Scaling a Content Delivery system for Open Source Software

Niklas Edmundsson

May 25, 2015

Master's Thesis in Computing Science, 30 credits

Supervisor at CS-UmU: Per-Olov Östberg
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

This master's thesis addresses scaling of content distribution sites. In a case study, the thesis investigates issues encountered on ftp.acc.umu.se, a content distribution site run by the Academic Computer Club (ACC) of Umeå University. This site is characterized by the unusual situation of the external network connectivity having higher bandwidth than the components of the system, which differs from the norm of the external connectivity being the limiting factor. To address this imbalance, a caching approach is proposed to architect a system that is able to fully utilize the available network capacity, while still providing a homogeneous resource to the end user. A set of modifications are made to standard open source solutions to make caching perform as required, and results from production deployment of the system are evaluated. In addition, time series analysis and forecasting techniques are introduced as tools to improve the system further, resulting in the implementation of a method to automatically detect bursts and handle load distribution of unusually popular files.

Contents

List of Figures

Chapter 1

Introduction

The Academic Computer Club[1] (ACC) is a student organization at Umeå University hosting a public file archive[2], most often called an FTP mirror, that distributes files for various open source projects such as Linux distributions. This category of site is commonly bandwidth-limited. Being located at a Swedish university, ACC has always had the privilege of being connected to the Swedish University Network (SUNET)[3], internationally acknowledged as a powerful and reliable research network [13]. With the announcement of the tentative plans for the next-generation SUNET network, with 100 gigabit connectivity in the near future [17], the ftp.acc.umu.se archive is yet again to face an interesting challenge in scaling.

The ACC network situation is different from the common ones, where external bandwidth is the limiting factor. In those cases solutions such as load balancers are viable. This is not possible in the ftp.acc.umu.se system, where any single component is unable to handle the bandwidth required. In order to overcome this issue caching is leveraged, and modifications of existing solutions are required to meet the system's bandwidth demands.

The first part of this work provides an overview of the ftp.acc.umu.se system, and modifications proposed to scale the system to meet demand. The benefits and drawbacks of this solution are summarized, highlighting the need to be able to automatically identify burst situations in order to dynamically redistribute the load when needed.

In the second part time series analysis and forecasting techniques are introduced, with the intention of showing how they can be applied to the problem of burst detection. An automated method of handling of files causing bursts is implemented and deployed in production on the ftp.acc.umu.se system. We present promising results from experiences gained during the latest Debian Linux operating system release.

[1] `http://www.acc.umu.se/`
[2] `http://ftp.acc.umu.se/`
[3] `http://www.sunet.se/`

The main contributions of this work are:

- Design and implementation of a cache subsystem optimized for large-file delivery, cooperating among multiple services while minimizing data duplication.
- Design and implementation of a cache-aware redirection subsystem, minimizing site-wide cache inflation by large files.
- Analysis and evaluation of results from multiple years of using the cache- and redirection subsystems in production.
- Formulation of a strategy to preprocess file transfer logs to obtain more accurate statistics.
- A survey of time series analysis and forecasting techniques, suitable for beginners coming from the computing science field.
- Design and implementation of an automated file burst detection and load redistribution system.
- A first analysis of results from the automated burst detection system when used in production during a large burst, caused by the latest Debian Linux release.

Information on open source tools used, both in the ftp.acc.umu.se system and during the work on this report, are listed in the Appendix.

Throughout this work the reader is provided footnotes with links to Internet sites with more information on a topic or term used. Note however that many references are Open access[4] as well, including links to the Internet resources.

[4] `http://en.wikipedia.org/wiki/Open_access`

Chapter 2

Background

ACC has been running ftp.acc.umu.se, a public file archive, since the computer club was founded in 1997[1] but it was not until the site became the official Swedish Debian[2] Linux operating system mirror in May 2000 that it started to get any noticeable amount of traffic. The mirror grew in popularity and soon ACC could not expand the server machine, a SUN 690MP with 2x60 MHz CPU's and 80 GiB of storage, to meet the demands in storage capacity and bandwidth. In 2001 the archive had to be scaled to more than one server to overcome this limitation.

The solution chosen at the time was based on a classic model with a cluster of servers interconnected by a high-speed network, using a high-performance distributed file system. The main reason for this decision was the availability of hardware, as ACC got hold of a couple of nodes from a decommissioned IBM SP cluster located at the Center for High Performance Computing (PDC) at the Royal Institute of Technology (KTH). At the time the university had a 155 Mbps link to SUNET, which was later upgraded to 622 Mbps to meet demands before GigaSUNET with 2.5 Gbps came online in 2002 [35].

When it became obvious that there would not be any easy upgrades to this solution (there were no newer IBM SP systems in Sweden at the time) work began on designing a replacement. The goal was an architecture that would leverage the kind of hardware being donated to the computer club while meeting the ever increasing bandwidth demands. At the time the plans for OptoSUNET with 10 Gbps connectivity to universities were known, and the file archive architecture would ideally scale to those bandwidths.

The choice fell on using separate servers for storing the content and doing the actual distribution of data, and to leverage caching in the publicly visible servers to work around the inevitable bandwidth bottlenecks of such a solution. The top caching layer would be a RAM file system cache implemented in the operating system of each server, but given the size of the content working set there was no chance enough data would fit into the comparatively small amount of RAM of those servers. To alleviate this, a disk-based caching layer in each server was needed. It turned out that the open source cache solutions available at the time, with Squid[3] and the caching subsystem of

[1] The predecessor, *Teknologsektionens Datorförening* (TSDF) dates back to 1994

[2] `http://www.debian.org/`

[3] `http://www.squid-cache.org/`

the Apache HTTP Server Project[4] (Apache httpd) being the most prominent, were all heavily geared towards small-file many-request workloads and were thus not well suited for the large-file oriented workload of the ftp.acc.umu.se site.

To overcome these limitations it was decided to adapt the caching subsystem of Apache httpd. The main factors leading to that decision were performance and architectural simplicity. Apache httpd was able to deliver the performance needed, while the code base for the caching subsystem by itself was rather small and easy to understand. This was a prerequisite to have a chance at realizing the vision of utilizing the disk cache layer not only for the HTTP protocol [12], but also for the FTP protocol [28] and the rsync application [33].

This work implements the components needed, which were taken into production during 2006-2008. This enables drawing definitive conclusions on the performance of the components over time.

The workload pattern of today is surprisingly similar to the one identified 10 years ago. The HTTP protocol still accounts for the majority of usage with 94% of the delivered data, the rsync application is still being used to synchronize file sets between mirror sites, and the FTP protocol is mostly used as a last resort. Large files are still the limiting factor when it comes to bandwidth, with files larger than 1 MiB accounting for more than 97% of the sent data.

As newer and more capable server hardware has been donated to ACC during the years, fewer servers are now required to meet demands. While it is likely that the next generation donated hardware will be able to meet demands posed by the current 10 Gbps connection to SUNET, clustering with efficient caching will still be required to serve the upcoming 100 Gbps SUNET connection.

[4] `http://httpd.apache.org/`

Chapter 3

Architecture analysis

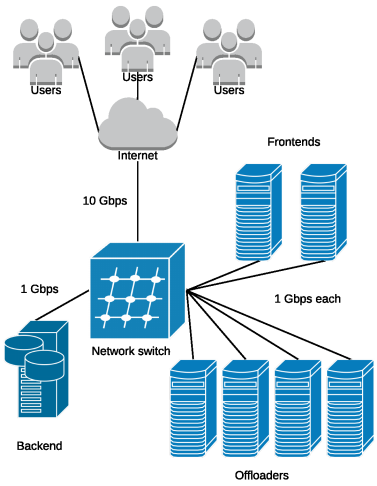
In this chapter an overview of the ftp.acc.umu.se system is provided, along with an in-depth discussion on the motivation and design of the adapted components. A summary discusses benefits and drawbacks of the system, where a scaling limitation is identified and addressed further in Chapter 4.

3.1 Overview

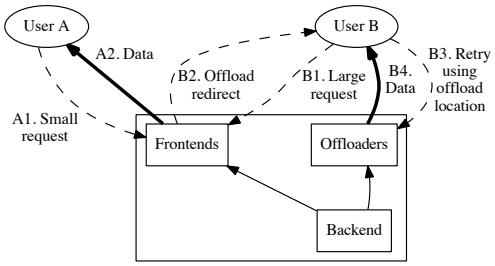
The current incarnation of the file archive at ftp.acc.umu.se is comprised of the following classes of server machines, described using the vocabulary established at ACC:

- Backend - the file server where all data is stored. After having initially experimented with multiple Backend servers it was concluded to have only one, due to the administrative complexity of balancing the use of multiple servers. The file system is exported using the Network File System (NFS) version 4 protocol [31] and mounted on all other servers.
- Frontends - publicly accessible, these are the servers reached when users contact them by the ftp.acc.umu.se name. They serve requests directly, with larger files being redirected to Offloaders using HTTP redirects when possible.
- Offloaders - servers delivering larger files as redirected from the Frontends.

Figure 3.1 shows the ftp.acc.umu.se system. The logic diagram in Figure 3.1b gives an overview of the system function from the user perspective, when contacting the system to retrieve files. As server and client software interact the system functions as a homogeneous unit, making the fact that the system consists of multiple components opaque to the end user. Looking closer at the network diagram in Figure 3.1a it is clear that all components, including the Backend system, is limited by a 1 Gbps network connection compared to the 10 Gbps uplink connecting to the Internet. Without a good caching solution this would severely limit the system performance.



(a) Network diagram illustrating the external network connectivity providing higher bandwidth than the components of the system.



(b) Logic diagram depicting the two common use cases. *User A* requests a small file while *User B* requests a large file. Dashed arrows represents signalling while bold arrows shows data flow to end users.

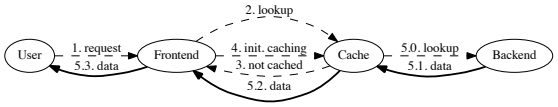
Figure 3.1: Overview of the ftp.acc.umu.se system.

Incoming requests from end users are handled by the Frontends. If the request is to be handled locally, as determined by protocol, file size and special considerations regarding client support, the request is handled as a normal request by the server subsystem in accordance to Figures 3.2a and 3.2b. The cache subsystem components `mod_cache_disk_largefile` (for HTTP) or `libhttpcacheopen` (for FTP and rsync) are involved as required.

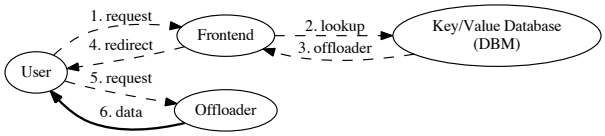
The HTTP protocol can send a reply to a client, redirecting it to another resource. This is used to implement true offload handling by having the client reissue the request to the server providing the resource, enabling use of the aggregated bandwidth of the Offloaders. The Apache httpd `mod_rewrite`[1] subsystem is used to drive the offload handling. If the request is a candidate to be handled by an Offloader, a look-up is made into a key/value database to see whether a request for this file has already been evaluated by the redirection subsystem. If found, redirection is handled as illustrated in Figure 3.2c. If not, the request is passed on to the redirection subsystem `redirprg.pl` for evaluation, with the decision stored into the key/value database to avoid doing the evaluation more often than necessary.

[1] `http://httpd.apache.org/docs/current/mod/mod_rewrite.html`

(a) Cached file. A lookup is made to the local cache and the data found is sent to the user.



(b) Uncached file. Data is fetched from the Backend in order to populate the cache and send to user.



(c) Large files. The assigned Offloader is retrieved from a key/value store and a reply is sent to the client, instructing it to contact the Offloader in order to access the data.

Figure 3.2: Cache subsystem, data flow in the common cases. Graphs originally by Mattias Wadenstein for `http://ftp.acc.umu.se/about/`

To avoid overflowing the cache of the Offloaders each file is served by a specific Offloader. The inode number (the index node number uniquely identifying a file in a file system) of the Backend file is used as the base for selecting the Offloader to assign the request. This avoids aliasing effects by multiple site names, file/directory names etc, which can cause requests for the same backing file to be sent to multiple Offloaders. The requests are split evenly among the Offloaders based on the file inode number. This has the potential of causing hot spots if multiple popular files happen to be assigned the same Offloader, a fact addressed further in Chapter 4. As assignments are deterministic there is no need for communication between the `redirprg.pl` instances on the different servers.

In a similar manner, the `mod_cache_disk_largefile` and `libhttpcacheopen` cache subsystem components also use the inode of the backing file to avoid storing multiple copies of the same file. In the case of the `mod_cache_disk_largefile` Apache HTTP module, the cache has two levels. A cached header entity is keyed on the URL/URI of the request, and the header entity then contains a reference to the local cached file of the request. The preload wrapper library `libhttpcacheopen` only operates on the file system level and caches/redirects accesses for files on the Backend file system into files residing in the local cache.

3.2 Components

This is a more detailed analysis of the custom components in the ftp.acc.umu.se system, responsible for realizing the logic shown in Figure 3.2. Identified major issues and considerations leading up to the current design are listed, together with results of using the system in production for a long period of time. A summarizing discussion wraps up the section. For more information on standard components used, see Appendix A.

3.2.1 mod_cache_disk_largefile - Apache httpd disk cache

The `mod_cache_disk_largefile` module for Apache httpd is a major adaption and re-engineering of the Apache httpd `mod_disk_cache`[2] module, which was later renamed `mod_cache_disk`[3] in the 2.4-release of Apache httpd. These modules all use the infrastructure provided by the Apache httpd `mod_cache`[4] module to handle the logic of caching, leaving them to handle backing media storage/retrieval.

The original `mod_disk_cache` module is heavily geared towards a workload comprised of many requests for small files. When exposed to what can be seen as the opposite, a workload of few requests for large files, some of the design decisions made proves to be detrimental to the performance and behaviour under such workloads.

The original design of `mod_disk_cache` stores a file to the cache before starting to reply to the client. During retrieval of large files, most users, and client software, gives up and retries before the caching operation is complete. Each request of an uncached file triggers a caching operation, regardless of whether that file was already in the process of being cached or not. This causes a huge inflation of the space used on the cache backing store, and makes caching very slow due to multiple processes fighting for bandwidth.

Considering future plans, the backing store layout not being suitable to handle sharing of already cached content with other access methods is also identified as an issue. This is due to the fact that the data stored is keyed on the URL of the file, a fact also responsible for causing duplicates of the same backing files to be stored in the cache.

To address these issues the logic of the storage and retrieval of cached files is re-engineered with the following design key points:

- The first access to an uncached file triggers caching.
- Data is transferred to clients as files are cached.
- Multiple requests for files are served from cache as files are being cached.
- No explicit locking using separate lock files. Rely on the POSIX [15] O_EXCL flag to open a file exclusively for writing.
- The cache backing store should allow for cooperation with other services.

Apache httpd has an internal API based on APR-util [3] that allows a representation of content in any format. Simplified, the API allows for representing a piece of data, a bucket[5], with all pieces combined into a complete data stream in a bucket brigade[6]. To solve the problem of being able to send data to the client while a file is being cached a bucket type used to represent a cached file is implemented. In order to leverage optimizations in Apache httpd for delivering regular files, the most common use case, the buckets morph into regular file buckets as file data becomes available. While this allows delivering data with virtually no performance impact during caching, the process of determining whether new data is available is performed for each concurrent request to files being cached, which can waste server resources.

[2] `http://httpd.apache.org/docs/2.2/mod/mod_disk_cache.html`
[3] `http://httpd.apache.org/docs/2.4/mod/mod_cache_disk.html`
[4] `http://httpd.apache.org/docs/2.4/mod/mod_cache.html`
[5] `http://apr.apache.org/docs/apr-util/1.5/structapr__bucket.html`
[6] `http://apr.apache.org/docs/apr-util/1.5/structapr__bucket__brigade.html`

The underlying POSIX API [15] used on Unix/Linux ensures that content written to an unbuffered file is visible atomically as it is written, so care has been taken to write files in consistent chunks using `writev()` or similar functions. Timeouts are used for certain operations to achieve a robust system. As an example, a request determines that another process is currently performing a caching operation, and it waits for valid content to show up in the appropriate header file. If that caching process fails, for example because the Backend server was restarted, that file will never have valid content. For cases like this, a processing timeout ensures that the offending file is removed so the cache operation can be retried. As this processing is abstracted from the request, the only noticeable effect to an end user is the response being slightly delayed.

In order to have a cache backing store layout that allows cooperation with other services, separate indexes are used for storing the header and the matching file, called body in HTTP [12, section 4.3] and other protocols. An early approach of using the backing file name as index for the body reduced the data duplication somewhat, but it was concluded that there was still a considerable amount of data duplication in the cache. Experiments with indexing files based on content were abandoned due to not performing as required. Finally it was found that indexing the body based on the inode numbers of the files on the Backend file system solves the issue with minimum performance impact.

While there have been numerous other small improvements and fixes in the `mod_cache_disk_largefile` module to obtain the best possible cache efficiency and performance, the items detailed here are the key ones for the cache to be useful in a bandwidth-constrained environment handling mostly large files such as the ftp.acc.umu.se file archive.

3.2.2 libhttpcacheopen - using the httpd disk cache for other services

In order to be able to leverage the same cache backing store for other services, namely FTP and rsync, a wrapper library is implemented that detects accesses to the Backend file system and redirects those to the cache backing store.

The subsystem is comprised of two components, `libhttpcacheopen` and `copyd`. `libhttpcacheopen` is the actual wrapper library that injects itself between the application and the operating system using the LD_PRELOAD mechanism of the Executable and Linkable Format (ELF) dynamic linker/loader ld.so [20] [21] used in Unix-like operating systems. `copyd` is a background process, daemon, that handles background caching of larger files in order for the service wrapped to be able to send data while a file is being cached.

The design and implementation of `libhttpcacheopen` is rather straight-forward. All library routines such as `open()` and `read()` that access files are identified and required functionality to utilize the cache is implemented. This includes redirecting accesses to files, initializing caching if not already cached, etc. While the general algorithms are the same as the ones developed for the `mod_cache_disk_largefile` Apache httpd module, the code has to be ported from the APR API [3] to the standard POSIX API [15] in order to reduce the number of library dependencies and interactions that preloading libraries can induce.

The `copyd` daemon simply runs in the background, listening to a Unix domain socket for cache-requests to handle. It uses the same code base as `libhttpcacheopen` for copying/caching files, keeping the `copyd`-specific code base to a minimum.

Even though the overall design is rather simple, there are a number of small details that have to be taken care of for the implementation to work properly. As an example, Linux uses inlined wrapper functions for the `stat`-family library calls. In order to catch those the `__xstat`-family must be wrapped instead.

3.2.3 redirprg.pl - redirection subsystem

The ftp.acc.umu.se system started out with a number of servers all providing access to clients using FTP, HTTP and rsync. Load distribution was achieved by having the DNS [24] name ftp.acc.umu.se point to all target servers. This is called DNS Load Balancing [5], or more commonly Round-robin DNS[7]. The HTTP protocol accounts for most of the accesses, with close to 95% of the delivered data[8]. As using Round-robin DNS for HTTP load distribution proved inadequate due to large site-wide cache inflation, a way of doing cache-aware load distribution was called for.

The HTTP protocol provides a mechanism that uses the Location response-header to inform a client where the requested content can be found [12, section 14.30]. This operation is often called a HTTP redirect based on the Apache httpd configuration directive Redirect[9]. The concept is also known by many other names, for example URL redirection or URL forwarding[10]. This mechanism can be used to implement a load distribution scheme where the client software accesses the server providing the resource directly, avoiding congestion, while the user only have to access the site with a known address.

To provide a degree of load distribution, and alleviate the issues of Round-robin DNS and cache inflation, manually assigned HTTP redirects were used. This quickly proves impractical on a large scale, and a method of automatically doing cache-friendly HTTP load distribution is needed.

The Apache httpd `mod_rewrite` subsystem provides mechanisms to do table look-ups called RewriteMap[11]. These maps can be based on static data, such as text files, dbm files (standalone key/value databases) or SQL databases; or dynamic such as random values from a list or driven by external programs. Using these building blocks a proof of concept redirection subsystem is designed and implemented using the Perl [32] script language. The proof of concept turns out to have good enough performance to be able to handle loads multiple orders of magnitude higher than seen on the ftp.acc.umu.se system.

The main goal of the ftp.acc.umu.se redirection subsystem is to increase system efficiency and capacity by doing cache-aware load distribution. The same scheme of using the backing file inode numbers, as used in the `mod_cache_disk_largefile` Apache httpd module, is used to define mappings between resources requested by users and offload

[7] `http://en.wikipedia.org/wiki/Round-robin_DNS`
[8] `http://www.acc.umu.se/technical/statistics/ftp/index.html.en#user`
[9] `http://httpd.apache.org/docs/2.4/mod/mod_alias.html#redirect`
[10] `http://en.wikipedia.org/wiki/URL_redirection`
[11] `http://httpd.apache.org/docs/2.4/rewrite/rewritemap.html`

targets providing those resources. Using this scheme a mapping function that produces the same mappings on all instances of `redirprg.pl` is implemented. This is made possible by the fact that the inode numbers of the backing files are identical on all servers.

`redirprg.pl` implements a handler for a prg[12] RewriteMap. This is an external program that given a key (a file name in our case) returns a value (an offload target host in this setting). Static map look-ups in the Apache httpd `mod_rewrite` subsystem are heavily optimized by using in-memory caches and other methods. Writing the request/reply pair to a dbm file as well enables taking advantage of those optimizations. An optimized redirection subsystem is implemented by using Apache httpd `mod_rewrite` directives to first check if the resource has been previously requested, and use a cached reply if that is the case. If not, a look-up is made to the `redirprg.pl` subsystem which is subsequently cached and used to reply to the user with a suitable redirection to the target server providing the resource.

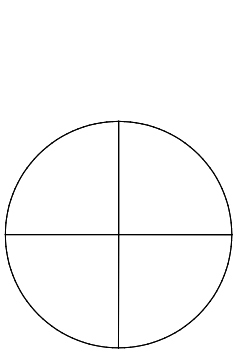
Having solved the major implementation hurdles by leveraging `mod_rewrite` we list the major requirements of the redirection subsystem:

- Cache-aware mapping, assign all instances of a file to the same offload server.
- Consistent mapping, to avoid the need for communication between the `redirprg.pl` instances running on different Frontend servers.
- Detect if Offload servers are offline/overloaded.
- Minimize impact of missing servers, if mappings change too much this causes caching storms which effectively can bring the Backend file server to a standstill.
- Allowing manual/static mappings, to single and multiple targets.
- Detect if files change inode number, this happens when files are updated.

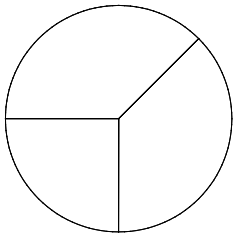
The major challenge is having a mapping scheme that is robust in the face of server outage. The mapping scheme used in this work is named the Pie-Chart Algorithm. It stems from discussions with fellow ACC system administrators on how to solve this problem in a way that is easy to understand and straight-forward to implement and debug. The basic idea is to assign each offload server a slice in a pie-chart, and in the event of an outage splitting that slice in half and assigning those slices to the closest neighbors. This provides a method that changes as little as possible of the mapping should a server go missing while redistributing the load in a predictable manner. The files are mapped to static positions in the pie-chart by having a fixed number of very small slices for this purpose. Although similar to Consistent Hashing [18] [19], the Pie-Chart Algorithm is not based on that work.

To illustrate, Figure 3.3a illustrates four offload target servers and Figure 3.3c the possible file placement positions. Should the upper-right target go missing the neighbor targets will grow in that direction to share the load as shown in Figure 3.3b, which will cause caching activity on those targets. When the missing target returns it will usually have most files already cached, and can resume its duties without having to re-cache all data. In an overload situation this sequence can repeat itself multiple times as popular content becomes cached.

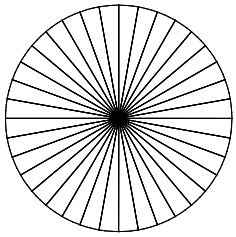
[12] `http://httpd.apache.org/docs/2.4/rewrite/rewritemap.html#prg`



(a) A pie-chart with four slices, illustrating four servers.



(b) A pie-chart with the upper-right slice missing, the neighboring slices growing to fill up the gap, demonstrating rebalancing should a server go missing.



(c) A pie-chart with many slices, depicting the placement of files in the pie chart.

Figure 3.3: Pie-charts for illustrating the redirection subsystem mapping scheme.

Static mappings with a single file mapping to multiple targets proved to be a challenge due to the fact that a key/value lookup can only provide a single offload target (value) per file (key). This is handled by periodically rotating the target returned for those lookups. For the ftp.acc.umu.se workload with large files and large transfer times a decent load distribution is achieved when rotating the targets a few times per minute.

Detecting targets missing or overloaded is done as a sub process to `redirprg.pl` in order not to interfere with responding to incoming queries. The current method is to send HTTP HEAD requests [12, section 9.4] for a known file on each target and flag the server as down if the response takes too long or fails, requiring a much quicker successful response before flagging the server as up again to avoid unnecessary flapping.

As all mappings are stored in a dbm file it is easy to update the mappings in response to events such as servers going offline/online, updated files causing new inode numbers, changing targets for static mappings etc. This comes with a small performance impact as `mod_rewrite` needs to read the updated entries into memory. That impact is however small compared to having to feed the requests into `redirprg.pl`, which has to be done in a atomic manner as `mod_rewrite` prg RewriteMaps can only process a single request at a time.

3.3 Results

Experience from multiple years of service indicates that the basic design of the system is sound. Table 3.1 shows bytes sent compared to bytes transferred from the Backend system. As can be seen the solution performs as required, with the caching subsystem reducing the load on the Backend system be nearly an order of magnitude. There are fluctuations over the years, due to accounting errors (corrupted RRDs caused by server

crashes), changes in workload, software bugs and software enhancements. The long term trend is a ten-fold difference in the amount of data transferred to end users compared to the amount of data transferred from the Backend system.

Year	data sent (TiB)	Backend transfer (TiB)
2007	1424	367
2008	3198	371
2009	4022	270
2010	4586	310
2011	5900	425
2012	5199	568
2013	4804	527
2014	5127	717
2015-01-01 - 2015-05-15	2105	191

Table 3.1: Data sent to end users from the ftp.acc.umu.se system compared to data transferred from Backend, gathered from `http://www.acc.umu.se/technical/statistics/ftp/monitordata/`

Translating this to network bandwidth, the system is capable of transforming the 1 Gbps bandwidth provided by the Backend server to close to 10 Gbps in outgoing bandwidth, as displayed in Figure 3.4. We expect this system-wide cache efficiency ratio to apply for the future as well, as the workload pattern has proven to be quite stable over the years.

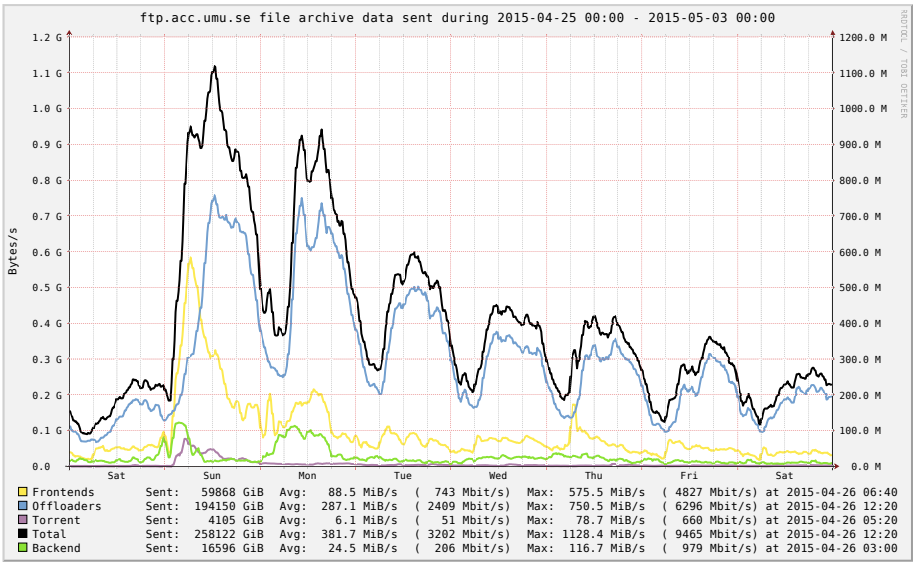


Figure 3.4: Transfer summary during the Debian 8 release, showing the system capability of saturating the available network bandwidth.

3.4 Summary and Discussion

Caching is essential in order to build a cost-effective infrastructure for content distribution. We have shown that by designing and implementing a few select components providing large-file oriented cache functionality, and combining them with standard server software components such as Apache httpd [2], vsftpd [11] and rsync [33], we can adapt off-the-shelf components into a solution highly optimized for open source content delivery with caching adapted to the needs of the ftp.acc.umu.se site.

3.4.1 Limitations

While meeting the intended system goals, our proposed solution has some limitations. For example, there is no offload support for FTP and rsync. These protocols do not provide support for telling the client to contact another server to retrieve the requested content, which means that the logic outlined in Figure 3.1b can not be realized. This results in traffic always having to pass through the server the client initially made contact with, even if a load balancing scheme is used behind the scenes to split the load over multiple servers. While this is unfortunate, there is not much we can do about it and we do not believe that this situation will change for the following reasons:

- The FTP protocol has the notion of third party transfers[13] which is used in, for example, GridFTP[14].
- Third party FTP transfers are disabled in general as it is considered a security risk in conjunction with unauthenticated FTP access[15].
- The rsync application has no notion of offload support. This is understandable due to the common bottleneck usually being file tree traversals when rsync sessions obtain lists of files, and this is observed to not scale well with distributed file systems.

For ftp.acc.umu.se, this means that the FTP and rsync traffic is served by the Frontend servers. As listed in Table 3.2 the amount of data for FTP is nearly negligible, and while rsync accounts for a greater share of transferred data it is still manageable. Rsync is mainly used for synchronizing file sets between mirror sites so the number of users is relatively constant, with the amount of data sent being coupled to the archive size and update rate. If the number of end users increase they will mostly access the archive using the HTTP protocol, for which our offload strategy applies.

Application protocol	bytes sent (% of total)
HTTP	94%
rsync	5%
FTP	1%

Table 3.2: Statistics of bytes sent using different application protocols for the time period 2015-03-01 - 2015-03-30, gathered from `http://www.acc.umu.se/technical/statistics/ftp/index.html.en`

[13] `http://en.wikipedia.org/wiki/File_eXchange_Protocol`
[14] `http://en.wikipedia.org/wiki/GridFTP`
[15] `http://en.wikipedia.org/wiki/FTP_bounce_attack`

System administrators do need to monitor the system for changes, as a small increase in system usage might cause passing a threshold making the cache infrastructure perform sub-optimally. Such an example was uncovered during the latest Debian Linux release, where a slight increase in the data set caused the cache on one of the Frontend servers to be overflowed by the daily rsync sessions to update other mirror sites. This in turn caused Frontend servers to continuously contact the Backend server to handle requests, a behavior that is not good for overall system cache efficiency.

Looking into the adapted components, `mod_cache_disk_largefile` is resource-hungry during caching. This is an implementation-specific artifact due to the historic processing model of Apache httpd. This can today be solved by leveraging the `inotify` API [16] and use event notifications for file changes instead of polling.

The redirection component `redirprg.pl` is responsible for offloading requests to the Offload servers. There are two known limitations in this area. One limitation is the fact that `mod_rewrite` sends requests one by one to the `redirprg.pl` program RewriteMap. While this limitation exists we do not see this affecting the workload of ftp.acc.umu.se in the foreseeable future. The other limitation is the fact that `redirprg.pl` assigns a single target for each file. While there is a mechanism to do manual assignments this is only used in extreme circumstances today due to the effort required. This limitation will be addressed in more detail in the following chapter, where we outline and implement a method of automated detection and load distribution for such files.

The stable mapping scheme of our Pie-Chart Algorithm is comparable to Consistent Hashing [18]. Although the schemes are similar in design, we believe Karger et al. [19] to be more refined with faster algorithms. To scale the system beyond tens of Offloaders, an unlikely scenario for the ftp.acc.umu.se system, the redirection component `redirprg.pl` might perform better by using an optimized library for Consistent Hashing such as Hash::ConsistentHash[16] or Set::ConsistentHash[17]. An alternative is to use a distributed key/value store such as Redis[18], which allows implementing a redirection subsystem with a coherent system view. This would enable using more advanced schemes of load distribution.

[16] `http://search.cpan.org/perldoc?Hash::ConsistentHash`
[17] `http://search.cpan.org/perldoc?Set::ConsistentHash`
[18] `http://redis.io/`

Chapter 4

Improving load balancing with time series analysis and burst detection

This chapter takes a closer look at the problem of bottlenecks caused by popular files, as identified earlier in Section 3.4.1. The available data sources in the system are investigated, along with anomalies in the data and suggestions on how to handle that issue. Time series analysis and forecasting techniques are surveyed with the intention of investigating how they can be applied to the problem. Given a usable data source, and techniques to apply to the problem at hand, a design of a burst detection and load redistribution subsystem in the context of the ftp.acc.umu.se system is detailed. This is followed by results from using the system in production, concluded with a summarizing discussion.

4.1 Problem Statement

Today the ftp.acc.umu.se file archive is dependent on manual detection and handling of bursts and bottlenecks caused by excessive popularity of single large files, the most common example being install images downloaded by users in conjunction with the release of a new version of a Linux distribution. This manual detection requires administrators to be very alert and constantly monitor the system if they are to have a chance to react in time. In practice this only happens when alerted of big events beforehand, leading to bursts caused by unannounced releases to be missed. In order to solve this an automated solution is needed.

As a first step in order to devise an automated solution we study the possibility of detecting load bursts. While the problem might seem trivial at first, daily/weekly/seasonal variations in the workload makes reliable detection challenging. To tackle this we investigate Time Series Analysis, a branch of mathematical statistics where there are tools to handle these kinds of issues.

There is also the question on whether the cause of the bursts can be detected in enough detail to provide an automated response, or if obtaining such a detailed answer would require too much computing/analysis power to be feasible on a production system. As the automated response would happen in the redirection subsystem this is quite essential for the solution to be practically useful.

The first challenge is modeling the current system to establish a baseline for normal system load. This is preferably done using log data detailed enough to make it possible to later answer the question of what file(s) are causing the peak, as this will be needed to formulate an automated response.

Even though there are a lot of logs available, both current and historical data, those logs might not be appropriate for this task. As the system is bandwidth limited, the logs are geared towards this fact, with only the essential transfer statistics logged. A complicating factor is that most subsystems tend to write log entries when transfers are complete. This makes real-time detection of bursts hard considering the transfer times of large files. It has also been shown that the distribution of file transfer times across the Internet is heavy-tailed [25, pp. 35-36], which gives a high probability for large transfer times. If the current logs proves inadequate the system needs to be augmented with additional logging to solve the task. Related to this there are also a few known usage patterns caused by a group of client software called Download Managers that can cause log inflation and severely skew analysis.

Having a baseline enables burst detection, the process of detecting load peaks due to sudden increased popularity. Depending on how the solution for this problem performs this can be verified on the production system, on a scaled down test system or on a simulation system.

4.2 Data sources

Systems usually have a logging facility that logs individual transfers, and the most common is for log entries to be written upon completion of the request. This is true for all components of the ftp.acc.umu.se system.

Logs are written locally on each server and collected daily to a central server to be able to generate statistics[1]. These logs have been saved from 2002 onward, and for historical reasons the logs collected from Apache httpd are in the `xferlog`[2] format. This log format details transfers made, including transfer times and actual amount transferred.

The fact that log entries are written upon completion were early on in this work suspected to be an issue, as a burst detection system needs to have a reasonably quick reaction time. There is also the fact that while the logs are written on the Offloaders, the incoming requests are handled by the Frontends where actions needs to be taken upon detecting a burst condition.

To get more details on offload operations, the ftp.acc.umu.se system was augmented with a dedicated offload-log that logs each request resulting in an offload operation. Added benefits for this work are the facts that the logs are stored on the Frontends where the

[1] `http://www.acc.umu.se/technical/statistics/ftp/`
[2] `http://manpages.ubuntu.com/manpages/trusty/en/man5/xferlog.5.html`

offload operations happen, and that log entries are written when the offload/redirect reply is sent to the client. This enables implementing automated processing and reaction to events without the need to transfer log files between components of the system.

In addition to this there are also network bandwidth logs from all servers involved. These are stored as Round Robin Databases (RRDs) using RRDtool[3] and shown on the statistics web page[4]. RRDs has the benefit of being able to store statistics for long time periods in comparatively small files. This is done by aggregating data points for older data in larger time steps as configured upon setup of the RRD.

4.3 Simulating server network bandwidth using transfer log files

In order to construct a model for server bandwidth using log files, a simple simulator was implemented that, given log records in `xferlog` format, emits a time series with simulated network transfer rates. Statistics such as average transfer rate and maximum transfer time are also generated. The simulator falls into the category of a discrete-event simulation (DES) [4] in a reduced form, as only the processing of the transfers need to be simulated.

The `xferlog` records has information on transfer start time, duration and file size. However, they are logged at the completion of the request which means that records have to be read and then ordered by start time during processing. The time step resolution chosen is one second, based on the `xferlog` format time stamp resolution. The granularity of the time series output defaults to 60 seconds to keep the resulting files in a manageable size.

To test the simulator, logs from a day when a burst occurred was chosen. Figure 4.1 displays a comparison of the simulated network transfer rate and the measured transfer rate. As has been mentioned earlier the network transfer rates are logged in RRDs, and are configured to store data with high resolution, small time steps, for the last 12 hours only. The data are extracted from the system more than a month later, yielding a much lower resolution hiding most details. The spikes only shown in the simulation data during the initial burst period are attributed to the fact that the offload target server became overwhelmed by the number of concurrent requests to an uncached file, causing the redirection subsystem to flag the server as down. This issue was investigated further to reveal other configuration errors as well, leading to a revised configuration of the RRDs used for logging network traffic. Due to these configuration errors, the measured network transfer rates logged before the reconfiguration can not be trusted to reveal short bursts.

[3] `http://oss.oetiker.ch/rrdtool/`
[4] `http://www.acc.umu.se/technical/statistics/ftp/monitordata/`

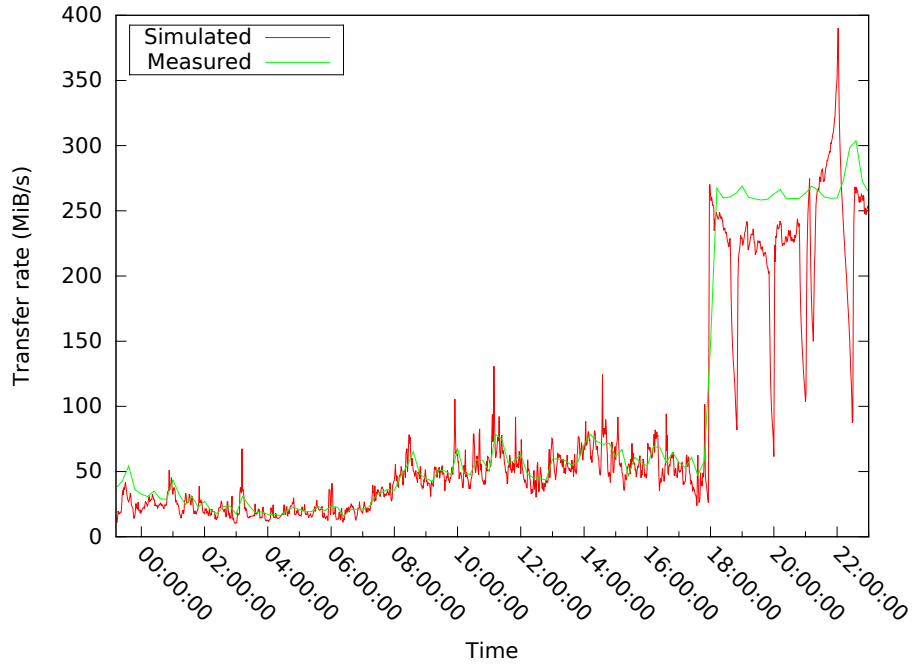


Figure 4.1: Simulated and measured network transfer rate of gensho.acc.umu.se on 2015-01-21.

Statistics output by the simulator can also be used to strengthen the argument that the transfer logs are an unsuitable base for building a burst detection system, due to the fact that log entries are written upon transfer completion. The run illustrated in Figure 4.1 yields an average transfer rate of approximately 84 kiB/s[5], with a median transfer rate of approximately 22 kiB/s. In addition to showing that the transfer rates during these conditions are not normally distributed, it indicates that most transfers are long running. This causes a long delay between transfer initiation and writing the corresponding log entries. As an example a 3.7 GiB DVD image takes 13 hours to transfer at 84 kiB/s and 48 hours at 22 kiB/s, potentially causing a large delay before the log records are written.

Early detection of bursts are important when dealing with long running processes where load balancing happens upon process initiation, at the start of transfer in this use case. As an example, a DVD image containing a freely distributable movie is released. The movie is long-awaited by its fans, who starts to download it immediately within the first four hours of release causing a huge burst taxing the capacity of a single Offload server. The transfer logs incur an average 13 hour delay between start of transfer and writing the corresponding log record. In this case, an automated burst detection based on the transfer logs would react far too late to have any effect.

The transfer logs incur unavoidable delays that severely impacts the usefulness of an automated burst detection scheme. This prompts investigating whether other logs are better suited as data sources for burst detection.

[5] `http://en.wikipedia.org/wiki/Data_rate_units`

4.4 Using the offload-log as a base for burst detection

The offload-log entries are written when the offload reply has been sent to the client, which means that the information is available for a burst detection system at the same time as the client receives the reply directing it to an Offloader. The big drawback however is the fact that at this stage nothing is known of the size of the transfer, and this needs to be taken into consideration as the system is bandwidth limited. As an example, a transfer a 4 MiB MP3 file has less long-term impact than a transfer of a 4 GiB DVD image.

Well behaved clients will normally transfer the whole file in one request but some client software, commonly Download Managers, can issue massive amounts of requests in order to transfer a file in multiple small chunks. While the HTTP protocol allows for partial transfers, previous experience shows that some Download Managers neglect to specify the accurate length of the transfer beforehand and just closes the connection when the appropriate amount has been received. This means that headers indicating partial transfers can not be seen as a reliable indicator of the actual amount of data to be transferred. The unfiltered data in Figure 4.2 illustrates a typical 100-fold inflation in offload traffic caused by this category of clients. For these graphs the offload transfer rate is estimated as a moving average over 30 minutes (1800 seconds).

For the purpose of burst detection, processing only the first request seen for a file from a specific IP address is proposed. While this would hide requests for the same file from multiple people sitting behind NAT or proxy connections, it would prevent a single client causing a 100-fold inflation of requests. Keeping a table of IP address and file pairs in memory will require a manageable amount of space, in the order of 200k entries to keep a table for the last 24 hours according to past logs.

The size of the requested file is assumed to be the amount of data to be transferred. Partial transfers to a single client occur, as some Download Managers are able to download single files from multiple sites simultaneously. However, previous investigations on the ftp.acc.umu.se production systems indicate those transfers to represent a negligible fraction of transfers made. The `xferlog` log format currently used prohibits doing this analysis on historical data.

The results of applying such a filter is shown in Figure 4.2. It can be seen that offloaded traffic is slightly inflated compared to the measured transfer rate, but roughly follows the same daily cycle. Had the traffic been underinflated, a significant presence of NAT/proxy transfers of identical files could have been suspected. This shows that the correct decision in this use case is to filter duplicates as shown.

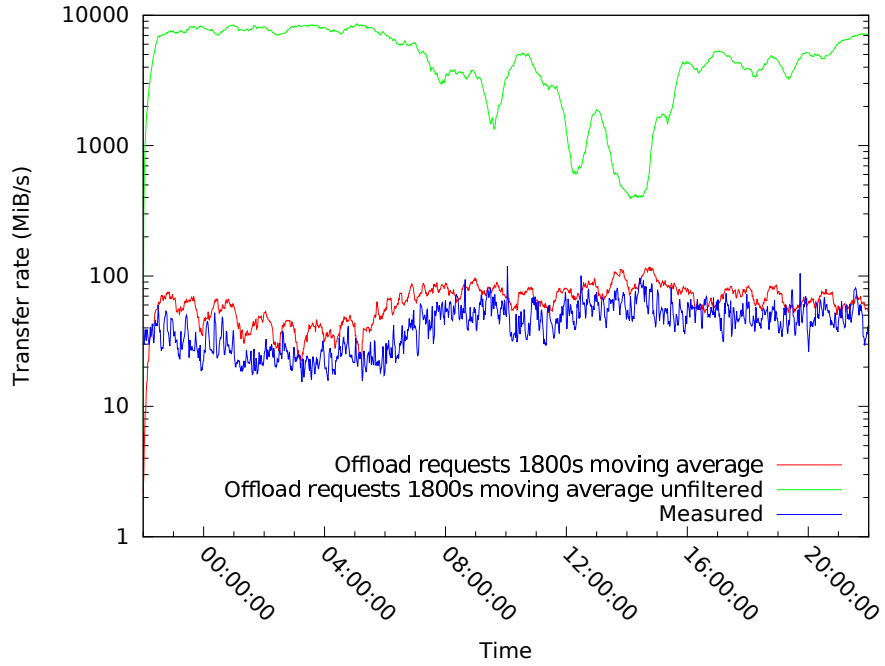


Figure 4.2: Traffic offloaded to saimei.acc.umu.se on 2015-04-01 (normal day) showing the inflation primarily caused by Download Managers (unfiltered) compared to filtered by first-seen ip/file pair and measured network transfer rate.

In order to determine whether this method also works during a burst period the `xferlog` used in Figure 4.1 was processed into an offload-log and the same filter and plots were applied. The result can be seen in Figure 4.3. Note that the actual burst starts around 18:00. The large discrepancy between the measured transfer rate and the filtered offload rate can be attributed mostly to demand being higher than the offload target can handle and the fact that at this point many slow transfers are being initiated. However, if this behaviour is consistent for all bursts it will make detecting them straight-forward.

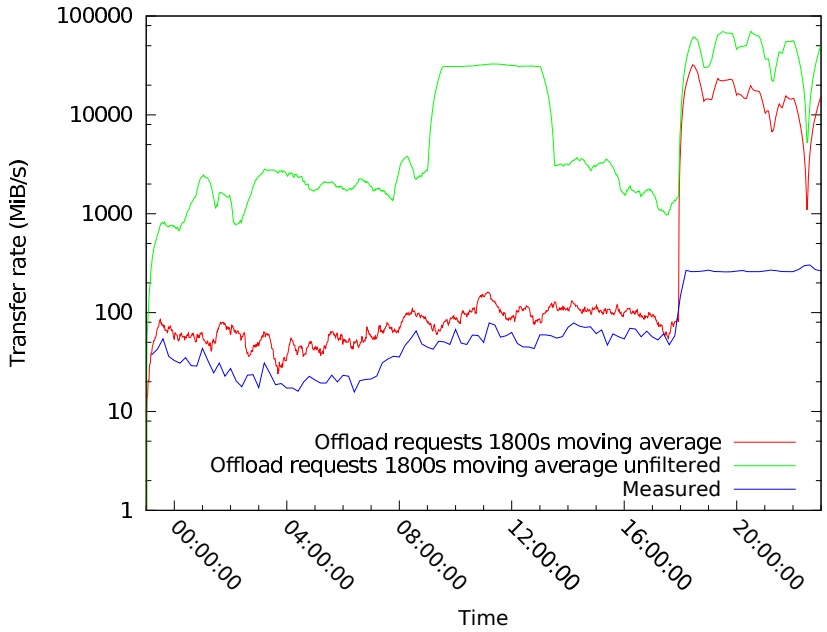
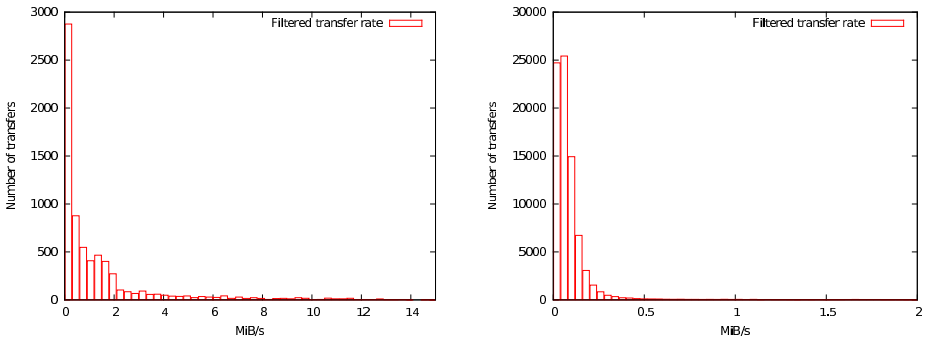


Figure 4.3: Traffic offloaded to gensho.acc.umu.se on 2015-01-21 (burst day) showing the inflation primarily caused by Download Managers (unfiltered) compared to filtered by first-seen ip/file pair and measured network transfer rate.

In light of the impact of filtering transfers by first-seen ip/file pairs, a similar processing was made to investigate whether the transfer rate statistics were skewed as well. For this purpose the average transfer rate for each ip/file pair occurrence are calculated. This causes no change for whole-file transfers but emits a single record instead of multiple records for multiple partial transfers of the same file from a single client. The transfers from gensho.acc.umu.se on 2015-01-21, when a burst occurred, now average at 175 kiB/s with a median of 65 kiB/s, a significant difference from the unfiltered results previously shown in Section 4.3. As comparison, on a normal day (2015-04-01) the transfers from saimei.acc.umu.se averages at 2.6 MiB/s with a median of 441 kiB/s.

Figure 4.4 indicates that the transfer time distribution is heavy-tailed as suspected earlier. It should be noted that the plots have been truncated to be able to show some detail and that there are occurrences of transfer rates up to the single-session physical maximum of 120 MiB/s. These data sets can be fitted reasonably well to a log-normal distribution[6], this coincides with other observations of log-normal distributions in the scope of internet traffic [1] [10].

[6] `http://en.wikipedia.org/wiki/Log-normal_distribution`

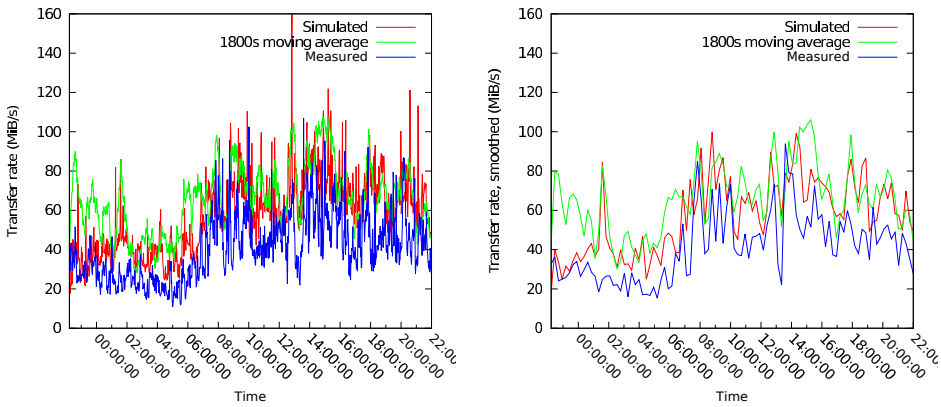


(a) saimei.acc.umu.se on 2015-04-01 (normal day).

(b) gensho.acc.umu.se on 2015-01-21 (burst day).

Figure 4.4: Transfer rates averaged by first-seen ip/file pair.

Using a log-normal distribution to assign random transfer rates to the transfers logged in the offload-log enables the possibility to simulate the server network bandwidth. A typical simulation run is shown in Figure 4.5. As can be seen the raw data is very noisy, making it hard to draw conclusions, so a smoothed representation is included as well. The smoothing algorithm chosen is cubic splines [14, chapter 5.6]. The simulation captures the changes in transfer rates, but as can be seen in Figure 4.5b the simulated transfer rate is inflated compared to the measured transfer rate. In fact, the simple moving average estimation of the offload requests made are closer to the simulated transfer rate than the simulated transfer rates are to the measured transfer rate. One of the reasons for this can be a larger occurrence of partial transfers than anticipated. Another potential factor contributing to the error can be the random distribution of transfer rates not being a close enough match to the measured transfer rates. Determining the exact cause is beyond the scope of this work due to time constraints.



(a) Raw data.

(b) Data smoothed using cubic splines.

Figure 4.5: Transfer rates from gensho.acc.umu.se on 2015-04-01, simulated from offload-log filtered by first-seen ip/file pair.

As it stands, this is the data available being logged close to real time. The estimate obtained of the server network bandwidth is rather inaccurate. However, based on historical data the bursts of interest seem to be large enough to be easily detectable despite the discrepancy.

4.5 Logs and time series

Time series analysis is a large area involving many disciplines and it can be hard to identify suitable literature to begin with. We find Hyndman and Athanasopoulos [14] to be a good introduction and primer on how to work with time series using the R software [29] and Cryer and Chan [9] to be a good resource for the theoretical aspects.

A common task is to use time series to predict/forecast a property at a future time. Methods range from naïve ones, such as using the last seen value as a prediction of the future (known as the persistence method in weather forecasting[7]), to highly complex models evaluating multiple time series using high performance compute clusters, commonly used in modern weather forecasting[8]. It is worth to note that simple methods can be quite useful and should not be discarded before having been evaluated. An example is long-range weather forecasts, where using the average seasonal data is more accurate than current weather models[9].

Time series are data sets where a property is logged sequentially in time. However, most methods and literature constrain themselves to time series with properties logged at regular intervals, also known as equidistant sampling. The network transfer rates logged in the ftp.acc.umu.se system are examples of such time series.

The other various logs in the ftp.acc.umu.se system have to be preprocessed before using established methods of analysis, modeling and forecasting/prediction. This is commonly done via various forms of interpolation, such as averaging or summing properties over a selected time interval, and unless otherwise noted this is the method used in this work. The process of changing the data interval is sometimes referred to as subsampling, extending the data interval by aggregating data points, or supersampling, reducing the data interval by calculating artificial intermediate data points.

For readers unfamiliar with time series analysis, we now provide a brief introduction to commonly used terms and methods.

4.6 Initial time series analysis

Analyzing time series starts with plotting the data to identify properties of the data set. A time plot is the most common way to represent time series data. Figures 4.6 and 4.7 illustrates typical time plots for the outgoing network bandwidth from the ftp.acc.umu-.se Offloaders, produced by RRDtool [26] and R [29] respectively. As can be seen the data rate is not constant, contains spikes, outages and what appears to be a repetitive pattern.

[7] http://en.wikipedia.org/wiki/Weather_forecasting#Persistence
[8] http://www.smhi.se/kunskapsbanken/meteorologi/meteorologiska-modeller-1.5932
[9] http://www.forecastadvisor.com/blog/2009/03/06/week-out-weather-forecasts/

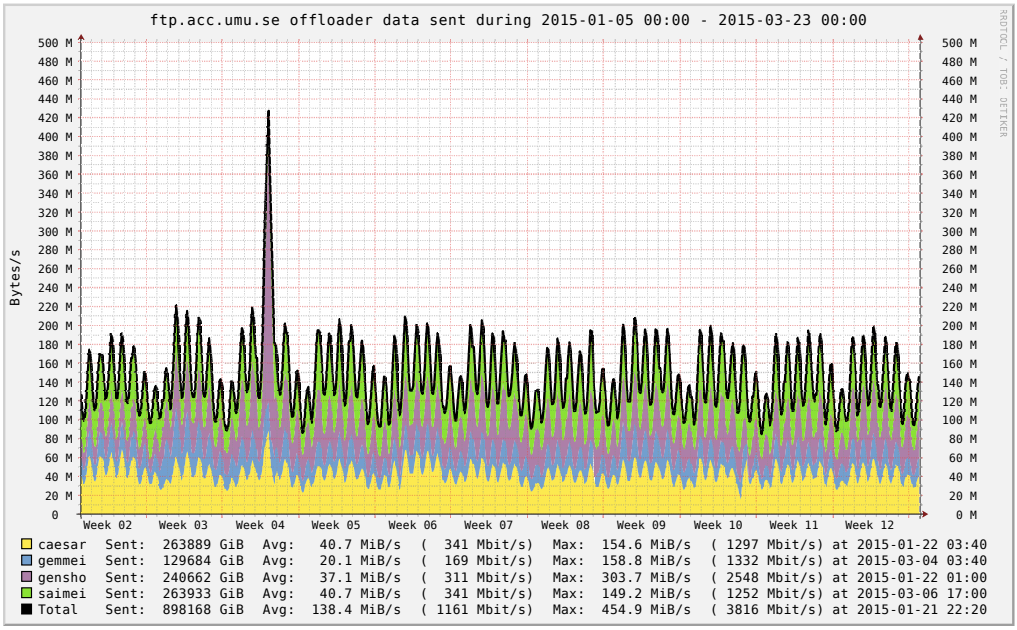
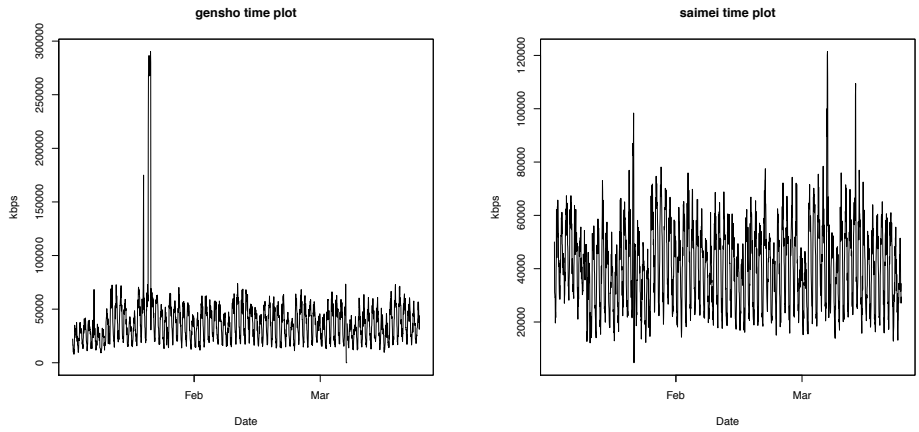


Figure 4.6: Transfer rates from ftp.acc.umu.se Offloaders 2015-01-05 - 2015-03-23, produced by RRDtool.



(a) gensho (b) saimei

Figure 4.7: Example individual transfer rates from ftp.acc.umu.se Offloaders 2015-01-05 - 2015-03-23, produced by R.

Depending on the resolution of the plot different patterns can be identified. *Outliers* are values that stand out for some reason. These can be outages, peaks due to changed usage patterns, measurement errors etc. *Level shifts* are events that shifts the entire level

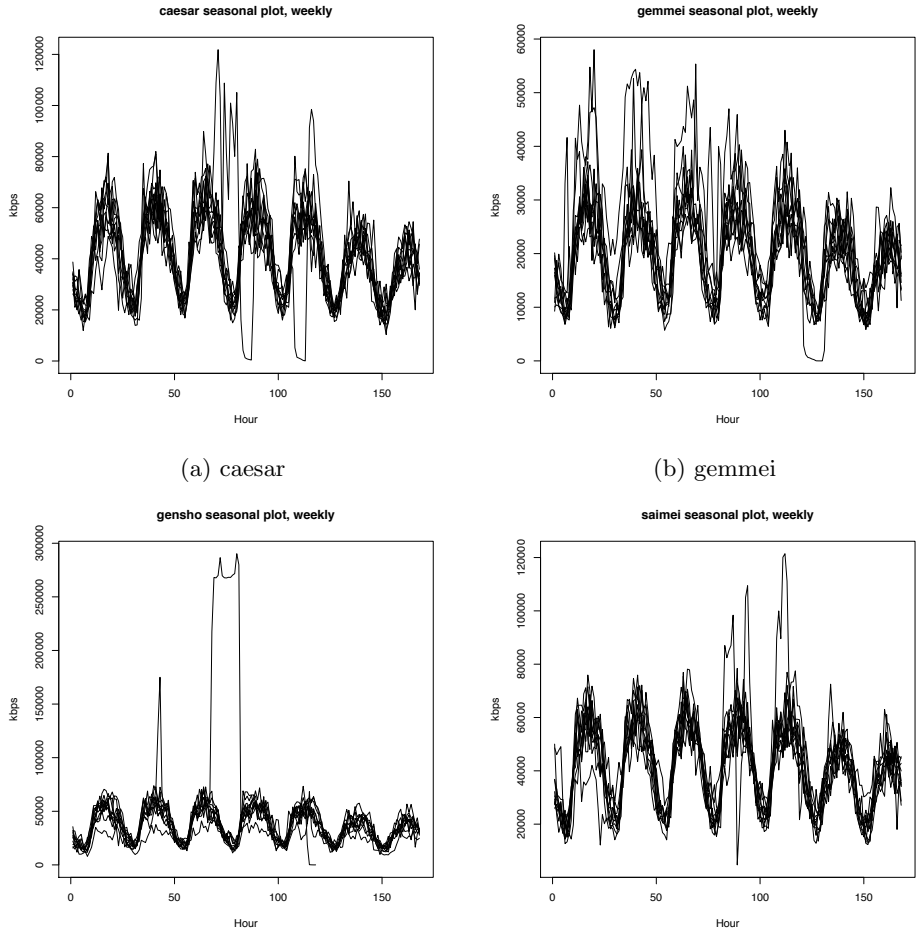
of the time series, for example a new Linux distribution becoming extremely popular, dominating the number of downloads from the ftp.acc.umu.se system. These events are sometimes referred to as *structure changes* [34].

Knowledge of the occurrence of outliers and level shifts can be beneficial as they commonly cause issues when choosing and using forecast methods [34] [7]. While many methods of automated detection exists, such as Chen and Liu [6] implemented in the R `tsoutliers` package[10], the handling of these events are tightly coupled to the data set at hand as shown earlier in Section 4.4. If the outliers and level shifts are known this knowledge can be used to assess their impact, but also as a benchmark for automated methods of detection. In any case it should be noted that outliers caused by measurement and data entry errors are normally discarded from the data set [14, chapter 4.4]. A *trend* is a long-term change, increase or decrease, in the data. This is usually easiest to spot in a plot with a low resolution, for example a multi-year plot with quarterly results. An example of too high resolution making trends hard to see is the data shown earlier in Figure 4.5.

Seasonal patterns are coupled to the time of the observations. Examples of these are hourly, daily, monthly or yearly fluctuations. It should be noted that seasonal patterns have fixed lengths that are known. To see the seasonality the resolution of the plot must be high enough, for example hourly values to identify a daily seasonal pattern. *Cycles* are not to be confused with seasonal patterns, as they are disconnected from the time of the observations and are due to external factors, such as the global economy. They are commonly longer than the seasonality, and have larger impact than seasonal changes [14, chapter 2.1].

If a seasonal pattern is identified a seasonal plot can help reveal further details. Figure 4.8 demonstrates typical seasonal plots for the ftp.acc.umu.se Offloaders. Here the data is plotted with a weekly season, meaning that all weeks are plotted on top of each other. A few outliers such as down times and bursts can be seen, but in general there are compelling evidence for a weekly seasonal pattern.

[10] `http://cran.r-project.org/web/packages/tsoutliers/index.html`



(a) caesar

(b) gemmei

(c) gensho

(d) saimei

Figure 4.8: Seasonal plot of transfer rates from ftp.acc.umu.se Offloaders 2015-01-05 - 2015-03-23, showing a strong likelihood of a weekly seasonal pattern.

There is also descriptive statistics that can be leveraged to identify features of time series. Common measures include the mean(average) value, min/max values, median values, percentiles/quantiles and standard deviations. It should be noted that outliers can have a large impact on average values, using the median value or other quantiles is usually preferred when data is prone to contain outliers [22].

Using time series data to plan/predict peak capacity needed has a few pitfalls that requires attention. Commonly, loads are bursty and graphical representations are smoothed to make it easier to identify trends and patterns, or purely for visualization reasons. However, these kinds of plots can be deceiving if there are needs to cater for the peaks. As an example, the logged network bandwidth shown in Figure 4.1 is not close to the server physical maximum of approximately 400 MiB/s, but the peaks turned out to be

hidden due to low resolution (big time steps) of the logged data. Another pitfall is to only look at average or median values. While helpful to understand the workload patterns, they commonly hide peaks and other interesting phenomena.

In many situations it is more useful to consider the percentile[11], also called quantile[12] in a more generic definition. A percentile is the value below which a given percentage of the data falls. For example, the 98th percentile for a network bandwidth time series tells that 98% of the time the bandwidth is below the given value. Conversely, 2% of the time the bandwidth is above the given value. When using percentiles as a base for sizing it is again important to consider system behavior when going above the given percentile. A system slow-down might be accepted, but a complete system crash is probably not.

The Pearson product-moment correlation coefficient[13], usually called just the *correlation coefficient*, is commonly used to assess the linear relationship between variables in cross-sectional data. For time series data this concept can be extended to measure the relationship between values in a time series, and is then called *autocorrelation* [14, chapter 2.2].

The term *lag* is used in conjunction with autocorrelation to describe the relationship between time steps. For example, lag 1 describes the relationship between y_t and y_{t-1}; lag 4 describes the relationship between y_t and y_{t-4}. The autocorrelation is usually plotted as the *autocorrelation function* (ACF) also known as a *correlogram* [14, chapter 2.2] [9, p. 46].

The ACF plot is commonly used to help select an appropriate method for forecasting, and to evaluate the suitability of a chosen forecasting method by using residual, forecast error, diagnostics. In all cases, significant spikes in lags are of interest.

4.7 Forecasting method overview

Having established basic knowledge of the time series of interest a suitable method for the intended forecast can now be selected. For doing burst detection in the ftp.acc-.umu.se redirection subsystem, a short-term forecast is needed detailing the expected network bandwidth usage of the offload target servers. This will then serve as the base for selecting a threshold level used to decide if a burst has occurred. When selecting a method it is wise to adhere to the principle of parsimony. For computer scientists it is best explained as the KISS principle for statistics: when faced with multiple methods with similar performance choose the simplest one [14, chapter 2.3] [23].

Another concept to be aware of is over-fitting. This commonly happens when too much, or too fine-grained, data is used to select/train/fit a model resulting in a more complex system than necessary. The result is a forecasting system for noise instead of the property of interest. Before discussing the selection of a method for this specific use case a summary is provided of a selection of methods that computer scientists and system administrators might encounter, together with a small discussion on situations where to apply them.

[11] `http://www.mathsisfun.com/data/percentiles.html`
[12] `http://en.wikipedia.org/wiki/Quantile`
[13] `http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient`

4.7.1 Simple methods

The average method predicts the future to be the average of all past observations.

The naïve method sets future predictions to be the value of the last observation. A variant is the seasonal naïve method that uses the last observed value from the previous season, for example predicting the temperature at 08:00 today to be the same as 08:00 yesterday. A variation is the drift method that allows the naïve prediction to change based on the average change seen in the historical data.

These methods are very simple to implement and often provide good enough predictions for the problem at hand [14, chapter 2.3].

Applications leveraging RRDtool [26] for time series storage can use the PREDICT[14] operand to implement a variant of seasonal naïve forecasting that can also use values from multiple seasons. This is illustrated by Figure 4.9 showing a forecast a week into the future using the average of the eight previous weekly seasons.

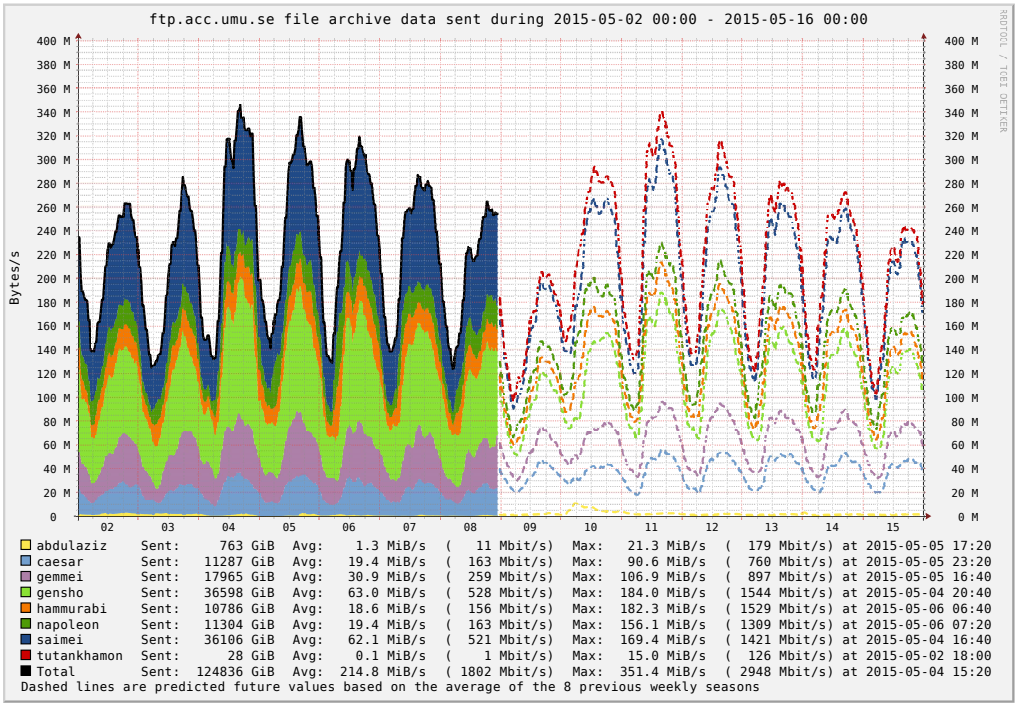


Figure 4.9: Example seasonal naïve forecast of the ftp.acc.umu.se system using the PREDICT operator of RRDtool.

[14] https://oss.oetiker.ch/rrdtool/doc/rrdgraph_rpn.en.html#ISet_Operations

4.7.2 Linear regression

Linear regression, or fitting a straight line as an approximation of a time series using least squares estimation, is often the simplest way of predicting the future development of a trend in a time series. It is also possible to obtain prediction intervals which gives an indication on the accuracy of the predictions [14, chapter 4.8].

RRDtool users can easily implement these predictions using the built-in functions for least square line approximation[15]. Figure 4.10 demonstrates a linear trend prediction of the network bandwidth for the ftp.acc.umu.se system for the coming 3 years, based on data from the 3 previous years.

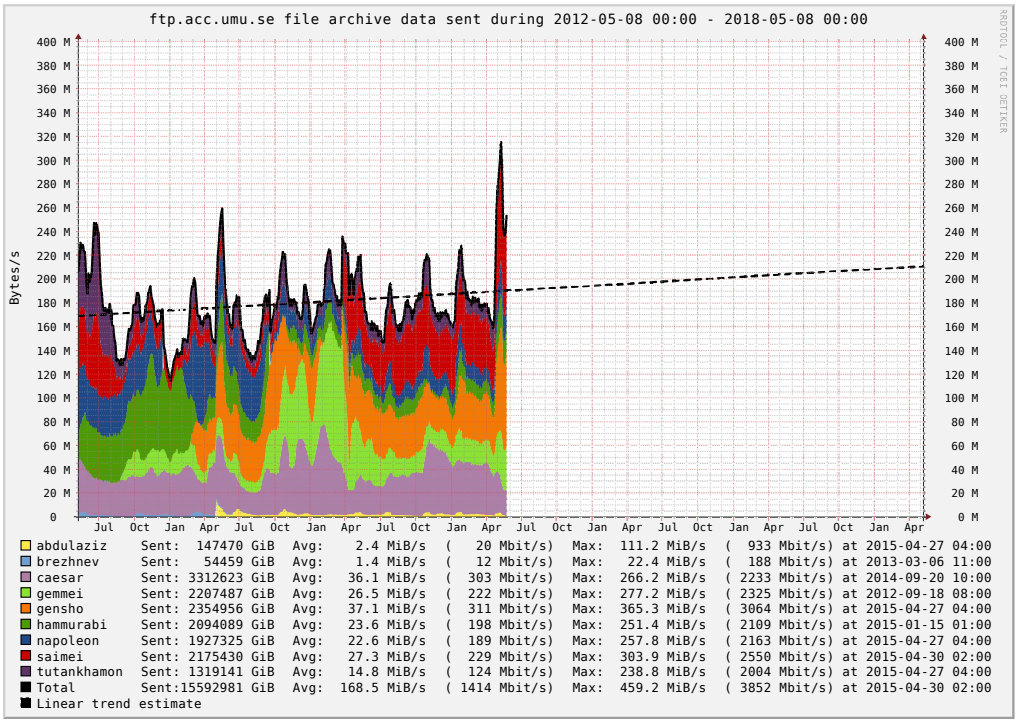


Figure 4.10: Example linear trend prediction of the ftp.acc.umu.se system using the least square line operators of RRDtool.

4.7.3 Decomposition

Decomposition aims to isolate patterns such as trends, seasonality and cycles in time series data. This is commonly done to get better understanding of a time series data which can then be used to improve forecasts.

Moving average smoothing is a classic method used mainly during the 1920s to the 1950s that exposes trends and cycles by averaging data centered around each observation [14, chapter 6.2]. This eliminates noise in the data making it easy to spot longer trends. An

[15] `https://oss.oetiker.ch/rrdtool/doc/rrdgraph_rpn.en.html#ILSLSLOPE__LSLINT__LSLCORREL`

m-MA uses m values and is also called a moving average of order m. For example, a *3-MA* provides a smoothed time series by estimating each data point by the average of the neighboring data points and the data point itself.

Mostly used as an analysis tool, employing a statistical software package such as R is recommended. Seasonal and Trend decomposition using Loess (STL) [8] is one of the preferred automated methods [14, chapter 6.5] to use for decomposition in R.

Figure 4.11 shows an example decomposition of network transfer rates from the offload server caesar.acc.umu.se using STL. The topmost *data* panel shows a regular time plot of the raw data. Here a weekly pattern can be observed with more traffic on work days, less traffic during weekends, and nights being less busy than days. There are also some bursts and outages. The *seasonal* panel displays the identified seasonal pattern. While a daily pattern has been identified the weekly aspect has been missed by the decomposition. As expected from the *data* panel the *trend* panel depicts a weak trend, essentially a flat line if the scales are considered. Finally, the *remainder* panel shows what can not be explained by the seasonal and trend components. Here it can be seen that there are indeed indications of a weekly pattern remaining, as well as the same bursts and outages observed in the *data* panel.

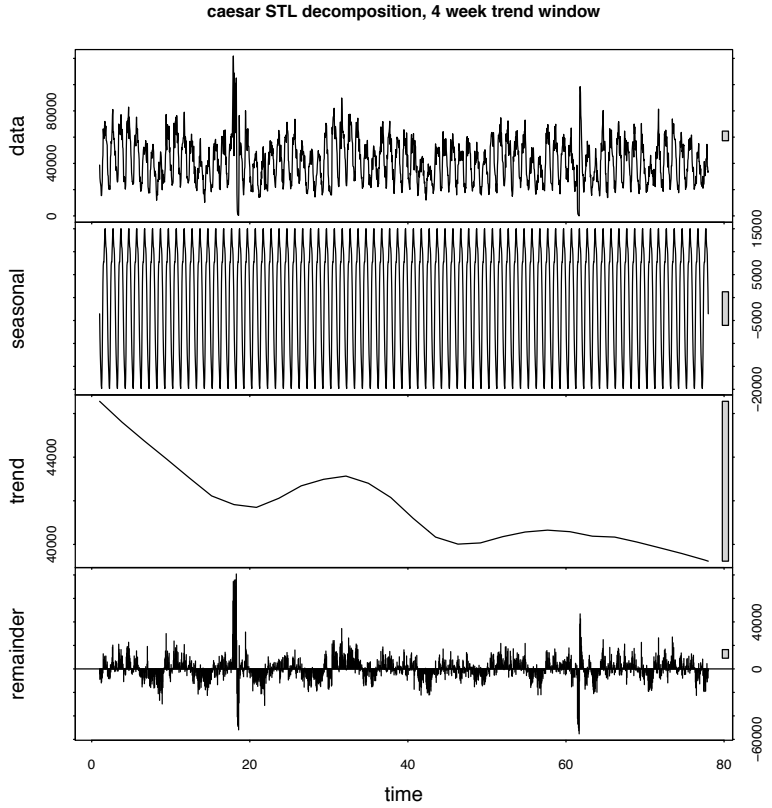


Figure 4.11: Decomposition of network transfer rates from caesar.acc.umu.se during 2015-01-05 - 2015-03-23 using the `stl()` R function with a periodic season window and a four week trend window.

Decomposition can also be leveraged to do forecasts. Since the patterns of the time series are identified as components it is possible to predict future values by forecasting the components separately, for example by using the seasonal naïve method for the seasonal component and linear regression for the remaining component.

4.7.4 Exponential smoothing

Exponential smoothing methods can be seen as more sophisticated versions of the average method. While all past observations have equal impact in the average method, exponential smoothing methods introduce weights that decays exponentially given the age of the observation, called smoothing equations. This gives the effect of observations close in time having larger impact on the prediction of future values than older observations, and enables to describe a time series with more accuracy [14, chapters 7, 7.1].

The Holt-Winters seasonal method is an example. This method has smoothing equations for the level, trend and seasonal components with the corresponding parameters α (alpha), β^* (beta-star) and γ (gamma). The nature of the seasonal component is a deciding factor whether the additive or multiplicative variant of this method is to be chosen. The additive method is preferred if the amplitude of the seasonal component is relatively constant while the multiplicative method is preferred if the amplitude of the seasonal component varies with the amplitude of the time series [14, chapter 7.5].

Holt-Winters is also implemented in RRDtool[16], which enables more complex forecasts to be done directly from such time series. However, due to the relative complexity of setting up RRDtool for such forecasts, evaluating the benefit first using statistics software such as R is recommended. R is also helpful by estimating the smoothing parameters if these are not known.

The underlying statistical models for the exponential smoothing methods are labeled $ETS(Error, Trend, Seasonal)$, with the components being labeled N for none, A for additive and M for multiplicative with a few special cases and sub notations for a total of 30 models [14, chapters 7.6, 7.7]. Being true models the statistical framework exists to enable automated model selection, as has been implemented in the R `ets()` function.

4.7.5 ARIMA models

AutoRegressive Integrated Moving Average ($ARIMA$) models are a different approach to time series modeling and forecasting. These models intend to describe autocorrelations in the data to be able to do predictions. Together with exponential smoothing models they are the most commonly used tools in forecasting [14, chapter 8]. The nomenclature can be rather confusing, as multiple naming schemes have evolved and merged.

Stationarity is an important concept when dealing with $ARIMA$ models. Hyndman and Athanasopoulos [14, chapter 8.1] explains this as "a stationary time series is one whose properties do not depend on the time at which the series is observed". A more formal definition can be found in Cryer and Chan [9, pp. 16-19]. In addition to a common time

[16] `http://oss.oetiker.ch/rrdtool/doc/rrdtool.en.html#IAberrant_Behavior_Detection`

plot, an ACF plot or formal unit root tests such as Augmented Dickey-Fuller (ADF, R function `adf.test()`) and Kwiatkowski-Phillips-Schmidt-Shin ($KPSS$, R function `kpss.test()`) can be used to assess whether a time series is stationary.

An autoregressive, $AR(p)$, model predicts future values based on p linear combinations of past, lagged, values. For example, an $AR(1)$ model uses one lagged value while an $AR(3)$ model uses three lagged values. $AR(p)$ models are usually restricted to stationary time series [14, chapter 8.3] [9, p. 71].

A moving average, $MA(q)$, model uses the q past prediction errors to predict future values. For example, an $MA(1)$ model uses the last error and an $MA(3)$ model uses the past three errors. It should be noted that an $MA(q)$ forecast model should not be confused with moving average smoothing, m-MA, used to estimate trends and cycles [14, chapter 8.4]. A more in-depth discussion on $MA(q)$ models can be found in Cryer and Chan [9, pp. 57-65].

Combining these two models yields the autoregressive moving average, $ARMA(p, q)$, model [9, p. 77]. An $AR(2)$ model can also be described as $ARMA(2, 0)$, $MA(1)$ as $ARMA(0, 1)$ and so on.

Autoregressive integrated moving average, $ARIMA(p, d, q)$, models are the next step with an addition of an integration, degree of differencing, term d. This term corresponds to the degree of differencing required to obtain a stationary time series. In practice, $d = 1$ (first difference) or at most $d = 2$ (second difference) [9, p. 92]. Analogous to the previous examples, an $ARMA(3, 4)$ model can be described as $ARIMA(3, 0, 4)$ and so on [14, chapter 8.5].

Seasonal models are written as $ARIMA(p, d, q)(P, D, Q)_m$, where p, d, q are the non-seasonal components; P, D, Q are the seasonal components and m denotes the number of periods per season [14, chapter 8.9]. The seasonal components adds seasonal awareness by operating on lagged values of the corresponding season.

The $ARIMA$ models also allows for building a framework of automated model estimation based on a set of data. However, the results needs to be verified before the model suggested can be trusted. Hyndman and Athanasopoulos [14, chapter 8.7] provides a detailed workflow on how to do $ARIMA$ modelling in R.

The choice between $ARIMA$ and ETS depends on the problem and data at hand. When using automated functions for model suggestion it is relatively easy to compare the models suggested and select the simplest model that is best suited for the task.

4.8 Evaluating a forecasting method

An evaluation of a forecasting method is comparing results from a forecast with the actual outcome. It is important to note that the data used for such a comparison should not have been used during the selection/fitting process.

For this purpose it is common to divide the available data into a training set used for selection/fitting, and a test set, also called out-of-sample data or hold-out data, to evaluate forecasting performance. Hyndman and Athanasopoulos [14, chapter 2.5]

recommends the test set to be about 20% of the total data, or at least as long the forecast spans. There are also recommendations to have test sets as large as 50% if there is a lot of data available[17].

There are a number of metrics that can be computed to assess the accuracy of a forecast with the Root Mean Squared Error ($RMSE$) and Mean Absolute Error (MAE) being frequently used to provide scale-dependent metrics. These metrics can not be used to compare forecast performance between data sets, for this purpose the Mean Absolute Percentage Error ($MAPE$) is available. It should however be noted that percentage errors are unsuitable if fractions make no sense in the current context. In those cases the Mean Absolute Scaled Error ($MASE$) or Mean Squared Scaled Error ($MSSE$) might be applicable. A common method is to calculate multiple metrics when evaluating and favor methods that all metrics agree on [14, chapter 2.5]. A more elaborate method is to choose metrics based on their properties[18].

In all cases, it is useful to plot the forecast together with the test set to assess the behavior of the forecast. There are cases where metrics can be misleading compared to the properties needed for a forecast in a specific application.

A residual is the difference between a predicted value and an observed value. The residuals should ideally be just noise, meaning that the forecast is using all information available [14, chapter 2.6]. The ACF plot is used to analyze the residual properties, where significant spikes on an ACF plot suggests that the residuals contain information that a better forecasting model can use to produce more accurate results.

The need for forecast accuracy should also be considered, as well as stability in the face of outliers, missing values, trend/level changes etc. In environments where the forecasting method is likely to stay fixed once implemented, tests should also encompass data with multiple possible scenarios to expose any unwanted behavior that might endanger the system in the future. The behavior of simpler methods and models are likely more predictable when subjected to changing conditions.

4.9 Forecast accuracy

It is often of interest to be able to assess beforehand whether a forecast is likely to be correct or not. Most forecasting methods provides a tool for this, a prediction interval. While the forecast produces a future prediction of a property of interest, the prediction interval is used to assess the uncertainty of that prediction. A common example is a 95% prediction interval, which means that there is a 95% probability that the actual value will be within a given range.

Prediction intervals, also called forecast intervals, are commonly based on an estimate of the standard deviation $\hat{\sigma}$ (sigma-hat) of the forecast distribution. For parameterless methods such as the naïve method this is identical to the standard deviation of the forecast errors, the residuals. For other methods there is a slight difference, but this is often ignored in practice [14, chapter 2.7]. When faced with multiple methods which produces similar forecasts it can be of interest to study whether the methods differ in their prediction intervals.

[17] `http://people.duke.edu/~rnau/three.htm`
[18] `http://people.duke.edu/~rnau/compare.htm`

To illustrate, Figure 4.12 demonstrates a forecast based on linear regression with 50%, 80% and 95% forecast intervals shown in dark to light shade respectively. As can be seen, the higher the probability the larger the range of the forecast needs to be.

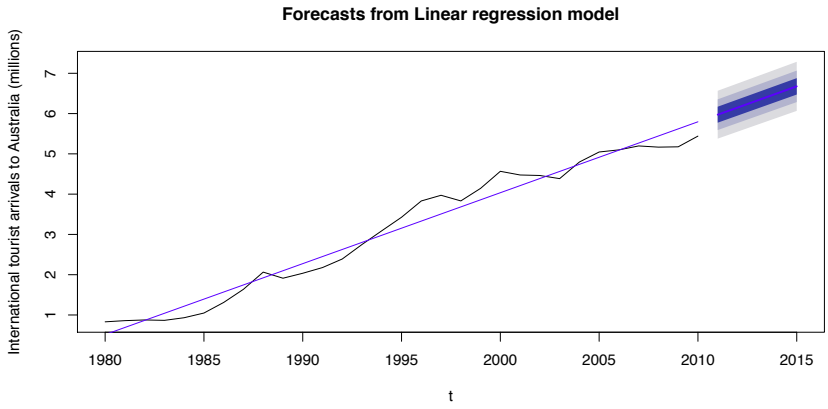


Figure 4.12: Example demonstrating 50%, 80% and 95% forecast intervals, data from Hyndman and Athanasopoulos [14, Figure 4.12].

4.10 Forecasting method selection

Having provided an overview of forecasting methods, including the topics of selection, evaluation and accuracy, the process of choosing a forecasting method for the problem at hand can now proceed. Aiming to detect significant bursts that warrants adding additional offload servers to serve popular files, the forecast is needed to provide a baseline for a burst detection scheme. For this, only a short-term forecast is needed. The forecasting implementation will be fixed once implemented, and as such must be very robust in the face of changing conditions. Accuracy is not of high priority as the bandwidth estimations used are rather imprecise.

Reviewing data from the ftp.acc.umu.se offload servers, for example Figures 4.6, 4.7 and 4.8, the following properties of the network transfer rates of the ftp.acc.umu.se offload servers can be established:

- A highly seasonal weekly pattern. This indicates that a significant amount of the downloads are initiated manually by users located in European time zones.
- Level changes occur. New Linux OS releases commonly cause a different mapping between popular files and offload servers, and the impact on network bandwidth is hard to predict.
- Outages occur. The most common causes are routine system maintenance and hardware failure.
- Trends are detectable over longer time periods. Larger projects such as Debian tend to release biannually, so multiple years of data needs to be analyzed to isolate the effect of individual releases in order to see conclusive proofs of trends.
- Bursts occur. Mainly caused by popular files, the goal is to automatically detect these.
- As shown in Section 4.4 on page 23 estimating the server bandwidth from the offload-log yields a low accuracy compared to the measured bandwidth.

For this application using a seasonal naïve method is proposed. Leveraging the median of multiple seasons of data provides a robust method of detecting level shifts, while not being sensitive to outliers such as outages and bursts. The seasonal plot demonstrated in Figure 4.8 indicates that the predictions will be accurate enough for this purpose.

One of the key motivations for a seasonal naïve method is that it is straightforward to understand and easy to implement. This is important should the system need to be changed by someone without advanced forecasting knowledge.

A downside of a seasonal naïve method is that it relies on saved historical data. Loss of this data, for example due to a server re-install, would mean that forecasts are impossible until enough data is again available. This can be worked around by having a bootstrap data set used in such cases.

As a compromise between robustness and sensitivity to level changes and outliers, three valid observations are used as a base for the forecast. Attempts are first made to find three observations from the past five weekly seasons. If this fails, a new attempt is made using daily seasons. Should this also fail, naïve prediction using only the previous observations is used as a last resort. This strategy ensures that multiple values are always used when producing a forecast, while setting a limit on how old observations can be in order to contribute to the result.

4.11 Burst detection

To be able to detect a burst there must first be a definition of what a burst is. For an Offloader in the ftp.acc.umu.se system, a burst is defined as an access pattern causing significantly more network traffic than normal. In the context of providing an response, this can be amended with enough accesses to a single file to warrant more Offloaders to handle the load. While these definitions are good enough for a human doing manual burst detection, which has previously been the case, an automated system needs a more precise definition which we will now address.

As illustrated earlier in Figure 4.4b bursts are likely to be very pronounced. This motivates choosing a simple threshold based method for burst detection, comparing the current network bandwidth with a threshold set to comply with the above definition.

The threshold should not be higher than the maximum bandwidth physically possible for a server. However, allowances need to be made for the low accuracy of the bandwidth estimations used, and the fact that each `redirprg.pl` instance running on a Frontend server will only see a part of the total amount of traffic handled by a single offload server. As these numbers more or less cancel each other out in the current setup, the maximum bandwidth is used as is when determining a maximum possible threshold to avoid the need of a configuration parameter having little impact.

The ideal threshold is low enough to enable an early detection, yet high enough to have as few false positives as possible. An automated way of approximating this is suggested by Rezaie [30], who uses the standard deviation of the residuals to calculate thresholds for outliers, the equivalent to bursts in this use case. For detecting a burst situation on an offload target server the same parameters are used, four standard deviations above the predicted network bandwidth (the 99.997% quantile of the residual distribution).

In order to detect which file is causing the burst, there are two simple iterative methods that can be used:

- Remove a file from the bandwidth calculation and see if the burst condition vanishes.
- Calculate the bandwidth of a single file and see if the accesses to that file triggers a burst condition.

As bursts tend to be rather pronounced the second method is chosen, as it has a lower possibility of false positives. The threshold is set to to three standard deviations above the predicted network bandwidth, the 99.9% quantile, when determining the file causing the burst. Tests with historical logged data indicates this to be a reasonable compromise between early detection and avoiding false positives.

To detect when the burst state ends comparison is made against the threshold used when the detection occurred. Previous experiences indicate the system state before a burst being more deterministic than the state during a burst.

The expected behaviour during a large release, for example the Debian Linux operating system, is for all servers in the ftp.acc.umu.se system to see a higher load than normal. The impact of this on the burst detection system is a high probability of detecting a burst situation on an offload target server, while not detecting a single file responsible for the burst situation. With this in mind it can be argued that the target server burst detection is excessive, and that only doing file-level burst detection would suffice. Future experiences from using the system in production will show whether this is the case, or if the target server burst detection proves useful.

Files determined to cause a burst are excluded from network transfer rate summaries and saved data used for future forecasts, as these files can have a very large impact on the total transfer rate and do not represent the normal state the forecasts should reflect.

Upon system (re)start the entire log file of the current day is processed. Using the available historical logs allows the system to be able to start doing burst detection virtually immediately after start-up, compared to only using near-live log data. As the methods chosen are computationally simple this incurs only a few seconds increase in start-up time.

4.12 Burst file handling and the redirection subsystem redirprg.pl

As indicated earlier in Section 3.2.3, the offload target servers only accept requests for files that they have been assigned to serve using the Pie-Chart Algorithm. Since there is no communication between the `redirprg.pl` instances, there is no method for a Frontend to signal an Offloader that a burst is occurring for a specific file and that it should start accepting requests for that file.

To work around this problem the assignment on the Offloaders are relaxed, allowing them to serve files that has neighboring servers as primary targets. As an example, consider Figure 3.3a and a file positioned in the upper-right position in the pie-chart. A request for that file sent to a server assigned to either the upper-left or lower-right positions will now also be accepted.

This solution allows assigning up to two extra offload target servers to handle a popular file, while not requiring communication between the `redirprg.pl` instances.

4.13 Results

The burst file handling implementation has been running in production on the ftp.acc.umu.se system since 2015-04-18. In order to bootstrap the forecast engine three weeks of logs from the corresponding ftp.acc.umu.se Frontend servers were used. Here we analyze data from the period 2015-04-18 - 2015-05-11. The first 9 days can be categorized as normal service, while the following days saw the release of Debian 8[19] with an initial burst that slowly declines.

Investigating the performance of the forecast engine, we randomly choose to focus on the Offloader saimei.acc.umu.se as seen on the Frontend hammurabi.acc.umu.se. Other combinations of Frontends and Offloaders shows similar characteristics. The forecasts for the first 9 days seems reasonably accurate, as depicted in Figure 4.13.

[19] `https://www.debian.org/News/2015/20150426`

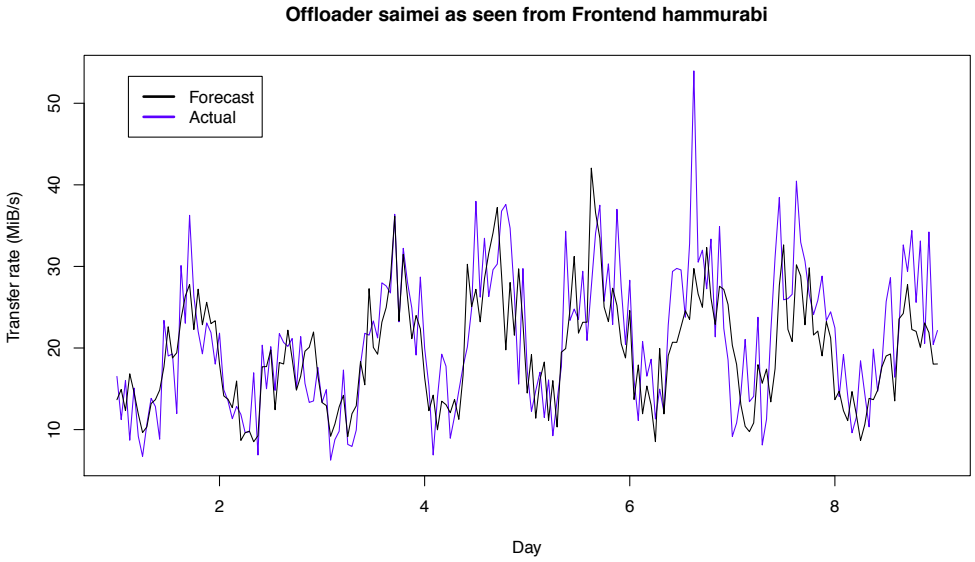


Figure 4.13: Offloader saimei.acc.umu.se network rate as seen on Frontend hammurabi-.acc.umu.se, day 0-9, normal traffic.

Figure 4.14 shows the burst happening, with the elevated transfer rate being ignored by the forecast engine during the first 14 days. This is as expected when doing a naïve forecast based on the median of the values from the last three weekly seasons. After 14 days there are clear indications of the forecasting engine starting to adapt to the post-burst data rates.

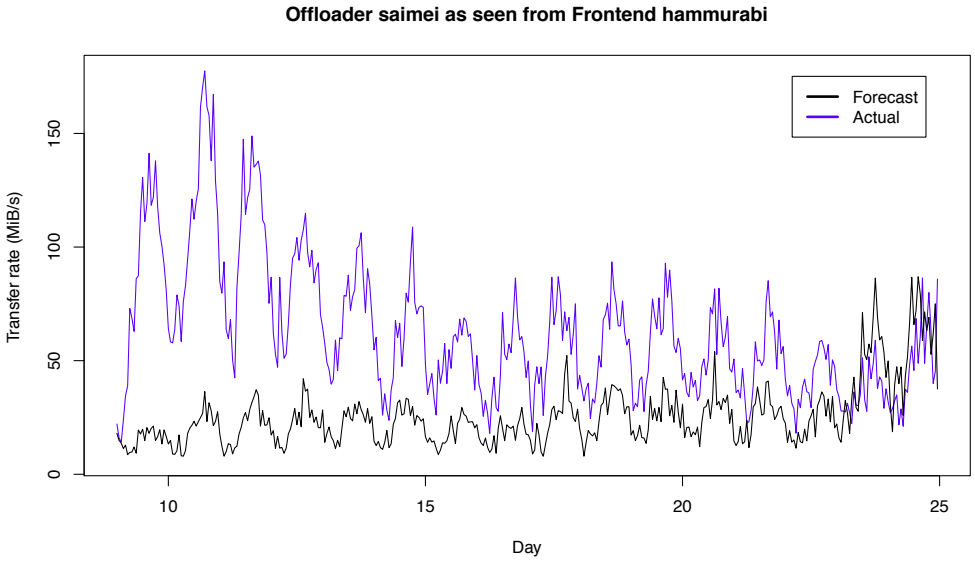


Figure 4.14: Offloader saimei.acc.umu.se network rate as seen on Frontend hammurabi-.acc.umu.se, day 9-25, burst traffic due to the release of Debian 8.

Having determined that the forecasting engine performs as expected, the accuracy of the Offloader network rates as estimated on the Frontend servers needs to be assessed. The results are expected to be sub-par but tolerable, given the experience gained in Section 4.4. There is also a scaling error, due to the simplistic implementation detailed earlier in Section 4.11, to be anticipated. Figure 4.15 compares the estimated Offloader network rates as seen on the Frontend hammurabi.acc.umu.se to the network rates measured on the actual Offloaders. The behavior is as expected, except for the Offloader gensho.acc.umu.se. This Offloader experiences the largest measured transfer rate, but the estimation seems unperturbed by this.

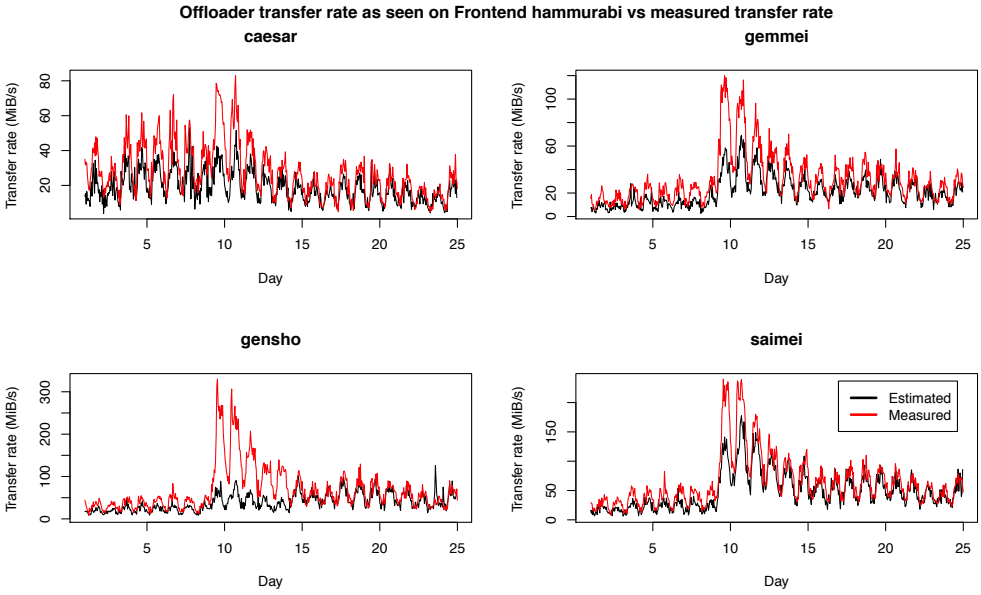


Figure 4.15: Offloader network rates as seen on Frontend hammurabi.acc.umu.se versus measured network rates, day 0-25.

The explanation can be found in the burst detection engine, where files detected as being in burst state are disregarded when calculating the forecast-related transfer rates (as explained earlier in Section 4.11). The most popular file[20] was assigned gensho.acc-.umu.se as an Offloader. A burst was early on detected, leading to the assignment of gemmei.acc.umu.se as an additional Offloader. However, reviewing the logs uncovers an unwanted behavior causing the file to be erroneously signaled as reverted to normal state, canceling the additional Offloader assignment. Internally, the file was still being treated as in burst mode, and consequently being disregarded from the forecast rate calculations.

This file was the only one detected as a burst during the Debian 8 release, and reviewing the logs available supports this decision. Overall system performance is dependent upon keeping as many files as possible in the cache, and assigning additional Offloaders to files should only be done when absolutely necessary.

4.14 Summary and Discussion

Time Series Analysis comes with a powerful toolbox, methods range from simple to very complex. System administrators frequently use RRDtool to log time series, a tool which is capable of doing much more than just logging and displaying historic events. Basic knowledge of time series analysis and forecasting techniques enables utilizing the full potential of RRDtool, enabling automated forecasts to be shown together with the historical data.

[20] `http://cdimage.debian.org/debian-cd/8.0.0/amd64/iso-dvd/debian-8.0.0-amd64-DVD-1.iso`

For this work we have leveraged our new knowledge to design and implement an automated burst file detection and handling scheme for the ftp.acc.umu.se file archive. We have shown that by choosing simple methods suited for the task an effective system can be implemented that not only fulfills the requirements of automated burst detection and handling, but also keeps the possibility of understanding, maintaining and debugging the system without deep knowledge of time series analysis and forecasting techniques.

The script language Perl [32] is used in the `redirprg.pl` implementation, and consequently for the additions made during this work. Perl proves to be a quite effective tool to use in many situations that a system administrator comes across. Regardless of the programming language used, good support for flexible data types is a must to enable quick and efficient problem solving.

Chapter 5

Acknowledgements

The author would like to extend his gratitude to a number of people whose insights has helped bringing success to this project.

Per-Olov Östberg and Ali Rezaie for useful ideas, feedback and constructive criticism during the Master's thesis work. Without you this report would not be quite as polished...

Fellow ACC admins Mattias Wadenstein, Tomas Ögren and Anton Lundin. We have had lengthy discussions on how to solve all these issues, and some of the ideas even worked (never underestimate the Pie!).

Tommy Norling deserves a special mention for proofreading obstinate C-code in the early days of `mod_cache_disk_largefile`. There are only so much bugs you can find by yourself!

Mikael Rännar is to credit for the introduction to wrapper libraries on the Operating Systems course.

The users of the ftp.acc.umu.se file archive, without you there would not be much to evaluate.

And finally, my beloved wife Helena Lindblom. In addition to being the one earning money when I worked on this, you did a fantastic job of proofreading and challenging me to do better. This would definitely not have been possible without you!

References

[1] I. Antoniou et al. "On the log-normal distribution of network traffic". In: *Physica D: Nonlinear Phenomena* 167.1–2 (2002), pp. 72–85. ISSN: 0167-2789. DOI: 10.1016/S0167-2789(02)00431-1. URL: http://www.sciencedirect.com/science/article/pii/S0167278902004311.

[2] *Apache HTTP Server Project*. The Apache Software Foundation. URL: http://httpd.apache.org/ (visited on 2015-05-06).

[3] *Apache Portable Runtime (APR) project*. The Apache Software Foundation. URL: http://apr.apache.org/ (visited on 2015-05-06).

[4] J. Banks and J. S. Carson II. "Introduction to Discrete-event Simulation". In: *Proceedings of the 18th Conference on Winter Simulation*. WSC '86. Washington, D.C., USA: ACM, 1986, pp. 17–23. ISBN: 0-911801-11-1. DOI: 10.1145/318242.318253.

[5] T. Brisco. *DNS Support for Load Balancing*. RFC 1794 (Informational). Internet Engineering Task Force, 1995-04. URL: http://www.ietf.org/rfc/rfc1794.txt.

[6] C. Chen and L.-M. Liu. "Joint Estimation of Model Parameters and Outlier Effects in Time Series". English. In: *Journal of the American Statistical Association* 88.421 (1993), pp. 284–297. ISSN: 01621459. URL: http://www.jstor.org/stable/2290724.

[7] C. Chen and G. C. Tiao. "Random Level-Shift Time Series Models, ARIMA Approximations, and Level-Shift Detection". English. In: *Journal of Business & Economic Statistics* 8.1 (1990), pp. 83–97. ISSN: 07350015. URL: http://www.jstor.org/stable/1391755.

[8] R. B. Cleveland et al. "STL: A Seasonal-Trend Decomposition Procedure Based on Loess (with Discussion)". In: *Journal of Official Statistics* 6 (1990), pp. 3–73. URL: http://www.jos.nu/Articles/abstract.asp?article=613.

[9] J. D. Cryer and K.-S. Chan. *Time Series Analysis. With Applications in R*. 2nd ed. Springer Texts in Statistics. Springer-Verlag New York, 2008. 491 pp. ISBN: 978-0-387-75958-6. DOI: 10.1007/978-0-387-75959-3. URL: http://www.springer.com/gp/book/9780387759586.

[10] A. B. Downey. "Lognormal and Pareto Distributions in the Internet". In: *Comput. Commun.* 28.7 (2005-05), pp. 790–801. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2004.11.001.

[11] C. Evans. *vsftpd. Very Secure FTP Daemon.*
URL: `http://security.appspot.com/vsftpd.html` (visited on 2015-05-13).

[12] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, 1999-06.
URL: `http://www.ietf.org/rfc/rfc2616.txt`.

[13] D. Holtstam et al. *SUNET (THE SWEDISH UNIVERSITY COMPUTER NETWORK) UNDER SCRUTINY. Evaluation report from the international expert panel.* 2013. ISBN: 978-91-7307-234-2.
URL: `http://publikationer.vr.se/`.

[14] R. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice.* 2013.
URL: `http://otexts.org/fpp/` (visited on 2015-03/2015-05).

[15] "Information technology - Portable Operating System Interface (POSIX) Operating System Interface (POSIX)". In: *ISO/IEC/IEEE 9945 (First edition 2009-09-15)* (2009-09), pp. 1–3830.
DOI: `10.1109/IEEESTD.2009.5393893`.

[16] *inotify. monitoring filesystem events.* Version 3.54. The Linux Kernel Organization. 2013-09-07.
URL: `http://manpages.ubuntu.com/manpages/trusty/man7/inotify.7.html`.

[17] B. Josefsson. *Sunets nät 2016.* Presented at SUNET-dagarna 11, Göteborg. 2014-10-06.
URL: `http://play.sunet.se/`.

[18] D. Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing.* STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6.
DOI: `10.1145/258533.258660`.

[19] D. Karger et al. "Web caching with consistent hashing". In: *Computer Networks* 31.11–16 (1999), pp. 1203–1213. ISSN: 1389-1286.
DOI: `10.1016/S1389-1286(99)00055-9`.
URL: `http://www.sciencedirect.com/science/article/pii/S1389128699000559`.

[20] *ld.so, ld-linux.so*. dynamic linker/loader.* Version 3.54. The Linux Kernel Organization. 2013-09-07.
URL: `http://manpages.ubuntu.com/manpages/trusty/man8/ld.so.8.html`.

[21] *ld.so.1. runtime linker for dynamic objects.* Version 10. Oracle. 2013-01.
URL: `http://docs.oracle.com/cd/E26505_01/html/816-5165/ld.so.1-1.html`.

[22] S. Manikandan. "Measures of central tendency. Median and mode". In: *Journal of Pharmacology and Pharmacotherapeutics* 2.3 (2011), pp. 214–215. ISSN: 0976-500X.
DOI: `10.4103/0976-500X.83300`.

[23] A. I. McLeod. "Parsimony, Model Adequacy and Periodic Correlation in Time Series Forecasting". In: *International Statistical Review / Revue Internationale de Statistique* 61.3 (1993), pp. 387–393. ISSN: 03067734.
URL: `http://www.jstor.org/stable/1403750`.

[24] P. Mockapetris. *Domain names - concepts and facilities*. RFC 1034 (INTERNET STANDARD). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. Internet Engineering Task Force, 1987-11.
URL: `http://www.ietf.org/rfc/rfc1034.txt`.

[25] T. Neame. "Characterisation and modelling of internet traffic streams". PhD thesis. University of Melbourne, 2003.
URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.1247`.

[26] T. Oetiker. *The Round Robin Database Tool (RRDtool)*.
URL: `http://oss.oetiker.ch/rrdtool/` (visited on 2015-05-11).

[27] T. Oetiker et al. "The Not So Short Introduction to LaTeX 2_ε".
URL: `http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf` (visited on 2015-03/2015-05).

[28] J. Postel and J. Reynolds. *File Transfer Protocol*. RFC 959 (INTERNET STANDARD). Updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151. Internet Engineering Task Force, 1985-10.
URL: `http://www.ietf.org/rfc/rfc959.txt`.

[29] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2013.
URL: `http://www.R-project.org/`.

[30] A.-R. Rezaie. "An automated forecasting method for workloads on web-based systems. Employing an adaptive method using splines to forecast seasonal time series with outliers". MA thesis. Umeå University, Department of Mathematics and Mathematical Statistics, 2014. 47 pp.
URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-98024`.

[31] S. Shepler et al. *Network File System (NFS) version 4 Protocol*. RFC 3530 (Proposed Standard). Obsoleted by RFC 7530. Internet Engineering Task Force, 2003-04.
URL: `http://www.ietf.org/rfc/rfc3530.txt`.

[32] *The Perl programming language*. The Perl Foundation.
URL: `http://www.perl.org/` (visited on 2015-05-11).

[33] A. Tridgell, P. Mackerras, and W. Davison. *rsync. a file transfer program for Unix systems*.
URL: `http://rsync.samba.org/` (visited on 2015-05-13).

[34] R. S. Tsay. "Outliers, level shifts, and variance changes in time series". In: *Journal of Forecasting* 7.1 (1988), pp. 1–20. ISSN: 1099-131X.
DOI: `10.1002/for.3980070102`.

[35] H. Wallberg. *SUNET during more than 20 years*. Presented at NSC'09, Linköping. 2009-10.
URL: `http://www.nsc.liu.se/nsc09/pres/HansWallberg.pdf`.

[36] T. Williams, C. Kelley, and the Gnuplot development team. *Gnuplot. an interactive plotting program*. 2012-03.
URL: `http://www.gnuplot.info/`.

Appendix A

Tools

Here a short description of tools used in the ftp.acc.umu.se system, and during the work on this Master's thesis, is provided.

Apache httpd [2]

`http://httpd.apache.org/`

The Apache HTTP Server Project is an open-source HTTP server. Apache httpd is highly modular and extensible, a fact that has been leveraged in the ftp.acc.umu.se system.

rsync [33]

`http://rsync.samba.org/`

rsync is a utility providing fast incremental file transfers. It is the de-facto standard for syncing file sets.

vsftpd [11]

`http://security.appspot.com/vsftpd.html`

The Very Secure FTP Daemon (vsftpd) is a fast and easily configurable implementation of a server for the FTP protocol [28].

APR [3]

`http://apr.apache.org/`

The Apache Portable Runtime (APR) is used in Apache httpd to provide a consistent API regardless of the underlying operating system. Any development on Apache httpd requires to get familiar with APR and the related APR-util.

RRDtool [26]

`http://oss.oetiker.ch/rrdtool/`

RRDtool has become the de-facto standard for data logging and graphing of time series data in open source applications and stand alone systems. RRDtool is used in the ftp.acc.umu.se system to log network bandwidth on the different servers.

Perl [32]

`http://www.perl.org/`

Perl is a general-purpose programming language that is commonly found wherever there was a need for a quick hack to do text manipulation and has become especially popular among system administrators. While ideally suited for smaller hacks due to its compact C/shell-like syntax, a disciplined programmer can write very large projects in Perl in the same manner as Python or Java (which are notoriously long-winded for quick hacks in our opinion). There are numerous modules that extend the functionality of perl.

PDL

`http://pdl.perl.org/`

The Perl Data Language (PDL) is a Perl extension that gives Perl the ability to efficiently store and process N-dimensional data arrays. Despite being largely unknown in the scientific community, PDL provides similar capabilities as numpy/scipy (Python), IDL and MatLab.

Gnuplot [36]

`http://www.gnuplot.info/`

Gnuplot is a command-line driven utility for creating graphs and plots. There are very few things you can not do with Gnuplot, the problem is usually to figure out how to do it.

R [29]

`http://www.R-project.org/`

R is a program aimed at statistics, both computing and graphs/plots. It enjoys a vivid community and an excellent set of plugin libraries.

vim

`http://www.vim.org/`

A text editor that builds upon the venerable vi editor found on most Unix systems. Vim provides more capabilities making it more suitable for programming etc. Basic knowledge of classic vi usage should be considered mandatory if you consider doing systems administration. It is commonly the only usable editor available on bare installs of classic Unix operating systems such as Solaris and AIX, and once you are used to vi style editing vim is the logical choice in environments where you can install an editor of your choice.

Graphviz

`http://www.graphviz.org/`

A tool to visualize graphs described in a text file. The ideal tool to create an elegant graph, especially for us coders and system administrators without much graphical skills. Just enter the graph in the text-based language and let the Graphviz utilities worry about the layout.

LaTeX

`http://www.latex-project.org/`

The de-facto standard for scientific reports. It allows you to concentrate on writing content, the result just looks good by default.

Wikibooks LaTeX

http://en.wikibooks.org/wiki/LaTeX

A good go-to resource for solving common LaTeX issues.

The not so Short Introduction to LaTeX 2$_\varepsilon$ [27]

http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf

A rather comprehensive introduction to writing documents in LaTeX, a bit too comprehensive perhaps but a good reference on how to do most everything you will need in a document.