# Gossips – System and Service Monitor

*Victor Götsch, Albert Wuersch, and Tobias Oetiker* – Swiss Federal Institute of Technology

## ABSTRACT

Gossips is a modular client/server based system monitor. It uses distributed monitoring tasks to define and report states of an IT-environment. A monitoring task includes probes to measure data and a test to evaluate them. Gossips does not only report problems, it can also suggest solutions to the problems by consulting a knowledge-base, which is maintained and easily extended by the system managers using the local system. The monitor software is easily extensible through a flexible plug-in system for tests and probes. The monitor software is written in object oriented Perl which allows new tasks to inherit large parts of the existing infrastructure of the program.

### Introduction

The problem of monitoring a group of networked hosts has been discussed at length only recently by John Sellens [1]. Many protocols and tools for monitoring are available, including SNMP [2], Big Brother [3], Swatch [4], Spong [5] and pikt [6]. These have different strengths and weaknesses. Our goal in this project was to address some of the problems we found with existing solutions, focusing on a clean architecture and easy extensibility. After an evaluation of the mentioned tools we defined the following criteria for a new design:

- The monitor relies on a scalable client-server architecture where the client only talks to the server when it finds a problem and periodically assures the server that everything is okay.
- The software design is flexible and expandable.
- Only free tools are used (e.g., GNU GPL).
- The monitoring system allows to archive solutions to known problems.

### State of the Art

#### Evaluation Criteria

- **Configuration**: The tool configuration should present the system manager with a good overview of the system and services monitored. The configuration is the instrument of a system manager using the tool. The system manager should be able to change configurations quickly without editing many files.
- **Design and Complexity**: The design of the tool should be as simple as possible, but not too simple. This concerns not just the code, but also the documentation and configuration of the tool.
- **Scalability**: The tool should work fine with five as well as with 5000 machines.
- **Extensibility**: The tool should offer an interface for adding new monitoring tasks without modification to the code of the tool.
- **Modularity**: This is in fact a specific aspect of extensibility. When a new extension is added to the tool, it should be possible to reuse this new extension, like in a lego system.
- **Messaging**: The tool should report exceptions. It should not primarily display a webpage with red and green 'lights.' Such a webpage gives the system manager kind of a secure feeling when he sees all shining in green. But he always has to look at the pages and is distracted from his work.

#### Comparison

A first evaluation in summer 2000 showed deficiencies in most tools mainly in the areas of extensibility and modularity. The table in Figure 1 shows an updated summary of the evaluation based on the latest versions (September 2001) of the most promising monitor tools after our first evaluation.

*Big Brother*

- **Configuration**: Big Brother needs a separate configuration file for each local test. Global tests, such as network tests are all configured in one additional file. In these configuration files you are able to define global configuration for

| Monitor | configuration | design | scalable | extensible | modular | messaging |
|---|---|---|---|---|---|---|
| **Big Brother** | ○ | + | + | ○ | - | + |
| **Swatch** | + | + | + | ○ | - | + |
| **Spong** | ○ | + | + | ○ | ○ | + |
| **pikt** | + | + | + | + | - | + |
| **gossips** | + | + | + | + | + | + |

**Figure 1**: Comparison: '+' good; '○' okay; '-' missing.

all client and local configuration for special clients. It is rather painful to make changes in this system of configuration files.

- **Extensibility**: There is an interface where you can add your own scripts. The script has to translate the state of your system to *green*, *yellow* and *red* states. This makes it difficult to write a test which looks for keywords in logfiles. Additionally it is rather complicated to configure tests from the configuration file.
- **Modularity**: Since the new tests are hard to configure from the configuration file, it is difficult to write reusable Big Brother tests.

*Swatch*

- **Configuration**: It takes just a few minutes to learn how to setup and use the monitoring system. Each system manager has his customized swatch configuration file, that contains pattern/action pairs that are personally interesting, or that pertain to his system responsibility.
- **Extensibility**: Swatch is a monitoring tool to observe syslogs. It is possible to implement tests that write their monitored data to the syslog, but Swatch does not support a test developer with any tools to write a new test.

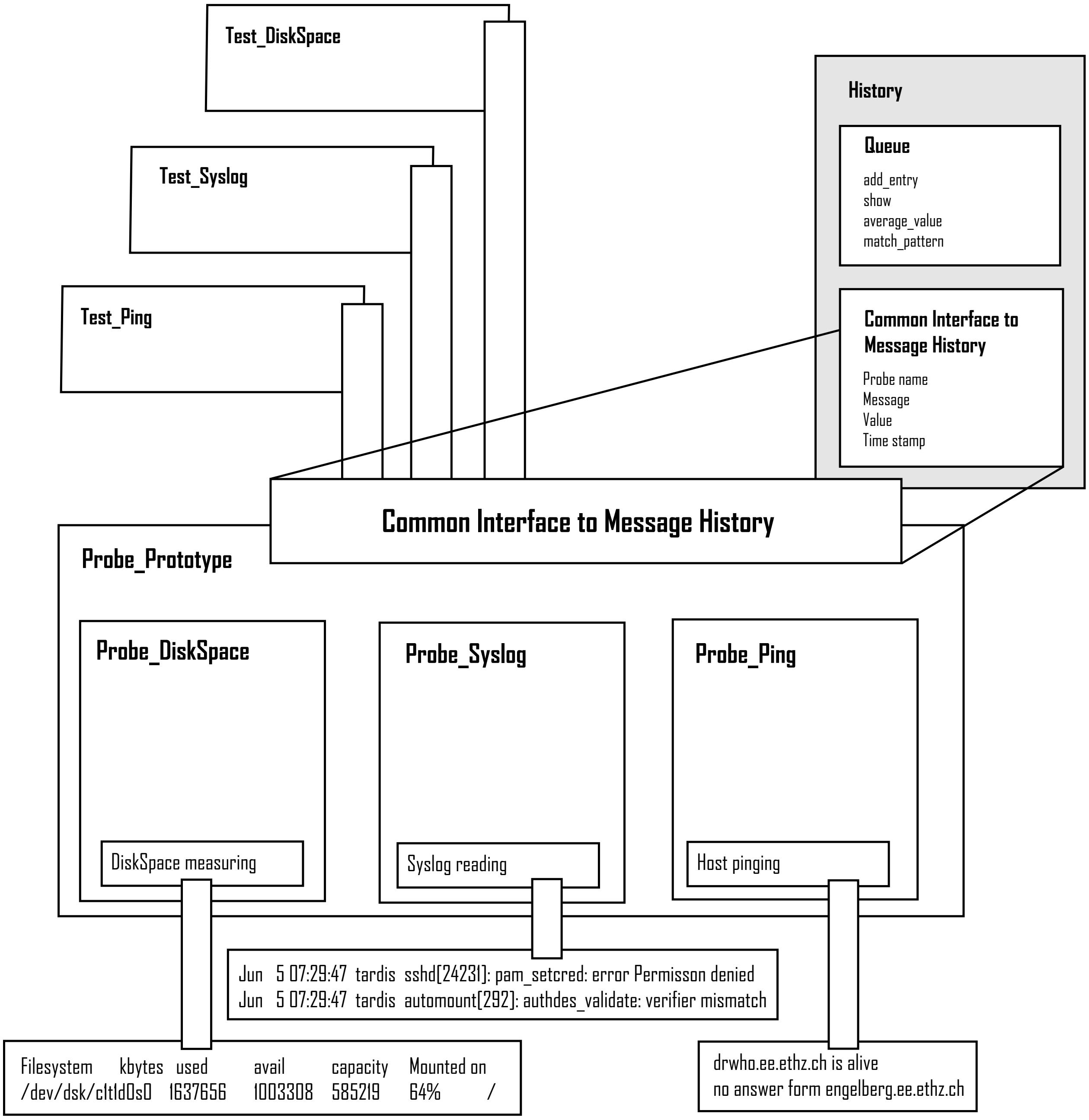


**Figure 2**: Objects.

- **Modularity**: Not everybody has access to the vendor's source code for system utilities that produce syslog entries. This makes it really difficult to enhance or reuse these utilities of Swatch.

*Spong*

- **Configuration**: Spong uses one global configuration file which sets internal values. If a specific client needs different settings you have to 'override' the default configuration with a new file. A monitoring system should be managed centrally even if it is a distributed system. The concept of grouping hosts for network tests should have been applied to local tests, too.
- **Extensibility**: You can write new tests by implementing new plugins. The measured state of the test has be mapped to status colors *red*, *yellow* or *green*.
- **Modularity**: Spong does not offer support reusing of implemented tests or parts of it. The plugin-system does not define an interface for intercommunication of different tests. A separation of data measuring and data analyzing would be a step to go towards modularity.

### Results

In the process of evaluating the other products we found lots of fascinating concepts and ideas. But no tool had a really flexible framework for writing new tests. The framework we envisioned would handle all the basic functionality of a monitoring system like *execution time*, *message handling* and *internal communication*. Such a toolkit helps to implement new monitoring tasks much faster as the developer can focus on the functionality of the new monitoring task. We have not found a system which separates data acquisition and data analysis, allowing the implementation of reusable monitoring tasks.

In the end no tool fulfilled our criteria to a degree which encouraged us to add the missing bits to an existing package, so we decided to implement the tool ourselves.

## Gossips Design

### Architecture

Gossips is a object oriented framework written in Perl. The software is designed as a distributed client/server architecture where all clients report to a central server. Gossips is configured through a central configuration file and controlled via a command-line interface. A message handling system on the server notifies the system manager about system-state-changes. This concept is similar to the messaging of cfengine [7]. Cfengine writes a message when it changes something on a system and gossips notifies the system manager if and only if a state-change occurs in the system. Thus there is no need for a graphical display of the system status, as most of the time nothing changes. For long-time monitoring of system status, a tool as for example RRDtool [8] can be used within the gossips frame work. (See Figure 9 for an example.)

### Probes and Tests – Separating Measurement from Analysis

Each participating client runs a gossips monitoring process. Each gossips process consists of a set of *probe* objects to acquire data about the state of the local system or anything else you want to observe. Data from these probes is then analyzed by a set of *test* objects. Each test can subscribe to any number of probes. This separation of data collection and data analysis was an important step toward simplifying the design, implementation and reuse of new monitoring components for the gossips system (see Figure 2).

Gossips uses a scheduler similar to the one implemented in pikt. The scheduler manages the execution of all tests and probes within a gossips instance. It executes the probes periodically. When a probe finds new data it adds all tests which have subscribed to its data to the scheduler. When a test is executed, it accesses the data acquired by the probe together with a history of old data. The test evaluates the data and decides about the state of its target.

### States – Describing Systems or Services

*Simple Monitoring Tasks*

The generation of states relies on the data gathered by the probes. States describe the condition of a system or a service. It is up to the developer of a test to decide what states best describe a certain system. Simple things like *working/broken* are possible but also more complex approaches with many different states of operation. For example, if *free disk space* is monitored, a system manager needs to know when a certain *threshold* is reached. Additionally, it would be helpful to predict if a disk will fill up in the next hour. Because the test does not only see current data from a probe but also data collected earlier, it can make much more in-depth decisions as if it had only access to the latest measurements. This feature makes is possible to do trend analysis of measured values. In the *free disk space*-test just mentioned, all the data that was accumulated is used to calculate an approximate time when the disk fills up. The *free disk space*-test can then use the following states to decide the condition of a disk, all the values in this example are thresholds:

- Everything okay with disk /scratch
- less than 200 M on disk /scratch
- less than 30 minutes until disk /scratch full
- less than 200 M on disk /scratch and less than 30 minutes left until disk full

*Combining Monitoring Tasks*

By assigning several probes to a test, a next level of defining states is reached. An ftp-test, for example, could just monitor an ftp connection to a host. It could use simple states like *working* or *broken*. The client monitor might also test the 'pingability' of a host. When the observed host crashes or is rebooted gossips would then come up with two messages, one noticing the broken ftp connection and the other that the host is not alive. This is redundant information. The important information at this time is that the host is not alive.

Therefore an ftp-test should be implemented that checks the ftp connection with an ftp-probe and simultaneously evaluates the 'pingability' of a host using a ping-probe. The test is then able to access information on status messages of these two probes and use states like:

- Everything okay with ftp connection to tardis
- no ftp connection to tardis
- no answer from tardis

A test that subscribes to several numbers of probes allows very comprehensive state assessments. As each instance of gossips is able to decide about the state of the system it monitors, it will only talk to the central server if something interesting happens (a state-change). Because normal operation is much more common than problems, this approach helps to keep communication between clients and server down to a minimal level.

## Configuration

### *Central Configuration*

One of the main design goals of the project was to keep the configuration files in one central location. Therefor gossips uses a central file for test parameters. Systems like Big Brother or Spong with their local config files for each client are much more cumbersome to change. If the parameters of a test must be edited for each client the system manager has to do lots of editing. With the complexity reducing group-design of gossips the system manager only has to edit some lines in the test.cfg file.

### *Distribution of the Configuration*

All instances of gossips get their configuration from a central configuration. When a gossips process on a client is started, it contacts the server and asks it for its configuration. The server can also push new configurations out to the clients as each client connects to the server in a regular interval to assure the server that it is still alive.

### *host.cfg*

Every host in the IT-environment is subscribed to groups. These groups describe hardware, network and organizational setup of a host. This design is similar to the class concept of cfengine. The difference between the two designs is that gossips uses a separate file to define a host-group relation whereas cfengine lets the host derive its memberships to the defined groups. This was made to be flexible enough to define abstract terms like *department names*, *computer room names*, *institutes* or even *disk size* as groups. See Figure 3 for an example of a host configuration file.

### *test.cfg*

Tests are configured by assigning parameters to groups. This allows to define a network wide configuration and also the specification of test parameters for a particular host. This is a similar approach as the configuration model of cfengine. For example every host is subscribed to a group called 'ee', meaning it is located in the department of electrical engineering. All of these hosts receive the same test parameters when the parameters are assigned to the 'ee' group. In Figure 4 part of a test configuration file is shown. Each test configuration section starts with its name encapsulated by three asterisks (***). Lines starting with '+' build a subsection to attach parameters to groups.

```
*** HOSTS ***
server     server,ignore
tardis     ee,tardis,sun,2cpu,link,ignore
engelberg  ee,isg,sun,1cpu,ignore
nova       ee,isg,sun,2cpu,ignore
jabba      ee,jabba,sun,1cpu,ignore
tardis-a4  ee,tardis-a,sun,1cpu,4gb,ignore
```

**Figure 3**: host.cfg – file.

```
*** Test_DiskS ***

+ ignore
  run = no

+ sun
  disk1 = scratch::100M::20min
  disk2 = tmp::100M::20min

+ sun&4gb
  disk = default::50M::30min

*** Test_Load ***

+ ignore
  run = no

+ 2cpu
  period = 60sec
  timeavg = 30min
  proclim = 3proc

+ 1cpu
  run = no
```

**Figure 4**: test.cfg – file.

Gossips can also handle more complex group structures in the test configuration. By chaining several groups with '&' it is possible to assign very specific parameters to selected hosts. If parameters for a group 'sun&4gb' are defined gossips would apply this configuration only for hosts belonging to both groups 'sun' and '4gb'.

### *Merging the Configuration Information*

The server process reads the configuration files and build an internal structure by merging the information. The merging algorithm searches in each test section of the config file shown in Figure 4 for a matching constellation with a host by seeking from the bottom to the top. On the top of every test section is a group called 'ignore'. It has the parameter 'run = no' which deactivates the test for a group. As you can see in Figure 3 all hosts are member of the 'ignore' group. If the merging algorithm finds for a host no other match than the 'ignore' group, the test is deactivated for the host. If no match can be found at all, meaning, a host is not a member in the 'ignore' group, gossips will tell the system manager to review his configuration files.

Based on the information available in the configuration file fragments shown in Figures 3 and 4 the host 'tardis-a4' would receive the configuration shown in Figure 5.

```
tardis-a4:
     Test_DiskS: sun&4gb
             default::50M::30min
```

**Figure 5**: Configuration of tardis-a4.

### The Knowledge Base

One of the functions of the gossips server is to provide a message handling system which notifies the system manager of state-changes found by tests running on the clients.

Because gossips reacts to state-changes and not to system conditions it will only report a broken disk once. If the disk breaks, this is a state-change, and gossips will report it. The disk will only be reported again when the state of the disk changes (e.g., miracle healing).

Depending on the nature of the state-change, the solution to the problem might not be obvious. When a problem occurs for the first time, there is no helping it, someone has to get to the bottom of the problem and find a solution. Once the solution is found, gossips allows to attach a description of the solution to the original message. Gossips stores this information in its knowledge base. When this particular state-change occurs again, gossips will not only inform the system manager about the new state, but will also tell about the solution which was found last time.

It is possible that in some cases many different causes will result in the same state-change. Lets look at a hard drive which is running out of space. When this happens for the first time, the system manager will add a description of the problem to the knowledge base. If the state-change occurs again at a later stage and the system manager finds a different cause for the problem, the knowledge base entry can easily be edited to explain the second possible cause as well. Otherwise the trigger can be adjusted to match the state-change more closely.

If the system manager notices that a certain problem occurs again and again, gossips could be used together with cfengine, which is able to do reparations or rebuild configurations.

### Message Handling System

Gossips does not maintain a fancy web page with red and green icons indicating the system health. Normally it is quiet and leaves the system manager alone. Only if a problem occurs gossips searches its knowledge base and initiates a message to the system manager about the new state of the system or service. The communication module at the moment uses email, but it can easily be extended to talk over other transports, e.g., a pager. Visual monitoring tasks can be implemented for long-time monitoring by using RRDTool as graphic library (see Figure 9).

## Implementation of Gossips

### Startup of a Process

Let's start at the beginning and see what happens when you start the monitoring system. A gossips

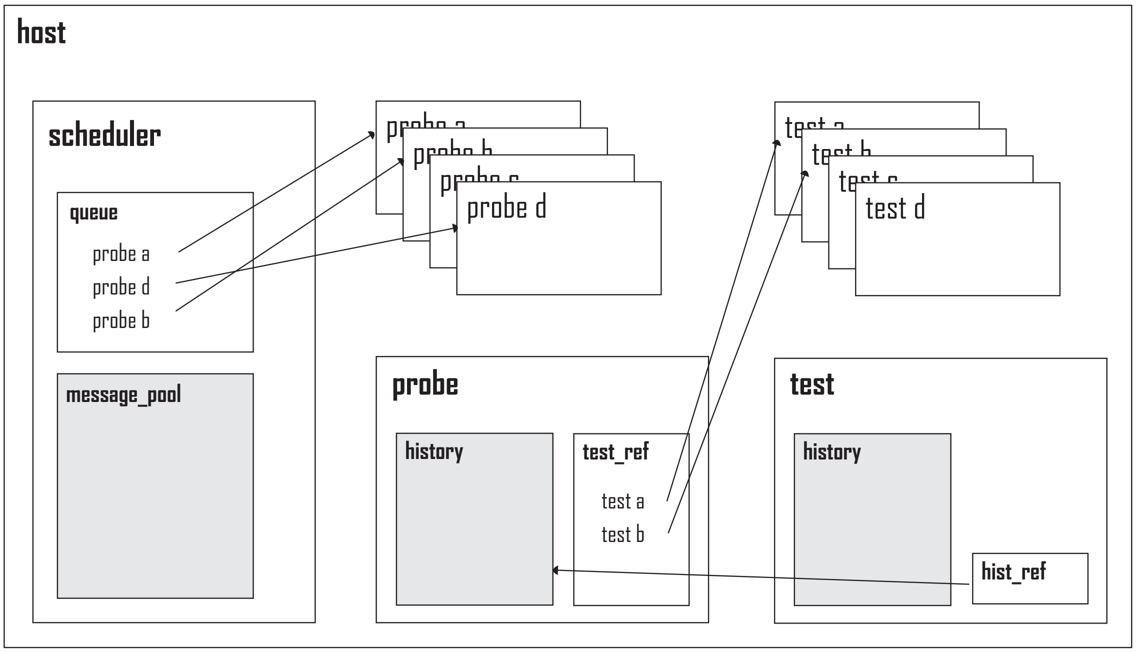


**Figure 6**: Structure of a host.

distribution contains two shell scripts which are designed to be executed as *init.d scripts*. The startup scripts *gossips-client-control* and *gossips-server-control* will each start the related process as daemons. Both scripts handle the command-line arguments *start*, *stop* and *restart*.

**Internal Organization of a Gossips Process**

*Server and Client Modules*

There is a single main gossips program. By using different startup parameters it loads either the server or the client modules. Every gossips process has the same objects, a *scheduler*, as well as several *probe-* and *test*-objects. In Figure 6 the internal structure of a host is illustrated. The next subsection will describe the function of each object.

*Objects in a Gossips Instance*

The scheduler object manages the internal operation of a gossips process. It uses a queue to control the *firetime* of probes and tests. Every probe object consists of a period. When a probe has finished its execution the scheduler puts it back into the queue and it will be re-executed after the specified time interval.

Every object in a gossips instance has a history object attached. The history object of the scheduler is called *message_pool*. To save the states of the related object the history uses a stack of constant length. In addition, the history supplies methods to evaluate its contents. For example, it provides a trend analyzing method which calculates a *gradient* of the numerical values stored in the history.

The probe objects gather the data for the monitor system. The data is stored in the attached history and accessible for the test objects through a reference. The test objects which evaluate the measured data are referenced in the probe object. At the end of its execution the probe inserts all the test objects that are subscribed to it into the scheduler queue. If a test is already scheduled it will not be added to the queue again.

*Client/Server Communication*

The gossips client/server architecture is implemented with probes and tests. The client and the server both use modules to communicate with each other. Each module uses a probe and a test object to implement its functionality. In Figure 7 the client/server architecture and the relation to the system manager is shown.

When a test on a client detects a new state, it pushes the related message into the message pool of

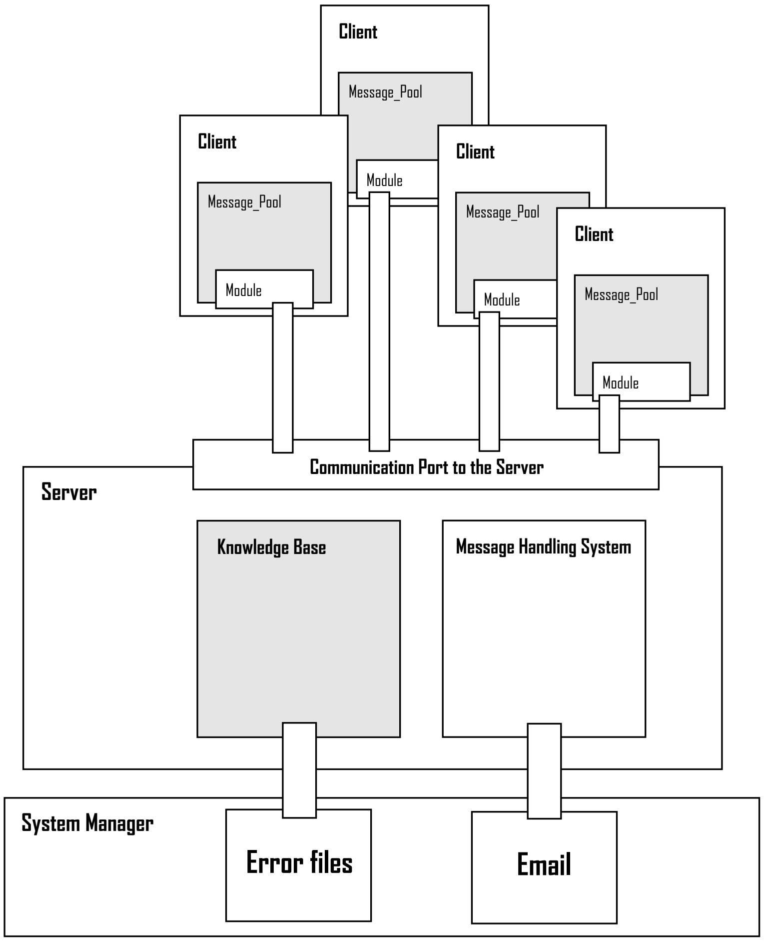


**Figure 7**: Client/Server architecture.

the scheduler. A probe monitors the message pool. If a new message is put in the message pool the probe schedules a test that connects to the server and forwards all new messages from the message pool to the server.

On the server a probe listens for client connections. The client authenticates itself using a *challenge/respond*-module. The communication socket itself is not encrypted by default, but it is possible to modify the client/server-modules to use the *IO-Socket-SSL*-perl-module which provides *SSL* functionality.

## Current Gossips Distribution

The current gossips distribution is not just a monitoring toolkit. The current release of the package consists of an installation system, the gossips base classes, several monitoring tasks, and full documentation. Figure 8 shows a listing of the currently implemented monitoring tasks.

## Extending

### Base Classes

One of the main reasons for designing gossips as an object oriented framework was to define a clear and simple interface for adding new tests and probes. Gossips comes with base classes for tests and probes including several methods. The base classes provide all the communication infrastructure required for tests and probes. They also handle the scheduler as well as a few other essential gossips services.

The first step to build a new monitoring task is to separate data collection from data evaluation. Data collection is done with the probe object that measures a device or a service. The evaluation of the collected data is done by the test object. Both objects are instances of a basic test and a probe class.

### Adding Probes

Probes often use UNIX-commands to collect data. Gossips supports the execution of external commands through a method called ‘safe_run’ which kills any started process if it does not complete within a given amount of time.

The main method of a probe object is the ‘my_script’-method. It must be overridden when inheriting from the basic probe class. The job of the scheduler is to execute the ‘my_script’-method. (See Figure 10 for an example of a method that pings hosts.)

### Adding Tests

It is a bit more complex to implement a new test class. Again the main method that is called by the scheduler is named ‘my_script.’ Additionally, a method must be added that defines a language to parse the desired parameters from the configuration file and one that links these parameters with the probes and the test. Those two methods are explained in the next section.

The new test object will determine a certain state from the data acquired by the probe. This state is the return value of the main method ‘my_script’ (see Figure 11 for an example of a method evaluating ping measurements). In this example the ‘my_script’ method uses a pattern analyzing feature of the history object. This

| | |
|---|---|
| **Measurement Classes** | |
| Probe_Logfile.pm | gathers lines of a logfile in its history. |
| Probe_DiskS.pm | measures free disk space using the UNIX command ‘df -l’. |
| Probe_Load.pm | measures the load on the local host using the UNIX command ‘uptime’. |
| Probe_Ping.pm | pinging hosts. (using ‘ping’) |
| Probe_MultiPing.pm | pinging hosts more than once. (using ‘fping’) |
| Probe_FTP.pm | checks ftp-connections to a host. |
| Probe_FileSize.pm | measures the size of a given file and returns the filename and its size in kilo bytes. |
| **Analysis Classes (basic)** | |
| Test_Logfile.pm | analyses logfiles using regular expressions. |
| Test_DiskS.pm | checks if there is enough space and enough time before a disk is filled up. It uses threshold values for available space on a disk and a time window in which the disk should not fill up. |
| Test_Load.pm | checks if the load of a local host is critical over a given period of time. |
| Test_FTP.pm | checks the ftp-connections and the pingability of a host simultaneously. |
| Test_MailWatcher.pm | checks if the size of the INBOX-file is beyond a given threshold. It sends an email to the respective user if the mailbox is too large. |
| **Analysis Classes (graphical/using RRDTool)** | |
| Test_DiskGraph.pm | builds a html-page with graphics which display free and used disk space of local disks. |
| Test_LinkUp.pm | draws a graph of the round trip times between the localhost and a given remote host (see Figure 9). |
| Test_MailGraph.pm | draws graphs about sent, received, bounced and rejected mails of your mailserver. |

**Figure 8**: Features of a gossips distribution.

method only returns the first message of the history if it was repeated at least twice in a row. This feature forces the test to verify a received probe message. The state is only returned when it was confirmed once again. This test directly uses the returned messages of the ping command as states. The ping command of a Solaris distribution returns messages like *hostname is alive*, *no answer from hostname* or *ping: unknown host hostname*. On a Linux system the 'my_script' method would be implemented differently.

```
sub my_script {
  my $self = shift;
  my $history = shift;
  my $message = $history->show_message();
  return $history->first_entries_eq(1);
}
```

**Figure 11**: Test_Ping.pm: my_script-method.

The history object provides several methods to handle the collected data of the probe. It has methods to show the content of history entries. A history entry contains the name of the owner object, a message field, a value field and a time-stamp. Value fields could, for example, store available disk space in a test monitoring a hard disk.

The history also provides methods that evaluate its value fields. One example is an *average*-method that calculates the arithmetic mean of all values in the history entries. The history provides the *gradient*-method to be able to predict trends of measured values. This method calculates a gradient using the values from the history entries along with its time-stamp.

The result of the 'my_script'-method is the identified state of the measured service. Gossips then decides if the result is a state-change. If it is, gossips puts the state message into the message pool of the scheduler object.

### Defining the Configuration

The configuration system of gossips gives the test developer the freedom to define his own 'parameter style.' Two methods are required in the test module to define the syntax of the parameter and the assignment of parameters to the test and the probes.

A 'my_syntax'-method defines the syntax of the test parameters in the configuration file seen in Figure 4. Figure 12 shows the corresponding 'my_syntax'-method of the 'Test_Load'-class.

- [1] defines the parameter key 'run'.
- [2] assigns a syntax to 'run'. The syntax is given by a regular expression (/^no$/). For the key 'run' the parser just accepts the line 'run = no'. Otherwise it throws the error message 'wrong run value'.
- [3] defines the parameter key 'period'.
- [4] assigns a syntax to 'period'. The regular expression (/^\d+sec$/) is the syntax. With this configuration the parser accepts only lines starting with a number and ending with the

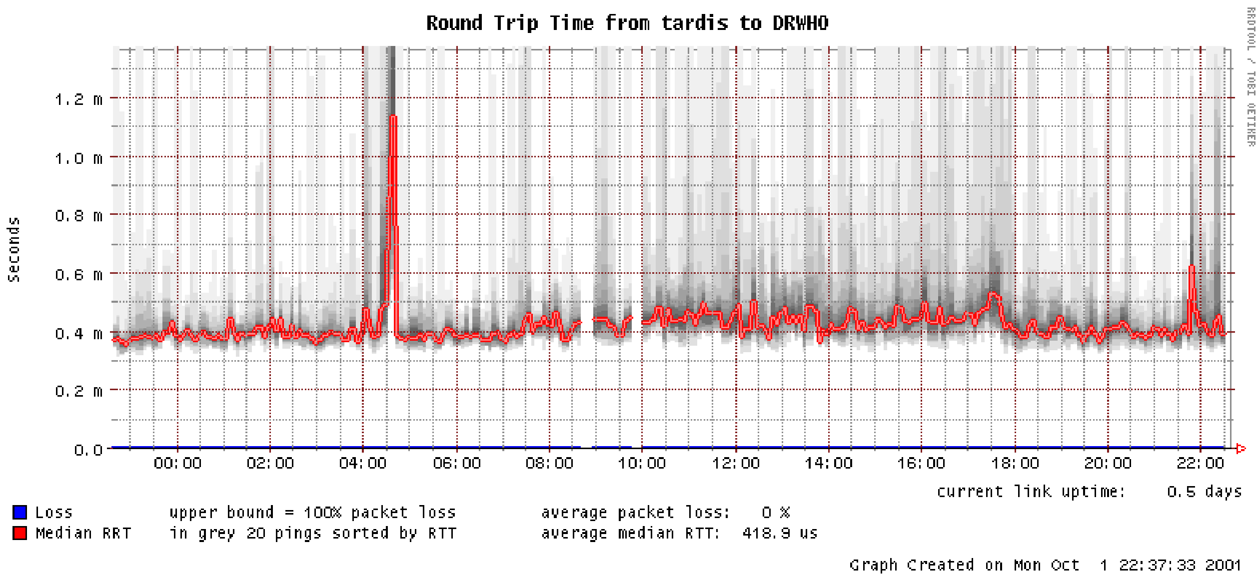


**Figure 9**: Example of a RRDTool-Graph used in gossips (Test_LinkUp).

```
sub my_script {
  my $self = shift;
  my $target = $self->argument;
  my $message = $self->safe_run("/usr/sbin/ping $target 5");
  return $message;
}
```

**Figure 10**: my_script-method of Probe_Ping.pm.

identifier 'sec'. On failure it will respond ''syntax error in 'period' parameter''.

To assign the different parameters to the test and probes the developer has to implement a 'my_struct'-method. Again, the test base class offers methods to define these relations.

In the 'my_struct'-method of the 'Test_ Load'-class seen in Figure 13 the developer first adds the keys defined in the configuration file.

- [A] defines the key 'period'.
- [B] adds a 'filter' to the first parameter of key 'period'. The filter '/^(\d+)sec/' is used to extract the information from the parameter. In this case the filter cuts off the identifier sec'.
- [C] defines the key 'timeavg'.
- [D] adds the filter '/(\d+)min/' to the first element of the key 'timeavg'. This filter lets pass just the minutes which are defined.
- [E] adds the probe 'Probe_Load' to the test.
- [F] links the first argument of key 'period' to the period of the probe 'Probe_Load'. This command sets the period of a probe.
- [G] links the first argument of key 'timeavg' to the test.
- [H] links the first argument of key 'proclim' to the test.

Figure 14 illustrates the relations between the configuration parameter, the parser, the test and the probe object.

**Defining the States**

The generation of states relies on the data gathered by the probes. For a simple test like a ping test the collected data already defines reasonable states like '*host is alive*' or '*no answer from host*.' In more difficult cases the test developer has to define his own set of states in the test class.

The main job of the 'my_script'-method in the test module is to handle the messages in the history of the probe. The message should be mapped to logical states. The definition of a sensible set of states is essential for successful monitoring. The type of information that flows from the history into the states is restricted in some points. Remember that gossips supplies state-changes. States should express if they are *good* or *bad*. By using such states gossips is able to tell the system manager if a monitored service just changed to a *bad* state. If gossips monitors, for example, a hard disk by collecting the free disk space it should use a threshold value. With such a value it can define states like '*Everything okay with disk /scratch*' when the free disk space is larger then the threshold or '*less than 100 M on disk /scratch*' if the free disk space shrinks under the defined mark of 100 MB. The important point for the design of states is that they should not contain changing elements like *actual disk-size*, *uptimes*, etc. Otherwise gossips will generate lots of state-change messages overwhelming the system manger. Gossips provides the possibility to log

```
sub my_syntax {
    my $self = shift;
[1] $self->add_syntax_key('run');
[2] $self->add_syntax_to_key('run','/^no$/',"wrong 'run' value");
[3] $self->add_syntax_key('period');
[4] $self->add_syntax_to_key('period','/^\d+sec$/',
                             "syntax error in 'period' parameter");
    $self->add_syntax_key('timeavg');
    $self->add_syntax_to_key('timeavg','/^\d+min$/',
                             "syntax error in 'time average' parameter");
    $self->add_syntax_key('proclim');
    $self->add_syntax_to_key('proclim','/^\d+\e.?\ed*proc$/',
                             "syntax error in 'proc limit' parameter");
}
```

**Figure 12**: Test_Load.pm: my_syntax-method.

```
sub my_struct {
    my $self = shift;
[A] $self->add_key_to_struct('period');
[B] $self->add_filter_to_parameter('period','/^(\d+)sec/',1);
[C] $self->add_key_to_struct('timeavg');
[D] $self->add_filter_to_parameter('timeavg','/^(\d+)min/',1);
    $self->add_key_to_struct('proclim');
    $self->add_filter_to_parameter('proclim','/^(\d+)proc/',1);
[E] $self->add_probe_to_struct('Probe_Load');
[F] $self->link_key_elem_to_probe_period('period',1,'Probe_Load');
[G] $self->link_key_elem_to_test('timeavg',1);
[H] $self->link_key_elem_to_test('proclim',1);
}
```

**Figure 13**: Test_Load.pm: my_struct-method.

changing values like '*free disk space*.' These values can be submitted to the server along with the state message. The server then stores these values in a logfile associated with the corresponding knowledge base file.

## Conclusions

The distributed architecture of gossips builds a scalable monitoring system. Through its flexible and central configuration environment, together with its command-line module, gossips is easily maintainable. The object oriented design of gossips builds a flexible and well defined framework for developing new monitoring tasks. The concept of separating data acquisition and data analysis makes defined monitoring tasks reusable and provides the possibility to build combined tests. The knowledge base allows to archive solutions to known problems in one place and to integrate the knowledge of the system manager.

By including cfengine, gossips could be extended into a automated repair tool. Development of an SNMP-probe-class would extend the monitor software to a low level device monitor.

## Availability

Gossips source and documentation along with its monitoring tests are available on the web-page http://isg.ee.ethz.ch/tools. There is a mailing list on gossips. Send an email with subject: *subscribe* to gossips-request@list.ee.ethz.ch to subscribe.

## Acknowledgments

We would like to thank the following people for their feedback and suggestions: our co-workers Andreas Karrer, David Schweikert, Edwin Thaler, Christoph Wicki, Fritz Zaucker, as well as our shepherds Mark Burgess and Todd K. Watson.

## Author Information

Victor Götsch is a third year Computer Science student at the Swiss Federal Institute of Technology, Zurich. After finishing his second year he started an internship with the IT Support Group of the Department of Electrical Engineering where he learned a lot about system management and spent most of his time developing the System and Service Monitor gossips. He will continue his studies in fall 2001 to get his degree in Computer Science.

Albert Wuersch got a degree in Electrical Engineering from the Swiss Federal Institute of Technology in 1999. He worked for nine months as a Trainee System Manager for the IT Support Group of the EE Department. During that time he designed the gossips concept and started the implementation.

Tobias Oetiker is a Senior System Manager with the above mentioned IT Support Group and has guided the gossips project over the last 18 months.

## References

[1] Sellens, John, "System and Network Monitoring," *;login:*, Vol 25, No. 3, June, 2000.

[2] Case, Fedor, Schoffstall, and Davin, "A Simple Network Management Protocol (SNMP)," *RFC 1157, SNMP*, May, 1990.

[3] MacGuire, Sean, "Big Brother, a tool for proactive network monitoring," http://www.bb4.com .

[4] Hansen, S. E., E. T. Atkins, "Automated System Monitoring and Notification With Swatch,"

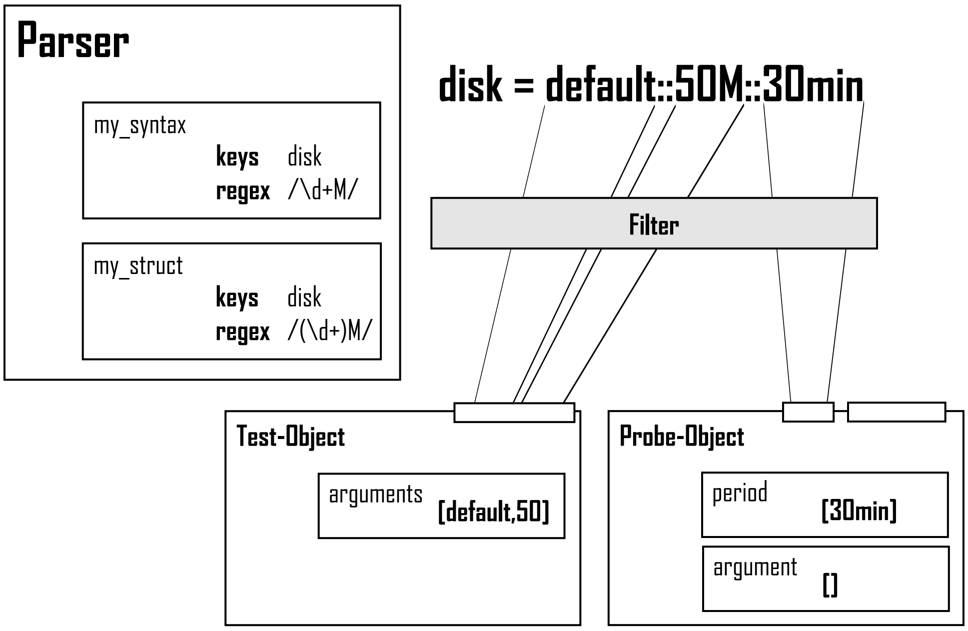


**Figure 14**: Filtering of test parameter and linking them to the test or probe object.

*Proceedings of the Seventh Systems Administration Conference (LISA VII)*, p. 145, USENIX Association, Berkeley, CA.

[5] Johnson, Stephen L., "Spong – Systems and Network Montoring," http://spong.sourceforge.net .

[6] Osterlund, R., "PIKT: Problem Informant/Killer tool, *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, p. 147, USENIX Association, Berkeley, CA.

[7] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, http://www.iu.hio.no/cfengine, Vol 8, No. 3, 1995,

[8] Oetiker, Tobias, "RRDTool, The Round Robin Database Tool for Long Time Monitoring," http://people.ee.ethz.ch/~oetiker/webtools/rrdtool .