department of computer science
faculty of sciences

vrije Universiteit amsterdam

Thesis: **Master of Science**
Specialization: **Computer Science**

Monitoring in Globule

Maarten Sander
1096427

July 2006

Supervisor:
Dr. Guillaume Pierre

Department of Computer Science
Vrije Universiteit Amsterdam

Contents

Abstract

This thesis presents a flexible framework for building complex monitoring systems. The system architecture is based on the pipes and filters, allowing one to create a complex monitor from many smaller components. This allows for great flexibility, since modifications to the monitor generally affect very few components.

The flexibility of the framework, however, comes at a cost: it is possible to incorrectly connect components and create invalid pipelines. As part of the monitoring framework, we introduce a simple way of validating pipelines based on the observation that pipelines can be represented as directed graphs.

Finally, this thesis introduces what-if analysis for Globule. What-if analysis allows an administrator to find the optimal configuration of a website given a certain traffic, without modifying the configuration of the actual server. Such analysis can become a valuable tool for system administrators to better understand the behavior of a complex system such as Globule.

Chapter 1

Introduction

The use of the internet has grown tremendously since its first steps in the seventies and eighties. Nowadays, millions of users are active on the Internet on a daily basis. Whenever a major news event occurs, those users tend to look for more information on the internet. Websites like CNN.com and BBC.co.uk receive hundreds of thousands of hits at peak times; during high-profile media events, visitors request so much information that a single server would not be able to keep up with the requests.

To help spread large amounts of requests amongst a group of servers, an interesting solution has been developed, namely the use of content distribution networks (CDNs) [19]. A CDN offloads a master server by distributing the requested documents over a network of edge servers, and redirecting users to a server that is able to handle their requests and is relatively close to them, topographically or otherwise.

To assess the status of their servers, administrators use specialized tools to gather information. Monitoring a single webserver is relatively simple, as it is for simple groups of multiple servers (like those behind a load balancer). A webserver's basic mode of operation consists of only a few steps: a client issues a request that arrives at the server, which in turn processes that request and sends the reply back to the client. Examples of relevant metrics in this scenario are the amount of requests that arrive during a specific period, the average time it takes to process a request and the location of the client.

Content distribution networks are much more complex in this regard. Apart from handling requests like a normal webserver, each node in the CDN also needs to interact with other nodes. For example, request logs need to be exchanged, clients need to be redirected, documents need to be distributed amongst the nodes and document consistency needs to be maintained. So, in a CDN, more metrics need to be measured than when monitoring an individual server. Also, site administrators often need information about an entire site, which is now spread over many locations. In some cases it may even be desirable to have status information on the entire network. Clearly, monitoring individual servers is not enough for a CDN.

Apart from passively monitoring the status of their servers, administrators often need to make decisions about them as well. When managing a group of web servers, questions on capacity planning are quite common. For CDNs, even more decisions need to be made, for example about the placement of new nodes. To make these kinds of decisions, the administrator will preferably need to know the effect of an action before actually performing that action. However, the complexity of the system makes such predictions quite difficult. Since interactions are so complex when dealing with a global network of CDN nodes, a tool that assists an administrator with making decisions would be extremely useful.

This thesis introduces a monitoring infrastructure, based on relatively small components that can be used to construct more complex monitor components in a Lego-like fashion. Using this

framework, I have designed a flexible and extensible monitor for the Globule content distribution network. Globule, created by the group supporting me in writing this thesis, had, at the time of starting this thesis, only basic monitoring functionally built in [20].

Using such a flexible monitor, however, comes at a cost. When connecting components to build a monitor, it is easy for the administrator to create a non-functional monitor. For example, it is possible to create data flows that contain cycles, causing the monitor to go into an endless loop, or to create constructions through which data do not flow at all, leaving the administrator with a monitor application that does not monitor. In order to solve this problem, I propose a method for validating a collection of connected components using standard graph checking algorithms, with the assistance of some component-specific knowledge. This method detects erroneous component configurations automatically and rejects them before they can be used.

This thesis also introduces a form of analysis which we will call 'what-if analysis'. It basically allows administrators to ask questions of the form "*what* would happen to my site *if* X happened, under these specific circumstances?" Using the new monitoring framework and a few simple changes to Globule, I have designed a tool that analyze all kinds of different hypothetical scenarios, assisting the administrator in making decisions about his web hosting system.

The rest of this thesis is structured as follows. Chapter 2 reviews existing monitoring solutions from the point of view of their features and their architecture. Chapter 3 introduces Globule and Apache, the webserver Globule is built upon. Chapter 4 explains the monitoring framework created for this thesis, and Chapter 5 describes the what-if analysis. Finally, Chapter 6 concludes.

Chapter 2

Related Work

Monitoring a distributed system does not only mean collecting data about it; in most cases it is desirable to process the data in some way such that it will be useful for the user (either a person or another application).

One of the most obvious uses for a monitor is debugging in general. In many cases, the data a monitor provides will only become useful when something is not working as advertised. When a system is not performing well or is generating errors, the monitor may provide answers on the cause of the problem. A monitoring solution like NetLogger can help a developer identify performance bottlenecks in an application, for example [11].

Applications can also use data generated by a monitor to adapt themselves to a changing execution context. For example, a node in the CoDeeN content distribution network uses information collected from other nodes in the network to create its own view of the network, and using this information it decides whether it should avoid certain nodes when redirecting clients [17].

Besides just processing data and presenting them to the user, a monitor can also be used to perform certain forms of analysis, like what-if analysis. It is often desirable to know in advance how a system will behave when its environment and input changes. However, it is not always possible to test such a change in a live system, since it may cost too much or may just be impractical. For example, testing what would happen if half of a CDN's nodes go down on a live system is not a good idea. One could think of running the test on a shadow copy of the CDN, but setting up an identical copy is very expensive. What-if analysis provides a means of performing those experiments without actually changing the system. A monitor can use metrics of a live system to perform this type of analysis.

An example of a monitor that can perform a basic form of what-if analysis is the Medusa proxy, a 'tool for exploring user-perceived web performance' [12]. It is a non-caching proxy server with the ability to mirror and compare requests to different web delivery systems simultaneously, and with the ability to transparently rewrite requests. These two features can be used to directly compare the difference between a normal request and a request to, for example, a CDN in real time, but also to perform basic what-if analysis.

The proxy records a log of the sites a user visits during a certain time period; at a later time, these logs can be replayed. Instead of directly replaying those logs, Medusa can rewrite the URLs, for example to remove references to CDNs. Now a comparison can be made of the perceived performance of the web with or without using CDNs [4]. These comparisons can also be done live (when the user accesses the proxy) instead of at a later time, but then bandwidth limitations may influence the results. The main strength of this approach is at the same time its weakness: Medusa uses real requests to get data, but this also means that there are no guarantees about the environment.

category	*static*	*dynamic*
hardware	connected devices	disk S.M.A.R.T. status; CPU temperature
operating system	kernel version	system uptime; number of users
processes	process limit	amount of running processes; CPU usage
file system	partition size	disk usage; swap activity
memory	installed memory	memory usage; memory fragmentation
network	link speed; IP address	bandwidth usage; amount of packet loss

Table 2.1: Examples of generic metrics

Finally, one could think of other uses for a monitor, such as monitoring user behavior for marketing purposes, like providing personalized advertisements on a website or sending marketing e-mails to people according to their tastes. HitBox Professional and ClickTracks Pro are products that both perform these kinds of analysis [25, 7]. However, these subjects are outside the scope of this thesis and will therefore not be discussed any further.

2.1 Features

Many monitoring solutions are specific to a given application, as different applications typically need to monitor different sets of metrics. However, classes of applications often have similar needs. Here we will introduce a selection of the various types of metrics that can be measured. First the different metrics themselves are discussed, then we will focus on how these metrics can be processed.

2.1.1 Metrics

In a given computer system, there are many things you may want to measure. Sometimes the data to collect are static and thus read only once, like the version of the operating system or the amount of memory installed. More often, measuring involves changing (dynamic) data, like the number of processes currently running or the amount of memory currently in use. This kind of data needs constant monitoring if we want to be able to draw conclusions from the changes.

In most cases, data is measured to assess system performance. There are many generic metrics that might be interesting when analyzing performance, like the current load of the machine, but in order to determine the performance of a given application, one will have to monitor other kinds of metrics that allow for a more detailed view of the application.

A lot of available monitoring solutions measure just a set of generic metrics, i.e. they measure data that are commonly available on every computer system. These metrics are mostly at the operating system level. Table 2.1 shows a categorization of such metrics and some examples for every category. These kind of metrics can be useful when debugging a faulty system or when restricting general resource usage, for example.

The are a lot of tools available for measuring generic metrics [10, 16]. ZABBIX is an example of a very complete generic monitoring solution [24]; it primarily provides functionality for monitoring network-related metrics, but can be configured to monitor practically any kind of metric with some effort. This can be accomplished by using user defined scripts and the SNMP protocol [6].

Apart from the common generic system monitors, there are more specialized monitors for all kinds of applications, like web and FTP servers. Systems running these applications need to monitor different metrics than the generic metrics discussed above, since most of the time

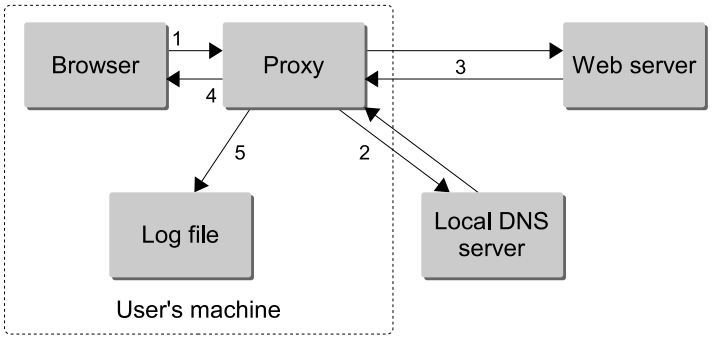


Figure 2.1: Architecture of the monitoring proxy by Liston and Zegura.

those metrics are too low-level. For example, when monitoring webserver software, metrics like the number of disk writes per second are not directly relevant, but metrics like document access times, request rate and the number of referrals per website certainly are.

Only when the application-specific metrics indicate that there may be a problem, the more generic metrics become relevant. It is often the case that problems in the lower levels of a system exhibit themselves at higher levels. For example, when a webserver suddenly starts serving pages at a much lower rate than usual, it might be the case that the system is running out of memory and starts using the swap space more and more, resulting in degraded performance.

To conserve resources, it would be useful if a monitor has the ability to switch monitoring components on and off at runtime. That way, it is possible to start measuring the more generic metrics only when the need arises.

An example of an application-specific monitor is the proxy created by Liston and Zegura [13]. When accessing the internet, users often experience performance problems without an obvious cause. This proxy can be used to identify such problems, but especially DNS server problems. As seen in figure 2.1, it intercepts client requests to a webserver (1), queries a DNS server for the domain (2) and sends the request to the destination webserver (3). It then returns the response received from the webserver to the client (4) and finally logs the requests the client makes, the time it takes to query the DNS server and the time it takes for the webserver to respond to the request (5). Unfortunately, this method of monitoring cannot be used on production web sites, since it requires the user to install additional software on his or her PC and requires modifications to the browser configuration. It is, however, a good example of how an application-specific monitor can be used to identify problems.

2.1.2 Metrics in CDNs

Monitoring distributed systems and applications is a very broad subject and most of the time the metrics depend entirely on the application domain. Apart from that, gathering metrics in a distributed system is usually much more difficult than gathering metrics in a stand-alone system; data are spread over multiple nodes, so there needs to be an efficient means of retrieving data from other nodes. A lot has been written on this subject already, see for example [2]. However, this is beyond the subject of this thesis, so we will only focus on metrics applicable to content distribution networks.

The metrics that are interesting when monitoring a CDN are basically a superset of metrics that are interesting when monitoring a stand-alone webserver. In addition to metrics like document request rate and bandwidth usage, metrics like the document replica placement policy, node status and the total number of issued redirects are very important.

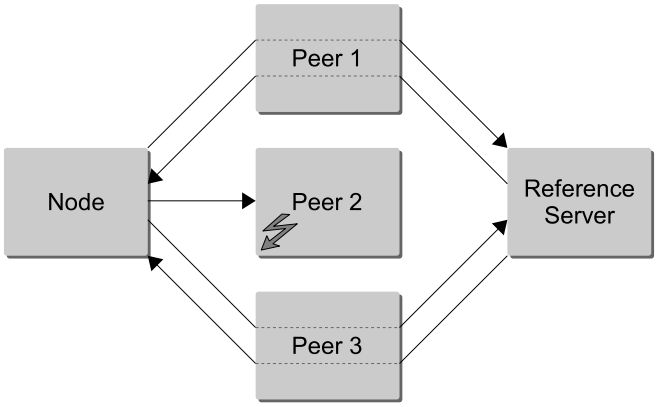


Figure 2.2: Checking HTTP availability in CoDeeN. Arrows indicate HTTP requests through a proxy. Peer 2 is currently failing.

In the CoDeeN network, each node maintains a database of information on all other nodes, which it uses to determine whether it should avoid certain nodes when redirecting incoming requests [17]. The metrics it uses to determine availability of other nodes are as follows:

- load average
- CPU usage
- availability of file descriptors
- proxy and node uptime
- average hourly traffic
- DNS failure statistics
- DNS latency statistics

These metrics are piggybacked on the response (ACK) to a UDP heart beat message that each node sends every second to one of its peers. However, these ACKs can get lost or delayed, providing two other metrics that can give some insight to the state of a node:

- amount of missed ACKs
- amount of late ACKs

A final metric CoDeeN uses is HTTP availability. Since network ports can be filtered or blocked, the UDP heart beat may work while HTTP connections are not possible. To check HTTP availability, a node iterates over each of its peers, performing a simple HTTP fetch test by requesting a non-cachable document on a known working server using the current peer as a proxy. This process is illustrated in figure 2.2.

2.1.3 Generating Data

After deciding which metrics must be monitored for a certain application, a decision has to be made on how to collect the data the monitor requires. From this point of view, roughly three categories of data can be distinguished: application-generated data, intercepted data and monitor-generated data.

Quite often, the data being measured is generated on purpose by the application itself. The application will, for example, write diagnostic messages to a log file or periodically display its status on standard output. A monitor can use this information to present them to the administrator. For example, log analyzers like Webalizer and AWStats parse and analyze the log files generated by the Apache webserver (or any other webserver that generates log files in the same format), generating statistics about the visitors and the popularity of the websites hosted on the server [1, 8].

The advantage of letting the application generate data is that it is possible to provide the monitor with very specific details on how the application is performing. It is also potentially very efficient, since in the ideal case the application does not have to do any extra work to send data to the monitor. The application does not even need to be aware that it is being monitored. A disadvantage of using only application-generated data in the monitor is that the application might not provide all the metrics that are needed for the monitor. If other metrics need monitoring, the application itself will need to be modified, or another monitoring method has to be used.

When monitoring a distributed system, retrieving these kinds of data from all nodes can be very difficult. Ganglia solves this problem by providing a command-line application and a client-side library that applications can use to publish data [15]. The data is sent to a multicast channel, to which monitor daemons can subscribe. Another possibility is to create a stub on each node that can accept connections from monitors. Such a stub reads application-generated data and on request provides the monitor with these metrics.

Intercepting the data streams to and from the application (or its components) is another way of generating data for a monitor. This technique, also called 'sniffing', is quite popular amongst network monitors. It can be used to gather standard network metrics like throughput and latency, but it is also possible to reconstruct low-level network packets into packets belonging to higher level protocols, allowing a monitor to measure application-specific metrics. Pandora tries to solve at least part of this problem by providing the user a way of reconstructing data from network packets. [18]

Of course, network connections are not the only kind of data streams that can be monitored by interception; it is also possible to capture other data streams that exist between components of an application. For example, if two components are connected by a UNIX pipe (using a shell command), it might be possible to insert a filter application between the two components that will intercept all data.

The biggest advantage of this approach is that it does not affect the application involved; the application is not aware that it is being monitored. Intercepting data also allows for monitoring metrics that the application itself does not provide. A disadvantage of intercepting data is that it may take a lot of work and a lot of application-specific knowledge to be able to reconstruct useful data from the streams.

A different way of tracking data as it 'flows' through an application is provided by the NetLogger toolkit [11]. However, the application needs to be modified for this to work: NetLogger allows a programmer to specify certain checkpoints in his application. When a piece of data (an object) reaches such a checkpoint, a message is sent to a log server. This message contains a timestamp, its origin (location in the application) and a unique identifier for that object, but could possibly contain more information relevant for the application. The timestamp and the identifier will assist the monitor in tracking the object through the system; at any given time the monitor will know where the object currently is.

The final category comprises monitors that work by actively providing the application with some kind of test input (a request), and then checking the output (the response) against a set of known valid responses. For this, the response to any request the monitor can send needs to

be well-defined. If the response does not meet expectations, the monitor will know something is going wrong; the monitor derives the application status from the response the application gives. An example of this kind of data generation is provided by CoDeeN. Its HTTP heart beat, as discussed in section 2.1.2, actively checks the response of other nodes. If a destination node responds incorrectly, or does not respond at all, there may be something wrong. Additionally to checking the response, it is also possible to measure other types of data, like the amount of time it takes to process a request or, for a networked application, the number of 'hops' between the client and the server.

2.1.4 Processing Data

Generally, data generated by taking measurements needs to be processed before being displayed to the user in order to be useful. For example, when looking at the average document request rate, it is usually more interesting to look at the combined average request rate of all documents instead of that of a single document.

Data can be processed either in the application itself, or in a separate application (see also section 2.2.2). If an external application is to process the data, the monitor needs to output the data in a structured way such that the external application can read it. It can output its data directly to a log file or to a log server, but it can also directly supply the data to the processor application on request [14].

When efficiency in terms of resource usage is desirable, a better option is to supply data only on request. This way, the monitor is only active when the data processor is active. On the other hand, when analyzing large sets of data, processing log files can be more efficient in terms of system performance, since it can be done when the system is not busy or even on a different machine. The next section will discuss this in more detail.

2.2 Architecture

The architecture of a monitor for a large part dictates how it affects the application being monitored, but it also influences the direct usefulness of the monitor. Depending on what time data analysis takes place, performance of the server may be affected; performing analysis real-time can use up precious CPU time. However, delaying analysis is probably not a good idea when the monitored application actually uses the results to change its own behavior.

When a monitor requires that the application is modified in order to be able to monitor certain metrics, it may affect the willingness of an administrator to install the monitor. Even worse, in the case of software where the source is not available, it is impossible to directly modify the application. Installing a stand-alone monitor that requires no changes to the existing application at all seems a better solution, were it not that there is no guarantee that all metrics needed by the monitor are measurable outside the application. If these metrics are indeed not measurable, the first option may be preferable after all.

Finally, a monitor's flexibility and its ability to be extended influences its usefulness. Extending the functionality of the monitor (to monitor new metrics, for example) by a script or writing a new monitoring component is preferable to depending on the monitor's vendor for new functionality. Also, turning on and off certain parts of the monitor or modifiable behavior of components is preferable over a single static system.

2.2.1 Time of Analysis

Depending on the application, the measurements made by the monitor may be analyzed at different times. In some cases you may want to analyze the data immediately (*online*) so the

application can benefit from the results, but in other cases it may be better to analyze the data after data collection has taken place (*offline*).

Real-Time Analysis Online analysis is the kind of analysis that is performed when the application is still running; measurements are sent directly to a module or application that will process them. Performing this type of *real-time* analysis is particularly useful when the behavior of an application needs to change when its execution context changes, or when live monitoring of the application is desired. For example, each node in the CoDeeN network uses the data it receives from other nodes to build its own view of the network, and uses it to determine which nodes to avoid when redirecting clients.

Another example that illustrates the use of real-time data analysis is an alert service. When an error (or an exceptional system state) in the monitored application is detected, the monitor can alert the system operator, for example by sending an e-mail or SMS message. The operator can then take the appropriate actions. ZABBIX provides a flexible way to implement notification. It knows about two kinds of media that can be used for notifications: e-mail and script. The script media indicates a user-defined script that can be used to define a custom notification type. ZABBIX provides this script with three parameters, namely the recipient, the subject and the message itself. The e-mail media just sends an e-mail to the specified recipient.

Online analysis may become a problem when a lot of data needs to be analyzed; the monitor might not be able to keep up with the stream of incoming measurements. In order to try to do as much as possible, the system may have to throw away important measurements, so it is important not to perform all types of analysis at all times.

Post-Mortem Analysis It is also possible to analyze offline, meaning that the measurements are stored in log files instead of directly monitoring a system and are processed after they have taken place (*post mortem*). For continuously running applications, the analysis may be performed periodically.

Post-mortem analysis is particularly useful if you need to correlate certain events that happen in your system. For example, when monitoring a webserver, you may want to track a certain user as he visits several pages of a web site. When doing this in an online fashion, you need to keep a history of visits, for example those during the last ten minutes, in memory. This may become a problem when analyzing high traffic web sites, since the history may use a lot of resources.

The data generated by the proxy created by Liston and Zigura is mostly processed offline. The proxy does some simple analysis by itself, like determining the beginning and end of the execution of JavaScripts on a web page, but grouping requests that belong to a single page is done in post processing.

Offline analysis is also very useful when you want to do more intensive analysis than usual. When processing two minutes worth of data takes, say, more than two hours, it is neither possible nor desirable to do this online. Processing the data offline is then the solution.

Semi-Real-Time Analysis When analyzing data offline for a continuously running system, it is possible to decrease the interval in which no analysis takes place. When the interval becomes small enough, for the user the monitor may appear to be analyzing the measurements in real time. In fact, this semi-real-time behavior may be good enough for a lot of applications. For example, a web master of a high traffic website may not care if the statistics are updated whenever a new visitor accesses his web site; an update every 5 minutes, or perhaps even every hour, may be good enough.

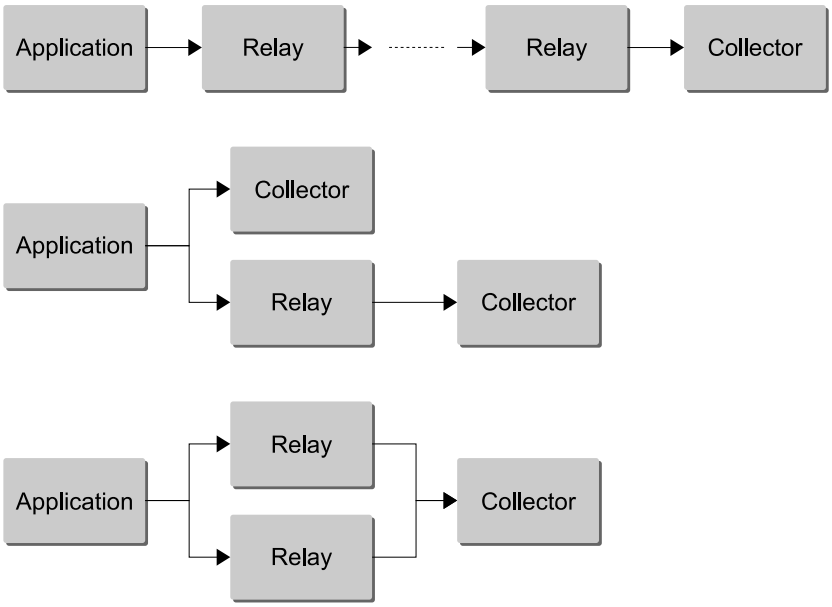


Figure 2.3: Examples of possible syslog architectures.

Log analyzers perform their measurements offline, but Webalizer can either analyze a complete log file with a month worth of data at once, or periodically (e.g. hourly or daily) analyze a log file containing only a part of the data. Processing partial log files is an example of semi-real-time analysis, especially when the period between runs is kept relative small, for example 5 minutes.

2.2.2 Structure

Once data have been measured, one needs to process them to generate actual analysis. The algorithms used differ per application, but in general, there are only a few places where this analysis can take place. The analysis can be performed in the application itself, in an external library or in a completely separate application.

Each method has its pros and cons, as discussed below. It is also possible to use a combination of the methods described above, for example by performing simple analysis in the application itself and the more complex ones in a different specialized application.

Performing analysis in the application itself is mostly used by applications that immediately need results from the analysis, for example to be able to adapt to certain changes in the environment. Data are sent to a component that will perform the analysis. After processing, the results are ready to be used. If the data are required by other applications as well, care has to be taken to make all these data available, for example in a log file.

An external library containing functionality for performing analysis may be specifically designed for the application, but is usually a third party library that is being integrated in the application. The application sends data to the library using a well-known API. The advantage of using an external library is that it is possible to modify (or swap) the analyzer without having to modify the application. This is particularly useful when monitoring an application without having access to the source code of that application, but one that does have a method to dynamically load libraries (i.e. some sort of a plug-in infrastructure).

When analyzing data in a separate application, the measurements are often communicated to the application using some sort of log file as transportation medium, but also by sending

data directly to the analyzer, either on request by the analyzer (*pull*) or when new data is available in the application (*push*). These two concepts can be combined by creating a log server. Such a server accepts log messages from different clients and writes them to a log file on permanent storage. Analyzers can then read those logs and analyze them. The BSD syslog standard implements and extends this particular scheme, supporting multiple log servers (collectors), but also servers that only forward requests (relays) [14]. Figure 2.3 illustrates the use of collectors and relays.

The NetLogger toolkit can also use log servers for gathering data [11]. The application that needs monitoring has to use the client-side library supplied with NetLogger to send data to a log. This log can be stored either in memory, a local file, or a remote log server (*syslogd* or NetLogger's *netlogd*, for example). Logging to a remote server is particularly useful when monitoring multiple applications or monitoring across a wide area network. For the monitored application there is no difference between sending logs to a file or to a server, so virtually no changes are necessary when it is decided that a different method is to be used.

At first sight, processing data in the application itself or in an external library seems not ideal when you want to monitor combined data from more than one application (for example in a distributed environment) or when doing real-time analysis. If you want to combine measurements from several applications, or if you want to be immediately notified when a significant event occurs, there needs to be a way to push data to a central application, either a 'master' application or a stand-alone specialized application that only gathers data and performs analysis. Processing data in a separate application is more suitable for this, as explained above. However, it is not impossible to use the other methods for analyzing measurements in a distributed environment. For example, it is possible to have an application periodically gather these metrics from each monitored application. However, using this pull based method is not really suitable when real-time analysis is needed, since it would require very small intervals in between requests, and thus could consume a lot of resources. It is more suitable for semi-real-time and offline analysis.

2.2.3 Flexibility and Extensibility

As seen in the previous section, it is often desirable to add or modify existing sensors. This can be because of the need to measure new metrics, or because existing sensors are not suitable for the current application. Many monitors only provide a fixed set of metrics that can be monitored, but there are also monitors that allow users to define their own sensors. However, most of the time the monitor is not aware of the specifics of a sensor and does not know exactly how to handle the data a sensor provides. This requires the user to adapt the sensor to the monitor's needs.

A solution to this problem is to allow users to also define their own ways of processing the generated data and to modify existing data processors. The monitor does not need to be aware of the specifics of the sensor, as long as the user defined data processors do. Unfortunately, this feature is not often found in existing monitors.

Many monitors either have a user interface for monitoring the status of the application, or log the status of the application to a file. It may be desirable to process the data in a different application, or generate reports from the monitor's output. However, there is a problem if the reporting application cannot read the data generated by the monitor. If that is the case, it may be desirably to change the output of the monitor or to create additional functionality for different output formats.

This all means that the monitor needs to be really flexible in terms of adding and modifying parts of the entire monitoring process. The ability of adding sensors is found reasonably often,

and many monitors offer different output formats, but the ability to modify the way data is processed and analyzed is practically unavailable in existing monitors. A positive exception is NetLogger, which includes functionality to visualize and analyze the logged data, but the applicability of NetLogger is very small when looking at content distribution networks.

2.2.4 Scalability

The architecture of a monitor also influences the scalability of a system. For example, Ganglia allows the user to set up a hierarchical network of monitor components, separating the nodes into clusters, with each cluster having one node that represents that cluster (the nodes in a cluster share all monitoring data from that cluster) [15]. Data is collected by nodes higher up the hierarchy by periodically polling the cluster representatives, eventually reaching the client. This allows for great scalability, since the amount of data sent is much smaller than when every node has to be polled individually.

2.3 Conclusion

The most important feature of a monitoring system is flexibility. Different users are not interested in the same metrics, and everybody has his own ideas on how to process and display the measurements. The architecture of a monitor defines for a great part its flexibility. Splitting up the system into smaller individual customizable parts helps increasing flexibility. Depending on the needs of the user, measurements can be processed and analyzed either online or offline. The monitor needs to be able to communicate with other applications using a standard format to allow them to analyze the measurements offline. Especially for content distribution networks it is important to adapt to a new environment, so the data a monitor generates should also be available to the monitored application, preferably in real-time. Finally, what-if analysis is a very useful tool for administrators that helps them make decisions, and should be offered as part of the monitor.

Chapter 3

Apache and Globule

The monitor described in this thesis is written for the Globule CDN. Since Globule is written as a module for the Apache webserver, it is important to know how Apache works internally. This chapter explains the basic architecture of Apache and how modules are integrated in the server. It also explains the architecture of Globule and the details of how this particular module is integrated into Apache, and finally it describes how the monitor fits into the Globule design.

3.1 Apache

Apache is the most widely used open source webserver software. It is a highly configurable and extensible webserver that can be customized by writing modules using the Apache module API [3].

This section will explain how these modules are integrated into Apache (version 2.0.49 is used for this thesis) and, since Apache uses multiple instances to serve clients, how it is possible to communicate with another another instance of a module. Apache has several ways of serving multiple clients at the same time (forking new processes, creating new threads and a hybrid) and selects one at compile time, depending on the underlying operating system, so we details this technique first.

3.1.1 Handling Requests

Whenever Apache receives a request, it splits the operations necessary to process the request into a number of phases [9]:

- URI-to-filename translation
- authentication
- authorization
- other access checks
- MIME media type determination of the requested object
- 'fix up'
- content handling
- request logging

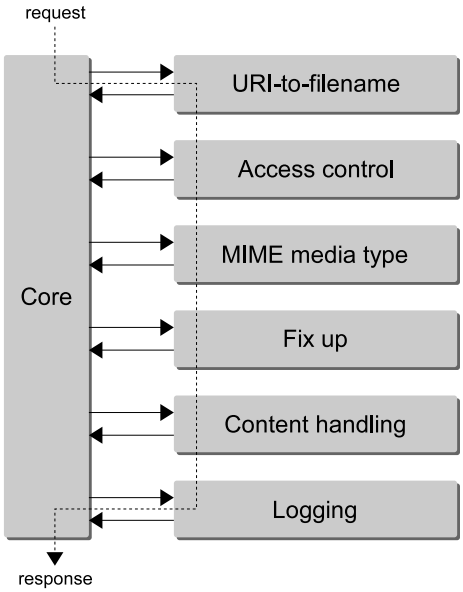


Figure 3.1: The Apache core controls the flow of the data.

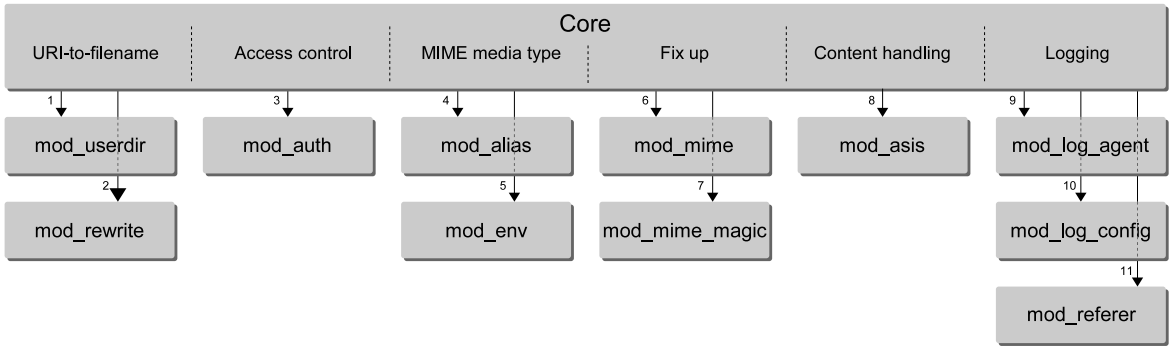


Figure 3.2: High level architecture of Apache. This figure also shows the predetermined order in which the example modules are called.

Each of these phases is handled by one or more modules. The Apache core controls the flow of the data, as illustrated in figure 3.1. URI-to-filename translation determines where the requested resource is located in storage. Access control performs authentication (checking if the user is really who he says he is), authorization (checking if the user is allowed to access this object) and other kinds of access checking. Determining the MIME type helps the client to identify the type of object that is returned, for example an image (image/jpeg), a text file (text/plain) or an HTML file (text/html). The fix up phase allows for miscellaneous fixes, for example fixing typos in URIs or replacing aliases by their actual paths. Content handling generates the content that will be included in the response to the client.

3.1.2 Modules

An overview of the Apache architecture is shown in figure 3.2. The modular structure of the webserver is represented by a single core component and a number of modules connected to the core. The core interacts with the client, handing off certain tasks to the modules.

A module in Apache is basically a set of functions that perform a specific task. For example,

a module could (partly) implement one of the major phases, or perhaps add new functionality altogether, such as the ability to compress the response sent to the client. Modules can be loaded dynamically, so it is a very convenient way to extend the basic functionality of Apache without having to modify the Apache source code itself.

If a module wants to be called in a certain phase, it needs to register itself for that phase. More specifically, it needs to register itself for a certain hook in that phase. A hook is a point in the execution of an Apache process where modules are called. A module can register itself for a hook during initialization by calling the right `ap_hook_`*`hook name`* function, with a function name as its argument. This argument names a function that will handle the hook, i.e. it is the name of the function that Apache will call at that hook. A module can install as many hook handlers as it wishes, so it is no problem for a module to be active in more than one phase.

To illustrate the wide range of available hooks, here follows a list of all hooks declared in the file *http_request.h*, in the order they run:

- *create_request*
- *translate_name*
- *map_to_storage*
- *access_checker*
- *check_user_id*
- *auth_checker*
- *type_checker*
- *fixups*
- *insert_filter*

3.1.3 Multi-Processing Modules

The previous series of Apache releases (versions 1.3.x) was built to be very portable, but lower-level functionality, like binding to network ports, accepting requests and dispatching children to handle requests, was less portable. For example, on Windows, Apache used the networking features of a POSIX compatibility layer instead of native ones. However, in the current releases of the Apache webserver (versions 2.x), things have changed.

Now, even the most basic functions are built to be modular. The lower-level functionality mentioned above is has moved to a module-based structure, just like the rest of Apache. The modules that provide these basic features are called Multi-Processing Modules, or MPMs for short. By using native networking features, *mpm_winnt* made the Windows version of Apache much more efficient, for example.

Currently, nine MPMs are available for Apache. The following list describes these MPMs briefly:

- *beos*: This is the default MPM for BeOS. It uses a single control process which creates worker threads to handle the incoming requests.
- *leader*: An experimental variant of *worker*.
- *mpm_netware*: This is the default MPM for Novell NetWare. Like *beos*, this MPM is fully threaded, but optimized for NetWare.

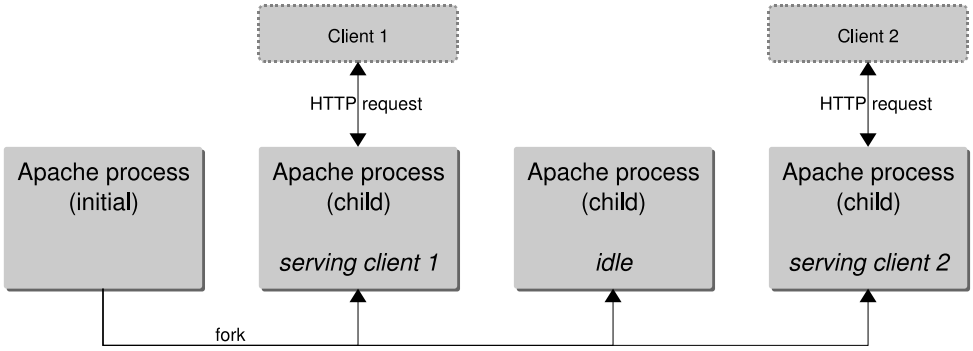


Figure 3.3: Apache has initially forked three children, making a total of four. One child is currently idle.

- *mpmt_os2*: This is the default MPM for OS/2. This is a hybrid multi-process, multi-threaded MPM. It uses a main parent process that manages a small static number of child processes. Each of those child processes consists of a pool of worker threads and a main thread that manages them.
- *perchild*: An MPM that allows for daemon processes that serve incoming requests, each assigned to different user ID.
- *prefork*: This is the default MPM for Unix-like operating systems. It implements a non-threaded preforking webserver. It handles incoming requests similar to the way Apache 1.3 did.
- *threadpool*: Another experimental variant of *worker*.
- *mpm_winnt*: This is the default MPM for Windows NT. It uses a single control process that launches a single process. This worker process in turn creates threads that handle the incoming requests.
- *worker*: An MPM that implements a hybrid multi-process, multi-threaded webserver.

The *prefork* MPM is illustrated in figure 3.3. In the rest of this chapter, it is assumed that Apache is configured using this module. Therefore, only the term 'process' is used, but this can be substituted with 'thread' if appropriate.

3.1.4 Apache Portable Runtime

Apache is designed to be portable to many different platforms. Since Globule was built as an Apache module, it is logical to design Globule and the monitor to be portable as well. To this end, software written for Apache can use a library that hides all platform specific issues, called the Apache Portable Runtime (APR). The APR is an API, originally created as part of the Apache webserver, that provides a platform independent interface to platform dependent implementations. This allows developers to write software that will run on multiple platforms without worrying about platform-specific issues.

3.1.5 Inter-Process Communication

Sometimes it is desirable to have some way of communicating between multiple Apache processes. This can be done using facilities provided by the operating system, like sockets, (named)

pipes or semaphores, but the availability of these facilities generally depends on the operating system Apache is running on. To avoid the dependency on operating system functionality, the APR supplies at least two commonly used methods of inter-process communication, namely shared memory and mutexes.

The APR allows the programmer to create, use and destroy segments of memory that are shared between all processes. This means that all processes can see data found in these segments of memory. This also means that those processes may try to read or write the same memory area at the same time. Simultaneous reading is not a problem, but when two processes try to alter some shared data, the outcome is undetermined. This problem can be solved by letting the process lock the area that will be needed.

For this, the APR provides functionality for creating so-called mutexes. A mutex, short for mutual exclusion, is a piece of memory that can be locked and unlocked in one atomic operation. When a process needs to perform an action on a shared resource without interference from other processes (for example, writing to shared memory), it tries to acquire a lock on the mutex. If this succeeds, the process proceeds by performing the action and finally unlocking the mutex. If the process is not able to acquire a lock, it blocks until the lock is freed and then acquires it. Of course, using mutexes will only work when all processes adhere to this protocol.

3.2 Globule

The Globule CDN was built as a module for Apache, called *mod_globule*. This has several advantages. First, it allows for easy integration of the CDN into existing webservers, relieving administrators from having to install and configure completely new software if they want to run a CDN. Another advantage is that Globule developers can always use the APR for portability.

3.2.1 Apache Module

The Globule module (version 1.1.3 is used for this thesis) registers itself for the following hooks and maps them to the mentioned functions:

- *post_config*: `globule_post_config`
- *child_init*: `globule_child_init`
- *translate_name*: `globule_http_redirect`
- *log_transaction*: `globule_log`
- *handler*: `globule_handler`

When DNS redirection is enabled, the module also registers itself for the following hooks:

- *process_connection*: `dns_process_connection`
- *log_transaction*: `dns_http_cleanup`

Apache executes the *post_config* hook after it reads the configuration file. Apart from processing configuration directives, Globule uses this hook to initialize shared memory, to initialize the cache, to initialize the resource pools and to create an 'event manager' (more on this later).

Whenever a requests comes in, Apache calls the *translate_name* hook. Globule uses this hook to check if the request should be redirected to a slave replica. After *translate_name*, the *handler* hook is run. Here the actual request is processed. When receiving a HTTP GET request, Globule creates an event containing that HTTP request, and then, using the event manager, sends it off to the component responsible for serving the document.

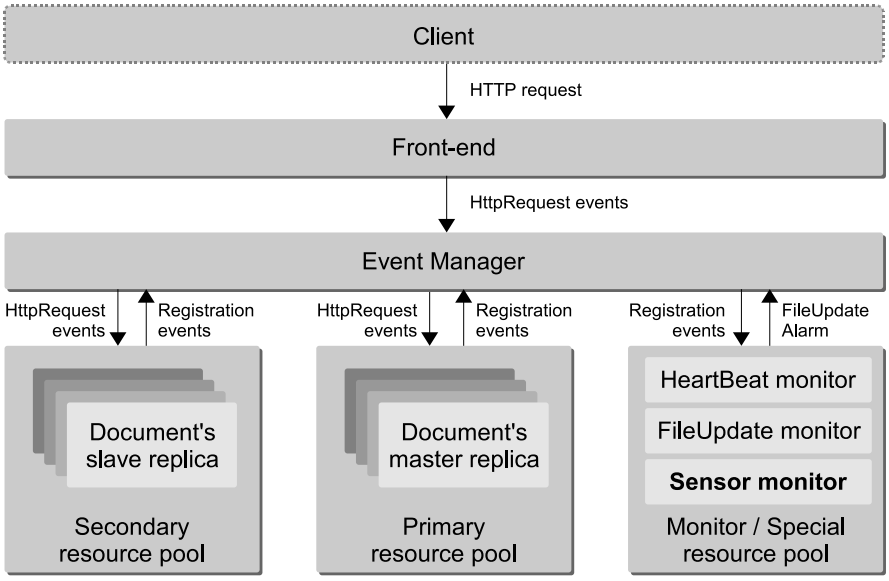


Figure 3.4: High level view of the Globule architecture.

3.2.2 Components

Figure 3.4 shows a high level view of the architecture of Globule[1]. It clearly shows the event-based nature of Globule. Almost all communication between components takes place using events. The EventManager accepts events derived from the GlobuleEvent base class. The destination of an event is specified by an EventReceiverId, consisting of a PoolId and a DocId. The former refers to a resource pool, the latter to a document.

This specific construction hints at the architecture of Globule. At startup Globule initializes several resource pools: one primary pool (also called the export pool since it represents a master copy of a site that is being replicated), zero or more secondary pools (representing replicated sites) and a different kind of pool, named the SpecialPool. The primary and secondary pools contain document objects (representing the documents on disk), but the SpecialPool contains so-called monitors. For example, the HeartBeat monitor generates HeartBeat events, and the FileUpdate monitor periodically (i.e. when a HeartBeat occurs) checks for changed files.

3.2.3 SensorMonitor

A *sensor monitor* is also included in figure 3.4. This monitor will be fully described in the next chapter, since it represents the monitor that this thesis discusses. However, in this section I will explain how the SensorMonitor object is integrated into Globule and how it receives external requests.

The SensorMonitor object is created by the SpecialPool when Globule calls the pool's `init` method. Mod_globule's handler for the *post_config* hook is modified to send the sensor monitor a notification when initialization is complete. This allows the sensor monitor to perform its own initialization that can only be done when Globule is fully up and running.

`Globule_http_redirect`, the *translate_name* handler, is also modified since it should never redirect the client when a request comes in for the sensor monitor. Currently, `/globulemon/` is used is used as the name of the (virtual) directory used for communication. As such, it is not

[1]Figure adapted from [20].

possible to have an identically named directory in the root of the webserver; Globule will ignore it.

The `globule_inner_handler` function is modified as well. It is called by `globule_handler` and does the actual work of handling a request. Modifications include the ability to parse special requests directed to the sensor monitor and to actually generate and send events containing the parsed requests.

Chapter 4

Monitor Design and Implementation

This chapter discusses the architecture of the Globule monitor. First the requirements are determined. Next, relevant design patterns are compared, followed by an explanation of the monitor's design. After this, the flexibility of the design is illustrated, and finally validation of pipelines is discussed.

4.1 Requirements

This section describes the requirements that a truly flexible CDN monitor will have to meet. First the functional requirements are determined, followed by the non-functional requirements that help meet the functional ones.

4.1.1 Functional Requirements

Basic Metrics First of all, the monitor should give interesting measures, so it needs to be able to monitor a wide range of metrics. The system should provide a large number of relevant metrics, but in addition the user should also ideally be able to supply his or her own sensors if the supplied sensors do not provide enough information.

Metrics on Remote Servers It should be possible to read metrics from all nodes in the CDN, and also to combine the readings from sensors at these remote sites with readings from local sensors.

Generic Data Processing There should not be any restriction on how data are being processed. The user should be given the ability to reuse existing components that are used for processing, but it should also be possible to add functionality in this respect.

Enabling and Disabling Parts of the Monitor Monitoring certain metrics may require quite some resources, so it is desirable to have the ability to toggle parts of the monitor on and off. For example, when a CDN needs to process more requests than usual, a number of monitoring components could be turned off to increase the amount of resources that can be spent on request processing, thereby improving performance. These components can then be turned on again in case of emergency or when traffic returns to normal levels.

Notifications The monitor should also be able to warn the user if a given condition becomes true (or when a given threshold is crossed), or give the user a daily update of the system's status. This could be by means of e-mail, SMS messages or a web page for example.

Interoperability Finally, it should be straightforward for third-party applications to interact with the monitor.

4.1.2 Non-functional Requirements

In order to reach the above goals, the following non-functional requirements have to be met:

Flexibility It should be possible to easily swap one component of the monitor with an alternative one, for example to adapt to a specific situation, to test a new implementation of a certain component or to upgrade a component. Also, it is necessary that the user is able to turn on and off parts of the monitor at runtime, so that currently irrelevant measurements will not be taken.

Extensibility It is impossible to know in advance all the metrics that need to be monitored and how the generated data will be processed, so it should be possible to add new components when desired. This allows the user to monitor different metrics than those monitored by the existing system, and allows the user to process data in all kinds of new ways.

Performance Since the Globule monitor will perform measurements online, it will have an impact on performance. However, the user visiting a monitored website should not be aware of the fact that measurements are being taken, so the monitor should have a minimal impact on system performance.

One way to provide flexibility and extensibility is to design the system to be modular. Each step in the system, such as reading, processing and outputting data, should be contained in its own component. This will help reaching those goals, but also help with debugging and maintaining code, since it keeps side effects and hidden dependencies minimal.

There are some major architectural patterns that can help us reach the above goals. The next section will discuss the advantages and disadvantages of two well-known and widely-used design patterns.

4.2 Architectural Design Patterns

The design goals presented above are quite common. To address them, [5] describes three major architectural design patterns that can be used when structuring an application into high level system subdivisions. We discuss the two that are applicable to our situation, namely the Layers pattern and the Pipes and Filters pattern, here. Using an existing pattern helps focusing on the structure of the application that is being built.

4.2.1 Layers

Structuring an application according to the Layers pattern consists of decomposing the application in separate subtasks that are at a particular level of abstraction, and placing them in a certain layer. The idea behind the Layers pattern is that a component in a given layer may only use the services in its own layer and those of the layer directly below it. This minimizes the dependencies between layers and keeps changes local. A client (usually the user, but could also be an external application) only interacts with the top layer and does not need any knowledge of the layers below the top one. The interfaces of the individual layers are strictly defined. This

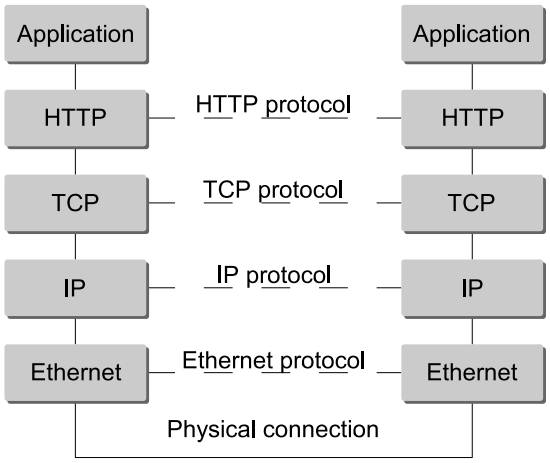


Figure 4.1: Communication between applications using HTTP

allows for exchanging a layer with a different one without having to modify components in the other layers.

Many applications are already subdivided in a kind of layered system, but often the layers are not truly separate. The Layers pattern tries to emphasize the boundaries between those layers to help improve the independence of the layers and consequently the flexibility of the application. Nice examples of layered systems are network protocols, like the well-known TCP/IP stack. TCP/IP is illustrated in figure 4.1.

Relaxed Layered System While the relations between layers in the standard Layers pattern are very restrictive, the relations in a Relaxed Layered System are much less strict. Instead of only using the one layer directly below the current layer, it is possible to use services from all layers below the current layer. Note that this does not mean that all services from all layers below the current one are available; a layer may be partially opaque and only allow access to a certain set of services. Relaxing the Layers pattern usually allows for better performance and better flexibility, but comes at the cost of losing maintainability. Individual layers depend on more than one layer, and changes in a particular layer will probably ripple through a bigger part of the application than when applying these changes to a Strict Layered System.

4.2.2 Pipes and Filters

The Pipes and Filters pattern is mainly used in systems that process streams of data, but it could be used for any system that processes data sequentially [23]. Each processing step is implemented by an independent component. A pipeline is a chain of components connected by so-called pipes. The component at the beginning of a such a chain is usually called a source or a sensor, while the component at the end is usually called a sink. Data are generated at the source, from where they 'flow' through all components until they reach a sink, where they are consumed (displayed on a screen or saved to disk, for example). The components between the source and the sink are called filters, since they have the ability to filter (i.e. transform) the data that flow through them.

Components The most important components in a processing pipeline are the filters; they are the actual processing units. The first filter in a pipeline receives its input from a data source, while the last filter outputs to a data sink. Pipes are used to synchronize and connect

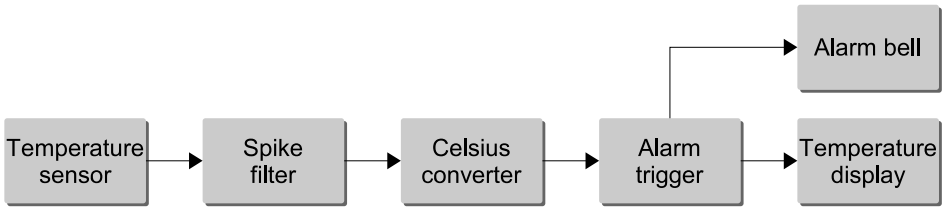


Figure 4.2: Temperature reading pipeline

the individual components. Examples of sources are (hardware) sensors, application output and text files; examples of sinks are terminals, visualization programs and files.

An example pipeline is shown in figure 4.2. The figure shows the different stages of a simple temperature sensor with alarm. First the raw values are read from the sensor and forwarded to the spike filter where unrealistically high or low values are filtered out. Next, the resulting raw values are converted to degrees Celsius. Now the processed values are fed into the alarm trigger. This component may trigger an alarm bell when, for example, the temperature goes below 0 degrees Celsius. The trigger also passes the values unmodified to another sink where the numbers are displayed on a screen.

Component Activity Components can either be active or passive. Active filters take control of the data flow, pulling its input from the previous component in the pipeline or pushing its output up the pipeline. Passive filters only respond to data flow triggered by other components, either when data are being pulled in by the next component or pushed out by the previous. An active source is a source that actively pushes data up to the first processing stage; a passive source only provides data when the first component needs it. Similarly, an active data sink will pull data from the pipeline, while a passive sink allows the final filter to push (i.e. write) data into it.

4.2.3 Conclusion

The Layers and Pipes-and-Filters patterns both have their advantages and disadvantages. They both make sure components are reusable, but in the Layers pattern, only complete layers can be reused instead of individual components. Both patterns make sure dependencies are kept local, but when introducing larger changes to a layered system, there is a chance that changes cascade to other layers or components. Sharing information between layers or components is usually very expensive (or at least inflexible). A Relaxed Layered System partly solves this problem for the Layers pattern, but sacrifices maintainability.

The Globule monitor uses a hybrid pattern. The monitor itself is separated into four layers, and the layer concerned with reading and processing data is internally structured according to the Pipes and Filters pattern. Both the communication with the user interface and the integration with Globule and Apache are placed in a separate layer. The layered approach is illustrated in figure 4.3, and is fully discussed in the next section.

The Pipes-and-Filters pattern is a natural choice for the data processing layer. Data are read from a sensor, then they are processed, perhaps multiple times, and finally the result is converted to a format suitable for output to the client. However, since the data processor is the only part of the system that works on a directed data flow, the rest of the system is structured as a Relaxed Layered System. It is not possible to use a stricter pattern since in some cases the communication layer needs direct access to services from the pipeline manager.

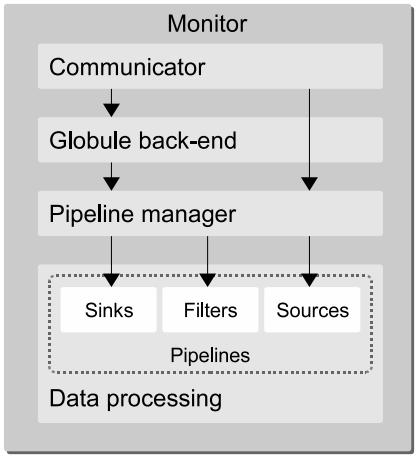


Figure 4.3: Basic architecture of the monitor.

4.3 Monitor Design

Figure 4.3 shows the basic architecture of the monitor and its connection with the *event manager*. Arrows indicate whether a component or layer calls another component or layer. Communication with clients and other servers is the task of the *communicator*, while the *Globule back-end* deals with communication with the Globule CDN. The *pipeline manager* manages the monitoring components that exist in the *data processing* layer, creating and destroying pipelines as required. This section will explain this architecture in detail.

4.3.1 Initialization

As explained in the previous chapter, the main monitor object is called the SensorMonitor. When it is instantiated, it first initializes the four layers it consists of. Preparing for initialization, the monitor first creates the pipeline manager. This component is very important, since it keeps track of all components that live in the data processing layer; all other layers depend on it.

As its name implies, the pipeline manager has the task of creating, destroying and maintaining pipelines. It stores a reference to each of the components of a pipeline (which we call *sensor components*) for bookkeeping.

Next the Globule back-end is initialized. This layer ties the monitor to Globule. It consists of an object that can communicate with Globule, and a few wrapper objects that encapsulate important Globule objects, such as resource pools and documents. This encapsulation allows the other parts of the monitor to access these objects without depending on Globule itself.

After the back-end, the communications layer is initialized. It is responsible for handling external communication. It consists of a communicator object that uses XML-RPC for external communication. Using XML-RPC allows any client application to call remote functions using requests formatted in XML, sent using HTTP [26]. The communicator also takes care of communication with sensors on different servers, again using XML-RPC.

The data processing layer is not explicitly initialized when Globule is started. The layer consists of the sensor components discussed before: sources, sinks, filters and pipes. Since components in this layer are created on the fly, the layer is initially empty, but will be populated whenever a new sensor (and its corresponding pipeline) in instantiated.

When all layers are initialized, the SensorMonitor acts an event relay. The components in the layers need to be able to receive certain events that originate from components in other parts

of Globule, like the HeartBeat event. By default, the Globule event system only accepts resource pools and documents as destinations for the events. Since the SensorMonitor is conceptually a document itself, events cannot reach the objects within the layers defined in the monitor[1]. This problem is solved by providing an event relay method in the SensorMonitor object. Every event that reaches the monitor is processed by this relay method. It decides to which layers the event needs to be forwarded, if it needs to be forwarded at all. When an event is forwarded, it is received by an object that is responsible for handling events in its layer. This object in turn forwards the event to other interested objects. For example, when the SensorMonitor receives a HeartBeat event, it forwards it to every layer. The pipeline manager is also responsible for the data processing layer. When it receives that event, it finds every sensor component that wants to receive the HeartBeat and forwards them the HeartBeat event.

4.3.2 Integration with Globule

The Globule back-end is the only component in the monitor that knows about Globule. Whenever Globule instantiates a resource pool object, the pool sends an event to the Globule back-end to register itself. Similarly, a document sends a registration event when it is instantiated by a resource pool. When the back-end receives such an event, the first thing it does is creating an appropriate wrapper for the document or pool. The monitor uses wrappers in order to ensure that the rest of the monitor does not depend on Globule. Next, it stores a reference to these wrappers in order to provide the following two methods:

- `getPools`
- `getDocuments`

`GetPools` returns a list of pools registered with the back-end, while `getDocuments` returns the list of all documents in a pool.

4.3.3 Managing Pipelines

Every time a sensor is created, its corresponding pipeline (if available) needs to be created as well. This is done by the pipeline manager. Sensors are normally created as soon as one of the wrappers mentioned in the previous section are instantiated. For example, a request-rate sensor is created by a document wrapper. Upon creation, it sends a registration event to the pipeline manager, which then stores a reference to the sensor. Next, the pipeline manager tries to find a pipeline configuration for this type of sensor. If found, it tries to create a pipeline according to the configuration and checks it for validity. Only when the pipeline is found to be valid, the pipeline manager stores the pipeline, ready to be used in the monitor.

Apart from processing sensor registration and unregistration events, the PipelineManager also provides the following methods:

- `generateId`
- `findComponent`
- `findSources`
- `findSinks`
- `getComponent`

[1]This issue has been resolved in more recent versions of Globule.

- `getSource`
- `getFilter`
- `getSink`

`GenerateId` is a static method that generates a unique ID. This is mainly used when a sensor component is created. The `findComponent`, `findSources` and `findSinks` methods are for finding components. `FindComponent` is a helper function for finding components that have a connection with another server. `FindSources` and `findSinks` both accept a number of parameters that allows the caller to filter the returned results. `GetComponent` allows the caller to get any component. `GetSource`, `getFilter` and `getSink` do the same, but additionally check the type of the component.

4.3.4 Data Processing

There are three major types of components in the data processing layer: sources, sinks and filters. Examples of these components are respectively `DiskSpaceSensor`, `XmlSink` and `HistoryFilter`. All components in the data processing layer derive from the same base class: `SensorComponent`. A sensor component has the following methods, which can be overridden if necessary:

- `id`
- `componentType`
- `enabled`
- `poolId`
- `documentId`
- `properties`, `isActive`, `performsRead`, `acceptsRead`, `performsWrite` and `acceptsWrite`
- `wantsHeartBeat`
- `tryPipeline`
- `toXml`

The `id` method returns the ID of the component. `ComponentType` returns the type, indicating if the component is a source, a sink or a filter. To temporarily enable or disable a component, the `enabled` function can be used. The function accessing the component's properties indicate whether the component actively reads or writes data or not, which is used to check the pipeline for validity. This will be discussed in details in section 4.5. `WantsHeartBeat` returns true if the component needs to receive HeartBeat events. The pipeline manager forwards the HeartBeat event to every component that has this property set to `true`. `tryPipeline` tells the PipelineManager whether it should try to create a pipeline for this component or not. `ToXml` is currently only used in source and sink objects, and is called when a formatted list of those components is created for inclusion in a response to a client.

Depending on the type of the component, a component extends either the `PipeInput` or `PipeOutput` class. `PipeInput` provides `read` and `peek` methods, while `PipeOutput` provides a `write` method. If the component can be used as an input for other components, it should extend the former, as is the case with a source component. If the component accepts input from other component, a sink for example, it should extend the latter. Note that it is also possible

to extend both classes at the same time; a filter is both an output component (it receives data from a previous component) and an input component (it outputs data to components next in the pipeline).

Depending on the type of component, more methods are made public. A filter does not have any extra methods at all, but sources and sinks implement a `type` method indicating the source or sink type. Every sensor component class has a different type, for example an `XmlSink` has type `XML_SINK` and a `DiskSpaceSensor` has type `DISK_SPACE_SENSOR`.

Additionally, a sink also implements the following methods:

- `contentType`
- `description`
- `kind`
- `output`
- `outputRaw`

`ContentType` returns the MIME content type of the output the sink will generate. For example, an `XmlSink` returns "text/xml" and an `HtmlSink` returns "text/html". `Description` returns a description of the component and the data it outputs, mainly for the user. To determine what kind of data the sink outputs, the `kind` function can be used. It returns a value like "filename", "graph-data" or "cost," which is received from the last component in the pipeline before the sink. `Output` and `outputRaw` both generate output from the current data. `Output` returns a string containing the value formatted according to the content type, while `outputRaw` directly returns the 'raw' value. These raw values are always wrapped in a PipeData object. Such an object provides the following methods:

- `description`
- `kind`
- `type`
- `value`
- `modified`
- `setModified`

`Type` returns the data type; currently number, string and vector types are supported. A vector can contain components of any of those data types. `Value` returns a union holding a variable of this data type, which are currently implemented as the Globule versions of respectively a `float`, `string` and `vector<PipeData *>`.

`Modified` returns a timestamp containing the date and time this value was last updated. Sensor components can explicitly update this timestamp by calling the `setModified` method.

Sensor components can be connected by pipes. A pipe is a simple object that forwards a `read` or `write` call from a component on one side to the component on the other side of the pipe. The reason for using a separate object to connect components instead of direct function calls is that in some cases, more advanced forms of communication are needed. For example, a component on another server cannot be accessed by simple function calls; in that case a pipe built on XML-RPC can be used. The next section will explain how this is done in the Globule monitor.

Function	*Description*	*Handler*
`pool.list`	List all resource pools	GB
`document.list`	List all documents, by pool	GB
`source.list`	List all sources, by type, pool or document	PM
`source.read`	Directly from a source	PM
`source.register`	Register remote component	PM
`source.unregister`	Remove remote component registration	PM
`source.toggle`	Enable or disable a source	PM
`source.setEnabled`	Enable or disable a source	PM
`sink.list`	List all sinks, by type, pool or document	PM
`sink.read`	Read from a sink	PM
`filter.read`	Directly read from a filter, used for remote components	PM
`filter.write`	Directly write to a filter, used for remote components	PM
`input`	Set user input	C

Table 4.1: Functions exported by the monitor. For handlers, 'GB' means GlobuleBackend, 'PM' means PipelineManager and 'C' means Communicator.

4.3.5 External Communication

The communication layer receives an event whenever a request to the monitor arrives in Globule. The *handler* hook had to be modified for detection of these kind of requests; the XML-RPC server listens for requests send to `/globulemon/`.

The communicator accepts two kinds of requests, namely 'internal' and 'external' requests. Internal requests are requests from the monitor itself, when there is communication with a sensor on another host.

External requests are requests from the user interface or from third-party applications, namely requests for listing components (sources and sinks), for reading the value of those components and for user input. Table 4.1 lists all functions exported by the communicator. A client can call those functions using XML-RPC. These calls are translated to normal function calls. All requests concerning sensor components are forwarded to the pipeline manager; the other requests are handled by the communicator and the Globule back-end.

4.4 Flexibility

An important goal of the monitor is to be flexible, but in many applications flexibility also leads to increased complexity. The Globule monitor tries to keep things as simple as possible for the user when it comes to adding or modifying functionality. Currently, the user needs to have some knowledge of the C++ programming language in order to be able to write new components, but the user can always use existing components.

4.4.1 Managing Pipelines

The user can define pipelines using a configuration file. This file is read when the monitor is instantiated. Currently, the format of the configuration file is fairly basic, and uses the actual class names to identify the components.

Figure 4.4 displays an example of a simple pipeline configuration. It creates the pipeline shown in figure 4.5. Each line starts with a keyword, followed by a number of parameters. A number of keywords are recognized, namely `BEGIN`, `END`, `FIND`, `CONNECT`, `CONNECT_MULTI` and

```
BEGIN DOCUMENT_REQUEST_RATE_SENSOR source
        SUMMATION_FILTER      filter1
        CONNECT_MULTI         source filter1

        SIMPLE_HISTORY_FILTER filter2 60 1440
        CONNECT               filter1 filter2

        DEFAULT_SINK          sink
        CONNECT               filter2 sink
END
```

Figure 4.4: Example pipeline configuration.

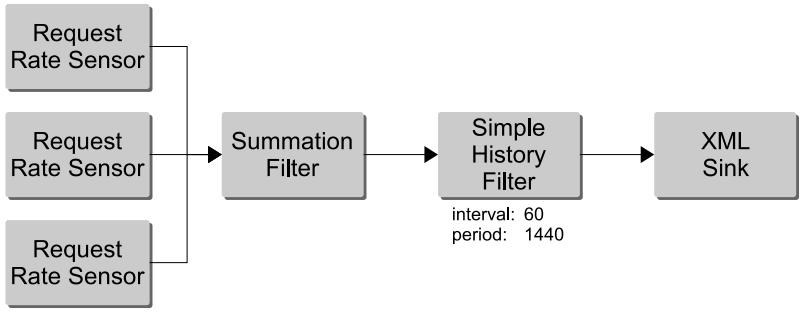


Figure 4.5: The pipeline resulting from the example configuration after three documents have been registered.

```
class BasicComponent: public PipeFilter
{
public:
        BasicComponent(const SensorComponentId componentId,
                EventMgr *eventManager);
        ~BasicComponent();
        PipeData *read(const char *kind = NULL);
        PipeData *peek(const char *kind = NULL);
        bool write(PipeData *data);
};
```

Figure 4.6: An interface for a simple component.

`CONNECT_REMOTE`. Component definitions start with the name of the component, followed by an identifier.

`BEGIN` indicates the start of a new pipeline definition; the rest of the line component definition, defining the kind of component this pipeline should be matched with. `FIND` searches for existing components of the specified type. It allows one source to be connected to multiple filters. `CONNECT` actually connects two components, the first being the input component, the second the output component. Similarly, `CONNECT_REMOTE` connects a remote component to a local one, creating a pipeline spanning multiple hosts.

Finally, the figure illustrates the `CONNECT_MULTI` keyword. This is also a variant of the `CONNECT` keyword, but it is used to indicate that this step can be done multiple times. Effectively, this means that, in this example, multiple sources can connect to the same SummationFilter. Currently, the `CONNECT_MULTI` keyword can only occur once in a pipeline definition.

This configuration file format currently provides a subset of all features available in the monitoring framework, but can be extended easily to provide more features.

To enable or disable pipelines, the pipeline manager has two methods: `createPipelines` and `destroyPipelines`. Individual components can be toggled as well; each component has an `enabled` method for this. It is up to the component to determine how to react to this.

4.4.2 Developing New Components

If the user wants to add functionality to the monitor, he needs to create an object representing the source, filter or sink he wants to create. To do this, the user needs to create a subclass of one of the following predefined classes in the appropriate directory (*sensor/sources/*, *sensor/filters/* or *sensor/sinks/*, respectively): `PipeSource`, `PipeFilter` or `PipeSink`.

Figure 4.6 shows the basic template of a sensor component (a filter). Sources only need to implement the `read` method, sinks the `write` method, and filters need to implement both.

After the user has created the component, the monitor needs to be made aware of it, so the component needs to be added to a few files. First, when the user has created a source or a sink, he needs to register a new type in *sensor/PipeDefs.hpp*. Next, the pipeline manager has to be able to create new components of this type, so the header of the new component needs to be included in *sensor/PipelineManager.cc*. Finally, the file needs to be compiled, so the component has to be added to *Makefile.am* before the user can 'make' the monitor with the new component included.

After the monitor has been recompiled, the user can modify the configuration file to use the newly created component in the pipelines.

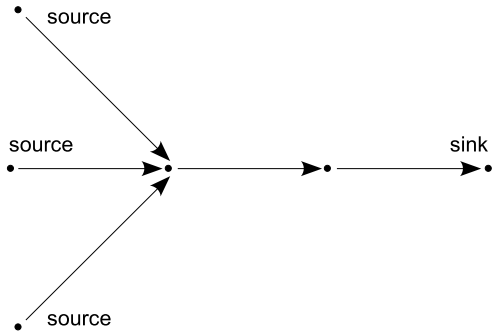


Figure 4.7: A simple weakly connected directed graph with three sources, two filters and one sink.

4.4.3 Modifying Existing Functionality

Since Globule is developed as an open source project, the source code to existing components is available for everyone to look at and modify as well. Currently, if an existing component is modified, the user needs to recompile and the monitor and restart Apache for the changes to take effect. However, the way the components are designed lends itself extremely well for dynamic loading in the form of plug-ins, which would be not too difficult to implement, and should be implemented in the future.

4.5 Validating Pipelines

Flexibility as introduced in the previous chapter allows the user to make mistakes easily when configuring the monitor. For example, pipelines may contain loops, may not produce any output or may even be incomplete. It is therefore important to validate the pipelines the user creates. This section discusses the algorithms used to perform this validation, but first we determine the actual criteria for forming a valid pipeline.

It is important to note that a valid pipeline is equivalent to a directed graph, see for example figure 4.7 for a graph version of figure 4.5. Pipes are represented as edges, with the data flow indicating their directions. Sensor components are the vertices in the graph, with source components directly mapping to the graph's sources (vertices having no entering edges), and sink components to sinks (vertices having no exiting edges). Filters correspond to the rest of the vertices. We can use this notion of a directed graph to establish some criteria for a valid pipeline.

4.5.1 Criteria

Although it is impossible to automatically certify that a given pipeline matches the intention of an administrator, we can devise a number of criteria that will necessarily avoid non-functional configurations. For example, a pipeline should not contain any cycles in the data flow. Since a pipeline can be seen as a directed graph, this statement can be reformulated as follows: the directed graph should not contain any directed cycles. Next, there should be an undirected path between any pair of vertices, ensuring that there are no disconnected vertices in the system. That is, each directed graph should be weakly connected. Finally, the graph should contain at least one source and one sink, but this property directly derives from the first criterion; having no source or sink implies having a cycle.

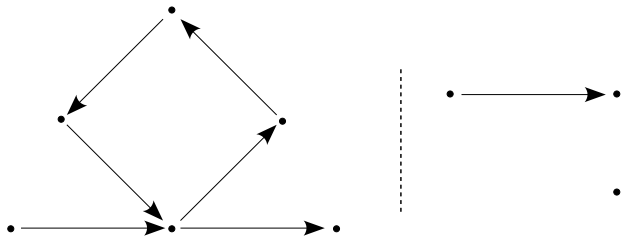


Figure 4.8: Two examples of invalid pipelines. The first contains a cycle, while the second is not weakly connected.

Apart from these graph-specific criteria, three more can help establish the notion of a valid pipeline. First, each pipeline should contain at least one active component that either pulls data from components up the pipeline, pushes data to components down the pipeline, or does both. Second, the activity of these components should 'reach' the entire pipeline. For example, if a filter only pulls data from an input on request, but there is no filter or sink that actively pulls the data from that filter, the pipeline is invalid. Finally, sources in the graph should all be source components as well. Similarly, sinks in the graph should be sink components in the system. Figure 4.8 illustrates these criteria.

To summarize, each graph, thus each pipeline, should have the following properties: a valid graph is acyclic, weakly connected and contains at least one source and at least one sink. Each pipeline should also contain at least one active component, have component activity reach every component in pipeline, have every source in the graph correspond to a source in the pipeline and have every sink in the graph corresponds to a sink in the pipeline.

4.5.2 Validating Graphs

Figure 4.9 shows a pseudo-code version of the algorithm that is used to verify the validity of the graphs representing the pipelines in the system. Validation is done in two major steps by the `checkGraphs` function.

In the first step, graphs are recursively checked for cycles and for the existence of at least one sink per graph. In the second step, the amount of components visited during the first step are compared to the actual amount of components registered in the system. If the visited amount is less than the actual amount, it means that at least one component is not connected to a graph (and thus at least one graph is not weakly connected).

4.5.3 Pipeline-Specific Checks

To ensure the lifeness of a pipeline, additional checks need to be performed. We need to make sure a pipeline's data flow will be initiated, and that there are no points in the pipeline where the data flow stalls. These requirements can be checked while creating a pipeline, so that invalid pipelines can be rejected as soon as an error is detected.

To assist in determining if the data flow in a pipeline can be started, each component sets a flag indicating whether the component is active or passive. Activate components initiate reads or writes, while passive component only react on reads and writes from other components. Checking if a pipeline can be started is now easy: each pipeline should have at least one active component. Note that a sink that performs reads is also considered active. As far as the pipeline is concerned, the sink initiates the read, even though the action may be triggered from outside the system.

```
FUNCTION checkGraphs()
        VECTOR allVertices
        VECTOR graphVertices
        VECTOR sources

        FOREACH sources AS source
                graphVertices.empty()

                IF checkGraph(source, graphVertices) == false
                        RETURN false
                IF graphVertices.contains(TYPE sink)
                        RETURN false

                allVertices.add(graphVertices)
        END

        IF allVertices.size() != allComponents.size()
                RETURN false

        RETURN true
END

FUNCTION checkGraph(component, visited)
        IF visited.contains(component)
                RETURN false

        FOREACH component.outgoingArrows AS arrow
                IF checkGraph(arrow.follow(), visited) == false
                        RETURN false
        END

        visited.add(component)

        RETURN true
END
```

Figure 4.9: Algorithm used to verify the validity of all graphs.

Additional flags indicating whether a component accepts reads, performs reads, accepts writes or performs writes (or a combination thereof) are used to determine is the data flow will stall. If a component *accepts* reads, then the next component in the pipeline needs to *perform* reads, and vice versa. Similarly, if a component *performs* writes, the next component needs to *accept* writes, and vice versa. If a component both accepts reads and performs writes, then only one of those two properties needs to be matched on the other side of the pipe.

Chapter 5

What-If Analysis

5.1 Introduction

Monitoring systems like Globule allows one to better understand the behavior of such complex distributed systems. However, it would be even more interesting if the monitor would allow the administrator to make good decisions about the way a site is hosted by this system.

In essence, as shown in figure 5.1, we can see a web hosting system as a single black box having two basic inputs: a system configuration and the traffic addressed to the system. The output of the black box is a measure of its performance, determined by the monitor.

The task of an administrator is to find the configuration that will give the best performance for a given incoming traffic. However, there is no simple way to determine a good configuration from a performance target and a traffic access pattern.

The first alternative consists of making (educated) guesses about the required changes to the configuration, and applying these changes to the live system. If the performance is not as good as desired, adjust the configuration again, and repeat. The advantage of this approach is its simplicity. However, it may take a lot of time to reach the desired goal, especially if the characteristics of traffic change over time. If the guess about configuration is wrong, the performance may get even worse after making the changes. Finally, tests are done on a live system so that it may take some time before enough traffic has been received to draw conclusions.

The second option consists of building a separate testbed. The idea is to make an exact copy of the existing system, and apply all configuration changes to this copy. This allows for controlled experimentation without disturbing the live system. Disadvantages of this approach are that the larger the system gets, the more expensive maintaining the copy gets, and that again it can take a while to complete the experiments.

The third alternative is to use a simulation to analyze changes in the system's configuration. In this approach, we replace the black box with a simulator, and the incoming traffic with logs of incoming traffic. The simulator loads these traces and replays them, simulating the behavior

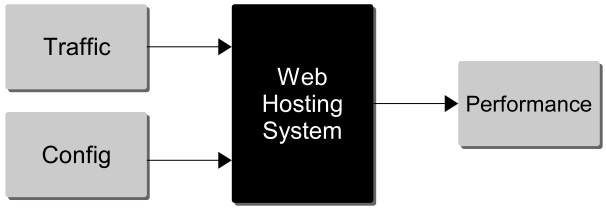


Figure 5.1: A web hosting system as a black box.

of the actual system as closely as possible. This allows us to run simulations with different configurations and compare the results to determine the best configuration for the current situation. It even allows us to change the characteristics of traffic easily and study their effect on system performance. Doing this is much harder in the other approaches. Using a simulator has several benefits over the other approaches, since it does not need a complete copy of the original system, and it does not interfere with the live system at all. On top of that, a simulator can execute much faster than the other alternatives, since it can replay traffic much faster than real time.

To implement this kind of what-if analysis, we need a simulator capable of reproducing the behavior of different system configurations under a given incoming traffic. In particular, it is crucial to accurately reproduce the effects of system load. We discuss the simulator and the modifications that had to be made on it in section 5.2. We then need to input the right configurations so that conclusions can be drawn. This often requires preprocessing the access traces before simulation, and is described in section 5.3. We then present the way simulation is integrated with the rest of Globule in 5.4. Finally, we show usage examples in 5.4.2 and discuss our experience drawn from this work in 5.5.

5.2 The Simulation

5.2.1 Requirements

The requirements for the simulator can be divided into two kinds: general requirements and requirements with regard to what-if analysis.

The first general requirement is that we need to replay existing access traces, to determine the effect of a change in configuration or in traffic access patterns. It needs to replay them as fast as possible, but it should also be accurate. Preferably, the result of a simulation run would be a single value indicating the cost of handling the requests in the trace. This cost value can then be used to compare different simulation runs.

Next, to provide an accurate simulation, the simulator needs to take all important steps of handling a request into account. A large document typically takes longer to process than a small one. Depending on the origin of the request and the requested document, the time spent inside Globule (processing the request) and on the network (sending the request and retrieving the response) will vary.

Document updates also need to be simulated. Apart from internal caching issues, they also have an effect on Globule; depending on the replication policy, the document may need to be updated on other replicas, resulting in even more requests. Of course, all replication policies available in Globule need to be simulated as well.

One important application of what-if analysis is testing server and resource provisioning. Globule is designed to be able to run on any server that is able to run Apache, so the server performance can vary to a very large extent. A cheap family PC is probably not as fast in processing a request as an expensive corporate webserver, so the simulator needs to be able to simulate different kinds of hardware. On top of that, server load impacts performance as well, so the simulator needs to be able to adjust performance based on the server load. We also want to be able to simulate different numbers of servers used in the system, and to apply different redirection strategies to the incoming requests.

Another important part of what-if analysis is simulating different amounts of traffic. For this, we want to be able to adjust the amount of incoming traffic.

5.2.2 The Globule Simulator

Globule already contains a simulator, which does much of what is described in the previous section, yet for a different purpose: it is currently used to determine the best replication policy for each document in the system [21]. This is done by taking a recent part of the trace and replaying it in a series of simulations. Each simulation run uses a different document replication policy, and uses a cost function to keep track of the best policy for each document. After the runs are complete, Globule instructs each document to use the policy that suits it best. Research has shown that recent activity is a good indication of activity in the near future, so the best policy for recent traces is likely to work well in the near future as well.

Even though this simulator does not entirely cover the full list of requirements, we decided to base our what-if analysis on it. We however need to improve it to incorporate the few missing features.

Most importantly, the simulator does currently not take server performance into account. Instead, each server is assumed to treat requests instantly, meaning that every server has infinity capacity. In reality a server's capacity depends on many factors such as CPU speed, amount of memory installed and hard disk speed. Simulating a server's internal performance is in particular essential to accurately reproduce server overload behavior, but also to determine the time it takes to process a request. This is currently a function of the size of the file.

Finally, a simulated server can currently handle any number of requests simultaneously, while in practice the number of concurrent requests a server can handle is limited. Usually, if the amount of incoming requests exceeds the amount the server can handle, the requests will either be queued or cause the server to return an error. Reproducing this behavior is essential to allow administrators to avoid this unpleasant situation.

For this, we need to improve the simulator as follows. First, we need to make server performance configurable. Next, the server needs configurable server limitations. Finally, we need to modify the simulator to make it simulate request serving times.

Another limitation of the Globule simulator is the simulation of network performance. Currently, available bandwidth is assumed to be unlimited, and each request is allocated exactly 64 KB/s. This is of course unrealistic, as in reality concurrent connections would need to consume the available bandwidth. We decided not to improve this aspect of the simulator. This clearly limits the ability of our what-if implementation to study the effects of bandwidth contention. However, should one want to incorporate this feature, all that would be needed is an improved simulator. The remaining part of the what-if implementation would stay untouched.

To simulate internal server cost, we first have to get an idea about how a server reacts to different amounts of traffic; this can be done by benchmarking several different servers and comparing the results. Using this results, we can devise the characteristics of a server. These characteristics can then be used to modify the simulator to calculate request serving times.

5.2.3 Benchmarks

The best way to characterize a server's reaction to different amounts of traffic, is to benchmark a number of different servers. I have used HammerHead to perform these benchmarks [27]. HammerHead is mostly used to stress-test webservers; it uses a user-defined number of threads that will each send requests to the web server, and wait for a response. This effectively means that this tool allows the user to specify the concurrency degree. In contrast, other tools like ab and httperf only allow the user to specify quantities in terms of request rate.

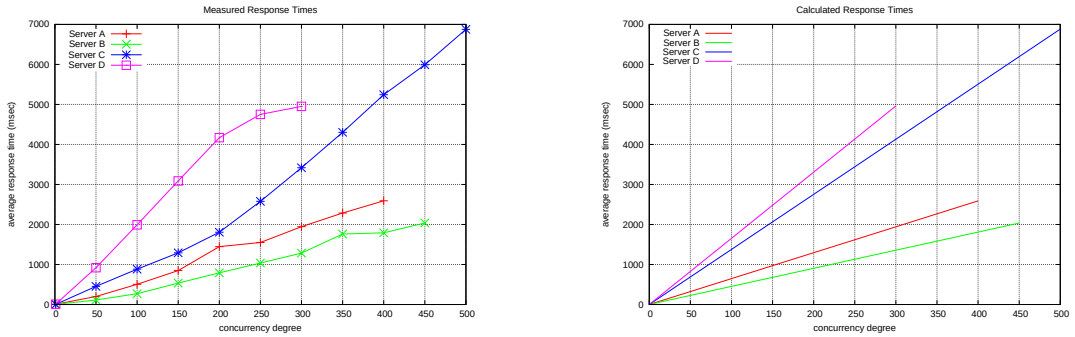
Figure 5.2 shows the output of a typical HammerHead run. The values we used for the benchmark is the average response time, which characterizes the latency that happens inside the webserver, regardless of what happens on the network. HammerHead behavior is specified in

```
Config File                   : server-b.conf
Machine                       : 192.168.1.117:8081
Time                          : Fri Sep 30 23:14:14 2005
SEED =                        : 1128114653
Parent Process PID =          : 7841
Total Run Time (sec)          : 180
Sessions                      : 300
Session Sleep Time (msec)     : 0
Startup Lag Time (seconds)    : 1
Failures                      : 0
NoVerify                      : 0
Total Requests Served         : 35929
Total Turnaround Time (msec) : 54355337
Average Request Time  (msec) : 1512
Total Responses               : 35929
Average Responses / sec       : 178.751251
Total Response Time (msec)    : 46173646
Average Response Time (msec) : 1285
Scenario Throughput           : 35929
Sequence Throughput           : 36229
Run Time (sec)                : 201
Scenarios / sec               : 178.751251
Sequences / sec               : 180.243774
Content Length bytes          : 590169754
Content Length bytes / sec    : 2936167
Read Length bytes             : 590169754
Read Length bytes / sec       : 2936167
```

Figure 5.2: Output of a typical HammerHead run.

Server	*CPU*	*Speed*	*Memory*	*Hard disk*
A	Intel Pentium III	550 MHz	192 MB	8 GB
B	Intel Pentium III	666 MHz	256 MB	20 GB
C	AMD Duron	800 MHz	512 MB	20 GB
D	Intel Pentium II	300 MHz	64 MB	4 GB

Table 5.1: Specifications of the benchmarked servers.



(a) Measured using HammerHead.

(b) Calculated using the server's characteristic constant.

Figure 5.3: Response times of the servers.

scenario files, allowing for good control of the clients. For this particular benchmark, our scenario consisted of simply requesting the same 16 KB file repeatedly. This has the effect that the server can keep the file in cache, not having to load it from hard disk. This way, we can measure the best-case performance of the web server. We expect that good server implementations will come close to this best case performance, even in more demanding scenarios.

Of course, the scenario described above is not very realistic, but properly benchmarking a web hosting system is not trivial. We opted for a simple benchmark, since the simulator can be easily adapted to new benchmark results in the future.

The benchmark setup is as follows. We benchmarked four machines with different hardware, all running the same version of Debian GNU/Linux. All non-essential system services were disabled. Version 1.1.3 of Globule was installed on each server, by compiling both Apache (2.0.49) and Globule from source. All servers were connected with a 100Mbit network, except one, since it only had a 10Mbit NIC. The specifications of the servers are show in table 5.1.

Benchmarks were performed using 50 to 500 simultaneous requests, in steps of 50. Benchmarks were stopped as soon as the server started generating errors; this point was noted as the overload point of the server. Figure 5.3(a) shows the average response time for every step in the concurrency degree. Each line stops at its respective server's overload point.

As can be seen in the graph, the response time almost follows the concurrency degree linearly; each line can thus be approximated by a simple linear equation $r = C * d$, where r is the response time, C is the slope of the graph and d the concurrency degree. Since C is the only part of the equation that differs on each server, we can use it as the server's *characteristic constant*, indicating the server's performance. The characteristic constants of our four test servers, together with their overload point (the maximum number of simultaneous requests they can handle), are summarized in table 5.2.

If we plot the graphs again, but now using the equation and each server's characteristic constant derived from the benchmark data, we get figure 5.3(b). As one can see, this matches the graph of figure 5.3(a) very closely.

Server	*Characteristic*	*Overload*
A	6,47	400
B	4,52	450
C	13,77	500
D	16,54	300

Table 5.2: Characteristic constant and overload point for each benchmarked server.

Figure 5.4: Steps necessary to properly reprocess a trace.

5.2.4 Simulator Improvements

To improve the simulator so it can also be used for performing what-if analysis, a few modifications need to be made. Currently, network latency and bandwidth are partially simulated, server performance is assumed to be infinite and there a no limitations on the amount of requests that can be served simultaneously.

In order to accurately reproduce request serving times, it is necessary to factor in the speed of the server, but also the server's limits. As seen above, we can derive a server's internal performance from its concurrency degree. This degree, in turn, can easily be calculated in the simulator by increasing the request count at the start of a request, and decreasing the count when the request is completed.

Given a measure of a server's current concurrency degree, the server's characteristic constant allows us to determine the time each request would have taken under the current load. We therefore updated the simulator to measure the concurrency degree of each server in the system. It then becomes easy to add a delay to each request which is the product of the server's characteristic constant and the current concurrency degree.

To simulate severe server overload, a large penalty is added to the cost of serving a request that exceeds the server's overload limit.

5.3 Trace Preprocessing

Apart from being able to simply replay existing traces in the simulator, it is also necessary that the user can analyze scenarios different from the current situation. One could think of increasing the number of requests sent to the site, using only a specific part of the trace, using the same trace on a different number of replicas and varying the client redirection policy.

It is possible to build these functionalities into the simulator itself, but this would unnecessarily increase the complexity of the simulator. However, we observe that the previously mentioned studies can be completed by preprocessing the existing request traces and keeping the simulator untouched. It is possible to reproduce the effect of different redirection policies by preprocessing the traces as well.

Using the pipeline system described in chapter 4, we can decompose what-if analysis into more simple components. As illustrated in figure 5.4, we can identify the following stages if we want to reprocess a trace for performing what-if analysis and simulating redirection policies:

First, the trace needs to be loaded from disk. This can either be the current live trace, but also a previously prepared trace, containing a specific scenario. A trace contains different kinds

of entries, but we are currently mostly interested in the lines containing requests. Such a line consists of the following fields:

- entry type (request, update, etc.)
- timestamp
- server and port where the request is issued
- request method (optional)
- client that requested the file
- time elapsed while processing the request
- path to the requested file

After loading a trace, we need to select the most interesting part of a trace. This is especially useful when using a live trace; in most cases it is only interesting to use recent entries in the trace, because recent activity is usually a good indication of future activity.

There are several options for simulating different traffic intensities, like generating traces containing the desired amount of traffic, or adding traffic to an existing trace. However, generating realistic traffic patterns is very difficult, so we decided to use a more simple solution: adjust trace speed. If we increase the speed of which a trace is replayed, the traffic seems more intense, retaining its characteristic statistical properties. If we decrease the replay speed, the traffic decreases as well. Of course, this could be implemented in the simulator, but it is easier to do this while processing the trace: just divide the timestamps by the desired speed-up factor.

Simulating different client redirection policies and different numbers of replica servers is also relatively simple when reprocessing a trace. The trace records the server to which each request was issued, so depending on the selected redirection policy, we can replace this server with the one selected by the policy.

Of course, it would also be possible to prepare a set of traces in advance, and feed them into the simulator one at a time. If we somehow store the trace, it would be possible to use the same trace for other kinds of output than the simulator. Examples of other outputs are graphs showing the development of the concurrency level per replica, or the total request rate.

In addition, for the system to be actually usable, the user needs to be able to specify the parameters for each step in the process. For example, the component that adjusts the speed of the trace needs a speedup factor, and the component that select a part of the trace needs a time frame.

5.4 Monitor Integration

To integrate our what-if analysis into Globule, we need to extend the Globule simulator as discussed in section 5.2, create the monitor components implementing the steps discussed in the previous section, create the monitor components for input and output, and finally configure a pipeline that ties all components together.

User input is done using a single component that accepts XML-RPC requests containing (key, value)-pairs. The keys are name of parameters defined by the components.

To make the results of the simulation visible to the user, the system has to be able to output the results. Therefore, components need to be created that can output simulation results, request rate, concurrency degree and the final trace, after processing.

Name	*Description*	*User input*
WITraceLoader	Load trace	Trace name
WITimeFramer	Select desired part of trace	Time frame
WISpeedAdjuster	Adjust trace speed	Speed
WIClientRedirector	Redirect clients	Number of replicas, policy
WISimulator	Simulator wrapper	Performance, threshold
WITraceExporter	Resulting trace	
WIConcurrencyDegree	Concurrency degree	Sampling interval
WIRequestRate	Request rate	Sampling interval

Table 5.3: Components needed for what-if analysis

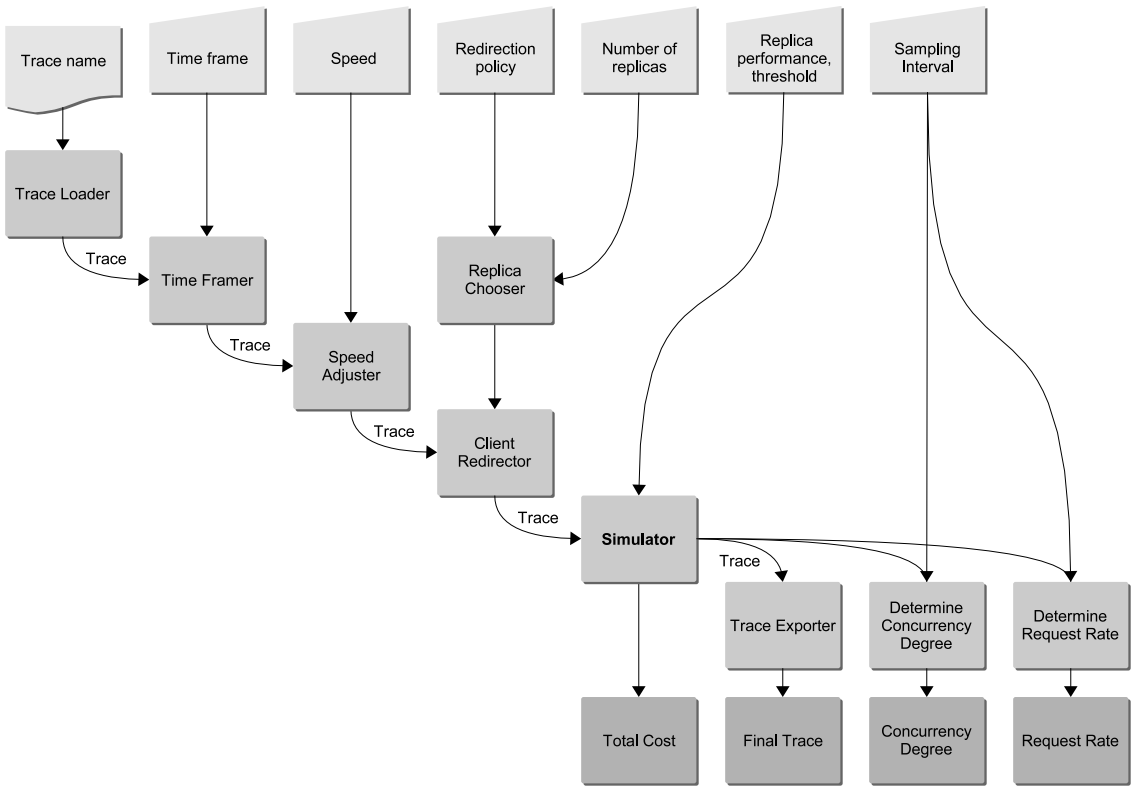


Figure 5.5: The resulting pipeline used for implementing what-if analysis.

More output components can be easily added to the system if desired. Table 5.3 summarizes all components that need to be created. Using these components, we can build a pipeline that reads the input from the user, retrieves the desired trace, processes it, feeds it into the simulator and outputs the desired result. The resulting pipeline is shown in figure 5.5.

5.4.1 User Input

To make it easier for the user to use the analyzer, I have created a simple web interface that assists the user in creating scenarios that can be simulated. The user interface also transforms the results from the analyzer into simple graphs, showing the effect of all changes the user makes.

First, the user selects the server and site that will be analyzed. The user is then presented with a screen, shown in figure 5.6, that allows the user to specify the desired scenario. After clicking on analyze, the web interface sends all details to the Globule server. After the analyzer is finished, the user sees the result of the analysis in the form of a set of graphs, as shown in figure 5.7.

5.4.2 Examples

One the next pages follow a number of example graphs, based on a part of an actual trace from the VU webserver. The trace contains a record of the first moments of a drastic increase in traffic. The examples were generated using the what-if analysis components described in the previous sections, although the resolution of the images has been increased. The overload point is set at 200 concurrent requests, and the server's characteristic constant is initially set at 20, representing a fairly slow server.

Figure 5.8(a) shows a graph of the incoming request rate, and figure 5.8(b) shows a graph of the corresponding concurrency degree. As we can see, the server starts to overload very quickly. The partial trace only spans about one and a half minutes, but it takes the server two minutes to process all requests.

However, if we speed up the server by setting it's characteristic constant to 10, the results improve greatly. It now only takes one and a half minutes to process the requests, as shown in figure 5.8(c). Finally, figure 5.8(d) shows the result of simulating an even faster server by setting the constant to 2.5.

5.5 Discussion

5.5.1 What-If Analysis

We believe that what-if analysis is a great tool that provides an administrator of a website with information to help make decisions with regard to the performance of the website. Traditional monitoring tools only allow the administrator to analyze existing behavior of a site; what-if analysis takes this to a further level allowing the administrator to simulate all kinds of scenarios that would often be too expensive to perform for real.

What is still missing from the implementation described in this section is a way to determine where to place a new replica, should the administrator decide to add one. This requires knowledge about distances in terms of network bandwidth and latency. Once this information has been acquired it should not be too difficult to add a component to the pipeline that implements this. For example, when one wants to implement this feature based on minimal client-to-replica latency, one could easily add a component implementing the HotZones algorithm to the what-if analyzer [22].

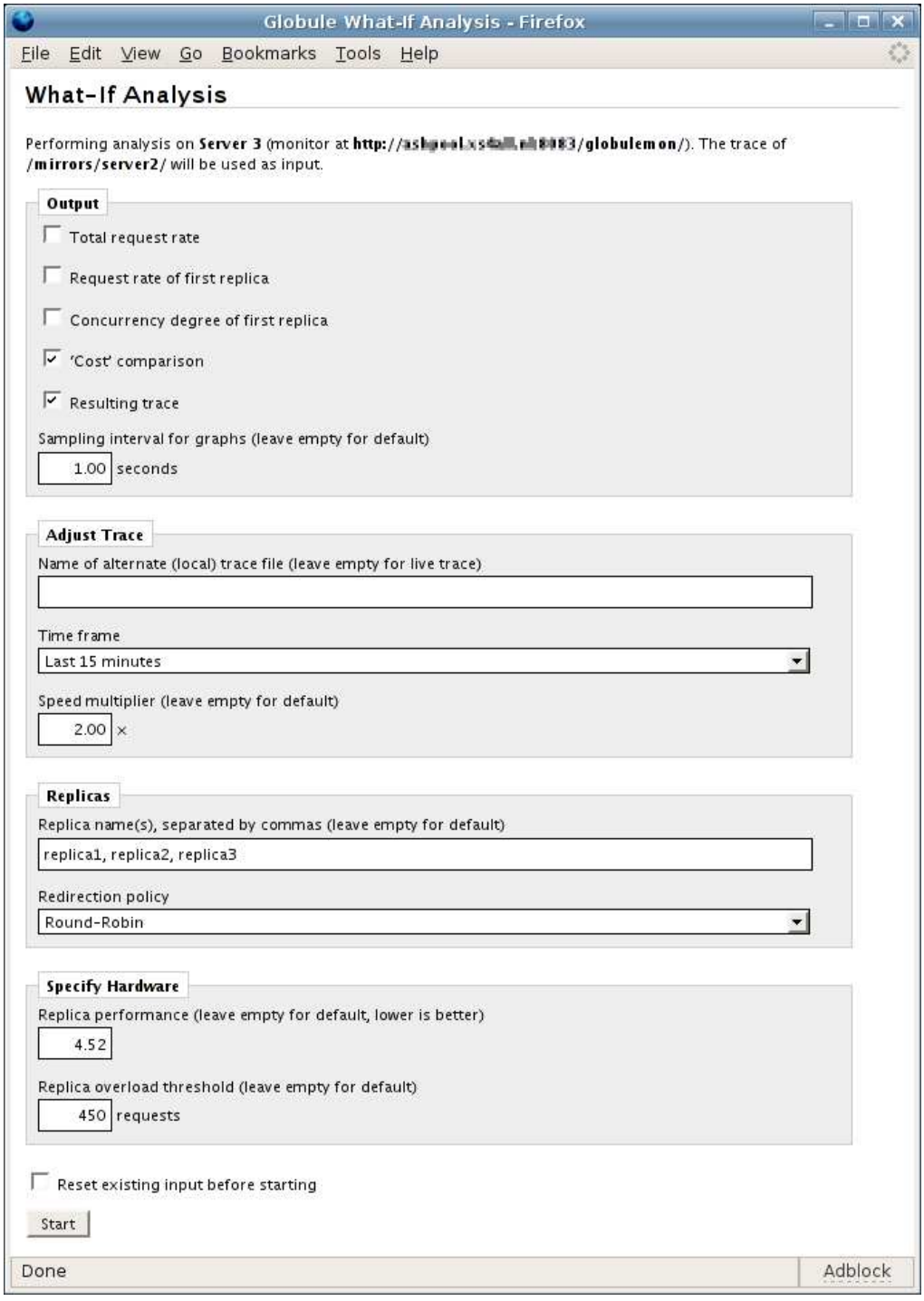


Figure 5.6: Getting user input using the web interface.

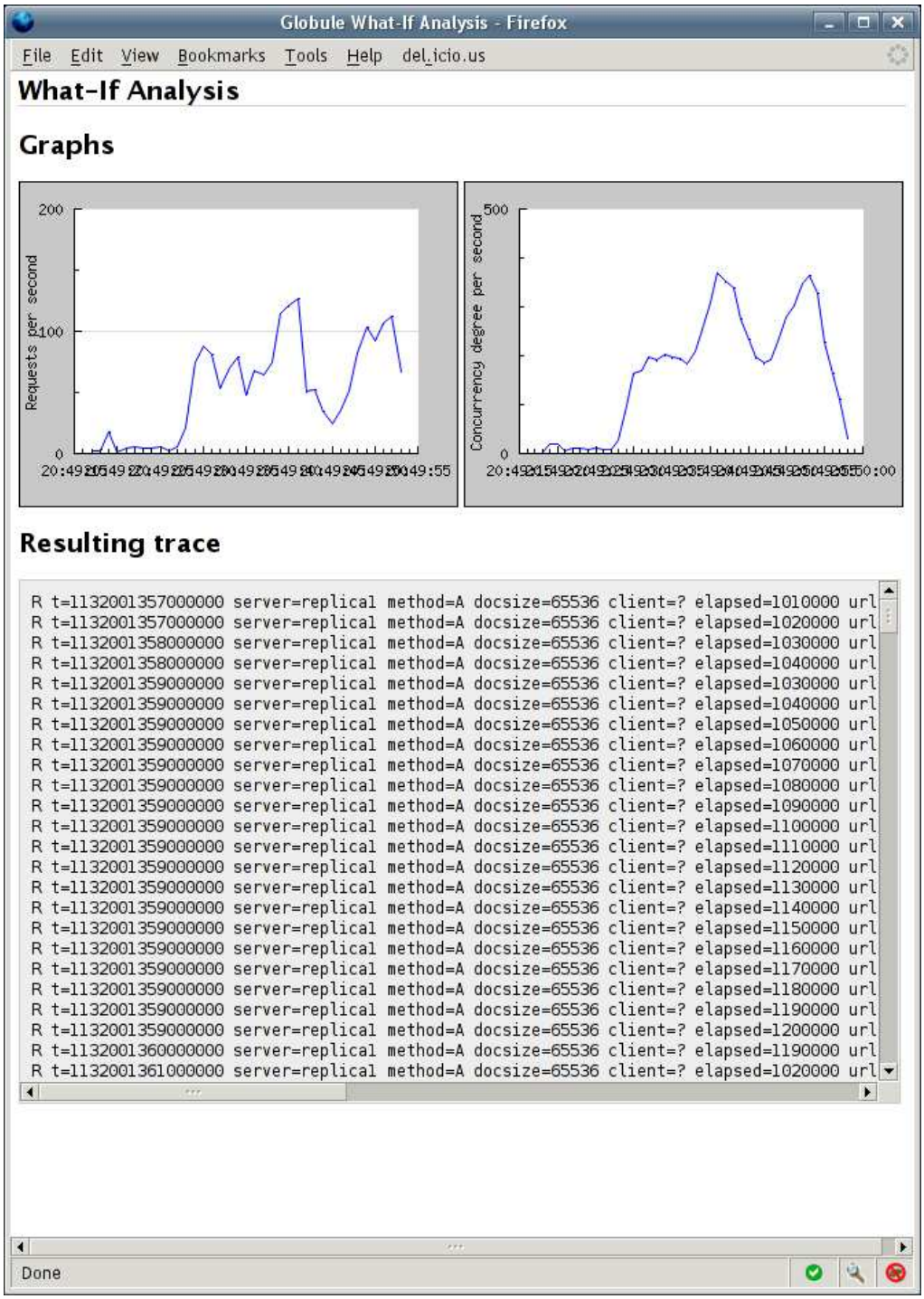
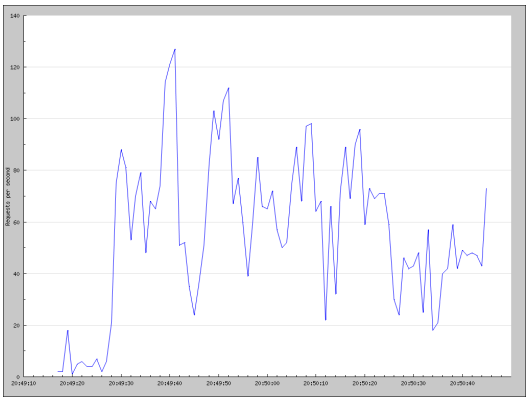
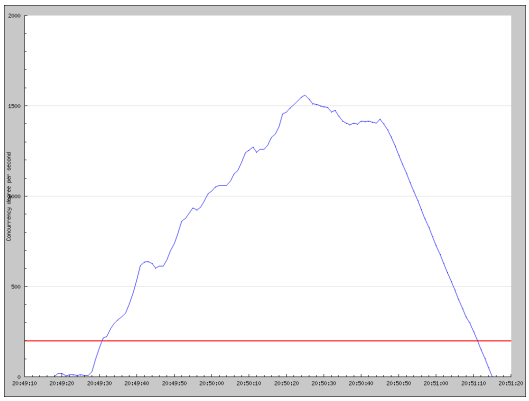


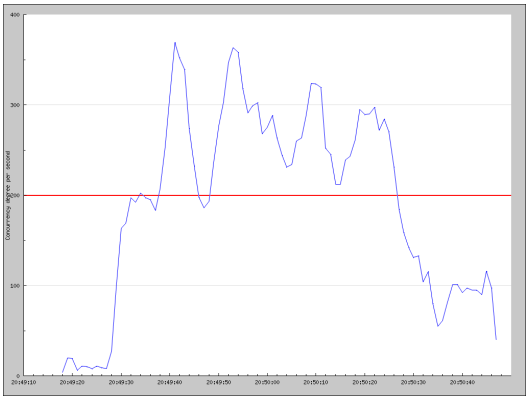
Figure 5.7: What-if analysis results, showing the request rate, concurrency degree and the resulting trace.



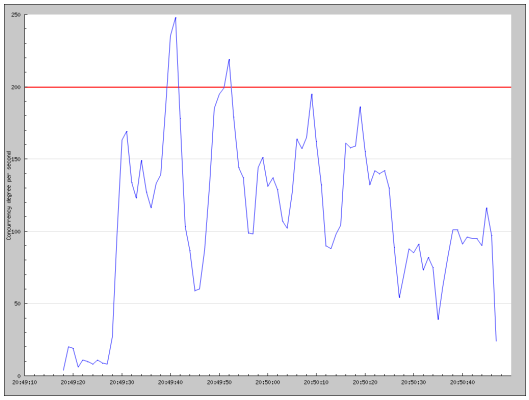
(a) Request rate.



(b) Concurrency degree, characteristic constant is set to 20.



(c) Concurrency degree, characteristic constant is set to 10.



(d) Concurrency degree, characteristic constant is set to 2.5.

Figure 5.8: Server performance improvements. The overload point is set at 200 and is shown in the graphs.

5.5.2 Monitor Integration

Building the what-if analysis also allowed me to validate the flexibility and performance of the pipeline framework described in Chapter 4, as this new pipeline is significantly more complex than all the other ones built so far.

We observe that the implementation of a complex system such as the what-if analyzer is greatly simplified by splitting up the analyzer into smaller components. The pipeline framework helps streamlining the flow of the data in the analyzer to the simulator, and allows us to generate multiple forms of output for the same set of data.

Chapter 6

Conclusion

This thesis has introduced a flexible framework for building complex monitoring systems. The structure based on pipes and filters allows one to create a monitor from smaller components. Using simple components to build a pipeline allows for great flexibility, as many generic components can be reused among multiple pipelines. The framework is specially targeted towards the Globule CDN, but can be integrated into other systems without much effort.

I have created a number of components for general system monitoring, as well as components that are specific to monitoring Globule. Since components are connected using pipe objects instead of a direct connection, adding support for pipelines spanning multiple servers was easily accomplished by replacing the pipe. No modifications to the components themselves were necessary.

The flexibility of the framework, however, comes at a cost. Since the user can connect components like Lego bricks, it is easy to incorrectly connect components and create invalid pipelines. To help solve this problem, this thesis introduces a simple way to validate pipelines. By mapping the components in a pipeline to the vertices in a directed graph, the validity of the pipeline can be checked by making sure that no cycles exist in the graph, and that the directed graph is weakly connected. Additional checks make sure that pipelines are unable to stall, and data are thus always transported from the sources to the sinks.

This monitoring framework provides a solid base which can be used for more than just simple monitoring. Using the framework and by making some slight modification to the existing Globule simulator, I have designed a pipeline that performs a completely different form of analysis, called what-if analysis. What-if allows an administrator to find the optimal configuration of a website given a certain traffic. By using a simulator and by varying the simulator input and configuration, an administrator can predict the behavior of a system under hypothetical circumstances, which allows him or her to make decisions about the system. To our knowledge, this form of analysis has no equivalent in any other academic CDN.

During the writing of this thesis, a monitoring framework similar to our framework has been added to Globule. However, this framework is not as flexible as the one introduced in this thesis and does not allow for what-if analysis. We therefore believe that the current implementation can greatly benefit from the work in this thesis.

Bibliography

[1] Barrett, B. L. Webalizer. Software. http://www.mrunix.net/webalizer/.

[2] Bearden, M., and Bianchini, R. *Efficient and fault-tolerant distributed host monitoring using system-level diagnosis*. Chapman & Hall, 1996, pp. 159–172.

[3] Behlendorf, B., Fielding, R. T., Hartill, R., Robinson, D., Skolnick, C., Terbush, R., Thau, R. S., and Wilson, A. Apache. Software, The Apache Software Foundation. http://httpd.apache.org/.

[4] Bent, L., and Voelker, G. M. Whole page performance. In *Proceedings of the 7th International Web Content Caching and Distribution Workshop* (August 2002).

[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture*, vol. 1. Wiley, 1996, ch. 2.2 From Mud to Structure.

[6] Case, J., Mundy, R., Partain, D., and Steward, B. Introduction and applicability statements for internet standard management framework. RFC 3410, December 2002.

[7] ClickTracks. ClickTracks Pro. Software. http://www.clicktracks.com/.

[8] Destailleur, L. Awstats. Software. http://awstats.sourceforge.net/.

[9] Dragoi, O. A., and Preston, J. E. The concrete architecture of the apache web server. Department of Computer Science, University of Waterloo, February 1999.

[10] Godard, S. Sysstat. Software. http://perso.wanadoo.fr/sebastien.godard/.

[11] Gunter, D., Tierney, B., Crowley, B., Holding, M., and Lee, J. NetLogger: A toolkit for distributed system performance analysis. Tech. Rep. 51276, Lawrence Berkeley National Laboratory, December 2002.

[12] Koletsou, M., and Voelker, G. M. The Medusa proxy. In *Proceedings of the 6th annual Web Caching Workshop* (June 2001).

[13] Liston, R., and Zegura, E. Using a proxy to measure user-perceived web performance. In *Proceedings of the 6th Web Caching and Content Distribution Workshop* (Boston, MA, 1999).

[14] Lonvick, C. The BSD syslog protocol. RFC 3164, August 2001.

[15] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing 30*, 7 (July 2004).

[16] McCanne, S., Leres, C., and Jacobson, V. Tcpdump. Software, The Tcpdump Group. http://www.tcpdump.org/.

[17] Pai, V. S., Wang, L., Park, K., Pang, R., and Peterson, L. The dark side of the web: An open proxy's view. In *Proceedings of the Second Workshop on Hot Topics in Networking (HotNets-II)* (Cambridge, MA, November 2003).

[18] Patarin, S., and Makpangou, M. Pandora: A flexible network monitoring platform. In *Proceedings of the USENIX 2000 Annual Technical Conference* (San Diego, June 2000).

[19] Peng, G. CDN: Content Distribution Network, January 2003.

[20] Pierre, G., and van Steen, M. Design and implementation of a user-centered content delivery network. In *Proceedings of the Third IEEE Workshop on Internet Applications* (June 2003).

[21] Pierre, G., van Steen, M., and Tanenbaum, A. S. Dynamically selecting optimal distribution strategies for Web documents. *IEEE Transactions on Computers 51*, 6 (June 2002), 637–651.

[22] Szymaniak, M., Pierre, G., and van Steen, M. Latency-driven replica placement. In *Proceedings of the International Symposium on Applications and the Internet (SAINT)* (Trento, Italy, Feb. 2005), pp. 399–405.

[23] Vermeulen, A., Beged-Dov, G., and Thompson, P. The pipeline design pattern. In *OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, October 1995.

[24] Vladishev, A. ZABBIX. Software. `http://www.zabbix.com/`.

[25] WebSideStory. HitBox® Professional. Software. `http://www.websidestory.com/`.

[26] Winer, D. XML-RPC specification. Web Page, 1999. `http://www.xmlrpc.com/spec/`.

[27] Wong, G. HammerHead. Software. `http://hammerhead.sourceforge.net/`.