Monitoring activity on multiple network servers

O'Connell, S.

Publisher's version / Version de l'éditeur:
https://doi.org/10.4224/8896205
Student Report (National Research Council of Canada. Institute for Ocean Technology); no. SR-2006-22, 2006

DOCUMENTATION PAGE

REPORT NUMBER	NRC REPORT NUMBER	DATE
SR-2006-22		August 2006

REPORT SECURITY CLASSIFICATION	DISTRIBUTION
Unclassified	Unlimited

TITLE
Monitoring Activity on Multiple Network Servers

AUTHOR(S)
Shane O'Connell

CORPORATE AUTHOR(S)/PERFORMING AGENCY(S)

Institute for Ocean Technology, National Research Council, St. John's, NL

PUBLICATION

SPONSORING AGENCY(S)

IMD PROJECT NUMBER	NRC FILE NUMBER

KEY WORDS	PAGES	FIGS.	TABLES
Network, Server, Monitoring, Logs, Syslog	43	8	

SUMMARY

Proactive monitoring of network servers is useful in predicting network problems so that they can be prevented. Log files frequently record errors which can be clues to predict larger problems. Centralized logging can be used to help monitor many log files from many servers. In this report I describe a system of centralized logging which I have set up at IOT, using freely-available standard tools, thus minimizing the training needed to maintain the system. A script is run nightly which e-mails network administrators a summary of the important log messages from all servers from the previous day. Administrators should read these e-mails and attempt to respond to every message. This can be either by fixing the problem that caused the message, or by deciding that the message is unimportant and adding it to the list of unimportant messages so that the nightly e-mail does not include it again.

ADDRESS National Research Council
Institute for Ocean Technology
Arctic Avenue, P. O. Box 12093
St. John's, NL A1B 3T5
Tel.: (709) 772-5185, Fax: (709) 772-2462

National Research Council Canada
Institute for Ocean Technology

Conseil national de recherches Canada
Institut des technologies océaniques

Monitoring Activity on Multiple Network Servers

SR-2006-22

Shane O'Connell

August 2006

TABLE OF CONTENTS

1.0 INTRODUCTION....1
2.0 LOG FILES....1
2.1 UNIX Syslog....1
2.2 Windows Event Logs....2
3.0 REMOTE LOGGING....3
3.1 UNIX/Linux....4
3.2 Network Devices....4
3.3 Microsoft Windows....4
3.3.1 Design of the windows syslog client....5
4.0 CENTRAL LOG FILE MANAGEMENT....6
4.1 Syslog-ng Configuration....6
4.2 Log File Rotation....7
4.3 Swatch Configuration....8
5.0 CONCLUSIONS....9
6.0 RECOMMENDATIONS....9

LIST OF FIGURES

Figure 1. Log file containing syslog messages....2
Figure 2. Windows event log entries viewed in the Event Viewer....3
Figure 3. Configuration file for syslog....4
Figure 4. Windows syslog client configuration....5
Figure 5. Configuration file for the Windows syslog client....6
Figure 6. Directory listing showing log file rotation....7
Figure 7. Swatch configuration....8
Figure 8. Alternate swatch configuration....8

APPENDICES

Appendix A. Source code listing for log rotation and swatch script
Appendix B. Example swatch configuration file
Appendix C. Extra options for syslog-ng configuration file
Appendix D. Windows syslog client source code

1.0 INTRODUCTION

A well-managed computer network requires proactive network policies. Problems can arise on a network at any time, and these problems can have very negative repercussions, depending on what the purpose of the network is and who is using it. In order to minimize the damage of these problems, network administrators need tools which will allow them to monitor the network closely. They need to receive notification of smaller errors as they happen, so that they can be fixed in order to prevent them from developing into larger problems.

This report documents the steps I have taken to put such a system in place at the Institute for Ocean Technology (IOT). I am a work term student with the computer systems group, working under the supervision of Paul Thorburn. We required a centralized logging server that could accept logs from several different sources, and then could provide notification of log entries that could indicate an error. Previously, all logging done by servers was stored on the server itself until it was deleted automatically. These logs were generally not checked unless there was a specific problem that required such action. We also required that any centralized logging solution would need to use standard programs and tools that are well-documented so that others could maintain the server without the need for any extra training.

2.0 LOG FILES

Many of the systems in use at IOT have the ability to record when important events take place into log files. These files are what allow us to monitor the network. By storing all log files in a central location, we have the ability to monitor many systems at once. There are two primary types of log files which we attempt to monitor. These are UNIX syslog-style logs, and event logs, used by Microsoft Windows. Syslog-style logs are used on UNIX servers, as well as on UNIX-like operating systems such as Linux, but are also the standard for network devices such as printers. This is because syslog is a very simple protocol, supports transferral of logs over a network, and has been in use for decades. Event logs are used only on Microsoft Windows and do not support remote logging natively.

2.1 UNIX Syslog

Syslog originated on the UNIX platform, and is a very simple logging system. On UNIX, the word “syslog” refers to the network protocol, the API function that sends log entries, and the name of the program that accepts and manages logs. In this report the term will refer to either the protocol or the program interchangeably.

On UNIX, logs are stored in text files. These text files can be read with any standard text editor. Each line contains one message, with various fields separated by spaces. Each message generally contains the date, the time, the

originating hostname, the process name, the process id (pid), and the message text itself. [Fig 1] The format of a syslog message, however, can vary from source to source. For example, some implementations may leave out the hostname or the process name.

```
bragg:~# tail -n 3 /var/log/syslog
Aug 15 13:58:02 bragg syslog-ng[14504]: STATS: dropped 0
Aug 15 14:08:02 bragg syslog-ng[14504]: STATS: dropped 0
Aug 15 14:09:01 bragg /USR/SBIN/CRON[24992]: (root) CMD (   [ -d
/var/lib/php4 ] && find /var/lib/php4/ -type f -cmin
+$(/usr/lib/php4/maxlifetime) -print0 | xargs -r -0 rm)
```

Figure 1. Log file containing syslog messages

Syslog messages are also assigned one more identifier not included in its line in the log file. In the syslog protocol itself, there is a provision for including a number known as the priority with each message. This number is generally not written into the log file, as its purpose is only to allow messages to be organized into separate files such that related messages are kept together.

The priority is composed of two parts, the facility and the level. The facility indicates the UNIX subsystem from which the message originated. Each facility is identified by a string which is then mapped to a number for use in the protocol. There are facilities for specific purposes, such as `kern`, `mail`, `lpr`, or `auth`, as well as general purpose facilities named `local0` through `local7`. The level is also identified by a string, and it indicates the importance of the message. They are, in increasing importance: `debug`, `info`, `notice`, `warning`, `crit`, `alert`, `emerg`.

Syslog messages have the capability to be sent over the network to a remote syslog server. The network protocol for syslog is very simple. The sender opens a UDP connection to port 514 on the destination server, and then sends the priority followed by the complete line to be inserted into the config file, including all fields.

2.2 Windows Event Logs

Event logs are a technology unique to Microsoft Windows. They are similar in some ways to syslog logs, but there are still a few significant differences. Event logs are stored in a proprietary file format which can only be accessed through the Windows API. To read event logs on Windows, the administrator must view them using the Event Viewer. [Fig 2]

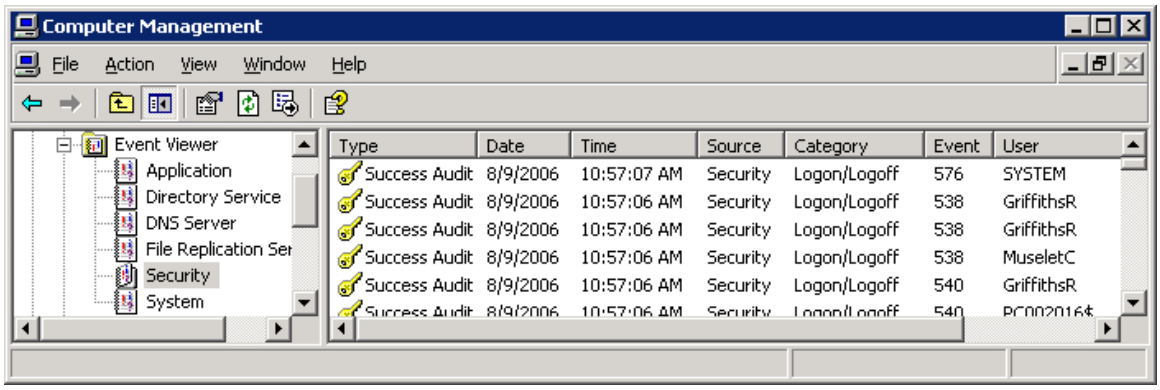

Figure 2. Windows event log entries viewed in the Event Viewer

There are several different event logs, each with different names. All Windows systems have the three basic event logs: Application, Security, and System. Windows Servers generally have a few more, and other programs can create new ones as needed, although that does not generally occur. Programs running on the system generally use the Application log, and Windows uses the Security and System logs.

Events written to these logs all have some information associated with them, such as the date, time, source, etc. Additionally, they also have a type, which is analogous to the syslog level. The type is chosen from a list, and indicates the importance of the message. In increasing levels of importance, they are: Information, Warning, and Error. The only exception is the Security event log, which uses Success Audit and Failure Audit as the possible types. In this case, Success Audit means that access was granted to a resource, and Failure Audit indicates that access was denied.

3.0 REMOTE LOGGING

Remote logging is accomplished by sending all logs over the network using the syslog protocol. Syslog has been chosen because it is a well-known standard with proven reliability. It is easy to maintain, as knowledge of how to use and maintain syslog logs is common, and thus no special training is required to use it. Syslog also already has the ability to log to a remote server, unlike windows event logs.

There are a large amount of network devices that support the syslog protocol, along with UNIX and UNIX-like operating systems such as Linux. These systems are relatively simple to configure to log remotely. Configuring Windows to use syslog, however, is more difficult. One solution is to install a program which reads the event log and converts the events to syslog messages before sending them over the network, and this is the approach we use.

3.1 UNIX/Linux

On UNIX, the syslog daemon is responsible both sending logs and receiving logs. In this sense, it is both the client and the server. All of the processes running locally on a system send their logs to the local syslog daemon, which then sends them over the network to the syslog daemon on the log server.

In order to configure syslog to work in this manner, we have to add a line to the syslog configuration file. [Fig 3] This line tells syslog the name of the log server as well as specifying which logs to send. There are two parts of the line, separated by spaces. The first part specifies which facility and level the line applies to, separated by a period. In our case, we want all facility and levels, so they are both specified with an asterisk. The second part specifies what to do with the logs. To send logs to a remote server, the second part must start with `@` followed by the hostname, which in this example is `log-server-hostname`.

```
#  /etc/syslog.conf     Configuration file for syslogd.
#
#                       For more information see syslog.conf(5)
#                       manpage.

*.*                             @log-server-hostname
```

Figure 3. Configuration file for syslog

3.2 Network Devices

Network devices that support sending events over syslog can have varying interfaces. However, configuring such a device for syslog is usually not difficult. Generally, there is a single setting which specifies the hostname or IP address of the log server.

3.3 Microsoft Windows

Microsoft Windows does not support syslog natively. Thus, for remote logging of windows event logs, I have written a program that runs as a service, watches the event log for entries, and forwards them to the syslog server. Installation of this program is simple; all that is required is that three files be together in one folder. There are two executables, one is for configuration, and one is the service itself. Once the files are in the same directory, running the configuration program allows you to choose a hostname to forward logs to and then start the service. [Fig 4]

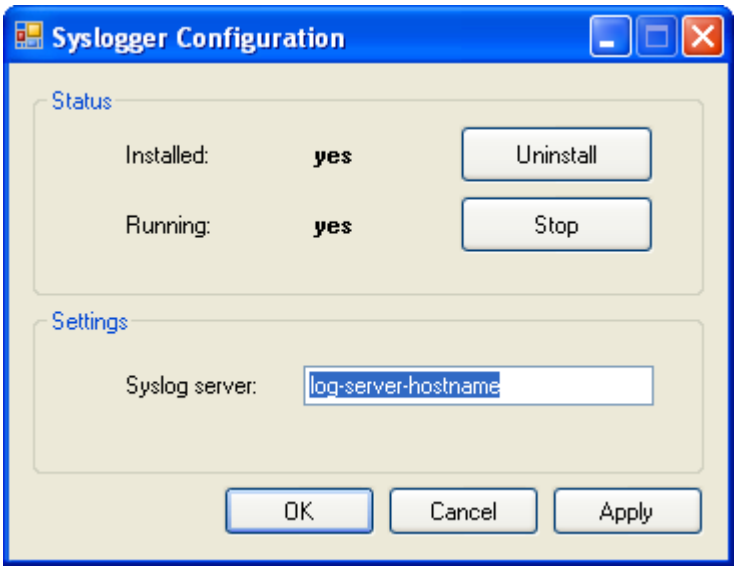

Figure 4. Windows syslog client configuration

3.3.1 Design of the windows syslog client

The design of the syslog client is relatively simple. It has been programmed in Microsoft Visual C#, and uses version 2.0 of the .NET runtime library. The source code has been included in Appendix D.

There are three files in the directory that the client is installed in. They are named `syslogger.exe`, `syslogger.exe.config`, and `sysloggercfg.exe`. The first one is the syslog client service itself, the second is the .NET XML configuration file for the service, and the third is a program that shows the configuration dialog as shown in Figure 4. The third program uses the .NET API to modify the configuration file for the service program.

The configuration file has a few options that are not visible in the configuration program. [Fig 5] The mapping between Windows event logs and syslog facilities can be changed here. This may be required in the event that a program on the server creates a new event log that needs to be monitored. There are two `ArrayOfString` XML tags, one for the event logs and one for the syslog facilities, given in numerical form. Facilities numbered 16 through 23 correspond to the `local0` through `local7` facilities. Although I have only used the local use facilities, any syslog facility can be specified here if necessary. It is important, however, that the syslog server is set up so that these facilities will not be directed into the log files for the systems that normally use them. Any change to these values will likely require restarting the service, unlike changes made with the configuration program itself, which notifies the service when changes are made.

```
            <setting name="EventLogs" serializeAs="Xml">
                <value>
                    <ArrayOfString
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                        <string>System</string>
                        <string>Security</string>
                        <string>Application</string>
                    </ArrayOfString>
                </value>
            </setting>
            <setting name="Facilities" serializeAs="Xml">
                <value>
                    <ArrayOfString
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                        <string>16</string>
                        <string>17</string>
                        <string>18</string>
                    </ArrayOfString>
                </value>
            </setting>
```

Figure 5. Configuration file for the Windows syslog client

4.0 CENTRAL LOG FILE MANAGEMENT

After all systems have been set up to forward to a log server, this server must be set up to properly manage the logs. The syslog daemon that comes with most UNIX and UNIX-like systems has the ability to receive logs over the network; however, its configuration file lacks flexibility in sorting logs based on the hostname that the messages are received from. Thus, all logs from all hosts would be saved into the same log files. In order to overcome this problem, another syslog daemon can be used, known as syslog-ng. Its configuration is far more flexible. Additionally, a nightly script is run, which scans the log files using a program called swatch, and then deletes older old files to save space.

4.1 Syslog-ng Configuration

Syslog-ng uses a far more flexible configuration file then the original syslog daemon. There are five main types of entries in the configuration file. The most important type of entry is the `log` entry. Each `log` entry references `source`, `filter`, and `destination` entries, which are three of the other types that are in the configuration file. A `source` entry specifies a place where log files are obtained, i.e. logs obtained over the network. A `filter` entry then allows you to choose which messages from the `source` entries are to be forwarded to the `destination` entries. A `destination` entry specifies where the logs will be stored, which could be a file, or the terminal, or even an SQL database. Each `log` entry can reference multiple entries of each type; however, there must always be a reference to at least one `source` and one `destination`. The fifth and final type of entry is the `options` entry, which specifies general options for syslog-ng.

For our configuration file, all of the default options are left in place. The default options, at least in the Debian GNU/Linux package, recreate the normal options that are commonly used with the normal syslog daemon. This means that the normal log files will be unaffected. Extra options are simply added to the file to configure our extra log files.

Configuration of the extra options is somewhat complex. Each host needs to have a directory for its log files, and each facility needs to have its own file. However, if the log is coming from a Windows system, syslog-ng has to convert the facility to a Windows event log name and use that as the filename. This is done using a `destination` for each event log name, and using `filter` entries to direct each corresponding facility into each `destination`. The `destination` also has the hostname included in its path as a variable. There is also one extra `destination`, which simply has the host name and facility name as variables inside it, and this is used for any non-event log messages. This configuration can be seen in Appendix C.

4.2 Log File Rotation

Log files can grow in size quite quickly, thus anytime one is used, there should be some way for older logs to be deleted in order to avoid wasted space. This can be accomplished by separating logs into multiple files. One file is the active log file, where new entries are being added. The rest of the files have the same name as the first, but they have a period and a number appended. [Fig 6]

```
bragg:/var/log/HOSTS/karfe# ls -l
total 520
-rw-r-----  1 root adm  39410 Aug 16 14:36 Application.log
-rw-r-----  1 root adm   2790 Aug 16 06:26 Application.log.1.gz
-rw-r-----  1 root adm   1927 Aug 15 06:26 Application.log.2.gz
-rw-r-----  1 root adm 120936 Aug 16 12:57 DNSServer.log
-rw-r-----  1 root adm   1835 Aug 15 13:18 DNSServer.log.1.gz
-rw-r-----  1 root adm    726 Aug 15 03:02 DNSServer.log.2.gz
-rw-r-----  1 root adm    285 Aug 16 13:17 DirectoryService.log
-rw-r-----  1 root adm    273 Aug 16 01:17 DirectoryService.log.1.gz
-rw-r-----  1 root adm    253 Aug 15 01:17 DirectoryService.log.2.gz
-rw-r-----  1 root adm    232 Aug 16 06:28 Security.log.1.gz
-rw-r-----  1 root adm 163043 Aug 15 06:28 Security.log.2.gz
-rw-r-----  1 root adm 138102 Aug 16 14:36 System.log
-rw-r-----  1 root adm  13549 Aug 16 05:56 System.log.1.gz
-rw-r-----  1 root adm   3197 Aug 14 19:32 System.log.2.gz
```

Figure 6. Directory listing showing log file rotation

Every night, a script is run which takes every log file ending in a number, increments the number, and then saves the current log file with a `.1` extension. This file is then compressed with `gzip` to save space. Then the script deletes any

log file with a `.10` extension, thus causing all logs older then 10 days to be automatically deleted.

4.3 Swatch Configuration

In order to have advanced notice of network problems, there needs to be a way of notifying administrators of important log messages. Swatch is a program that can do this. On our log server, the script that rotates the log files every night also runs swatch to scan the logs, and then e-mails the results of the scan to the administrators.

Swatch configuration is fairly straightforward. Its configuration file specifies a list of messages to look for, and what action to take when they are found. An example of this type of configuration is shown in Figure 7. In this example, you can see that swatch is configured to watch for the phrase “No space left on device”. Below that you can see two actions that it will take; swatch will display the message on the display in a red font, and then run a script which would presumably perform some action to solve the situation.

```
watchfor /No space left on device/
  echo red
  exec /home/bin/no-space.sh

watchfor /Hard drive failure/
  echo red
  exec /home/bin/hard-drive-failure.sh
```

Figure 7. Swatch configuration

However, when configured this way, swatch would require a list of every possibly important error message. This can be difficult to create, as there are thousands of possible errors, and there is no way to tell how the message would be phrased unless it had occurred and you could read the log file.

An easier and more foolproof method of configuring swatch is to give it a list of messages that are not important, and then tell it to notify you of any other error. This way, while you may accidentally get unimportant messages, you will definitely also receive any important messages. If an unimportant message is received, then it can be added to the list in order to prevent notification of it in the future. Figure 8 shows such a configuration; you can see that we have chosen to ignore all messages of the Information type, because these are not errors. The rest are saved into a file. This is the type of configuration used on our server.

```
ignore /\[Information\]/

watchfor /.*/
  pipe "tee –a /var/local/swatch/tmpfile"
```

Figure 8. Alternate swatch configuration

5.0 CONCLUSIONS

Proactive monitoring of a network is needed in order to prevent problems before they occur. Log files are one way of predicting problems; however, they are normally only stored on the computer that generated them. In order to easily monitor logs, a central log server can be set up, to which all logs are forwarded. The log server can then be set up to notify administrators of important messages. This can be accomplished by using standard, well-known tools, of which knowledge of their workings is common. This is an important consideration when future maintenance of the system is required.

6.0 RECOMMENDATIONS

To prevent network problems, log files should be monitored carefully. This report presents a system of monitoring several network servers centrally, which allows this. Network administrators should read swatch output and respond to all messages, either by adding the message to the swatch ignore list, or by solving the problem which caused the message.

Appendix A

Source code listing for log rotation and swatch script

```
#!/bin/bash

TEMPFILE=/var/local/swatch/tmpfile

echo "Here are the important log entries for yesterday, according to
 swatch." > $TEMPFILE
echo -n "If there are messages here that are actually unimportant, they
 should be " >> $TEMPFILE
echo -n "added to the appropriate config file in /etc/swatch/swatchrc.*
 on the log file server " >> $TEMPFILE
echo "so that they are not seen again." >> $TEMPFILE

echo >> $TEMPFILE
echo >> $TEMPFILE

for HOST in `ls /var/log/HOSTS` ; do
  if [ ! -f /var/log/HOSTS/$HOST/noswatch ]; then
    echo "----------------------------------------" >> $TEMPFILE
    echo "Notable log entries for $HOST" >> $TEMPFILE
    echo "----------------------------------------" >> $TEMPFILE

    echo >> $TEMPFILE
  fi

  if ls /var/log/HOSTS/$HOST/*.log > /dev/null ; then

    for LOGFILE in `ls /var/log/HOSTS/$HOST/*.log` ; do
      LOG=`echo $LOGFILE | cut -d / -f 6 | cut -d . -f 1`


      # Now we are rotating the log files
      for LOGNUMBER in 9 8 7 6 5 4 3 2 1 ; do
        if [ -f $LOGFILE.$LOGNUMBER.gz ]; then
          mv $LOGFILE.$LOGNUMBER.gz $LOGFILE.`expr $LOGNUMBER + 1`.gz
        fi
      done

      if [ -f $LOGFILE.10.gz ]; then
        rm $LOGFILE.10.gz
      fi

      mv $LOGFILE $LOGFILE.1


      if [ ! -f /var/log/HOSTS/$HOST/noswatch ]; then
        if [ -f /etc/swatch/swatchrc.$LOG ]; then
          echo \*\* $LOG log: >> $TEMPFILE
          swatch --config-file /etc/swatch/swatchrc.$LOG --script-
 dir=/var/local/swatch -f $LOGFILE.1 > /dev/null
        fi
      fi

      gzip $LOGFILE.1
    done

  else
```

```
    echo "Error: No log files found for $HOST, please check to make
  sure logs are being sent" >> $TEMPFILE
  fi

  if [ ! -f /var/log/HOSTS/$HOST/noswatch ]; then
    echo >> $TEMPFILE
  fi
done


SMTPSERVER=`host -t MX nrc-cnrc.gc.ca | grep -o -m 1 "[A-Za-z0-
  9\.]\+$"`

sendEmail -f helpdesk.iot@nrc-cnrc.gc.ca -t helpdesk.iot@nrc-cnrc.gc.ca
  -u "Nightly swatch output" -s $SMTPSERVER -o message-
  file=/var/local/swatch/tmpfile > /dev/null

rm /var/local/swatch/tmpfile
```

Appendix B

Example swatch configuration file

```
# This is swatchrc.System

# Each /etc/swatch/swatchrc.* cooresponds to logfiles named
/var/log/HOSTS/<host>/*.log
# This is because the errors in each type of log file may be different
enough that they should have their own
# list of messages to ignore

# The syntax is "ignore /<something>/" where the <something> is a
regular expression which matches the message to ignore



# We can ignore all events of the type "Information" because they do
not indicate errors
ignore /^\w{3}\s+\d{1,2}\s\d{2}:\d{2}:\d{2}\s\S+\s\S+\[Information\]/

ignore /Software Update Services successfully synchronized all content/
ignore /Print\[Warning\]: Printer .+ (is|was) (created|purged|pending
deletion)/


# All other messages are sent in the e-mail
# Here we pipe the message into tee, which appends it to
/var/local/swatch/tmpfile and displays it on the screen
# We then echo newlines onto the screen and into the file, because the
message that is piped into tee doesn't have any
watchfor /.*/
  pipe "tee -a /var/local/swatch/tmpfile; echo >>
/var/local/swatch/tmpfile; echo >> /var/local/swatch/tmpfile; echo;
echo"
  throttle 1:00:00,use=regex
```

Appendix C

Extra options for syslog-ng configuration file

```
source s_remote {
        # listens for connections on localhost, UDP port 514
        udp();
};

# destinations for remote hosts
destination df_remote_host_generic {
        file("/var/log/HOSTS/$HOST/$FACILITY.log");
};

destination df_remote_host_system {
        file("/var/log/HOSTS/$HOST/System.log");
};
destination df_remote_host_security {
        file("/var/log/HOSTS/$HOST/Security.log");
};
destination df_remote_host_application {
        file("/var/log/HOSTS/$HOST/Application.log");
};
destination df_remote_host_directoryservice {
        file("/var/log/HOSTS/$HOST/DirectoryService.log");
};
destination df_remote_host_dnsserver {
        file("/var/log/HOSTS/$HOST/DNSServer.log");
};
destination df_remote_host_filereplicationservice {
        file("/var/log/HOSTS/$HOST/FileReplicationService.log");
};


filter f_local0 { facility(local0); };
filter f_local1 { facility(local1); };
filter f_local2 { facility(local2); };
filter f_local3 { facility(local3); };
filter f_local4 { facility(local4); };
filter f_local5 { facility(local5); };

filter f_notlocal012345 { not facility(local0) and not facility(local1)
and not facility(local2) and not facility(local3) and not
facility(local4) and not facility(local5); };

filter f_windows_error { level(err,crit,alert,emerg); };
filter f_windows_host { host("snekke") or host("skiede")
                        or host("karfe") or host("knarr"); };
filter f_nonwindows_host { not filter(f_windows_host); };

# Logfiles for windows hosts here (System, Security, and Application
are kept in seperate files)
# We assume local0 = System, local1 = Security, and local2 =
Application
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_local0);
        destination(df_remote_host_system);
};
log {
```

```
        source(s_remote);
        filter(f_windows_host);
        filter(f_local1);
        destination(df_remote_host_security);
};
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_local2);
        destination(df_remote_host_application);
};
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_local3);
        destination(df_remote_host_directoryservice);
};
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_local4);
        destination(df_remote_host_dnsserver);
};
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_local5);
        destination(df_remote_host_filereplicationservice);
};

# Catch any log files from windows that don't seem to be in the
facilities they should be
log {
        source(s_remote);
        filter(f_windows_host);
        filter(f_notlocal012345);
        destination(df_remote_host_generic);
};

# Logfiles for non-windows remote hosts
log {
        source(s_remote);
        filter(f_nonwindows_host);
        destination(df_remote_host_generic);
};
```

Appendix D

Windows syslog client source code

File: Service\Program.cs

```
//using System.Collections.Generic;
using System.ServiceProcess;
//using System.Text;

namespace Syslogger
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main()
        {
            ServiceBase[] ServicesToRun;

            // More than one user Service may run within the same
 process. To add
            // another service to this process, change the following
 line to
            // create a second service object. For example,
            //
            //   ServicesToRun = new ServiceBase[] {new Service1(), new
 MySecondUserService()};
            //
            ServicesToRun = new ServiceBase[] { new SysloggerService()
 };

            ServiceBase.Run(ServicesToRun);
        }
    }
}
```

File: Service\ProjectInstaller.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Configuration.Install;

namespace Syslogger
{
    [RunInstaller(true)]
    public partial class ProjectInstaller : Installer
    {
        public ProjectInstaller()
        {
            InitializeComponent();
        }
    }
}
```

File: Service\ProjectInstaller.Designer.cs

```
namespace Syslogger
{
    partial class ProjectInstaller
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Component Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.ProcessInstaller = new
System.ServiceProcess.ServiceProcessInstaller();
            this.ServiceInstaller = new
System.ServiceProcess.ServiceInstaller();
            //
            // ProcessInstaller
            //
            this.ProcessInstaller.Account =
System.ServiceProcess.ServiceAccount.LocalSystem;
            this.ProcessInstaller.Password = null;
            this.ProcessInstaller.Username = null;
            //
            // ServiceInstaller
            //
            this.ServiceInstaller.ServiceName = "Syslogger Service";
            this.ServiceInstaller.StartType =
System.ServiceProcess.ServiceStartMode.Automatic;
            //
            // ProjectInstaller
            //
            this.Installers.AddRange(new
System.Configuration.Install.Installer[] {
            this.ProcessInstaller,
```

```
            this.ServiceInstaller});

        }

        #endregion

        private System.ServiceProcess.ServiceProcessInstaller
ProcessInstaller;
        private System.ServiceProcess.ServiceInstaller
ServiceInstaller;
    }
}
```

File: Service\Settings.cs

```
namespace Syslogger.Properties {


    // This class allows you to handle specific events on the settings
class:
    //  The SettingChanging event is raised before a setting's value is
changed.
    //  The PropertyChanged event is raised after a setting's value is
changed.
    //  The SettingsLoaded event is raised after the setting values are
loaded.
    //  The SettingsSaving event is raised before the setting values
are saved.
    internal sealed partial class Settings {

        public Settings() {
            // // To add event handlers for saving and changing
settings, uncomment the lines below:
            //
            // this.SettingChanging +=
this.SettingChangingEventHandler;
            //
            // this.SettingsSaving += this.SettingsSavingEventHandler;
            //
        }

        private void SettingChangingEventHandler(object sender,
System.Configuration.SettingChangingEventArgs e) {
            // Add code to handle the SettingChangingEvent event here.
        }

        private void SettingsSavingEventHandler(object sender,
System.ComponentModel.CancelEventArgs e) {
            // Add code to handle the SettingsSaving event here.
        }
    }
}
```

File: Service\SysloggerService.cs

```
using System;
```

```
using System.Collections.Generic;
//using System.ComponentModel;
//using System.Data;
using System.Diagnostics;
using System.ServiceProcess;
//using System.Text;
//using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text.RegularExpressions;

namespace Syslogger
{
    public partial class SysloggerService : ServiceBase
    {
        List<EventLog> eventLogs;

        Socket syslogSocket;
        IPEndPoint socketEndPoint;

        public SysloggerService()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {

            eventLogs = new
List<EventLog>(Properties.Settings.Default.EventLogs.Count);
            List<string> errorList = new List<string>();
            foreach (string logName in
Properties.Settings.Default.EventLogs)
            {
                try
                {
                    eventLogs.Add(new EventLog(logName));
                    eventLogs[eventLogs.Count - 1].EntryWritten += new
EntryWrittenEventHandler(LogEntryWritten);
                    eventLogs[eventLogs.Count - 1].EnableRaisingEvents
= true;
                }
                catch (Exception e)
                {   // We do not actually put this into the event log
til after the foreach loop
                    // to ensure that the error messages will be send
to the syslog server
                    // (i.e., we are making sure that Application
events will be sent to syslog
                    // before we start making them ourselves)
                    errorList.Add("Could not get logs from event log
named '" + logName + "': " + e.Message);
                }
            }
            foreach (string error in errorList)
                EventLog.WriteEntry("Syslogger Service", error,
EventLogEntryType.Error);
```

```
            try
            {
                syslogSocket = new Socket(AddressFamily.InterNetwork,
SocketType.Dgram, ProtocolType.Udp);
                IPHostEntry hostEntry =
Dns.GetHostEntry(Properties.Settings.Default.Hostname);
                socketEndPoint = new
IPEndPoint(hostEntry.AddressList[0], 514);
                syslogSocket.Blocking = false;
            }
            catch (Exception e)
            {
                EventLog.WriteEntry("Syslogger Service", "Error
connecting to syslog host: " + e.Message, EventLogEntryType.Error);
                EventLog.WriteEntry("Syslogger Service", "Service now
stopping due to inability to connect to syslog host.",
EventLogEntryType.Error);
                this.Stop();
            }



        }

        void LogEntryWritten(object sender, EntryWrittenEventArgs e)
        {
            // On Windows clusters, each node receives events from all
other nodes in the cluster
            // in their own event logs, but we only want the events
from this particular node
            if (e.Entry.MachineName != Environment.MachineName) return;

            // This retreives the facility specified in the config file
for this event log name
            int facility =
int.Parse(Properties.Settings.Default.Facilities[Properties.Settings.De
fault.EventLogs.IndexOf(((EventLog)sender).Log)]);
            int severity = 0;
            switch (e.Entry.EntryType)
            {
                case EventLogEntryType.Error:
                    severity = 3; // syslog code for "error"
                    break;
                case EventLogEntryType.FailureAudit:
                    severity = 4; // syslog code for "warning"
                    break;
                case EventLogEntryType.Information:
                    severity = 6; // syslog code for "informational"
                    break;
                case EventLogEntryType.SuccessAudit:
                    severity = 5; // syslog code for "notice"
                    break;
                case EventLogEntryType.Warning:
                    severity = 4; // syslog code for "warning"
                    break;
            }
```

```
            string[] months = {"Jan", "Feb", "Mar", "Apr", "May",
"Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

            // Now we make sure the message only has ASCII characters
between
            // 32 and 126, inclusive, as required by the syslog RFC
            // All other characters are replaced with spaces
            Regex rx = new Regex(@"[^\x20-\x7E]");
            string message = rx.Replace(e.Entry.Message, " ");

            int part = 1;   // Specifies which part of the message we
are sending, when the message is
                            // long enough to require being sent in
seperate parts (syslog messages
                            // cannot be greater then 1024 characters
in length, event log entries can
                            // be longer)
            bool isSplit = false;

            while (message.Length > 0)
            {
                string buf =
                    "<" + (facility * 8 + severity) + ">" +

                    months[e.Entry.TimeGenerated.Month - 1] + " " +
e.Entry.TimeGenerated.Day.ToString().PadLeft(2, ' ') +
                    " " +
e.Entry.TimeGenerated.Hour.ToString().PadLeft(2, '0') +
                    ":" +
e.Entry.TimeGenerated.Minute.ToString().PadLeft(2, '0') +
                    ":" +
e.Entry.TimeGenerated.Second.ToString().PadLeft(2, '0') +

                    " " + Dns.GetHostName() +
                            // rx.Replace removes characters not
between 32 and 126, and
                            // e.Entry.Source.Replace removes spaces
(space is used to
                            // differentiate fields in the message)
                    " " + rx.Replace(e.Entry.Source.Replace(" ", ""),
"") + "[" + e.Entry.EntryType.ToString() + "]" +

                    ": ";

                if ((buf.Length + message.Length) > 1024)
                {
                    // The length is over 1024, so we need to send 1024
bytes now
                    // and then send the rest in the next pass of the
loop
                    string splitEventMessage = "(split event, part " +
part.ToString() + "): ";
                    int messageRemovedLength = 1024 - buf.Length -
splitEventMessage.Length;
                    buf += splitEventMessage + message.Substring(0,
messageRemovedLength);
```

```
                    message = message.Substring(messageRemovedLength);
                    isSplit = true;
                    part++;
                }
                else if (isSplit)
                {
                    // This is called when we have a split event and we
are on the last piece of the message
                    string splitEventMessage = "(split event, part " +
part.ToString() + "): ";
                    if ((buf.Length + splitEventMessage.Length +
message.Length) > 1024)
                    {
                        // Sometimes adding the (split event) stuff to
the message puts us over 1024 characters
                        // so this code sends only the first 1024
characters and then goes through the loop again
                        int messageRemovedLength = 1024 - buf.Length -
splitEventMessage.Length;
                        buf += splitEventMessage + message.Substring(0,
messageRemovedLength);
                        message =
message.Substring(messageRemovedLength);
                        part++;
                    }
                    else
                    {
                        buf += splitEventMessage + message;
                        message = "";
                    }
                }
                else
                {
                    // The message is under 1024 characters. Yay!
                    buf += message;
                    message = "";
                }

                try
                {

syslogSocket.SendTo(System.Text.Encoding.ASCII.GetBytes(buf),
socketEndPoint);
                }
                catch
                {
                    // There's not much we can do if this fails. We
shouldn't end the program, because
                    // it may start working again and we want to always
send logs if it is possible
                    // If we mark this error in the event log, this
function will get called again
                    // for our own error event and we may get stuck in
an infinite loop

                    // Instead, if we can't send the packet, we just
silently give up
```

```
                }
            }
        }

        protected override void OnStop()
        {
            foreach (EventLog eventLog in eventLogs)
            {
                eventLog.EnableRaisingEvents = false;
            }
        }

        protected override void OnCustomCommand(int command)
        {
            // We log beforehand and after because the syslog server
address may have changed in the middle
            // and we should tell both servers about the change
            EventLog.WriteEntry("Syslogger Service", "Received request
to update config file.", EventLogEntryType.Information);

            try
            {
                // We only have one custom command, and that's the one
that says to reload the config file
                Properties.Settings.Default.Reload();

                EventLog.WriteEntry("Syslogger Service", "Sucessfully
updated config file.");
            }
            catch (Exception e)
            {
                EventLog.WriteEntry("Syslogger Service", "Error
accessing config file: " + e.Message, EventLogEntryType.Error);
            }

            try
            {   // We also have to update the socketEndPoint to point
to the new syslog host in case it has changed
                IPHostEntry hostEntry =
Dns.GetHostEntry(Properties.Settings.Default.Hostname);
                socketEndPoint = new
IPEndPoint(hostEntry.AddressList[0], 514);
            }
            catch (Exception e)
            {
                EventLog.WriteEntry("Syslogger Service", "Error
connecting to syslog host: " + e.Message, EventLogEntryType.Error);
                EventLog.WriteEntry("Syslogger Service", "Service now
stopping due to inability to connect to syslog host.",
EventLogEntryType.Error);
                this.Stop();
            }
        }
    }
}
```

File: Service\SysloggerService.Designer.cs

```
namespace Syslogger
{
    partial class SysloggerService
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Component Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            //
            // SysloggerService
            //
            this.ServiceName = "Syslogger Service";

        }

        #endregion
    }
}
```

File: Configuration\frmConfig.cs

```
using System;
//using System.Collections.Generic;
//using System.ComponentModel;
//using System.Data;
using System.Drawing;
//using System.Text;
using System.Windows.Forms;
using System.ServiceProcess;
using System.Configuration;
```

```
namespace Syslogger
{
    public partial class frmConfig : Form
    {
        private ServiceController serviceController;

        // These hold the status of the service at any given time, they
are updated every 500 ms
        private bool serviceRunning = false;
        private bool serviceInstalled = false;

        private int serviceRunningTimeout = -1;
        private int serviceInstalledTimeout = -1;

        // These are for accessing the config file for syslogger.exe
        Configuration serviceConfig;
        ClientSettingsSection settingsSection;

        public frmConfig()
        {
            InitializeComponent();
        }

        private void frmConfig_Load(object sender, EventArgs e)
        {
            // If syslogger.exe is not in this folder, we cannot change
it's syslogger.exe.config file
            if (!System.IO.File.Exists("syslogger.exe"))
            {
                MessageBox.Show("This program must be run in the same
folder as syslogger.exe", "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                this.Close();
            }

            UpdateStatus(); // Updates the running/installed status on
the form
            txtSyslogServer.Select();

            try
            {
                // We are using the configuration API to indirectly
access syslogger.exe.config
                serviceConfig =
ConfigurationManager.OpenExeConfiguration("syslogger.exe");
                settingsSection =
(ClientSettingsSection)serviceConfig.GetSectionGroup("applicationSettin
gs").Sections["Syslogger.Properties.Settings"];

                txtSyslogServer.Text =
settingsSection.Settings.Get("Hostname").Value.ValueXml.InnerText;
                cmdApply.Enabled = false;   // Event handler for
TextChanged automatically changes Enabled
                                            // to true after the
previous line, so we change it back
            }
            catch (Exception ex)
```

```
            {
                MessageBox.Show("Unable to load configuration file for
syslogger.exe: " + ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                this.Close();
            }
        }

        private void tmrUpdateStatus_Tick(object sender, EventArgs e)
        {
            UpdateStatus(); // Updates the running/installed status on
the form

            // Check on the timeout counters
            if (serviceRunningTimeout != -1)
            {
                serviceRunningTimeout--;
                if (serviceRunningTimeout == 0 && serviceRunning ==
false)
                {
                    MessageBox.Show("Could not start the service: Timed
out.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceRunningTimeout = -1;
                }
                if (serviceRunningTimeout == 0 && serviceRunning ==
true)
                {
                    MessageBox.Show("Could not stop the service: Timed
out.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceRunningTimeout = -1;
                }
            }

            if (serviceInstalledTimeout != -1)
            {
                serviceInstalledTimeout--;
                if (serviceInstalledTimeout == 0 && serviceInstalled ==
false)
                {
                    MessageBox.Show("Could not install service: Timed
out.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceInstalledTimeout = -1;
                }
                if (serviceInstalledTimeout == 0 && serviceInstalled ==
true)
                {
                    MessageBox.Show("Could not uninstall service: Timed
out.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceInstalledTimeout = -1;
                }
            }
        }

        private void UpdateStatus()
        {
            // Find out if the service is installed
            bool installedNow = false;
```

```
            try
            {
                ServiceController[] services =
ServiceController.GetServices();

                foreach (ServiceController service in services)
                    if (service.DisplayName == "Syslogger Service")
                    {
                        serviceController = service;
                        installedNow = true;
                    }
            }
            catch (Exception e)
            {
                MessageBox.Show("Unable to find status of service: " +
e.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }


            // Cancel any timeout counters if the installed status has
changed
            if (installedNow == true && serviceInstalled == false)
serviceInstalledTimeout = -1;
            if (installedNow == false && serviceInstalled == true)
serviceInstalledTimeout = -1;
            serviceInstalled = installedNow;



            // Find out if the service is running
            bool runningNow = false;
            if (serviceInstalled && serviceController.Status ==
ServiceControllerStatus.Running)
                runningNow = true;

            // Cancel any timeout counters of the running status has
changed
            if (runningNow == true && serviceRunning == false)
serviceRunningTimeout = -1;
            if (runningNow == false && serviceRunning == true)
serviceRunningTimeout = -1;
            serviceRunning = runningNow;



            // Update "Installed" status and button
            if (serviceInstalled)
            {
                lblInstalled.Text = "yes";
                lblInstalled.ForeColor =
System.Drawing.SystemColors.ControlText;
                cmdInstallUninstall.Text = "Uninstall";
            }
            else
            {
                lblInstalled.Text = "no";
```

```
                lblInstalled.ForeColor = Color.Red;
                cmdInstallUninstall.Text = "Install";
            }
            if (serviceInstalledTimeout == -1)
                cmdInstallUninstall.Enabled = true;
            else
                cmdInstallUninstall.Enabled = false;

            // Update "Running" status and button
            if (serviceRunning)
            {
                lblRunning.Text = "yes";
                lblRunning.ForeColor =
System.Drawing.SystemColors.ControlText;
                cmdStartStop.Text = "Stop";
            }
            else
            {
                lblRunning.Text = "no";
                lblRunning.ForeColor = Color.Red;
                cmdStartStop.Text = "Start";
            }
            if (serviceRunningTimeout == -1 && serviceInstalled)
                cmdStartStop.Enabled = true;
            else
                cmdStartStop.Enabled = false;


        }

        private void cmdInstallUninstall_Click(object sender, EventArgs
e)
        {
            // Under .NET, a service is installed by creating a
"service installer" class
            // inside the service itself, which specifies things that
should be done when
            // it is installed as a service. To do the installation, we
have to use the
            // System.Configuration.Install API to access this class
inside syslogger.exe
            // and tell it to install itself
            System.Configuration.Install.AssemblyInstaller
serviceInstaller;
            try
            {
                serviceInstaller = new
System.Configuration.Install.AssemblyInstaller();
                serviceInstaller.Path = "syslogger.exe";
                serviceInstaller.UseNewContext = true;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Unable to access service installer: "
+ ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
```

```
            // The API requires us to call .Install and .Commit
methods, passing this
            // saved state thing to both functions
            System.Collections.IDictionary savedState = new
System.Collections.Hashtable();

            if (!serviceInstalled)
            {
                try
                {
                    serviceInstalledTimeout = 10;   // number of ticks
of tmrUpdateStatus to wait before timing out,
                                                    // at time of
writing, 1 tick = 500 ms = 0.5 s
                    cmdInstallUninstall.Enabled = false;
                    savedState.Clear();
                    serviceInstaller.Install(savedState);
                    serviceInstaller.Commit(savedState);
                    UpdateStatus();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Could not install service: " +
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceInstalledTimeout = -1; // Put these values
back to normal, install cancelled
                    cmdInstallUninstall.Enabled = true;
                    UpdateStatus();
                }
            }
            else
            {
                try
                {
                    serviceInstalledTimeout = 10;   // number of ticks
of tmrUpdateStatus to wait before timing out,
                    // at time of writing, 1 tick = 500 ms = 0.5 s
                    cmdInstallUninstall.Enabled = false;
                    savedState.Clear();
                    serviceInstaller.Uninstall(savedState);
                    UpdateStatus();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Could not uninstall service: " +
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceInstalledTimeout = -1; // Put these values
back to normal, uninstall cancelled
                    cmdInstallUninstall.Enabled = true;
                    UpdateStatus();
                }
            }

        }

        private void cmdStartStop_Click(object sender, EventArgs ev)
```

```
        {
            if (serviceInstalled && serviceRunning)
            {   // Stop the service
                try
                {
                    serviceRunningTimeout = 10; // number of ticks of
tmrUpdateStatus to wait before timing out,
                                                // at time of writing,
1 tick = 500 ms = 0.5 s
                    cmdStartStop.Enabled = false;
                    serviceController.Stop();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Could not stop service: " +
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceRunningTimeout = -1; // Put these values
back to normal, stop cancelled
                    cmdStartStop.Enabled = true;
                }
                UpdateStatus();
            }
            else if (serviceInstalled && !serviceRunning)
            {   // Start the service
                try
                {
                    serviceRunningTimeout = 10; // number of ticks of
tmrUpdateStatus to wait before timing out,
                                                // at time of writing,
1 tick = 500 ms = 0.5 s
                    cmdStartStop.Enabled = false;
                    serviceController.Start();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Could not start service: " +
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    serviceRunningTimeout = -1; // Put these values
back to normal, start cancelled
                    cmdStartStop.Enabled = true;
                }
                UpdateStatus();
            }
        }

        private void cmdOK_Click(object sender, EventArgs e)
        {
            try
            {

settingsSection.Settings.Get("Hostname").Value.ValueXml.InnerText =
txtSyslogServer.Text;
                serviceConfig.Save(ConfigurationSaveMode.Minimal,
true);
            }
            catch (Exception ex)
            {
```

```
                MessageBox.Show("Unable to save configuration file for
syslogger.exe: " + ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }

            // This tells the service to reload it's config file. The
service doesn't actually use the
            // command number itself, all numbers tell it to reload the
config, however windows requires that
            // it be a number between 128 and 256
            if (serviceRunning)
            {
                try
                {
                    serviceController.ExecuteCommand(200);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Unable to tell service to refresh
it's config file: " + ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                }
            }


            this.Close();
        }

        private void cmdCancel_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void cmdApply_Click(object sender, EventArgs e)
        {
            try
            {

settingsSection.Settings.Get("Hostname").Value.ValueXml.InnerText =
txtSyslogServer.Text;
                serviceConfig.Save(ConfigurationSaveMode.Minimal,
true);
                cmdApply.Enabled = false;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Unable to save configuration file for
syslogger.exe: " + ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }

            // This tells the service to reload it's config file. The
service doesn't actually use the
            // command number itself, all numbers tell it to reload the
config, however windows requires that
            // it be a number between 128 and 256
            if (serviceRunning)
```

```
            {
                try
                {
                    serviceController.ExecuteCommand(200);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Unable to tell service to refresh
it's config file: " + ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                }
            }
        }

        private void txtSyslogServer_TextChanged(object sender,
EventArgs e)
        {
            cmdApply.Enabled = true;
        }


    }
}
```

File: Configuration\frmConfig.Designer.cs

```
namespace Syslogger
{
    partial class frmConfig
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
```

```
        {
            this.components = new System.ComponentModel.Container();
            System.Windows.Forms.GroupBox groupBox1;
            System.Windows.Forms.Label label2;
            System.Windows.Forms.Label label1;
            System.Windows.Forms.GroupBox groupBox2;
            System.Windows.Forms.Label label5;
            this.cmdStartStop = new System.Windows.Forms.Button();
            this.cmdInstallUninstall = new
System.Windows.Forms.Button();
            this.lblRunning = new System.Windows.Forms.Label();
            this.lblInstalled = new System.Windows.Forms.Label();
            this.chkProcesses = new System.Windows.Forms.CheckBox();
            this.txtSyslogServer = new System.Windows.Forms.TextBox();
            this.cmdCancel = new System.Windows.Forms.Button();
            this.cmdOK = new System.Windows.Forms.Button();
            this.tmrUpdateStatus = new
System.Windows.Forms.Timer(this.components);
            this.cmdApply = new System.Windows.Forms.Button();
            groupBox1 = new System.Windows.Forms.GroupBox();
            label2 = new System.Windows.Forms.Label();
            label1 = new System.Windows.Forms.Label();
            groupBox2 = new System.Windows.Forms.GroupBox();
            label5 = new System.Windows.Forms.Label();
            groupBox1.SuspendLayout();
            groupBox2.SuspendLayout();
            this.SuspendLayout();
            //
            // groupBox1
            //
            groupBox1.Controls.Add(this.cmdStartStop);
            groupBox1.Controls.Add(this.cmdInstallUninstall);
            groupBox1.Controls.Add(this.lblRunning);
            groupBox1.Controls.Add(this.lblInstalled);
            groupBox1.Controls.Add(label2);
            groupBox1.Controls.Add(label1);
            groupBox1.Location = new System.Drawing.Point(12, 12);
            groupBox1.Name = "groupBox1";
            groupBox1.Size = new System.Drawing.Size(335, 104);
            groupBox1.TabIndex = 0;
            groupBox1.TabStop = false;
            groupBox1.Text = "Status";
            //
            // cmdStartStop
            //
            this.cmdStartStop.Location = new System.Drawing.Point(213,
54);
            this.cmdStartStop.Name = "cmdStartStop";
            this.cmdStartStop.Size = new System.Drawing.Size(97, 29);
            this.cmdStartStop.TabIndex = 7;
            this.cmdStartStop.Text = "Stop";
            this.cmdStartStop.UseVisualStyleBackColor = true;
            this.cmdStartStop.Click += new
System.EventHandler(this.cmdStartStop_Click);
            //
            // cmdInstallUninstall
            //
```

```
            this.cmdInstallUninstall.Location = new
System.Drawing.Point(213, 19);
            this.cmdInstallUninstall.Name = "cmdInstallUninstall";
            this.cmdInstallUninstall.Size = new System.Drawing.Size(97,
29);
            this.cmdInstallUninstall.TabIndex = 6;
            this.cmdInstallUninstall.Text = "Uninstall";
            this.cmdInstallUninstall.UseVisualStyleBackColor = true;
            this.cmdInstallUninstall.Click += new
System.EventHandler(this.cmdInstallUninstall_Click);
            //
            // lblRunning
            //
            this.lblRunning.AutoSize = true;
            this.lblRunning.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblRunning.Location = new System.Drawing.Point(137,
62);
            this.lblRunning.Name = "lblRunning";
            this.lblRunning.Size = new System.Drawing.Size(26, 13);
            this.lblRunning.TabIndex = 3;
            this.lblRunning.Text = "yes";
            //
            // lblInstalled
            //
            this.lblInstalled.AutoSize = true;
            this.lblInstalled.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblInstalled.Location = new System.Drawing.Point(137,
27);
            this.lblInstalled.Name = "lblInstalled";
            this.lblInstalled.Size = new System.Drawing.Size(26, 13);
            this.lblInstalled.TabIndex = 2;
            this.lblInstalled.Text = "yes";
            //
            // label2
            //
            label2.AutoSize = true;
            label2.Location = new System.Drawing.Point(43, 62);
            label2.Name = "label2";
            label2.Size = new System.Drawing.Size(50, 13);
            label2.TabIndex = 1;
            label2.Text = "Running:";
            //
            // label1
            //
            label1.AutoSize = true;
            label1.Location = new System.Drawing.Point(43, 27);
            label1.Name = "label1";
            label1.Size = new System.Drawing.Size(49, 13);
            label1.TabIndex = 0;
            label1.Text = "Installed:";
            //
            // groupBox2
            //
```

```
            groupBox2.Controls.Add(this.chkProcesses);
            groupBox2.Controls.Add(this.txtSyslogServer);
            groupBox2.Controls.Add(label5);
            groupBox2.Location = new System.Drawing.Point(12, 122);
            groupBox2.Name = "groupBox2";
            groupBox2.Size = new System.Drawing.Size(335, 83);
            groupBox2.TabIndex = 1;
            groupBox2.TabStop = false;
            groupBox2.Text = "Settings";
            //
            // chkProcesses
            //
            this.chkProcesses.AutoSize = true;
            this.chkProcesses.Location = new System.Drawing.Point(205,
60);
            this.chkProcesses.Name = "chkProcesses";
            this.chkProcesses.Size = new System.Drawing.Size(105, 17);
            this.chkProcesses.TabIndex = 2;
            this.chkProcesses.Text = "Track processes";
            this.chkProcesses.UseVisualStyleBackColor = true;
            this.chkProcesses.Visible = false;
            //
            // txtSyslogServer
            //
            this.txtSyslogServer.Location = new
System.Drawing.Point(135, 29);
            this.txtSyslogServer.Name = "txtSyslogServer";
            this.txtSyslogServer.Size = new System.Drawing.Size(175,
20);
            this.txtSyslogServer.TabIndex = 1;
            this.txtSyslogServer.TextChanged += new
System.EventHandler(this.txtSyslogServer_TextChanged);
            //
            // label5
            //
            label5.AutoSize = true;
            label5.Location = new System.Drawing.Point(43, 32);
            label5.Name = "label5";
            label5.Size = new System.Drawing.Size(73, 13);
            label5.TabIndex = 0;
            label5.Text = "Syslog server:";
            //
            // cmdCancel
            //
            this.cmdCancel.DialogResult =
System.Windows.Forms.DialogResult.Cancel;
            this.cmdCancel.Location = new System.Drawing.Point(189,
211);
            this.cmdCancel.Name = "cmdCancel";
            this.cmdCancel.Size = new System.Drawing.Size(76, 25);
            this.cmdCancel.TabIndex = 4;
            this.cmdCancel.Text = "Cancel";
            this.cmdCancel.UseVisualStyleBackColor = true;
            this.cmdCancel.Click += new
System.EventHandler(this.cmdCancel_Click);
            //
            // cmdOK
```

```
            //
            this.cmdOK.Location = new System.Drawing.Point(107, 211);
            this.cmdOK.Name = "cmdOK";
            this.cmdOK.Size = new System.Drawing.Size(76, 25);
            this.cmdOK.TabIndex = 3;
            this.cmdOK.Text = "OK";
            this.cmdOK.UseVisualStyleBackColor = true;
            this.cmdOK.Click += new
System.EventHandler(this.cmdOK_Click);
            //
            // tmrUpdateStatus
            //
            this.tmrUpdateStatus.Enabled = true;
            this.tmrUpdateStatus.Interval = 500;
            this.tmrUpdateStatus.Tick += new
System.EventHandler(this.tmrUpdateStatus_Tick);
            //
            // cmdApply
            //
            this.cmdApply.Enabled = false;
            this.cmdApply.Location = new System.Drawing.Point(271,
211);
            this.cmdApply.Name = "cmdApply";
            this.cmdApply.Size = new System.Drawing.Size(76, 25);
            this.cmdApply.TabIndex = 5;
            this.cmdApply.Text = "Apply";
            this.cmdApply.UseVisualStyleBackColor = true;
            this.cmdApply.Click += new
System.EventHandler(this.cmdApply_Click);
            //
            // frmConfig
            //
            this.AcceptButton = this.cmdOK;
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F,
13F);
            this.AutoScaleMode =
System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(359, 248);
            this.Controls.Add(this.cmdApply);
            this.Controls.Add(this.cmdOK);
            this.Controls.Add(this.cmdCancel);
            this.Controls.Add(groupBox2);
            this.Controls.Add(groupBox1);
            this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedSingle;
            this.MaximizeBox = false;
            this.Name = "frmConfig";
            this.Text = "Syslogger Configuration";
            this.Load += new System.EventHandler(this.frmConfig_Load);
            groupBox1.ResumeLayout(false);
            groupBox1.PerformLayout();
            groupBox2.ResumeLayout(false);
            groupBox2.PerformLayout();
            this.ResumeLayout(false);

        }
```

```
        #endregion

        private System.Windows.Forms.Button cmdStartStop;
        private System.Windows.Forms.Button cmdInstallUninstall;
        private System.Windows.Forms.Label lblRunning;
        private System.Windows.Forms.Label lblInstalled;
        private System.Windows.Forms.Button cmdCancel;
        private System.Windows.Forms.TextBox txtSyslogServer;
        private System.Windows.Forms.Button cmdOK;
        private System.Windows.Forms.CheckBox chkProcesses;
        private System.Windows.Forms.Timer tmrUpdateStatus;
        private System.Windows.Forms.Button cmdApply;
    }
}
```

File: Configuration\Program.cs

```
using System;
//using System.Collections.Generic;
using System.Windows.Forms;

namespace Syslogger
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new frmConfig());
        }
    }
}
```