make sure /var/log/aicache is on a partition that has enough space left to accommodate for the log files that will get created.

############
3. Now you need to choose what user name and group name to use to run aiCache. Some systems come preconfigured with suitable user and group names - such as www-data, apache etc. Use those or create a new user and group called aiCache.

```
    # groupadd aicache
    # useradd -g aicache aicache
```

Set aiCache's password, shell and other user attributes to your liking.

Copy the distribution files to /usr/local/aicache:

```
    # cp aicache example.cfg *demo README /usr/local/aicache
```

Change ownership of /var/log/aicache and /usr/local/aicache to aicache.aicache.

```
    # chown aicache.aicache /var/log/aicache
    # chown -R aicache.aicache /usr/local/aicache
```

############
4. That's it, now we need to setup the configuration file. Use example.cfg as a starting point. Copy it to whatever other file name you want. We will use "aicache.cfg"

```
    # cd /usr/local/aicache
    # cp example.cfg aicache.cfg
```

Assuming common defaults:
- you want to run over HTTP port 80
- you want to accept connections over any configured IP address
- you're fine with default CLI port 2233
- you're fine with default CLI password of "secret"

all you need to do is to define:

- you website's DNS name (let's assume it is www.acme.com)
- the IP addresses or port numbers of one or more origin web servers
(let's assume it is 1.2.3.4, 1.2.3.5 - on port 80).

Using editor of your choice, edit aicache.cfg. Locate the "hostname" setting and set it to www.acme.com:

```
    hostname www.acme.com
```

Locate "origin" directive and add your origin web servers:

```
    origin 1.2.3.4 80
    origin 1.2.3.5 80
```

###########

5. We are done with configuration and now can test it.

As we test, there is absolutely no impact on the site we're testing the acceleration of. None, zip, zilch, nada. No extra load against the website itself. All of the site's visitors are still going to the site just the way they did before and nothing has changed as far as they are concerned. Their requests are not hitting the aiCache server and are still going to the origin web servers. Again, 0 impact.

You can test the aiCache from a dedicated desktop or set of systems in a contained fashion.

First we start aiCache.

```
    # cd /usr/local/aicache
    # ./aicache -f acme.cfg -l *.demo
```

You point to the config file via "-f acme.cfg" and the demo license file via "-l *.demo" .

As aiCache starts up, you will see some messages appear in the console window.

To finish the testing, you now need to get a system to test the setup via a browser . It can be done via a "hosts" file trick.

Assuming aiCache was installed on a server with IP of 1.1.1.1, you'd modify the hosts file to point the "www.acme.com" to 1.1.1.1. On Windows system, the hosts file is located in the C:\WINDOWS_INSTALL_DIR\system32\drivers\etc\hosts . If you're testing from a Unix/Linux host, the file to edit is called /etc/hosts. Open it with your favorite editor and add:

```
1.1.1.1 www.acme.com
```

Save the file. You don't have to restart the computer. Fire up a browser of your choice - Firefox, IE or whatever (you have to start the browser _after_ the hosts file is modified).

Go to www.acme.com . Now your browser and your browser only is hitting the aiCache instance. In other words, you're seeing your website: www.acme.com "through" or via aiCache. It should look the same as it did before aiCache, all the functionality should be there.

Take a look at /var/log/aicache/access log file - you should see your requests appear in it.

Congratulations ! You have accelerated your site with aiCache !

Now all you need to do is:

- fine tune the caching times
- make sure you don't cache/share private data by mistake
- ideally, enable and configure origin server health monitoring
- take care of regular chores: startup, log rotation, monitoring etc

Everything you need to know is explained in detail in aiCache Admin Guide.

Appendix C: Maximizing aiCache benefits: Client-Side Personalization Processing.

Introduction.

Many a web site use a customary "greeting" section on their web pages, showing something to the effect of "***Hello Mr.Smith***" or "***Welcome smith11***" when a *logged-in* user is detected. When an *anonymous* visitor opens these pages, a login form (or a link to one) is displayed instead, with customary "username" and "password" fields.

Frequently, the user name or login id text (such as ***Mr.Smith*** or ***smith11***) within the "greeting" message is a link , such as *userprofile.php* or *settings.asp,* to user settings screen where user preferences can be changed, messages read etc.

Some site don't require visitors to register for all or most of their content (major news sites are like that), some only offer small subset of content to anonymous users and require registration and login in order to the see the "premium" content.

Most of the time such "greeting" section is built via server-side code (PHP, Java, ASP etc). When login form is first submitted and after successful authentication of the user, a new HTTP session is created and *username*, *userid* and some other information might get stored in it. As such logged-in user then visits different pages on the site, every time the page is generated a new and the appropriate greeting gets incorporated into the page. The session is frequently associated with a session cookie. Some common session cookie names include "JSESSIONID", "PHPSESSID" etc.

The problem with server-side personalization.

Outside of the greeting message, the rest of the web pages look the same for all visitors - both *logged-in registered* users and *anonymous* visitors alike. Yet a piece of server-side code might be executed every time pages are rendered (served), inserting a personalized greeting message into the pages, making them non cacheable - all at a regular price of stressing most of your infrastructure - from web servers to application code to DB servers. It also complicates the setup, as you need to assure the session state gets replicated so all of the application code can see it or a provision is made for sticky (persistent, pinned) origin server connections.

However, there's an easy to eliminate such bottlenecks and enable caching of content - by moving "greeting" processing logic to client side. Just a few lines of JavaScript and viola : you can accomplish 95% caching ratios while personalizing content still ! Have your cake and eat it too. Here's how.

Client-Side Solution.

The provided example is a small html page that showcases an example of client-side greeting logic processing. There're no changes to your server-side code if your login logic generates a cookie that we can use as *userid* or *username* in the greeting. If no such cookie is being generated, it is trivial to have one implemented.

We assume that we can use value of a cookie called "*username*" in the greeting section. So, if *username* is set to *smith11*, the greeting will read: "Hello smith11" . We don't HREF (link) username to a URL (such as "editprofile.jsp") to save space, yet it is trivial to do so. You can also have links to regular "Profile", "Messages" added as part of the same code.

The code (in red) simply checks to see if a *username* cookie is set and if it is, it replaces the login form with the greeting. As simple as that. Users still have to login as usual to have the cookie set - no changes there. When a logged-in user decides to go to a profile page, the regular code will still execute to make sure the user is logged in (that one normally relies on a session cookie). No compromises in security.

Here's the code. Copy and save it in a file of your choosing. The code relies on *prototype.js* (you can obtain this very commonly used JS library at http://www.prototypejs.org/) and Cookie convenience code you can obtain at http://www.red91.com/2008/02/04/cookies-persistence-javascript.

```
---------------------- Copy from here --------------------

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html><head><meta http-equiv="Content-type" content="text/html; charset=utf-8">

<title>Blank Page</title>

<script src="prototype.js" type="text/javascript"></script>

</head><body><h1>Client-Side Greeting/Login processing.</h1>

<div id="bucket">

<form action="login.php?action=login" method="post" id="login">
Username: <input type="text" name="user"/><br/>
Password: <input type="password" name="password"/><br/>

<a id="sign_in" onclick="login.submit()" title="Sign in">Sign in</a>
<a id="sign_up" href="/registration.php" title="Register.">Register</a>
<a id="forgot" href="/passreminder.php" title="Reminder">Reminder</a><br/>
</form></div>

<div class="news-item" id="item_1">

<h3>Declaration of Independence</h3>
<p id="summary_1">When, in the course of human events ...</p>
</div>
```

```
<script type="text/javascript">
<!--

if (Cookie.get('username') != null ) {
  $('bucket').update('<h3> Hello Mr.' + Cookie.get('username') + '<\h3>');
}

//-->

</script></body></html>
------------------- To here -------------------------------
```

Testing.

To test, we use Firebug, the de-facto golden standard of all JS developers in the world. Assuming you have Firebug installed in a recent Firefox browser, simply open, in Firefox, the file you have created .

As no *username* cookie is presently set, you will see a login form on top of the page, with common login form attributes.

Let's set the *username* cookie and see what happens then. Click on Firebug icon and get to the Firebug console.

With the page open, type at Firebug's console prompt (>>>):

```
Cookie.set('username','maximus')
```

We have set a session cookie with the name of *username* and value of *maximus*. To verify that it is now set, you can type :

```
Cookie.get('username')
```

You should see the *'maximus'* value displayed, indicating presence of the *username* cookie.

Refresh the page. You will see that login form is gone and instead, an appropriate user greeting is displayed. Mission accomplished ! Now, this page and any other page that relies on client-side personalization, can be safely cached , with all of the numerous benefits of caching, yet it will be personalized via a friendly greeting that users expect to see after they log-in.

Conclusion.

We have shown, via a simple example, how to move personalization logic from server side to client-side, enabling page caching in process. There's nothing stopping us from extending this example to include more evolved personalization. Things like checking for messages (via XMLHTTPRequest, for example) and displaying the "*you have NNN messages*" link, showing a low balance warnings, hiding or showing links to premium content, etc .

Caching can be truly a life saver for many a web site - a difference between staying up and being down, quick response times and unacceptably long waits, 10 servers vs. 100 servers, $50,000 vs. a million dollar spend on your infrastructure, life or death ... well, that was an exaggeration, but you get the point.

Appendix D: Performance and stress testing of aiCache setups.

Most of website and system admins are the type of people that only believe vendor claims when they can see it for themselves. And it is especially true when it comes down to performance testing. One of the very first things folks want to do when testing anything that claims to improve performance of website is to put these claims to test. This appendix serves to point out some basic things about such stress and performance testing.

Enable caching.

Not much improvement can be seen unless you're letting aiCache to do what it is designed to do: cache content. Make sure you define some patterns with non-zero TTLs. A very simple pattern that would enable caching of every single URL for 60 seconds could be:

```
pattern / simple 60
```

Be aware that first request, for any given URL, to a cold aiCache instance would result in cache miss and the request would have to go all the way to the origin server to be filled, but the subsequent requests, for 60 seconds in our example, would come from cache.

Clearly, you wouldn't want to go to production with a cache-all pattern like that, but it is good enough for some quick and dirty testing.

Tools to use to generate a simple load test.

You can use Apache Benchmark or "ab" – it is normally installed along with Apache Web server and therefore could be found, free of charge, on almost any Linux system. To test a URL with 10 concurrent load threads, for total of 2000 requests, you'd run something like this:

```
ab -c 10 -n 2000 a.b.com/a_url.html
```

The report that *ab* produces is rather self-explanatory. You want to compare the before and after numbers of requests-per-second (RPS) and time-per-request, time-to-first-byte. It is also the easiest tool to use for some quick testing.

A more evolved tool for benchmarking is Jmeter. It is capable of running in a master-slave configuration, with main instance controlling a number of slave load generators. It has a flexible reporting API with a number of plugins available that can graph all kinds of stats in real time. You can create fairly evolved test scripts – crafting custom URLs based on values you read from files etc. Refer to Jmeter manual for more information.

Beware of network limitations.

Let's say you have 2 servers. One is running aiCache and the other one you use to generate the load from. Make sure you understand the network connectivity between the two. Are they on the same network/switch or are they a thousand miles apart? What is the max throughout you can have between the two?

A better case scenario would place both systems onto the same network, in the same datacenter and ideally on the same switch, at 1Gbps. Let's say the network switch is not oversubscribed and can deliver full 1Gbps in both directions (using full duplex links). What it means is that you can expect to pass about 100MBps in each direction.

Say the URL you're testing is such that the response is 20KB in size. You fire up ***ab***, point at aiCache instance at the other server, tweak the number of load generating threads for a while, but quickly grow frustrated – the aiCache doesn't seem to be able to handle more than 5000 RPS, no matter what you do. Both servers are barely registering any load during the test, so you suspect something must be seriously wrong somewhere. A smirk might cross your face: another overhyped product, clearly incapable of reaching stated performance numbers, you've seen plenty of those before. Guess what! Nothing is wrong, it is just that at 5000 RPS, each at 20KB of response payload, you have fully saturated your network! Re-read the previous sentence till you fully understand what is happening.

Now, if the 2 servers are in different datacenter, you can end with much smaller RPS numbers. In more severe cases, you'd become limited not only by available bandwidth, but also by latency.

Beware of load generator limitations.

Let's say you have 2 servers. One is running aiCache and the other one you use to generate the load from. Both are on the same network/switch and you do have full 1Gbps of bandwidth available. The response size is mere 40 bytes, with header and all, so using simple math, you expect to see around 250,000 RPS when performing the stress test.

In reality, however, you will likely discover that a single load client is incapable of generating enough load to load up aiCache server. You'll likely need 4 or more load clients, to get to 250K RPS numbers.

General performance improvement suggestions.

Enable Caching and Compression.

Assuming enough of content is cacheable, you will discover that in most cases, you are limited by bandwidth available between aiCache servers and your clients.

By all means make sure you compress everything that is compressible. This way you can expect 4-fold+ increase in RPS numbers.

Increase available network bandwidth.

Consider connecting aiCache servers at 10Gbps. A less expensive proposition is to use Ethernet trunking (aka NIC teaming) in load-balanced mode – so with 4 1Gbps interfaces you can have 4Gbps of available BW (assuming the switches/routers are not overloaded). Of course we assume this is the BW available all the way to the end users. For example, having aiCache servers connected at 2Gbps won't do you any good if you only have 1Gbps Internet link and all of your users are on the Internet.

Increase system-wide and per-process number of file descriptors.

Make sure to bump up number of open descriptors available to aiCache servers if you expect to serve any serious number of RPS, see aiCache User Guide for more information.

Another limit on max number of open connections might come from a *system-wide* limit. To see what it is set to and change it, read/write from **/proc/sys/fs/file-max** file:

```
# cat /proc/sys/fs/file-max
128000
# echo 512000 > /proc/sys/fs/file-max
# cat /proc/sys/fs/file-max
512000
```

Increase number of per-process open file descriptors via *ulimit* command.

```
ulimit –n 128000
```

Breaking the 64K open connections limit.

Operating system would normally impose a limit of maximum of 64K open connection per server IP address. While you're not likely to see it becoming an issue under most traffic conditions, extreme traffic volume might require maintaining higher number of open connections.

If that's the case, you'd need to setup multiple IPs on a single aiCache server (and remember that single of anything is never a good idea) or setup multiple aiCache servers to handle the traffic. You can then direct the traffic at multiple IP addresses via multiple DNS A-records.

Streamline handling of TIME_WAIT timeout.

When running such high-connection rate websites, you will discover that at any point in time you might have thousands upon thousands of connections in TIME_WAIT state. Now such connections do not translate to any extra load on aiCache, yet should they create a problem for your setup, you can try reducing number of such connection by turning on **client_linger** and **os_linger** options.

With these options set, the TCP/IP connection close takes a shortcut -instead of an orderly termination, a TCP/IP reset is sent instead and connection is disposed of immediately, without going through TIME_WAIT state. You must test this before enabling it in production setting. Some client browsers might not appreciate getting such TCP/IP resets none too much.

However, this should be much safer with origin server connections - as by the time aiCache issues a reset, it has obtained a complete response from origin servers. In addition, it is not only aiCache that will show reduced number of connections in TIME_WAIT state, origin servers will also see similar reduction. Yet again, please test before enabling it in production.

Alternatively and/or in addition , you can explore setting Linux's own TIME_WAIT interval to a lower value (some heavy traffic sites set these it to as low as 1 sec):

```
echo 5 > /proc/sys/net/ipv4/tcp_fin_timeout
```

Likewise, you may consider setting TCP_TW_REUSE to 1.

```
echo 5 > /proc/sys/net/ipv4/tcp_rw_reuse
```

You can also enable quicker reuse of sockets that are in TIME_WAIT state by setting TCP_TW_RECYCLE to 1. Note that the system default value is 0. As always, please re-test after enabling this setting.

```
echo 5 > /proc/sys/net/ipv4/tcp_tw_recycle
```

Client-Side Keep-Alive connections.

When dealing with client requests that are such that a number of them are likely to be issued in rapid succession, we recommend enabling client-side HTTP keep alive.

Normally aiCache does not force connection close after serving a response to a client, allowing clients to send more requests over already-established connection. This is knows as connection persistence or connection Keep-Alive feature. It has a potential to speed up user access to your web site by amortizing TCP/IP connection establishing overhead over many client requests.

Instead of closing client connection after serving a response, aiCache indicates to clients, via Keep-Alive HTTP header, its Keep-Alive preferences as to for how long the connection could be kept open by the client. However not all clients (read: browsers) may obey this hint. In order to prevent abnormally high number of open, yet mostly idle, client connections having to be maintained by aiCache and server's operating system, you can configure aiCache to drop idle Keep-Alive connections if there was no client input on particular connection for more than **maxkeepalivetime *seconds***. Default value: 10 secs.

You can limit maximum number of requests allowed to be served over a single Keep-Alive client connection - use **maxkeepalivereq** directive in global and/or website sections of the configuration file. Default value: 20 requests.

aiCache reports number of keep-alive requests served over each keep-alive client connection in its access log file. If you're kind of person that likes to fine tune stuff, try setting different increasing **maxkeepalivereq** values till you the reported number of served keep-alive requests stops growing. Doing so will speed up the loading of the page for your visitors.

Sometimes you know that a certain request URL results on in a response that is unlikely to be followed by additional Keep-Alive requests from the clients. In this case letting connection to go into Keep-Alive state is a straight waste of resources - as it is never used again. For such URLs you can set **conn_close** setting in the matching pattern section.

You can see gauge the effectiveness of client Keep-Alive connections by observing average number of client requests per client connection - it is reported both in Global and Website sections of Web self-refreshing monitoring pages. The higher the number, the better the experience for your visitors, however, larger number might mean more open connections on your aiCache serve, so you might need to adjust aiCache's client Keep-Alive settings to find a middle ground.

Server-Side Keep-Alive connections.

aiCache is capable of maintaining Keep-Alive connections to origin servers. Similar to client-side Keep-Alive connections, having OS-side Keep Alive connection allows you to speed up request processing time by amortizing TCP/IP connection establishing overhead over many requests. It is less beneficial however, compared to client-side Keep-Alives, as aiCache and origin servers are frequently located not only in the same hosting facility/datacenter, but are attached to the same switch - with latency in microseconds, not tens or hundred(s) of milliseconds as often is the case with client-side connections. However, if the origin servers are hosted in a geographically remote Datacenter, a significant latency away, consider using OS Keep-Alive, it will make responses much faster.

Please note that you *should test* Origin Server Keep-Alive feature before turning it on for production use. You must ensure that origin web servers support Keep-Alive connection (most do) and also configure it to persist for the reasonably long time and number of request, to realize maximum benefits.

To specify maximum per-origin-server number of open Keep-Alive connections to maintain, set it via:

```
max_os_ka_connections NNN
```

To specify maximum number of requests to be allowed per Keep-Alive origin server connection, set it via:

```
max_os_ka_req MMM
```

Please note that you might see a very long response time in case of conditional keep-alive requests to some origin servers. To fix this issue, simply set **max_os_ka_req** to 0, at global or website level:

```
max_os_ka_req 0
```

Most origin web servers are configured to only use a Keep-Alive connection for a few seconds, before discarding it and requiring opening of a new connection . The reasons for this include logic "resets" - to catch and stop any memory leaks in the application code and reducing number of processes and open connections that

operating system needs to maintain on origin server. You might want to fine-tune aiCache by setting **max_os_ka_time** parameter in global section match origin web server setting .

Please note that these OS Keep-Alive settings can be specified at global or website level, latter overriding global values.

You can see judge the effectiveness of Origin Server Keep-Alive connections by observing average number of OS requests per OS connection - it is reported both in Global and Website sections of Web self-refreshing monitoring pages. The higher the number, the faster it is to obtain a response from Origin Server, as it reduces (amortizes) overhead of TCP connection handshake. Again, the savings are most noticeable when aiCache and Origin Servers are located in different Datacenters.