# TUC Transfer Protocol Daemon
# &
# TUC Alert Daemon:

## System Documentation

Martin Thorsen Ranang

October 17, 2011

# Contents

# Abbreviations

| | |
|---|---|
| API | application program interface |
| CPU | central processing unit |
| I/O | input/output |
| SMS | Short Message Service |
| TAD | TUC Alert Daemon |
| TTPD | TUC Transfer Protocol Daemon |
| TUC | The Understanding Computer |
| XML | Extensible Markup Language |

# Abstract

This document provides an overview of both the TUC Transfer Protocol Daemon and the TUC Alert Daemon software. The software package contains two programs, named `ttpd` and `ttpc`. These programs use a common application program interface, which is also documented herein.

# Chapter 1

# Overview

The main purpose of TUC Transfer Protocol Daemon *(TTPD)* is to work as a mediator for requests between external programs and The Understanding Computer *(TUC)*. In addition, if used with the TUC Alert Daemon *(TAD)* it will also handle 'alert'[1] requests when used in conjunction with the Short Message Service *(SMS)* interface. Figure 1.1 shows an overview of the system architecture.

Both programs were implemented in Python version 2.3.4.

## 1.1 TUC Transfer Protocol Daemon

The TTPD is implemented as a *threading* server, listening for connections from *clients* on a user-specified network port (by using a *socket*). When a request is received, the server starts a *handler thread* responsible for handling the request. First, the nature of the request is determined. If it is an 'alert cancellation' request, the request is handled by the handler thread through communication with TAD and the *client* (see Section 1.2 for more information about this behavior). However, if the request is not of that nature, it is forwarded to TUC for processing. This is done by having the server act as a *producer thread* that places incoming *tasks* in a *thread pool* with $n \in \langle 0, k]$ *consumer threads*. Each of the consumer threads controls *one* external TUC process each.

The external TUC processes are started by the server before it starts accepting connections. This is done because it would take too long if an external TUC process should be started in order to handle each request. In relation

[1] That is, messages like "Kan du varsle meg 15 minutter før bussen fra Nardo til Gløshaugen går?"

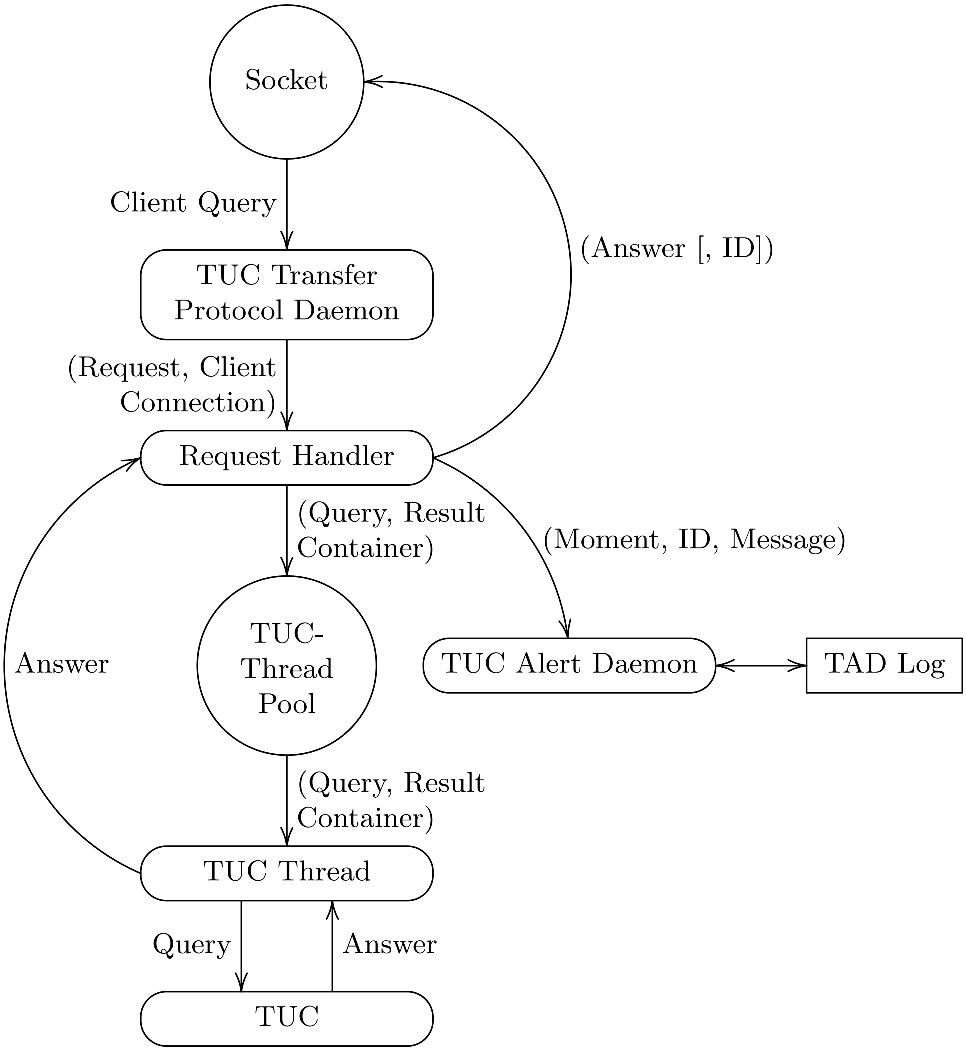


Figure 1.1: Overview of the TTPD and TAD system.

to the main server process, the TUC are *forked* processes. The communication between each *consumer* thread and its TUC process is done through *pipes* that function as the TUC process' `stdin` and `stdout` file streams.

When the result from TUC has been received, the *consumer thread* stores it in a *thread-safe* container that only the original *handler thread* and the consumer thread shares. Hence, the response for handling the request is again returned to the handler thread. Now, the result is parsed, the appropriate information is logged and an answer is given to the client.

## 1.2 TUC Alert Daemon

The TAD is built-in as a part of TTPD, but has its own responsibilities.

The most central component of TAD is a *scheduler* responsible for the timing of sending out alert-messages at the moments specified by users.

TAD consists mainly of two threads that for the most of the time run independently of the TTPD process. One thread constitutes an *alert scheduler* while the other thread handles requests of adding and removing alerts and is responsible for communicating with the *database engine*. The communication between a *TTPD request handler* and the *TAD request handler* is done through a shared request pool (thread-safely protected by *lock*).

# Chapter 2

# Features

This chapter presents the design criteria and features that characterizes TUC Transfer Protocol Daemon *(TTPD)* and TUC Alert Daemon *(TAD)*.

## 2.1 Design Criteria

The main goals of this work was to make the system more robust and to implement an alert service. In addition to this, where there have been several possible solutions to a problem, the guiding principles has been as follows: choose solutions that are *easily maintainable* (use standard modules and programs if available, and write easily readable code and documentation), *highly scalable* and that will *probably not need to be changed* in the near future. In addition, the system has been designed in the spirit of *Unix* philosophy; embracing modularity, the use of stream redirection and process control as described in (Ranang 2003). For a very concise introduction to the theory and pragmatics of server/daemon design, please see (Griffin and Nelson 1998a; Griffin and Nelson 1998b; Griffin, Donnelly, and Nelson 1998).

## 2.2 Scalability and Robustness

First of all, the *scalability* of the system has been improved in several ways:

- The design of TTPD allows a computer to serve multiple The Understanding Computer *(TUC)* requests in parallel. This is ensured through the use of the pool of *TUC-threads* where each thread controls an externally running TUC process. This means that if more central processing unit *(CPU)* are added to the computer, each processes can run on its own CPU. Even without multiple CPU, the computer can serve multiple requests in multiplexed parallel (but possibly with a longer delay than when serving single requests, depending both on CPU- and input/output *(I/O)* intensity).

- By combining the use of *socket* with *tread* (could have used *forked processes* but they take longer to create) the *daemon* is able to stack up incoming requests and serve them as soon as possible. This feature ensures that the clients can deliver their requests even when other requests are already being processed.

Secondly, the following features have been implemented to ensure a high level of *robustness* of the system:

- The use of socket ensure that *if* the daemon process should crash, any transaction received but not finished will silently be lost. That is, the log will contain information about the reception of the requests, but when the daemon is restarted, it will not result in a bombardment of the users with their old requests. An exception from this is the handling of any due TAD alerts. They will be sent out as soon as possible after the alert moment have passed.
- Efforts have been made to graciously handle extremely high workloads. The daemon has a configurable "soft" high-load limit. This results in a predefined message being sent back to users—when the current number of concurrently handled requests is above this limit—telling the user that due to very high demand, the request cannot be handled at this moment.
- The daemon tries to cope with non-perfect conditions in a controlled way:
  - If one of the encapsulated TUC processes die unexpectedly, the daemon will start a replacement process immediately (this constitutes a built-in *watch dog* feature).
  - All alerts that are registered by the TAD is immediately saved to disk. This is done to ensure that during startup, the daemon consults the old log file to restore it state.
  - Because the *protocol* for *communication* with the Short Message Service *(SMS) message switch* defines the communication as asynchronous, it is possible that the daemon experiences situations when it cannot connect to the remote server to deliver its answer. If such a situation should arise—and it has, during intensive testing of the system—a *retry/resend algorithm* has been implemented. The *handler* of that request will sleep for a random number of seconds (within a predefined interval) and

then try to resend the answer when it awakens. If it still does not succeed, it repeats this behavior until a total time limit has been reached (suggestively sat to 15 minutes).

- Since the *SMS message switch communication protocol* uses Extensible Markup Language *(XML)*-based headers, the daemon parses these with the default *Python SAX-parser* (usually an *expat parser*). This ensures a robust handling of incoming messages and an automatic check for XML compliance.
- The TTPD is designed to handle random connections to its socket, so that non-protocol clients will not crash it.

# Chapter 3

# Program Usage

The programs `ttpd` (the *daemon*) and `ttpc` (the client interface application) both contain some built-in help information that is always available from the command line. All you need to do is to supply one of the flags `-help` or `-h` to the program on the command line.

## 3.1 TUC Transfer Protocol Daemon

The help menu in `ttpd` looks like:

## 3.2 TUC Transfer Protocol Daemon Controller

The help menu in `ttpdctl` looks like:

```
Usage: ttpdctl [options] start|stop|restart|store_alerts

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -c FILENAME, --config-file=FILENAME
                        the FILENAME containing the default configuration; the
                        default is '/etc/default/ttpd'
  -a ADDRESS, --ip-address=ADDRESS
                        the ADDRESS of the interface on which to listen for
                        connections from clients; if ADDRESS is 0 (the
                        default) the server will listen on all available
                        network interfaces
  -p PORT, --port=PORT  listen for connections from clients on PORT; the
```

```
                        default is 2004
  -A ADDRESS, --remote-ip-address=ADDRESS
                        the ADDRESS of the interface on which to send outgoi
                        communication; default is 'localhost'
  -P PORT, --remote-port=PORT
                        the remote PORT; the default is 2005
  --service-name=SMS_SERVICE_NAME
                        the SERVICE_NAME for the SMS service.  The default i
                        'RUTE'.
  -q SIZE, --socket-queue-size=SIZE
                        the SIZE of the socket queue; the default is 5
  -Q THREADS, --high-load-limit=THREADS
                        maximum number of concurrently running THREADS; the
                        default is 50
  -L LEVEL, --log-level=LEVEL
                        the filter LEVEL used when logging; possible values
                        are 'notset' < 'protocol' < 'debug' < 'info' <
                        'warning' < 'error' < 'critical'; the default is
                        'info'
  -f FILE, --log-file=FILE
                        store the log in FILE; the default is 'ttpdctl.log'
  -t, --without-tad     don't start the TUC Alert Daemon (TAD)
  -T, --without-tuc     don't start the TUC processes
  -n NUMBER, --tuc-threads=NUMBER
                        start NUMBER concurrent TUC processes; the default i
                        3
  -s COMMAND, --path-to-tuc=COMMAND
                        run COMMAND as TUC subprocess; the default is
                        ./busestuc.sav
  -E ENCODING, --tuc-external-encoding=ENCODING
                        tell SICStus Prolog that it runs in an ENCODING
                        environment.  If "None", TTPD tries to set it
                        according to its stdin encoding.  The default is Non
  -d, --no-daemon       don't run as a daemon process
  --pid-file=FILENAME   store the pid of the server process in FILENAME; the
                        default is /var/run/ttpd/ttpd.pid
  -g PROVIDER, --sms-gateway=PROVIDER
                        the PROVIDER to use when sending SMS; possible value
                        are 'esolutions', 'payex' and 'pswincom'.  The defau
                        is 'payex'.
  -U USERNAME, --user=USERNAME
                        the USERNAME that the daemon process will run as.  T
                        default is 'ttpd'.
  -G GROUPNAME, --group=GROUPNAME
                        the GROUPNAME that the daemon process will run as.
                        The default is 'ttpd'.
  --originating-address=PHONE_NUMBER
                        the PHONE_NUMBER of the SMS service.  The default is
                        '1939'.
```

```
  --payex-test          use the PayEx test, instead of the production,
                        interface.  The default is 'False'.
  --payex-trace=FILENAME
                        store PayEx communication traces in FILENAME.  If
                        option is not used, no trace will be performed.
  --payex-key=ENCRYPTION_KEY
                        the ENCRYPTION_KEY to use when sending SMS via PayEx
                        interface.  The default is 'PgHzip4b2RH8u43XSE6V'.
  --payex-account=ACCOUNT_NUMBER
                        the ACCOUNT_NUMBER to use when sending SMS via PayEx
                        interface.  The default is '21217859'.
  --payex-test-key=ENCRYPTION_KEY
                        the ENCRYPTION_KEY to use for sending SMS via PayEx
                        test interface.  The default is
                        '2PuM2YTbK3VfUypbN2bU'.
  --payex-test-account=ACCOUNT_NUMBER
                        the ACCOUNT_NUMBER to use when sending SMS via PayEx
                        test interface.  The default is '50017893'.
  -o FILE, --old-log-file=FILE
                        location of the old log FILE, used to retrieve the PID
                        of the current process, when restarting after a file
                        has been moved (e.g., log rotated)
  -e EXECUTABLE, --executable=EXECUTABLE
                        location of the ttpd EXECUTABLE; the default is ttpd
```

## 3.3 TUC Transfer Protocol Client Application

The help menu in `ttpc` looks like:

```
Usage: ttpc [options] [request]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -c FILENAME, --config-file=FILENAME
                        the FILENAME containing the default configuration; the
                        default is '/etc/default/ttpd'
  -a ADDRESS, --ip-address=ADDRESS
                        the ADDRESS of the interface on which to listen for
                        connections from clients; if ADDRESS is 0 (the
                        default) the server will listen on all available
                        network interfaces
  -p PORT, --port=PORT  listen for connections from clients on PORT; the
                        default is 2004
  -A ADDRESS, --remote-ip-address=ADDRESS
                        the ADDRESS of the interface on which to send outgoing
                        communication; default is 'localhost'
  -P PORT, --remote-port=PORT
```

```
                      the remote PORT; the default is 2005
--service-name=SMS_SERVICE_NAME
                      the SERVICE_NAME for the SMS service.  The default i
                      'RUTE'.
-q SIZE, --socket-queue-size=SIZE
                      the SIZE of the socket queue; the default is 5
-Q THREADS, --high-load-limit=THREADS
                      maximum number of concurrently running THREADS; the
                      default is 50
-L LEVEL, --log-level=LEVEL
                      the filter LEVEL used when logging; possible values
                      are 'notset' < 'protocol' < 'debug' < 'info' <
                      'warning' < 'error' < 'critical'; the default is
                      'info'
-f FILE, --log-file=FILE
                      store the log in FILE; the default is 'ttpc.log'
-I TRANS_ID, --transaction-id=TRANS_ID
                      the TRANS_ID to use when communicating with a remote
                      server.  The default is 'LINGSMSIN'.
-l, --listen-mode     listen for inbound connections
-t, --test            run in looping test-mode
-T PHONE, --phone-number=PHONE
                      the number that the message supposedly was sent from
                      the deault is 'None'
-F, --fake-outgoing   fake an outgoing message from the server
-i FILE, --input-file=FILE
                      read input from FILE
-w, --web             generate output suitable for the Web; this implicate
                      --transaction-id=WEB
-j, --json            generate JSON output suitable for Web Services; this
                      implicates --transaction-id=JSON
-W, --show-technical  show technical information (semantics)
```

# Appendix A

# Module Application Program Interface

It should be noted that the SocketServer module was not developed by the author, but since the modules TTP.Server and TTP.Handler contain classes that inherit from classes in SocketServer, it has been included here to make the application program interface *(API)* documentation meaningful.

# Bibliography

[Griffin, Donnelly, and Nelson1998] Griffin, Ivan, Mark Donnelly, and John Nelson.
1998.
"Linux Network Programming, Part 3."
*Linux J.* 1998 (48es): 7 (April).
[Griffin and Nelson1998a] Griffin, Ivan, and John Nelson.
1998a.
"Linux Network Programming, Part 1."
*Linux J.* 1998 (46es): 5 (February).
[Griffin and Nelson1998b] ———.
1998b.
"Linux Network Programming, Part 2."
*Linux J.* 1998 (47es): 6 (March).
[Ranang2003] Ranang, Martin Thorsen.
2003.
"The Fragrance of Unix."
Compendium.
Version 1.1, available on-line.