TheSquid

P. Baillehache

September 9, 2019

Contents

Introduction

TheSquid is a C library providing structures and functions to perform cluster computing.

Cluster computing consists of performing the computation of a task by dividing it into several subtasks, ran in parallel on several physical devices and/or independant processes. A process, called the Squad, running on one device request the execution of tasks to one or more processes, called the Squidlets, on the same or other devices. TheSquid is the combination of a Squad and one or more Squidlets. TheSquid library takes care of dividing the main task into subtasks and managing the computation of these subtasks (communication with the Squidlets and processing of the results of subtasks).

Available tasks to be computed by TheSquid are:

- Dummy: a task to perform test
- Benchmark: a task to benchmark the performance of TheSquid
- PovRay: a task to render computer graphic image using POV-Ray
- ResetStats: a task to reset Squidlets' internal statistics

The library can be extended to other tasks.

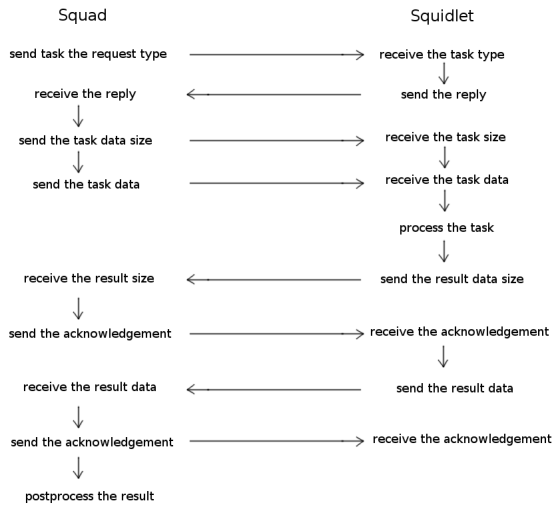
TheSquid provides two executable files (`squad` and `squidlet`). They can be used as stand alone to perform cluster computing, the detail of executed tasks and cluster configuration being given through configuration files in JSON format. The library can also be integrated into another application.

TheSquid has been tested on a cluster of Raspberry Pi, however the cluster may be any set of heteregenous devices able to comunicate between each other through TCP/IP protocol. This document describes in detail the implementation on a cluster of Raspberry Pi only.

It uses the `PBErr`, `PBMath`, `GSet`, `ResPublish`, `PBJson`, `PBCExtension`, `GenBrush` and `PBFileSys` libraries.

1 Protocol

The communication between a Squad and a Squidlet for one given task is as describe below. Communication is made via TCP/IP sockets. The listening socket of a Squildet is known to the Squad thanks to its configuration file which contains the IP and port of this socket. The Squidlet creates a socket to reply to the Squad upon reception of a task. The Squidlet gives up the processing of a task when an expected reply from the Squad is not received in a certain limit of time. The Squad gives up the processing of a task when an expected reply from the Squidlet is not received, and request again the execution of the failed task as soon as a Squidlet, eventually the same which failed to process the task, becomes available.



2 Tasks

2.1 File format

When a task is saved into a text file with JSON format the following properties must be specified:
`{"SquidletTaskType":"1", "id":"1", "maxWait":"1"}`
where

- "SquidletTaskType" is the type of task (see below)

- "id" is the id of the task
- "maxWait" is the number of seconds the Sqaud will wait for the result from the Squidlet before giving up and trying again the task on another Squidlet

In addition, the data sent by the Squad to the Squidlet as described below must be added.

2.2 Dummy

Type: 1

Data for the task request from the Squad to the Squidlet:
`{"v":"1"}`
The value of "v" is actually copied from the "id" and needs not to be specified.

Data added to the JSON structure of a task saved in a text file: same as above.

Task action: waits for "v" seconds.

Data of the result of the task request from the Squidlet to the Squad, if successful:
`{"success":"1","temperature":"0.0","v":"-1"}`
where

- "success" is the success flag
- "temperature" is the temperature of the device of the Squidlet if available
- "v" is equal to the opposite of the "v" in the task request

If failed:
`{"success":"0","temperature":"0.0","err":"Invalid input"}`
where

- "success" is the failure flag
- "temperature" is the temperature of the device of the Squidlet if available
- "err" is the error message

2.3 Benchmark

Type: 2

Data for the task request from the Squad to the Squidlet:
`{"nb":"1", "payloadSize":" "}`
where

- "nb" is the number of sort
- "payloadSize" is a string whose content is ignored but size is used to set the quantity of data sent over the network

Data added to the JSON structure of a task saved in a text file : same as above.

Task action: sorts "nb" times a set of GSet of 10*length(payloadSize).

Data of the result of the task request from the Squidlet to the Squad, if successful:
`{"success":"1","temperature":"0.0","v":"1"}`
where

- "success" is the success flag
- "temperature" is the temperature of the device of the Squidlet if available
- "v" is equal to the sorting value of the first element of the sorted GSet

If failed:
`{"success":"0","temperature":"0.0","err":"Invalid input"}`
where

- "success" is the failure flag
- "temperature" is the temperature of the device of the Squidlet if available
- "err" is the error message

2.4 PovRay

Type: 3

Data for the task request from the Squad to the Squidlet:
{"subid":"1", "ini":"./testPov.ini",
"tga":"./img.tga", "top":"0", "left":"0", "bottom":"99",
"right":"99","width:"100","height":"100","outTga":"./img-00001.tga"}
where

- "ini" is the path to the POV-Ray ini file describing the scene to render. It must contains at least the following parameters: Width, Height, Output_File_Name
- "top", "left", "bottom", "right" are the coordinates of the fragment inside the final image to be rendered by this task
- "width" and "height" are the dimension of the fragment
- "subid" is the id of the subtask corresponding to this fragment
- "tga" is the path of the final image
- "outTga" is the path of the fragment

Data added to the JSON structure of a task saved in a text file:
{"ini":"./testPov.ini", "sizeMinFragment":"100", "sizeMaxFragment":"1000"}
where

- "ini" is the path to the POV-Ray ini file describing the scene to render. It must contain at least the following parameters: Width, Height, Output_File_Name
- "sizeMinFragment" is the minimum size (width and height) of one square fragment corresponding to a given subtask
- "sizeMaxFragment" is the maximum size (width and height) of one square fragment corresponding to a given subtask

Task action: Render the image according to the ini file and save the result into a tga image at Output_File_Name. Fragments' position (top,left)-(bottom,right) are automatically calculated by dividing the width and height of the final image by the number of available Squidlets at the time of task request. If the result of division is not an integer the size of the last fragment of each line/column is corrected as necessary. The Squad postprocess the

fragments processed by the Squidlets to recompose the final image.

Data of the result of the task request from the Squidlet to the Squad, if successful:
`{"success":"1","temperature":"0.0"}`
where

- "success" is the success flag
- "temperature" is the temperature of the device of the Squidlet if available

If failed:
`{"success":"0","temperature":"0.0","err":"Invalid input"}`
where

- "success" is the failure flag
- "temperature" is the temperature of the device of the Squidlet if available
- "err" is the error message

2.5 ResetStats

Type: 4

This task is a special task used by the function `SquadRequestSquidletToResetStats`.

2.6 Statistics data in task result

Statistics about processing of task are automatically added to the data of the result of the task:
`"nbAcceptedConnection":"1","nbAcceptedTask":"1","nbRefusedTask":"0",`
`"nbFailedReceptTaskSize":"0","nbFailedReceptTaskData":"0","nbSentResult":"0",`
`"nbFailedSendResult":"0","nbFailedSendResultSize":"0","nbFailedReceptAck":"0",`
`"nbTaskComplete":"1","timeToProcessMs":"0","timeWaitedTaskMs":"0",`
`"timeWaitedAckMs":"0","timeTransferSquidSquadMs":"0.000"`
where

- "nbAcceptedConnection" is the number of accepted connection by the Squidlet
- "nbAcceptedTask" is the number of accepted task requests

- "nbRefusedTask" is the number of refused task requests
- "nbFailedReceptTaskSize" is the number of failure to receive the size of the data of the task
- "nbFailedReceptTaskData" is the number of failure to receive the data of the task
- "nbSentResult" is the number of sent result of processing
- "nbFailedSendResult" is the number of failure to send the result of processing
- "nbFailedSendResultSize" is the number of failure to send the size of the result of processing
- "nbFailedReceptAck" is the number of failure to receive the acknowledgement from the Squad
- "nbTaskComplete" is the number of successfully processed task
- "timeToProcessMs" is the average of the total time in millisecond used to process the last `SQUID_RANGEAVGSTAT` tasks
- "timeWaitedTaskMs" is the average of the time in millisecond between two successive task processing over the last `SQUID_RANGEAVGSTAT` tasks
- "timeWaitedAckMs" is the average of the time in millisecond waiting for acknowledgement from the Squad during the last `SQUID_RANGEAVGSTAT` tasks
- "timeTransferSquidSquadMs" is the average of the time per byte in millisecond to transfer data from the Squid to the Squad during the last `SQUID_RANGEAVGSTAT` tasks

3 Setup of the cluster

This section introduces how to setup and configure a cluster on which to use TheSquid. It is important to remind that TheSquid doesn't necessarily need a physical cluster of devices. One physical device may be used to run all the Squad and Squidlets.

3.1 Raspberry Pi 3B+

If the OS is not yet installed on the Pi:

1. On a PC, download the Raspbian Stretch Lite image (1.9Gb) from `https://www.raspberrypi.org/downloads/raspbian/`
2. If not already installed, install Etcher, cf `https://www.balena.io/etcher/`
3. Launch Etcher
4. Plug the 8Gb microSD card (should be class 10 or higher for better results)
5. Select the downloaded image
6. Select the microSD card
7. Flash!
8. Create an empy file named `ssh` on the boot drive of the miccroSD card
9. Insert the microSD into the Raspberry Pi

Once the OS is intalled on the Pi:

1. Connect the Raspberry Pi to the network (should use a lan cable of class 7 or more for best results)
2. Turn on the Raspberry Pi
3. Get the IP adress of a PC connected to the local network with ifconfig, lets say its `a.b.c.d`
4. Scan the devices on the local network with the command `nmap -sP a.b.c.0/24`
5. Connect to the Raspberry Pi through ssh with the command `ssh pi@ip.addr.goes.here`, default password is raspberry
6. Setup the Raspberry Pi with `sudo raspi-config`
7. Change the password, locale and timezone, expand the file system, and exit

8. Set the hostname with the following commands `sudo hostname Squidlet001` `whatever name you chose`
   `sudo nano /etc/hostname  change the hostname here too`
   `sudo nano /etc/hosts  change "raspberrypi" to "Squidlet001"`
9. Ensure the system time is right with the command `sudo apt install ntpdate -y`
10. Reboot with the command `sudo reboot`
11. Make a directory to clone the git repositories with the command `mkdir /GitHub`
12. Move to the `/GitHub` directory
13. Clone the repository `PBMake` with the command `git clone https://github.com/BayashiPascal/PBMake.git`
14. Edit the root Makefile with the command `nano /GitHub/PBMake/Makefile.inc` and change the value of `ROOT_DIR` with `/home/pi/GitHub/` and the value of `BUILD_ARCH` with `2`
15. Clone the repository `TheSquid` with the command `git clone https://github.com/BayashiPascal/TheSquid.git`
16. Move to the `/GitHub/TheSquid` directory
17. Open the Makefile with the command `nano Makefile` and make sure the `BUILD_MODE` is set to `1`
18. Compile the repository TheSquid with the command `make`, others repository are automatically installed
19. Check everything works fine with `./main`.
20. Install `lsof` with `sudo apt update && sudo apt install lsof`

To exchange data between devices during computation, a common file system will be necessary. Below is given the example of how to mount a NAS station:

1. `sudo mkdir /mnt/NAS`
2. `sudo nano /etc/fstab //<IP_TO_NAS>/TheSquid /mnt/NAS cifs user,uid=1000,rw,suid,credentials=/etc/credentials 0 0`

3. `sudo nano /etc/credentials username=squidlet password=mypassword`
4. `sudo apt-get -y install cifs-utils`

If you wish to use the PovRay task to render computer graphics using TheSquid, you need to install POV-Ray:

1. sudo apt-get -y update
2. sudo apt-get -y install libboost-all-dev
3. sudo apt-get -y install zlib1g-dev
4. sudo apt-get -y install libpng-dev
5. sudo apt-get -y install libjpeg8-dev
6. sudo apt-get -y install libopenexr-dev
7. sudo apt-get -y install libtiff5-dev libtiff5 libjbig-dev
8. sudo apt-get -y install autoconf
9. cd /GitHub
10. git clone https://github.com/POV-Ray/povray.git
11. cd /GitHub/povray/unix/
12. git checkout 3.7-stable
13. ./prebuild.sh
14. cd ../
15. ./configure COMPILED_BY=”TheSquid <email@address>” –with-boost-libdir=/usr/lib/arm-linux-gnueabihf
16. make
17. sudo make install
18. make check

3.2 Batch file

Below are a few batch files useful to operate TheSquid. They should be modified to match the user environment and are given as examples only.

3.2.1 startSquidlets.bat

```
for i in 1 2 3; do ssh pi@1.1.1.$i "cd /home/pi/GitHub/TheSquid;
(./squidlet -ip 1.1.1.$i -port 9000  ./squidlet -ip 1.1.1.$i -port
9001); exit"; done
```

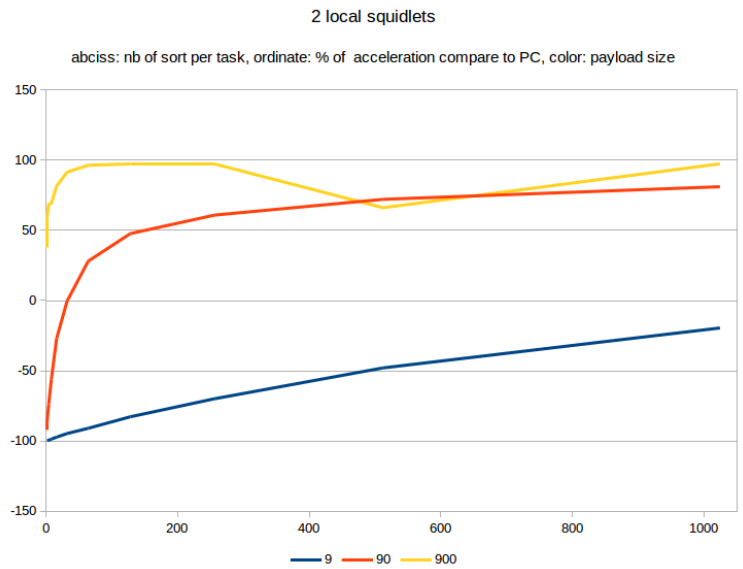
3.2.2 updateSquidlets.bat

```
for i in 1 2 3; do ssh pi@1.1.1.$i "cd GitHub/; rm -rf TheSquid;
git clone https://github.com/BayashiPascal/TheSquid.git; cd TheSquid;
make; exit"; done
```

4 Performance

4.1 PC

Benchmark executed on two Squidlets running on the same PC has the Squad, compared to the benchmark executed on this PC without using TheSquid.



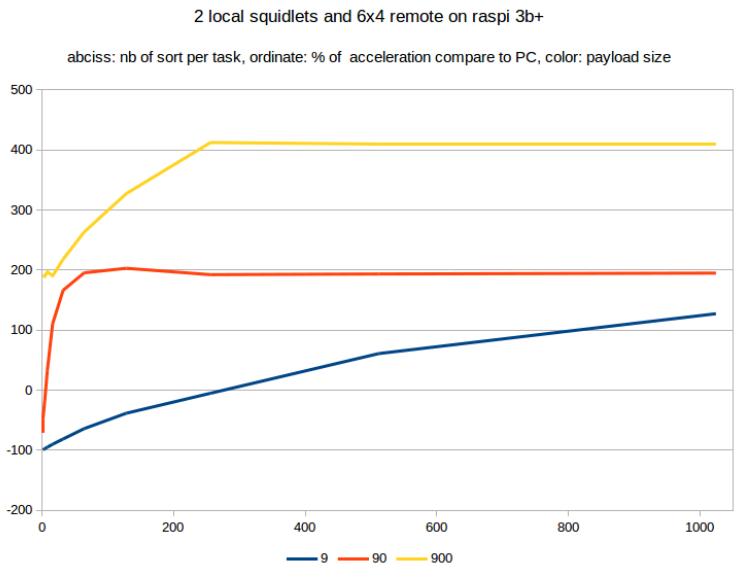
For a payload of 9 bytes, the execution is slower when using TheSquid whatever the number of sorts per task. The time lost during communication

between the Squad and the Squidlets, and the time lost managing the subtasks overcome the gain of running subtasks in parallel.

For a payload of 90 and 900 bytes, the gain of running subtasks in parallel quickly overcomes the lost and the gain converges toward the expected 100% (given that the PC had a quadcore processor, i.e. running two Squidlets locally was equivalent to running simultaneously two benchmarks compare to one for the reference without TheSquid).

4.2 Raspberry Pi 3B+

Benchmark executed on two Squidlets running on the same PC has the Squad plus 24 Squidlets running on 6 Raspbery Pi (4 Squidlets per Pi which are quadcore), compared to the benchmark executed on the PC without using TheSquid.



For a payload of 9 bytes, the execution on TheSquid starts to perform faster from around 300 sorts per tasks. The gain converges toward a little more 100%. For payloads of 90 and 900 bytes, the gain gets positive from, resp., 1 and 10 sorts per tasks, converging toward, resp., 200% and 400%.

Compare to two squidlets on the same PC, using a cluster of Raspberry Pi shows that the lost of performance due to network communication and task management is non negligeable if the tasks are small, but the gain can be important if the tasks get big enough.

The size from which TheSquid is getting advantageous depends on the hardware specification and cannot be foreseen, thus the user is adviced to run the benchmark on its own cluster to appreciate it. The benchmark function could also be used to estimate the gain of new tasks developped in the future.

5 Interface

```
// ============ THESQUID.H ================

#ifndef THESQUID_H
#define THESQUID_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <time.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"
#include "respublish.h"
#include "pbjson.h"
#include "pbcextension.h"
#include "pbfilesys.h"
#include "genbrush.h"
#include "gdataset.h"

// ================= Define ===================

#define THESQUID_NBMAXPENDINGCONN       1
#define THESQUID_PORTMIN                9000
#define THESQUID_PORTMAX                9999
#define THESQUID_TASKREFUSED            0
#define THESQUID_TASKACCEPTED           1
#define THESQUID_ACCEPT_TIMEOUT         1    // in seconds
#define THESQUID_PROC_TIMEOUT           60   // in seconds
#define THESQUID_MAXPAYLOADSIZE         1024 // bytes
#define THESQUID_WAITDATARECEPT_TIMEOUT 5    // in seconds

#define SQUAD_TXTOMETER_LINE1             \
```

```
  "NbRunning xxxxx NbQueued xxxxx NbSquidletAvail xxxxx\n"
#define SQUAD_TXTOMETER_FORMAT1           \
  "NbRunning %05ld NbQueued %05ld NbSquidletAvail %05ld\n"
#define SQUAD_TXTOMETER_FORMATHISTORY     "%s\n"
#define SQUAD_TXTOMETER_TASKHEADER        \
  "------------------    Tasks    ----------------------\n"
#define SQUAD_TXTOMETER_FORMATRUNNING     "Running: %s\n"
#define SQUAD_TXTOMETER_FORMATQUEUED      " Queued: %s\n"
#define SQUAD_TXTOMETER_NBLINEHISTORY     20
#define SQUAD_TXTOMETER_LENGTHLINEHISTORY 100
#define SQUAD_TXTOMETER_NBTASKDISPLAYED   32

// Range for the sliding average when computing stats
#define SQUID_RANGEAVGSTAT 100

// -------------- SquidletInfo

// ================= Data structure ===================

typedef struct SquidletInfoStats {
  unsigned long _nbAcceptedConnection;
  unsigned long _nbAcceptedTask;
  unsigned long _nbRefusedTask;
  unsigned long _nbFailedReceptTaskData;
  unsigned long _nbFailedReceptTaskSize;
  unsigned long _nbSentResult;
  unsigned long _nbFailedSendResult;
  unsigned long _nbFailedSendResultSize;
  unsigned long _nbFailedReceptAck;
  unsigned long _nbTaskComplete;
  float _timeToProcessMs[3];
  float _timeWaitedTaskMs[3];
  float _timeWaitedAckMs[3];
  float _temperature[3];
  float _timeTransferSquadSquidMs[3];
  float _timeTransferSquidSquadMs[3];
  float _timePerTask;
  float _nbTaskExpected;
} SquidletInfoStats;

typedef struct SquidletInfo {
  // Name of the squidlet
  char* _name;
  // IP of the squidlet
  char* _ip;
  // Port of the squidlet
  int _port;
  // Socket to communicate with this squidlet
  short _sock;
  // Statistics
  SquidletInfoStats _stats;
} SquidletInfo;

// ================ Functions declaration ====================

// Return a new SquidletInfo describing a Squidlet whose name is
// 'name', and whose attached to the address 'ip':'port'
SquidletInfo* SquidletInfoCreate(
  const char* const name,
  const char* const ip,
    const int const port);
```

```
// Free the memory used by the SquidletInfo 'that'
void SquidletInfoFree(
  SquidletInfo** that);

// Print the SquidletInfo 'that' on the file 'stream'
void SquidletInfoPrint(
  const SquidletInfo* const that,
                FILE* const stream);

// Print the SquidletInfoStats 'that' on the file 'stream'
void SquidletInfoStatsPrintln(
  const SquidletInfoStats* const that,
                    FILE* const stream);

// Init the stats of the SquidletInfoStats 'that'
void SquidletInfoStatsInit(
  SquidletInfoStats* const that);

// Return the stats of the SquidletInfo 'that'
#if BUILDMODE != 0
inline
#endif
const SquidletInfoStats* SquidletInfoStatistics(
  const SquidletInfo* const that);

// Update the statistics about the transfer time of the SquidletInfoStats
// 'that' given that it took 'deltams' millisecond to send 'len' bytes
// of data
void SquidletInfoStatsUpdateTimeTransfer(
  SquidletInfoStats* const that,
               const float deltams,
              const size_t len);

// -------------- SquidletTaskRequest

// ================= Data structure ==================

typedef enum SquidletTaskType {
  SquidletTaskType_Null,
  SquidletTaskType_Dummy,
  SquidletTaskType_Benchmark,
  SquidletTaskType_PovRay,
  SquidletTaskType_ResetStats,
  SquidletTaskType_EvalNeuranet} SquidletTaskType;

typedef struct SquidletTaskRequest {
  // Task type
  SquidletTaskType _type;
  // Task ID
  unsigned long _id;
  // Task sub ID
  unsigned long _subId;
  // Data associated to the request, as a string in JSON format
  char* _data;
  // Buffer to receive the result from the squidlet, as a string in
  // JSON format
  char* _bufferResult;
  // Time in second after which the Squad give up waiting for the
  // completion of this task
  time_t _maxWaitTime;
} SquidletTaskRequest;
```

```
// ================ Functions declaration ====================

// Return a new SquidletTaskRequest for a task of type 'type'
// The task is identified by its 'id'/'subId', it will have at
// maximum 'maxWait' seconds to complete. It holds a copy of 'data', a
// string in JSON format
SquidletTaskRequest* SquidletTaskRequestCreate(
   SquidletTaskType type,
      unsigned long id,
      unsigned long subId,
  const char* const data,
       const time_t maxWait);

// Free the memory used by the SquidletTaskRequest 'that'
void SquidletTaskRequestFree(
  SquidletTaskRequest** that);

// Print the SquidletTaskRequest 'that' on the file 'stream'
// Only a maximum of 100 first characters of the data are printed
void SquidletTaskRequestPrint(
  const SquidletTaskRequest* const that,
                       FILE* const stream);

// Return true if the SquidletTask 'that' has succeeded, else false
// The task is considered to have succeeded if its result buffer
// contains "success":"1"
#if BUILDMODE != 0
inline
#endif
bool SquidletTaskHasSucceeded(
  const SquidletTaskRequest* const that);

// Return the type of the task 'that' as a string
const char* SquidletTaskTypeAsStr(
  const SquidletTaskRequest* const that);

// Return the type of the task 'that'
#if BUILDMODE != 0
inline
#endif
SquidletTaskType SquidletTaskGetType(
  const SquidletTaskRequest* const that);

// Return the id of the task 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long SquidletTaskGetId(
  const SquidletTaskRequest* const that);

// Return the subid of the task 'that'
#if BUILDMODE != 0
inline
#endif
unsigned long SquidletTaskGetSubId(
  const SquidletTaskRequest* const that);

// Return the data of the task 'that'
#if BUILDMODE != 0
inline
#endif
const char* SquidletTaskData(
```

```
  const SquidletTaskRequest* const that);

// Return the buffer result of the task 'that'
#if BUILDMODE != 0
inline
#endif
const char* SquidletTaskBufferResult(
  const SquidletTaskRequest* const that);

// Return the max wait time of the task 'that'
#if BUILDMODE != 0
inline
#endif
time_t SquidletTaskGetMaxWaitTime(
  const SquidletTaskRequest* const that);

// -------------- SquadRunningTask

// ================= Data structure ===================

typedef struct SquadRunningTask {
  // The task
  SquidletTaskRequest* _request;
  // The squidlet
  SquidletInfo* _squidlet;
  // Time when the SquadRunningTask is created
  time_t _startTime;
} SquadRunningTask;

// ================ Functions declaration ====================

// Return a new SquadRunningTask for the SquidletTaskRequest 'request'
// running on the SquidletInfo 'squidlet'
SquadRunningTask* SquadRunningTaskCreate(
  SquidletTaskRequest* const request,
         SquidletInfo* const squidlet);

// Free the memory used by the SquadRunningTask 'that'
void SquadRunningTaskFree(
  SquadRunningTask** that);

// Print the SquadRunningTask 'that' on the file 'stream'
void SquadRunningTaskPrint(
  const SquadRunningTask* const that,
                    FILE* const stream);

// -------------- Squad

// ================= Data structure ===================

typedef struct Squad {
  // File descriptor of the socket
  short _fd;
  // Set of squidlets used by the Squad
  GSetSquidletInfo _squidlets;
  // Set of tasks to execute
  GSetSquidletTaskRequest _tasks;
  // Set of tasks currently under execution
  GSetSquadRunningTask _runningTasks;
  // Flag to memorize if info are displayed with a TextOMeter
  bool _flagTextOMeter;
  // TextOMeter to display info
```

```
  TextOMeter* _textOMeter;
  // Buffer used to display info in the TextOMeter
  char _history[SQUAD_TXTOMETER_NBLINEHISTORY] \
    [SQUAD_TXTOMETER_LENGTHLINEHISTORY];
  // Counter used to display info in the TextOMeter
  unsigned int _countLineHistory;
} Squad;

// ================ Functions declaration ====================

// Return a new Squad
Squad* SquadCreate(void);

// Free the memory used by the Squad 'that'
void SquadFree(
  Squad** that);

// Load a list of tasks stored in json format from the file 'stream'
// and add them to the set of tasks of the Squad 'that'
// If the Squad had already tasks, the loaded ones are added to them
// Return true if the tasks could be loaded, else false
// Example:
// {"tasks":[
//   {"SquidletTaskType":"1", "id":"1", "maxWait":"1"},
//   {"SquidletTaskType":"2", "id":"1", "maxWait":"1",
//    "nb":"1", "payloadSize":"1"},
//   {"SquidletTaskType":"3", "id":"1", "maxWait":"1",
//    "ini":"./testPov.ini", "sizeMinFragment":"100",
//    "sizeMaxFragment":"1000"}
// ]}
bool SquadLoadTasks(
  Squad* const that,
   FILE* const stream);

// Get the set of squidlets of the Squad 'that'
#if BUILDMODE != 0
inline
#endif
const GSetSquidletInfo* SquadSquidlets(
  const Squad* const that);

// Get the set of task to execute of the Squad 'that'
#if BUILDMODE != 0
inline
#endif
const GSetSquidletTaskRequest* SquadTasks(
  const Squad* const that);

// Get the set of running tasks of the Squad 'that'
#if BUILDMODE != 0
inline
#endif
const GSetSquadRunningTask* SquadRunningTasks(
  const Squad* const that);

// Load the Squidlet info from the file 'stream' into the Squad 'that'
// Return true if it could load the info, else false
bool SquadLoadSquidlets(
  Squad* const that,
   FILE* const stream);

// Load the Squidlet info from the string 'str' into the Squad 'that'
```

```
// Return true if it could load the info, else false
bool SquadLoadSquidletsFromStr(
  Squad* const that,
   char* const str);

// Send the task request 'request' from the Squad 'that' to its
// Squidlet 'squidlet'
// Create a socket, open a connection, ask the squidlet if it can
// execute the task and wait for its reply
// Return true if the request has been accepted by the squidlet,
// false else
bool SquadSendTaskRequest(
                      Squad* const that,
  const SquidletTaskRequest* const request,
              SquidletInfo* const squidlet);

// Send the data associated to the task request 'task' from the Squad
// 'that' to the Squidlet 'squidlet'
// First, send the size in byte of the data, then send the data
// Return true if the data could be sent, false else
// The size of the data must be less than THESQUID_MAXPAYLOADSIZE bytes
bool SquadSendTaskData(
               Squad* const that,
         SquidletInfo* const squidlet,
  SquidletTaskRequest* const task);

// Try to receive the result from the running task 'runningTask'
// If the result is ready it is stored in the _bufferResult of the
// SquidletTaskRequest of the 'runningTask'
// If the size of the result data is not ready and couldn't be received
// give up immediately
// If the size of the result data has been received, wait for
// (5 + sizeData / 100) seconds maximum to receive the data
// First receive the size of the result data, then send an
// acknowledgement signal to the squidlet for the size of data,
// then receive the data, and finally send an acknowledgement to the
// squidlet for the data
// Return true if it could receive the result data, false else
bool SquadReceiveTaskResult(
             Squad* const that,
  SquadRunningTask* const runningTask);

// Add a dummy task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
void SquadAddTask_Dummy(
         Squad* const that,
  const unsigned long id,
         const time_t maxWait);

// Add a benchmark task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
// Artificially set the size of the data for the task to 'payloadSize'
// bytes
// The benchmark function used is sorting 'nb' times a set of numbers
// cf TheSquidBenchmark()
void SquadAddTask_Benchmark(
```

```
        Squad* const that,
  const unsigned long id,
        const time_t maxWait,
   const unsigned int nb,
   const unsigned int payloadSize);

// Add a POV-Ray task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
// The POV-Ray task is described by the POV-Ray ini file 'ini' which
// must include at least the following options:
// Input_File_Name=<script.pov>
// Width=<width>
// Height=<height>
// Output_File_Name=<image.tga>
// The output format of the image must be TGA
// Video are not supported
// The image is splitted into as many squared fragments as
// SquadGetNbSquidlets(that)^2, but the size of the fragment is clipped
// to [sizeMinFragment, sizeMaxFragment]
// The size of the fragments are corrected to fit the size of the image
// if it's not a squared image
// Fragments are rendered on the squidlets in random order
// The final image is updated each time a fragment has been
// rendered
// The random generator must have been initialised before calling this
// function
void SquadAddTask_PovRay(
        Squad* const that,
  const unsigned long id,
        const time_t maxWait,
    const char* const ini,
   const unsigned int sizeMinFragment,
   const unsigned int sizeMaxFragment);

// Add a dummy task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
void SquadAddTask_EvalNeuraNet(
        Squad* const that,
  const unsigned long id,
        const time_t maxWait,
    const char* const datasetPath,
    const char* const workingDirPath,
 const VecLong* const nnids,
         const float curBest,
          const long cat);

// Send a request from the Squad 'that' to reset the stats of the
// Squidlet 'squid'
// Return true if the request was successfull, else false
bool SquadRequestSquidletToResetStats(
        Squad* const that,
  SquidletInfo* const squid);

// Send a request from the Squad 'that' to reset the stats of all its
// currently available Squidlets
// Return true if all the request were successfull, else false
```

```
bool SquadRequestAllSquidletToResetStats(
  Squad* const that);

// Return the number of tasks not yet completed
#if BUILDMODE != 0
inline
#endif
unsigned long SquadGetNbTaskToComplete(
  const Squad* const that);

// Return the number of running tasks
#if BUILDMODE != 0
inline
#endif
unsigned long SquadGetNbRunningTasks(
  const Squad* const that);

// Return the number of tasks to execute
#if BUILDMODE != 0
inline
#endif
unsigned long SquadGetNbRemainingTasks(
  const Squad* const that);

// Return the number of currently available squidlets (squidlets not
// executing a task for 'that')
#if BUILDMODE != 0
inline
#endif
unsigned long SquadGetNbSquidlets(
  const Squad* const that);

// Step the Squad 'that', i.e. tries to affect the remaining tasks to
// available Squidlets and check for completion of running tasks.
// Return the GSet of the completed SquadRunningTask at this step
// Non blocking, if there is no task to compute or no squidlet
// available, and no task completed, do nothing and return an empty set
GSetSquadRunningTask SquadStep(
  Squad* const that);

// Process the completed 'task' with the Squad 'that' after its
// reception in SquadStep()
void SquadProcessCompletedTask(
             Squad* const that,
  SquadRunningTask* const task);

// Process the completed Pov-Ray 'task' with the Squad 'that'
void SquadProcessCompletedTask_PovRay(
               Squad* const that,
  SquidletTaskRequest* const task);

// Set the flag memorizing if the TextOMeter is displayed for
// the Squad 'that' to 'flag'
void SquadSetFlagTextOMeter(
  Squad* const that,
  const bool flag);

// Return the flag for the TextOMeter of the Squad 'that'
#if BUILDMODE != 0
inline
#endif
bool SquadGetFlagTextOMeter(
```

```
  const Squad* const that);

// Put back the 'task' into the set of task to complete of the Squad
// 'that'
// Failed tasks (by timeout due to there 'maxWait' in
// SquadAddTask_xxx() or by failure code from the squidlet in the
// result data) are automatically put back into the set of task to
// complete
void SquadTryAgainTask(
                Squad* const that,
  SquidletTaskRequest* const task);

// Check all the squidlets of the Squad 'that' by processing a dummy
// task and display information about each one on the file 'stream'
// Return true if all the tasks could be performed, false else
bool SquadCheckSquidlets(
  Squad* const that,
   FILE* const stream);

// Run the benchmark with the squad 'that' and output the result on
// the file 'stream'
// The benchmark consists of executing benchmark tasks with several
// size and number of loop, first locally and then on the squidlet
// (which must have been loaded before calling this function)
void SquadBenchmark(
  Squad* const that,
   FILE* const stream);

// Print the statistics about the currently available Squidlets of
// the Squad 'that' on the 'stream'
void SquadPrintStatsSquidlets(
  const Squad* const that,
        FILE* const stream);

// -------------- Squidlet

// ================= Global variable ==================

// Variable to handle the signal Ctrl-C to kill the Squidlet running
// in background
extern bool Squidlet_CtrlC;

// ================= Data structure ===================

typedef struct Squidlet {
  // File descriptor of the socket used by the Squidlet ot listen for
  // connection
  short _fd;
  // Port the Squidlet is listening to
  int _port;
  // Info about the socket '_fd'
  struct sockaddr_in _sock;
  // PID of the process of the squidlet
  pid_t _pid;
  // Hostname of the device on which the Squidlet is running
  char _hostname[256];
  // Information about the host
  struct hostent* _host;
  // File descriptor of the socket to send the result of a task
  short _sockReply;
  // Stream to output infos, if null the squidlet is silent
  // By default it's null
```

```
  FILE* _streamInfo;
  // Variables for statistics
  unsigned long _nbAcceptedConnection;
  unsigned long _nbAcceptedTask;
  unsigned long _nbRefusedTask;
  unsigned long _nbFailedReceptTaskData;
  unsigned long _nbFailedReceptTaskSize;
  unsigned long _nbSentResult;
  unsigned long _nbFailedSendResult;
  unsigned long _nbFailedSendResultSize;
  unsigned long _nbFailedReceptAck;
  unsigned long _nbTaskComplete;
  unsigned long _timeToProcessMs;
  unsigned long _timeWaitedTaskMs;
  struct timeval _timeLastTaskComplete;
  unsigned long _timeWaitedAckMs;
  float _timeTransferSquidSquadMs;
  // Path of the last used GDataSet
  char* _datasetPath;
  // Last used GDataSet
  GDataSetVecFloat _dataset;
} Squidlet;

// ================ Functions declaration ====================

// Handler for the signal Ctrl-C
void SquidletHandlerCtrlC(
  const int sig);

// Handler for the signal SIGPIPE
void SquidletHandlerSigPipe(
  const int sig);

// Return a new Squidlet listening to the ip 'addr' and port 'port'
// If 'addr' equals 0, select automatically the first network address
// of the host
// If 'port' equals -1, select automatically one available between
// THESQUID_PORTMIN and THESQUID_PORTMAX
Squidlet* SquidletCreateOnPort(
  const uint32_t addr,
       const int port);

// Helper to create a squidlet with default ip and port
#define SquidletCreate() SquidletCreateOnPort(0, -1)

// Free the memory used by the Squidlet 'that'
void SquidletFree(
  Squidlet** that);

// Print the PID, Hostname, IP and Port of the Squidlet 'that' on the
// file 'stream'
// Example: 100 localhost 0.0.0.0:3000
void SquidletPrint(
  const Squidlet* const that,
            FILE* const stream);

// Reset the statistics of the Squidlet 'that'
void SquidletResetStats(
  Squidlet* const that);

// Wait for a task request to be received by the Squidlet 'that'
// Return the received task request, or give up after
```

```
// THESQUID_ACCEPT_TIMEOUT if there was no request and return a task
// request of type SquidletTaskType_Null
SquidletTaskRequest SquidletWaitRequest(
  Squidlet* const that);

// Process the task request 'request' with the Squidlet 'that'
void SquidletProcessRequest(
             Squidlet* const that,
  SquidletTaskRequest* const request);

// Process a dummy task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_Dummy(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult);

// Process a benchmark task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_Benchmark(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult);

// Process a Pov-Ray task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_PovRay(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult);

// Process a stats reset task request with the Squidlet 'that'
void SquidletProcessRequest_StatsReset(
    Squidlet* const that);

// Process a neuranet evaluation task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_EvalNeuranet(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult);

// Get the PID of the Squidlet 'that'
#if BUILDMODE != 0
inline
#endif
pid_t SquidletGetPID(
  const Squidlet* const that);

// Get the hostname of the Squidlet 'that'
```

```
#if BUILDMODE != 0
inline
#endif
const char* SquidletHostname(
  const Squidlet* const that);

// Get the IP of the Squidlet 'that'
#if BUILDMODE != 0
inline
#endif
const char* SquidletIP(
  const Squidlet* const that);

// Get the port of the Squidlet 'that'
#if BUILDMODE != 0
inline
#endif
int SquidletGetPort(
  const Squidlet* const that);

// Get the stream to output info of the Squidlet 'that'
#if BUILDMODE != 0
inline
#endif
FILE* SquidletStreamInfo(
  const Squidlet* const that);

// Set the stream to output info of the Squidlet 'that' to 'stream'
// 'stream' may be null to mute the Squidlet
#if BUILDMODE != 0
inline
#endif
void SquidletSetStreamInfo(
  Squidlet* const that,
      FILE* const stream);

// Return the temperature of the squidlet 'that' as a float.
// The result depends on the architecture on which the squidlet is
// running. It is '0.0' if the temperature is not available
float SquidletGetTemperature(
  const Squidlet* const that);

// -------------- TheSquid

// ================ Functions declaration ====================

// Function for benchmark purpose
int TheSquidBenchmark(
               int nbLoop,
  const char* const buffer);

// ================ Inliner ====================

#if BUILDMODE != 0
#include "thesquid-inline.c"
#endif

#endif
```

6 Code

6.1 thesquid.c

```
// ============ THESQUID.C ================

// ================ Include =================

#include "thesquid.h"
#if BUILDMODE == 0
  #include "thesquid-inline.c"
#endif

// ================ Module variables ====================

// Name of the tasks types
const char* squidletTaskTypeStr[] = {
  "Null", "Dummy", "Benchmark", "PovRay", "ResetStats", "EvalNeuranet"
};

// ================ Module functions declaration ====================

// Function to receive in blocking mode 'nb' bytes of data from
// the socket 'sock' and store them into 'buffer' (which must be big
// enough). Give up after 'timeout' seconds.
// Return true if we could read all the requested byte, false else
bool SocketRecv(
  short* sock,
  unsigned long nb,
  char* buffer,
  const time_t timeout);

// Append the statistical data about the squidlet 'that' to the JSON
// node 'json'
void SquidletAddStatsToJSON(
  const Squidlet* const that,
        JSONNode* const json);

// Update the statitics of the SquidletInfo 'that' with the result of
// the 'task'
void SquidletInfoUpdateStats(
         SquidletInfo* const that,
  SquidletTaskRequest* const task);

// Send the result 'bufferResult' of the processing of a task
// by the Squidlet 'that'
void SquidletSendResultData(
             Squidlet* const that,
           const char* const bufferResult);

// -------------- SquidletInfo

// ================ Functions implementation ====================

// Return a new SquidletInfo describing a Squidlet whose name is
// 'name', and whose attached to the address 'ip':'port'
SquidletInfo* SquidletInfoCreate(
  const char* const name,
  const char* const ip,
    const int const port) {
#if BUILDMODE == 0
```

```
  if (name == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'name' is null");
    PBErrCatch(TheSquidErr);
  }
  if (ip == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'ip' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Allocate memory for the squidletInfo
  SquidletInfo* that = PBErrMalloc(TheSquidErr, sizeof(SquidletInfo));

  // Init properties
  that->_name = strdup(name);
  that->_ip = strdup(ip);
  that->_port = port;
  that->_sock = -1;

  // Init the stats
  SquidletInfoStatsInit(&(that->_stats));

  // Return the new squidletInfo
  return that;
}

// Init the stats of the SquidletInfoStats 'that'
void SquidletInfoStatsInit(
  SquidletInfoStats* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  that->_nbAcceptedConnection = 0;
  that->_nbAcceptedTask = 0;
  that->_nbRefusedTask = 0;
  that->_nbFailedReceptTaskData = 0;
  that->_nbFailedReceptTaskSize = 0;
  that->_nbSentResult = 0;
  that->_nbFailedSendResult = 0;
  that->_nbFailedSendResultSize = 0;
  that->_nbFailedReceptAck = 0;
  that->_nbTaskComplete = 0;
  for (int i = 3; i--;) {
    that->_timeToProcessMs[i] = 0.0;
    that->_timeWaitedTaskMs[i] = 0.0;
    that->_timeWaitedAckMs[i] = 0.0;
    that->_timeTransferSquadSquidMs[i] = 0.0;
    that->_timeTransferSquidSquadMs[i] = 0.0;
  }
}

// Free the memory used by the SquidletInfo 'that'
void SquidletInfoFree(
  SquidletInfo** that) {
  // If the pointer is null there is nothing to do
  if (that == NULL || *that == NULL)
    return;
```

```
  // Close the socket if it's opened
  if ((*that)->_sock != -1)
    close((*that)->_sock);

  // Free memory
  if ((*that)->_name != NULL)
    free((*that)->_name);
  if ((*that)->_ip != NULL)
    free((*that)->_ip);
  free(*that);
  *that = NULL;
}

// Print the SquidletInfo 'that' on the file 'stream'
void SquidletInfoPrint(
  const SquidletInfo* const that,
                FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Print the info on the stream
  fprintf(stream, "%s(%s:%d)", that->_name, that->_ip, that->_port);
}

// Print the SquidletInfoStats 'that' on the file 'stream'
void SquidletInfoStatsPrintln(
  const SquidletInfoStats* const that,
                    FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Print the stats on the stream
  fprintf(stream, "    nbAcceptedConnection: %lu\n",
    that->_nbAcceptedConnection);
  fprintf(stream, "          nbAcceptedTask: %lu\n",
    that->_nbAcceptedTask);
  fprintf(stream, "           nbRefusedTask: %lu\n",
    that->_nbRefusedTask);
  fprintf(stream, "  nbFailedReceptTaskData: %lu\n",
    that->_nbFailedReceptTaskData);
  fprintf(stream, "  nbFailedReceptTaskSize: %lu\n",
    that->_nbFailedReceptTaskSize);
  fprintf(stream, "            nbSentResult: %lu\n",
```

```
    that->_nbSentResult);
  fprintf(stream, "      nbFailedSendResult: %lu\n",
    that->_nbFailedSendResult);
  fprintf(stream, "  nbFailedSendResultSize: %lu\n",
    that->_nbFailedSendResultSize);
  fprintf(stream, "       nbFailedReceptAck: %lu\n",
    that->_nbFailedReceptAck);
  fprintf(stream, "          nbTaskComplete: %lu\n",
    that->_nbTaskComplete);
  fprintf(stream, "         timeToProcessMs: %07.0f/%07.0f/%07.0f\n",
    that->_timeToProcessMs[0], that->_timeToProcessMs[1],
    that->_timeToProcessMs[2]);
  fprintf(stream, "        timeWaitedTaskMs: %07.0f/%07.0f/%07.0f\n",
    that->_timeWaitedTaskMs[0], that->_timeWaitedTaskMs[1],
    that->_timeWaitedTaskMs[2]);
  fprintf(stream, "         timeWaitedAckMs: %07.0f/%07.0f/%07.0f\n",
    that->_timeWaitedAckMs[0], that->_timeWaitedAckMs[1],
    that->_timeWaitedAckMs[2]);
  fprintf(stream, "             temperature: %03.1f/%03.1f/%03.1f\n",
    that->_temperature[0], that->_temperature[1],
    that->_temperature[2]);
  fprintf(stream, "timeTransferSquadSquidMs: %04.3f/%04.3f/%04.3f\n",
    that->_timeTransferSquadSquidMs[0], that->_timeTransferSquadSquidMs[1],
    that->_timeTransferSquadSquidMs[2]);
  fprintf(stream, "timeTransferSquidSquadMs: %04.3f/%04.3f/%04.3f\n",
    that->_timeTransferSquidSquadMs[0], that->_timeTransferSquidSquadMs[1],
    that->_timeTransferSquidSquadMs[2]);
}

// -------------- SquidletTaskRequest

// ================ Functions implementation ====================

// Return a new SquidletTaskRequest for a task of type 'type'
// The task is identified by its 'id'/'subId', it will have at
// maximum 'maxWait' seconds to complete. It holds a copy of 'data', a
// string in JSON format
SquidletTaskRequest* SquidletTaskRequestCreate(
   SquidletTaskType type,
      unsigned long id,
      unsigned long subId,
  const char* const data,
       const time_t maxWait) {
#if BUILDMODE == 0
  if (data == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'data' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Allocate memory for the new SquidletTaskRequest
  SquidletTaskRequest* that = PBErrMalloc(TheSquidErr,
    sizeof(SquidletTaskRequest));

  // Init properties
  that->_id = id;
  that->_subId = subId;
  that->_type = type;
  that->_data = strdup(data);
  that->_bufferResult = NULL;
  that->_maxWaitTime = maxWait;
```

```
  // Return the new SquidletTaskRequest
  return that;
}

// Free the memory used by the SquidletTaskRequest 'that'
void SquidletTaskRequestFree(
  SquidletTaskRequest** that) {
  // If the pointer is null there is nothing to do
  if (that == NULL || *that == NULL)
    return;

  // Free memory
  if ((*that)->_bufferResult != NULL)
    free((*that)->_bufferResult);
  if ((*that)->_data != NULL)
    free((*that)->_data);
  free(*that);
  *that = NULL;
}

// Print the SquidletTaskRequest 'that' on the file 'stream'
// Only a maximum of 100 first characters of the data are printed
void SquidletTaskRequestPrint(
  const SquidletTaskRequest* const that,
                       FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a buffer to truncate the data
  #define SquidletTaskRequestPrint_lengthTrunc 100
  char truncData[SquidletTaskRequestPrint_lengthTrunc];
  truncData[SquidletTaskRequestPrint_lengthTrunc - 1] = '\0';

  // Copy the data
  strncpy(truncData, SquidletTaskData(that),
    SquidletTaskRequestPrint_lengthTrunc - 1);

  // Add a mark if the data were too long
  if (strlen(SquidletTaskData(that)) >=
    SquidletTaskRequestPrint_lengthTrunc) {
    strcpy(truncData + SquidletTaskRequestPrint_lengthTrunc - 7,
      " (...)");
  }

  // Print the info on the stream
  fprintf(stream, "%s(#%lu-%lu) %s",
    SquidletTaskTypeAsStr(that),
    SquidletTaskGetId(that),
    SquidletTaskGetSubId(that),
    truncData);
}
```

```
// Return the type of the task 'that' as a string
const char* SquidletTaskTypeAsStr(
  const SquidletTaskRequest* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // If the type is valid
  if (that->_type >= 0 && that->_type < sizeof(squidletTaskTypeStr)) {

    // Return the name of the type
    return squidletTaskTypeStr[that->_type];

  // Else, the name is not valid
  } else {

    // Return the name of the default type
    return squidletTaskTypeStr[0];

  }
}


// -------------- SquadRunningTask

// ================ Functions implementation ====================

// Return a new SquadRunningTask for the SquidletTaskRequest 'request'
// running on the SquidletInfo 'squidlet'
SquadRunningTask* SquadRunningTaskCreate(
  SquidletTaskRequest* const request,
         SquidletInfo* const squidlet) {
#if BUILDMODE == 0
  if (request == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'request' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Allocate memory for the new SquadRunningTask
  SquadRunningTask* that = PBErrMalloc(TheSquidErr,
    sizeof(SquadRunningTask));

  // Init properties
  that->_request = request;
  that->_squidlet = squidlet;
  that->_startTime = time(NULL);

  // Return the new SquadRunningTask
  return that;
}

// Free the memory used by the SquadRunningTask 'that'
```

```
void SquadRunningTaskFree(
  SquadRunningTask** that) {
  // If the pointer is null there is nothing to do
  if (that == NULL || *that == NULL)
    return;

  // Free memory
  free(*that);
  *that = NULL;
}

// Print the SquadRunningTask 'that' on the file 'stream'
void SquadRunningTaskPrint(
  const SquadRunningTask* const that,
                    FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Print the info on the stream
  fprintf(stream, "[");
  if (that->_request != NULL) {
    SquidletTaskRequestPrint(that->_request, stream);
  } else {
    fprintf(stream, "<null>");
  }
  fprintf(stream, "]/[");
  if (that->_squidlet != NULL) {
    SquidletInfoPrint(that->_squidlet, stream);
  } else {
    fprintf(stream, "<null>");
  }
  fprintf(stream, "]");
}

// -------------- Squad

// ================ Functions declaration ====================

// Decode the JSON info of a Squad from the JSON node 'json'
bool SquadDecodeAsJSON(
     Squad* that,
  JSONNode* json);

// Refresh the content of the TextOMeter attached to the
// Squad 'that'
void SquadUpdateTextOMeter(
  const Squad* const that);

// Add one line to the history of messages for the TextOMeter
// 'msg' is truncated if it doesn't fit in one line of history
// If the TextOmeter is not turned on, do nothing
// Variadic function with the same signature as printf family
void SquadPushHistory(
```

```
  Squad* const that,
         char* msg,
               ...);

// Add the result of SquidletInfoPrint(squidlet) to the history
// of messages for the TextOMeter
// The output of SquidletInfoPrint is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquidletInfo(
               Squad* const that,
  const SquidletInfo* const squidlet);

// Add the result of SquidletTaskRequestPrint(request) to the history
// of messages for the TextOMeter
// The output of SquidletTaskRequest is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquidletTaskRequest(
                     Squad* const that,
  const SquidletTaskRequest* const request);

// Add the result of SquadRunningTaskPrint(request) to the history
// of messages for the TextOMeter
// The output of SquadRunningTaskPrint is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquadRunningTask(
                   Squad* const that,
  const SquadRunningTask* const task);

// Request the execution of a task on a squidlet for the squad 'that'
// Return true if the request was successfull, fals else
bool SquadSendTaskOnSquidlet(
  Squad* const that,
  SquidletInfo* const squidlet,
  SquidletTaskRequest* const task);

// ================ Functions implementation ====================

// Return a new Squad
Squad* SquadCreate(void) {
  // Allocate memory for the squad
  Squad* that = PBErrMalloc(TheSquidErr, sizeof(Squad));

  // Open the TCP/IP socket
  // AF_INET: IPv4 Internet protocols
  // SOCK_STREAM: Provides sequenced, reliable, two-way,
  // connection-based byte streams.
  int defaultProtocol = 0;
  that->_fd = socket(AF_INET, SOCK_STREAM, defaultProtocol);

  // If we couldn't open the socket
  if (that->_fd == -1) {

    // Free memory and return null
    free(that);
    return NULL;
  }

  // Init properties
  that->_squidlets = GSetSquidletInfoCreateStatic();
```

```
  that->_tasks = GSetSquidletTaskRequestCreateStatic();
  that->_runningTasks = GSetSquadRunningTaskCreateStatic();
  that->_flagTextOMeter = false;
  that->_textOMeter = NULL;
  for (int iLine = 0; iLine < SQUAD_TXTOMETER_NBLINEHISTORY; ++iLine) {
    that->_history[iLine][0] = '\n';
    that->_history[iLine][1] = '\0';
  }
  that->_countLineHistory = 0;

  // Return the new squad
  return that;
}

// Free the memory used by the Squad 'that'
void SquadFree(
  Squad** that) {
  // If the pointer is null there is nothing to do
  if (that == NULL || *that == NULL)
    return;

  // Close the socket
  close((*that)->_fd);

  // Free memory
  while (GSetNbElem(SquadSquidlets(*that)) > 0) {
    SquidletInfo* squidletInfo = GSetPop((GSet*)SquadSquidlets(*that));
    SquidletInfoFree(&squidletInfo);
  }
  while (GSetNbElem(SquadTasks(*that)) > 0) {
    SquidletTaskRequest* task = GSetPop((GSet*)SquadTasks(*that));
    SquidletTaskRequestFree(&task);
  }
  while (GSetNbElem(SquadRunningTasks(*that)) > 0) {
    SquadRunningTask* task = GSetPop((GSet*)SquadRunningTasks(*that));
    SquadRunningTaskFree(&task);
  }
  if ((*that)->_textOMeter != NULL) {
    TextOMeterFree(&((*that)->_textOMeter));
  }
  free(*that);
  *that = NULL;
}

// Load a list of tasks stored in json format from the file 'stream'
// and add them to the set of tasks of the Squad 'that'
// If the Squad had already tasks, the loaded ones are added to them
// Return true if the tasks could be loaded, else false
// Example of list of tasks:
// {"tasks":[
//   {"SquidletTaskType":"1", "id":"1", "maxWait":"1"},
//   {"SquidletTaskType":"2", "id":"1", "maxWait":"1",
//    "nb":"1", "payloadSize":"1"},
//   {"SquidletTaskType":"3", "id":"1", "maxWait":"1",
//    "ini":"./testPov.ini", "sizeMinFragment":"100",
//    "sizeMaxFragment":"1000"}
// ]}
bool SquadLoadTasks(
  Squad* const that,
   FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();

  // Load the whole encoded data
  if (JSONLoad(json, stream) == false) {
    TheSquidErr->_type = PBErrTypeIOError;
    sprintf(TheSquidErr->_msg, "JSONLoad failed");
    JSONFree(&json);
    return false;
  }

  // Get the list of tasks
  JSONNode* propTasks = JSONProperty(json, "tasks");
  if (propTasks == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "tasks not found");
    JSONFree(&json);
    return false;
  }

  // Get the number of tasks
  unsigned long nbTasks = JSONGetNbValue(propTasks);

  // Loop on tasks
  for (unsigned long iTask = 0; iTask < nbTasks; ++iTask) {

    // Get the task
    JSONNode* propTask = JSONValue(propTasks, iTask);

    // Get the type of task, id and time out
    JSONNode* propType = JSONProperty(propTask, "SquidletTaskType");
    if (propType == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'SquidletTaskType' not found");
      JSONFree(&json);
      return false;
    }
    JSONNode* propId = JSONProperty(propTask, "id");
    if (propId == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'id' not found");
      JSONFree(&json);
      return false;
    }
    JSONNode* propMaxWait = JSONProperty(propTask, "maxWait");
    if (propMaxWait == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'maxWait' not found");
      JSONFree(&json);
      return false;
    }
```

```
// Convert values
int type = atoi(JSONLblVal(propType));
unsigned long id = atol(JSONLblVal(propId));
time_t maxWait = atoi(JSONLblVal(propMaxWait));

// Switch according to the type of task and add the corresponding
// task
JSONNode* prop = NULL;
switch(type) {

  // Dummy task
  case SquidletTaskType_Dummy:

    // Add the task
    SquadAddTask_Dummy(that, id, maxWait);
    break;

  // Benchmark task
  case SquidletTaskType_Benchmark:

    // Get the extra arguments
    prop = JSONProperty(propTask, "nb");
    if (prop == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'nb' not found");
      JSONFree(&json);
      return false;
    }
    int nb = atoi(JSONLblVal(prop));
    prop = JSONProperty(propTask, "payloadSize");
    if (prop == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'payloadSize' not found");
      JSONFree(&json);
      return false;
    }
    size_t payloadSize = atol(JSONLblVal(prop));

    // Add the task
    SquadAddTask_Benchmark(that, id, maxWait, nb, payloadSize);
    break;

  // POV-Ray task
  case SquidletTaskType_PovRay:

    // Get the extra arguments
    prop = JSONProperty(propTask, "ini");
    if (prop == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "ini not found");
      JSONFree(&json);
      return false;
    }
    char* ini = strdup(JSONLblVal(prop));
    prop = JSONProperty(propTask, "sizeMinFragment");
    if (prop == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "sizeMinFragment not found");
      JSONFree(&json);
      return false;
    }
    int sizeMinFragment = atoi(JSONLblVal(prop));
```

```
  prop = JSONProperty(propTask, "sizeMaxFragment");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "sizeMaxFragment not found");
    JSONFree(&json);
    return false;
  }
  int sizeMaxFragment = atoi(JSONLblVal(prop));

  // Add the task
  SquadAddTask_PovRay(that, id, maxWait, ini,
    sizeMinFragment, sizeMaxFragment);

  // Free memory
  free(ini);
  break;

// Stats reset task
case SquidletTaskType_ResetStats:

  // Ignore this special task which can be only triggered
  // at runtime
  break;

// Neuranet evaluation task
case SquidletTaskType_EvalNeuranet:

  prop = JSONProperty(propTask, "dataset");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "dataset not found");
    JSONFree(&json);
    return false;
  }
  char* dataset = JSONLblVal(prop);
  prop = JSONProperty(propTask, "workingDir");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "workingDir not found");
    JSONFree(&json);
    return false;
  }
  char* workingDir = JSONLblVal(prop);
  prop = JSONProperty(propTask, "nnids");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "nnids not found");
    JSONFree(&json);
    return false;
  }
  VecLong* nnids = NULL;
  VecDecodeAsJSON(&nnids, prop);
  prop = JSONProperty(propTask, "best");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "best not found");
    JSONFree(&json);
    return false;
  }
  float bestVal = atof(JSONLblVal(prop));
  prop = JSONProperty(propTask, "cat");
  if (prop == NULL) {
```

```
          TheSquidErr->_type = PBErrTypeInvalidData;
          sprintf(TheSquidErr->_msg, "cat not found");
          JSONFree(&json);
          return false;
        }
        long cat = atol(JSONLblVal(prop));

        // Add the task
        SquadAddTask_EvalNeuraNet(that, id, maxWait,
          dataset, workingDir, nnids, bestVal, cat);

        // Free memory
        VecFree(&nnids);

        break;

      // Invalid task type
      default:
        // Set the error message
        TheSquidErr->_type = PBErrTypeInvalidData;
        sprintf(TheSquidErr->_msg, "invalid task type (%d)", type);

        // Free memory
        JSONFree(&json);

        // Return the error code
        return false;
    }
  }

  // Free the memory used by the JSON
  JSONFree(&json);

  // Return the success code
  return true;

}

// Load the Squidlet info from the file 'stream' into the Squad 'that'
// Return true if it could load the info, else false
bool SquadLoadSquidlets(
  Squad* const that,
   FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Discard the current squidlets info if any
  while (GSetNbElem(SquadSquidlets(that)) > 0) {
    SquidletInfo* squidletInfo = GSetPop((GSet*)SquadSquidlets(that));
    SquidletInfoFree(&squidletInfo);
  }

  // Declare a json to load the encoded data
```

```
  JSONNode* json = JSONCreate();

  // Load the whole encoded data
  if (JSONLoad(json, stream) == false) {
    TheSquidErr->_type = PBErrTypeIOError;
    sprintf(TheSquidErr->_msg, "JSONLoad failed");
    JSONFree(&json);
    return false;
  }

  // Decode the data from the JSON
  if (!SquadDecodeAsJSON(that, json)) {
    JSONFree(&json);
    return false;
  }

  // Free the memory used by the JSON
  JSONFree(&json);

  // Return the succes code
  return true;
}

// Load the Squidlet info from the string 'str' into the Squad 'that'
// Return true if it could load the info, else false
bool SquadLoadSquidletsFromStr(
  Squad* const that,
   char* const str) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (str == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'str' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Discard the current squidlets info if any
  while (GSetNbElem(SquadSquidlets(that)) > 0) {
    SquidletInfo* squidletInfo = GSetPop((GSet*)SquadSquidlets(that));
    SquidletInfoFree(&squidletInfo);
  }

  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();

  // Load the whole encoded data
  if (JSONLoadFromStr(json, str) == false) {
    TheSquidErr->_type = PBErrTypeIOError;
    sprintf(TheSquidErr->_msg, "JSONLoadFromStr failed");
    JSONFree(&json);
    return false;
  }

  // Decode the data from the JSON
  if (!SquadDecodeAsJSON(that, json)) {
    JSONFree(&json);
    return false;
  }
```

```
  // Free the memory used by the JSON
  JSONFree(&json);

  // Return the succes code
  return true;
}

// Decode the JSON info of a Squad from the JSON node 'json'
bool SquadDecodeAsJSON(
     Squad* that,
  JSONNode* json) {

  // Get the property _squidlets from the JSON
  JSONNode* prop = JSONProperty(json, "_squidlets");
  if (prop == NULL) {
    TheSquidErr->_type = PBErrTypeInvalidData;
    sprintf(TheSquidErr->_msg, "'_squidlets' not found");
    return false;
  }

  // Get the number of squidlets
  int nbSquidlet = JSONGetNbValue(prop);

  // Loop on squidlets
  for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {

    // Get the JSON node for this squidlet
    JSONNode* propSquidlet = JSONValue(prop, iSquidlet);

    // Get the property _name of the squidlet
    JSONNode* propName = JSONProperty(propSquidlet, "_name");
    if (propName == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'_name' not found for squidlet %d",
        iSquidlet);
      return false;
    }

    // Get the property _ip of the squidlet
    JSONNode* propIp = JSONProperty(propSquidlet, "_ip");
    if (propIp == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'_ip' not found for squidlet %d",
        iSquidlet);
      return false;
    }

    // Get the property _port of the squidlet
    JSONNode* propPort = JSONProperty(propSquidlet, "_port");
    if (propPort == NULL) {
      TheSquidErr->_type = PBErrTypeInvalidData;
      sprintf(TheSquidErr->_msg, "'_port' not found for squidlet %d",
        iSquidlet);
      return false;
    }

    // Create a SquidletInfo
    char* name = JSONLblVal(propName);
    char* ip = JSONLblVal(propIp);
    int port = atoi(JSONLblVal(propPort));
    SquidletInfo* squidletInfo = SquidletInfoCreate(name, ip, port);
```

```
    // Add the squidlet to the set of squidlets
    GSetAppend((GSet*)SquadSquidlets(that), squidletInfo);
  }

  // Return the success code
  return true;
}

// Send the task request 'request' from the Squad 'that' to its
// Squidlet 'squidlet'
// Create a socket, open a connection, ask the squidlet if it can
// execute the task and wait for its reply
// Return true if the request has been accepted by the squidlet,
// false else
bool SquadSendTaskRequest(
                      Squad* const that,
  const SquidletTaskRequest* const request,
               SquidletInfo* const squidlet) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (request == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'request' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // If the requested task is null
  if (request->_type == SquidletTaskType_Null) {

    // Nothing to do
    return true;
  }

  // Close the socket if it was opened
  if (squidlet->_sock != -1) {
    close(squidlet->_sock);
    squidlet->_sock = -1;
  }

  // Create the socket
  int protocol = 0;
  squidlet->_sock = socket(AF_INET, SOCK_STREAM, protocol);

  // If we couldn't create the socket
  if (squidlet->_sock == -1) {

    // Update history
    SquadPushHistory(that, "can't create socket to squidlet:");
    SquadPushHistorySquidletInfo(that, squidlet);

    // Return the failure code
    return false;
```

```
}

// Create the data for the connection to the squidlet from its
// ip and port
struct sockaddr_in remote = {0};
remote.sin_addr.s_addr = inet_addr(squidlet->_ip);
remote.sin_family = AF_INET;
remote.sin_port = htons(squidlet->_port);

// Connect to the squidlet
int retConnect = connect(squidlet->_sock, (struct sockaddr*)&remote,
  sizeof(struct sockaddr_in));

// If the connection failed
if (retConnect == -1) {

  // Close the socket
  close(squidlet->_sock);
  squidlet->_sock = -1;

  // Update history
  SquadPushHistory(that, "can't connect to squidlet:");
  SquadPushHistorySquidletInfo(that, squidlet);

  // Return the failure code
  return false;
}

// Update history
SquadPushHistory(that, "connected to squidlet:");
SquadPushHistorySquidletInfo(that, squidlet);

// Set the timeout of the socket for sending and receiving to 1us
// and allow the reuse of address
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 1;
int retSnd = setsockopt(squidlet->_sock, SOL_SOCKET, SO_SNDTIMEO,
  (char*)&tv, sizeof(tv));
int retRcv = setsockopt(squidlet->_sock, SOL_SOCKET, SO_RCVTIMEO,
  (char*)&tv, sizeof(tv));
int reuse = 1;
int retReuse = setsockopt(squidlet->_sock, SOL_SOCKET, SO_REUSEADDR,
  &reuse,sizeof(int));

// If we couldn't configure the socket
if (retSnd == -1 || retRcv == -1 || retReuse == -1) {

  // Close the socket
  close(squidlet->_sock);
  squidlet->_sock = -1;

  // Update history
  SquadPushHistory(that, "failed to configure socket to squidlet:");
  SquadPushHistorySquidletInfo(that, squidlet);

  // Return the failure code
  return false;
}

// Send the task request
int flags = 0;
```

```
  int retSend = send(squidlet->_sock,
    request, sizeof(SquidletTaskRequest), flags);

  // If we couldn't send the request
  if (retSend == -1) {

    // Close the socket
    close(squidlet->_sock);
    squidlet->_sock = -1;

    // Update history
    SquadPushHistory(that, "failed to send the request to squidlet:");
    SquadPushHistorySquidletInfo(that, squidlet);

    // Return the failure code
    return false;
  }

  // Wait for the reply from the squidlet up to 5s
  char reply = THESQUID_TASKREFUSED;
  time_t maxWait = 5;
  bool retRecv = SocketRecv(
    &(squidlet->_sock), sizeof(reply), &reply, maxWait);

  // If we couldn't receive the reply or the reply timed out or
  // the squidlet refused the task
  if (retRecv == false || reply == THESQUID_TASKREFUSED) {

    // Close the socket
    close(squidlet->_sock);
    squidlet->_sock = -1;

    // Update history
    SquadPushHistory(that, "task refused by squidlet:");
    SquadPushHistorySquidletInfo(that, squidlet);

    // Return the failure code
    return false;
  }

  // Update history
  SquadPushHistory(that, "request:");
  SquadPushHistorySquidletTaskRequest(that, request);
  SquadPushHistory(that, "accepted by squidlet:");
  SquadPushHistorySquidletInfo(that, squidlet);

  // Return the success code
  return true;
}

// Add a dummy task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
void SquadAddTask_Dummy(
         Squad* const that,
  const unsigned long id,
         const time_t maxWait) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Prepare the data as JSON
  char buffer[100];
  memset(buffer, 0, 100);
  sprintf(buffer, "{\"v\":\"%lu\"}", id);
  unsigned long subid = 0;

  // Create the new task
  SquidletTaskRequest* task = SquidletTaskRequestCreate(
    SquidletTaskType_Dummy, id, subid, buffer, maxWait);

  // Add the new task to the set of task to execute
  GSetAppend((GSet*)SquadTasks(that), task);
}

// Add a benchmark task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
// Artificially set the size of the data for the task to 'payloadSize'
// bytes
// The benchmark function used is sorting 'nb' times a set of numbers
// cf TheSquidBenchmark()
void SquadAddTask_Benchmark(
         Squad* const that,
  const unsigned long id,
         const time_t maxWait,
   const unsigned int nb,
   const unsigned int payloadSize) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Create a dummy buffer of length 'payloadSize'
  char* data = PBErrMalloc(TheSquidErr, payloadSize + 1);
  memset(data, ' ', payloadSize);
  data[payloadSize] = '\0';

  // Convert the arguments into strings to get their length
  char bufferNb[100];
  sprintf(bufferNb, "%d", nb);
  int bufferNbLen = strlen(bufferNb);
  char bufferId[100];
  sprintf(bufferId, "%ld", id);
  int bufferIdLen = strlen(bufferId);

  // Get the length of the json data without values
  int jsonFormatLen = \
    strlen("{\"id\":\"\",\"nb\":\"\",\"payloadSize\":\"\"}");

  // Get the length of the json data with values
  int bufferLength =
    jsonFormatLen + bufferIdLen + bufferNbLen + strlen(data) + 1;

  // Allocate memory
```

```
  char* buffer = PBErrMalloc(TheSquidErr, bufferLength);

  // Create the JSON data
  sprintf(buffer, "{\"id\":\"%s\",\"nb\":\"%s\",\"payloadSize\":\"%s\"}",
    bufferId, bufferNb, data);
  free(data);

  // Create the new task
  unsigned long subid = 0;
  SquidletTaskRequest* task = SquidletTaskRequestCreate(
    SquidletTaskType_Benchmark, id, subid, buffer, maxWait);
  free(buffer);

  // Add the new task to the set of task to execute
  GSetAppend((GSet*)SquadTasks(that), task);
}

// Add a POV-Ray task uniquely identified by its 'id' to the list of
// task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
// The POV-Ray task is described by the POV-Ray ini file 'ini' which
// must include at least the following options:
// Input_File_Name=<script.pov>
// Width=<width>
// Height=<height>
// Output_File_Name=<image.tga>
// The output format of the image must be TGA
// Video are not supported
// The image is splitted into as many squared fragments as
// SquadGetNbSquidlets(that)^2, but the size of the fragment is clipped
// to [sizeMinFragment, sizeMaxFragment]
// The size of the fragments are corrected to fit the size of the image
// if it's not a squared image
// Fragments are rendered on the squidlets in random order
// The final image is updated each time a fragment has been
// rendered
// The random generator must have been initialised before calling this
// function
void SquadAddTask_PovRay(
         Squad* const that,
  const unsigned long id,
         const time_t maxWait,
    const char* const ini,
   const unsigned int sizeMinFragment,
   const unsigned int sizeMaxFragment) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (ini == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'ini' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Init variables to memorize the dimensions and path of the final
  // output image
  unsigned long width = 0;
```

```
unsigned long height = 0;
char* outImgPath = NULL;

// Open the ini file
FILE* fp = fopen(ini, "r");

// If we couldn't open the ini file
if (fp == NULL) {

  // Report the error
  TheSquidErr->_type = PBErrTypeInvalidArg;
  sprintf(TheSquidErr->_msg, "Can't open %s", ini);
  PBErrCatch(TheSquidErr);

// Else, we could open the ini file
} else {

  // Read the ini file line by line
  char oneLine[THESQUID_MAXPAYLOADSIZE];
  while(fgets(oneLine, THESQUID_MAXPAYLOADSIZE, fp)) {

    // If we are on the line defining the width
    // and there is actually a value for the width
    if (strstr(oneLine, "Width=") &&
      strlen(oneLine) > 7) {

      // Decode the width
      sscanf(oneLine + 6, "%lu", &width);

    // If we are on the line defining the height
    // and there is actually a value for the height
    } else if (strstr(oneLine, "Height=") &&
      strlen(oneLine) > 8) {

      // Decode the height
      sscanf(oneLine + 7, "%lu", &height);

    // If we are on the line defining the path to the output file
    // and there is actually a value for the path
    } else if (strstr(oneLine, "Output_File_Name=") &&
      strlen(oneLine) > 17) {

      // Copy the path
      outImgPath = strdup(oneLine + 17);

      // Remove the return line
      if (outImgPath[strlen(outImgPath) - 1] == '\n') {
        outImgPath[strlen(outImgPath) - 1] = '\0';
      }

      // Make sure the output file doesn't exists
      char* cmd = PBErrMalloc(TheSquidErr, sizeof(char) *
        (strlen("rm -f ") + strlen(outImgPath) + 1));
      sprintf(cmd, "rm -f %s", outImgPath);
      int ret = system(cmd);
      free(cmd);
      (void)ret;
    }
  }
  fclose(fp);
}
```

```
// Check arguments
if (width == 0 || height == 0 || outImgPath == NULL) {
  TheSquidErr->_type = PBErrTypeInvalidArg;
  sprintf(TheSquidErr->_msg, "Can't decode arguments from %s", ini);
  PBErrCatch(TheSquidErr);
}

// Get the nb of squidlets, force it to 1 at least to avoid division
// by zero
unsigned long nbSquidlets = MAX(1, SquadGetNbSquidlets(that));

// Declare a variable to memorize the size of one fragment
unsigned long sizeFrag[2];

// Get the size of one fragment
sizeFrag[0] =
  MAX(sizeMinFragment, MIN(sizeMaxFragment, width / nbSquidlets));
sizeFrag[1] =
  MAX(sizeMinFragment, MIN(sizeMaxFragment, width / nbSquidlets));

// Declare a variable to memorize the nb of fragments
unsigned long nbFrag[2];

// Get the nb of fragments
nbFrag[0] = width / sizeFrag[0];
nbFrag[1] = height / sizeFrag[1];

// If the size of the image is not dividable we have to had one more
// fragment to render the last partial fragment
if (sizeFrag[0] * nbFrag[0] < width)
  ++(nbFrag[0]);
if (sizeFrag[1] * nbFrag[1] < height)
  ++(nbFrag[1]);

// Create a temporary GSet where to add the tasks to be able to
// shuffle it independantly of the eventual other task in the Squad
// Shuffling the task to render the fragments in random order
GSet set = GSetCreateStatic();

// Create the tasks for each fragment
for (unsigned long i = 0; i < nbFrag[0]; ++i) {
  for (unsigned long j = 0; j < nbFrag[1]; ++j) {

    // Get the id of the task
    unsigned long taskId = i * nbFrag[1] + j;

    // Get the coordinates of the fragment
    // Pov-Ray starts counting at 1, so the top left is (1,1)
    unsigned long top = j * sizeFrag[1] + 1;
    unsigned long left = i * sizeFrag[0] + 1;
    unsigned long bottom = (j + 1) * sizeFrag[1] + 1;
    if (bottom > height)
      bottom = height;
    unsigned long right = (i + 1) * sizeFrag[0] + 1;
    if (right > width)
      right = width;

    // Get the name of the output file for this fragment
    int len = strlen(outImgPath);
    char* tga = PBErrMalloc(TheSquidErr, len + 6);
    memset(tga, 0, len + 6);
    strcpy(tga, outImgPath);
```

```
      sprintf(tga + len - 4, "-%05lu.tga", taskId);

      // Prepare the data as JSON
      char buffer[THESQUID_MAXPAYLOADSIZE];
      memset(buffer, 0, THESQUID_MAXPAYLOADSIZE);
      sprintf(buffer,
        "{\"id\":\"%lu\",\"subid\":\"%lu\",\"ini\":\"%s\","
        "\"tga\":\"%s\",\"top\":\"%lu\",\"left\":\"%lu\","
        "\"bottom\":\"%lu\",\"right\":\"%lu\",\"width\":\"%lu\","
        "\"height\":\"%lu\",\"outTga\":\"%s\"}",
        id, taskId, ini, tga, top, left, bottom, right, width, height,
        outImgPath);

      // Add the new task to the set of task to execute
      SquidletTaskRequest* task = SquidletTaskRequestCreate(
        SquidletTaskType_PovRay, id, taskId, buffer, maxWait);
      GSetAppend(&set, task);

      // Free memory
      free(tga);
    }
  }

  // Shuffle the task and add them to the set of tasks
  GSetShuffle(&set);
  GSetAppendSet((GSet*)SquadTasks(that), &set);

  // Free memory
  GSetFlush(&set);
  free(outImgPath);

}

// Add a neuranet evaluation task uniquely identified by its 'id' to
// the list of task to execute by the squad 'that'
// The task will have a maximum of 'maxWait' seconds to complete from
// the time it's accepted by the squidlet or it will be considered
// as failed
void SquadAddTask_EvalNeuraNet(
         Squad* const that,
  const unsigned long id,
        const time_t maxWait,
    const char* const datasetPath,
    const char* const workingDirPath,
 const VecLong* const nnids,
          const float curBest,
           const long cat) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Prepare the data as JSON
  unsigned long subid = 0;
  char nnidsStr[THESQUID_MAXPAYLOADSIZE];
  JSONNode* json = VecEncodeAsJSON(nnids);
  JSONSaveToStr(json, nnidsStr, THESQUID_MAXPAYLOADSIZE, true);
  char buffer[THESQUID_MAXPAYLOADSIZE];
  JSONFree(&json);
  memset(buffer, 0, THESQUID_MAXPAYLOADSIZE);
```

```
  sprintf(buffer,
    "{\"id\":\"%lu\",\"subid\":\"%lu\",\"dataset\":\"%s\","
    "\"workingDir\":\"%s\",\"nnids\":%s,\"best\":\"%f\","
    "\"cat\":\"%ld\"}",
    id, subid, datasetPath, workingDirPath, nnidsStr, curBest, cat);

  // Create the new task
  SquidletTaskRequest* task = SquidletTaskRequestCreate(
    SquidletTaskType_EvalNeuranet, id, subid, buffer, maxWait);

  // Add the new task to the set of task to execute
  GSetAppend((GSet*)SquadTasks(that), task);
}

// Send a request from the Squad 'that' to reset the stats of the
// Squidlet 'squidlet'
// Return true if the request was successfull, else false
bool SquadRequestSquidletToResetStats(
         Squad* const that,
  SquidletInfo* const squidlet) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

    // Prepare the data as JSON
    char* buffer = "{\"id\":\"0\"}";

    // Create the task
    unsigned long id = 0;
    unsigned long subId = 0;
    time_t maxWait = 60;
    SquidletTaskRequest* task = SquidletTaskRequestCreate(
      SquidletTaskType_ResetStats, id, subId, buffer, maxWait);

    // Request the execution of the task by the squidlet
    bool ret = SquadSendTaskRequest(that, task, squidlet);
    if (ret) {

      // If the squidlet accepted to execute the task
      ret = SquadSendTaskData(that, squidlet, task);
    }

    // Free memory
    free(task);

    // Return the success of the request
    return ret;
}

// Send a request from the Squad 'that' to reset the stats of all its
// non-busy Squidlets
// Return true if all the request were successfull, else false
bool SquadRequestAllSquidletToResetStats(
```

```
  Squad* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a variable to memorize if all the requests were successfull
  bool flag = true;

  // If the Squad has non-busy squidlets
  if (SquadGetNbSquidlets(that) > 0) {

    // Loop on the squidlets
    GSetIterForward iter =
      GSetIterForwardCreateStatic(SquadSquidlets(that));
    do {

      // Get the current quidlet
      SquidletInfo* squidlet = GSetIterGet(&iter);

      // Request the reset of the stats of this squidlet
      // and update the flag with the returned flag
      flag &= SquadRequestSquidletToResetStats(that, squidlet);

    } while (GSetIterStep(&iter));
  }

  // Return the flag memorizing if all the requests were successfull
  return flag;
}

// Send the data associated to the task request 'task' from the Squad
// 'that' to the Squidlet 'squidlet'
// First, send the size in byte of the data, then send the data
// Return true if the data could be sent, false else
// The size of the data must be less than THESQUID_MAXPAYLOADSIZE bytes
bool SquadSendTaskData(
                Squad* const that,
         SquidletInfo* const squidlet,
  SquidletTaskRequest* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Send the task data size
```

```
  int flags = 0;
  size_t len = strlen(task->_data);
  int ret = send(squidlet->_sock, (char*)&len, sizeof(size_t), flags);

  // If we couldn't send the data size
  if (ret == -1) {

    // Update history
    SquadPushHistory(that, "couldn't send task data size %d", len);

    return false;

  // Else, we could send the data size
  } else {

    // Update history
    SquadPushHistory(that, "sent task data size %d to squidlet:", len);
    SquadPushHistorySquidletInfo(that, squidlet);

  }

  // Memorize the start time for the statistics
  struct timeval start;
  gettimeofday(&start, NULL);

  // Send the task data
  ret = send(squidlet->_sock, task->_data, len, flags);

  // If we couldn't send the data
  if (ret == -1) {

    // Update history
    SquadPushHistory(that, "couldn't send task data");

    return false;

  // Else, we could send the data
  } else {

    // Get the time to send the data
    struct timeval stop;
    gettimeofday(&stop, NULL);
    float deltams = (float)(stop.tv_sec - start.tv_sec) * 1000.0 +
      (float)(stop.tv_usec - start.tv_usec) / 1000.0;

    // Update the stats about transfer time
    SquidletInfoStatsUpdateTimeTransfer(
      (SquidletInfoStats*)SquidletInfoStatistics(squidlet),
      deltams, len);

  }

  // Return the success code
  return true;
}

// Update the statistics about the transfer time of the SquidletInfoStats
// 'that' given that it took 'deltams' millisecond to send 'len' bytes
// of data
void SquidletInfoStatsUpdateTimeTransfer(
  SquidletInfoStats* const that,
              const float deltams,
```

```
              const size_t len) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Get the delay per byte
  float deltamsUnit = deltams / (float)len;

  // Update the max
  if (that->_timeTransferSquadSquidMs[0] > deltamsUnit) {
    that->_timeTransferSquadSquidMs[0] = deltamsUnit;
  }

  // Update the sliding average
  // If the number of completed tasks is less than the length of the
  // sliding average and there are actually completed tasks
  if (that->_nbTaskComplete <= SQUID_RANGEAVGSTAT &&
    that->_nbTaskComplete > 0) {

    // Update the sliding average
    that->_timeTransferSquadSquidMs[1] =
      (that->_timeTransferSquadSquidMs[1] *
      (float)(that->_nbTaskComplete - 1) +
      deltamsUnit) / (float)(that->_nbTaskComplete);

  // Else, the number of completed tasks is more than the length of the
  // sliding average or there are no completed tasks yet
  } else {

    // Update the sliding average
    that->_timeTransferSquadSquidMs[1] =
      (that->_timeTransferSquadSquidMs[1] *
      (float)(SQUID_RANGEAVGSTAT - 1) +
      deltamsUnit) / (float)SQUID_RANGEAVGSTAT;
  }

  // Update the min
  if (that->_timeTransferSquadSquidMs[2] < deltamsUnit) {
    that->_timeTransferSquadSquidMs[2] = deltamsUnit;
  }
}

// Try to receive the result from the running task 'runningTask'
// If the result is ready it is stored in the _bufferResult of the
// SquidletTaskRequest of the 'runningTask'
// If the size of the result data is not ready and couldn't be received
// give up immediately
// If the size of the result data has been received, wait for
// (5 + sizeData / 100) seconds maximum to receive the data
// First receive the size of the result data, then send an
// acknowledgement signal to the squidlet for the size of data,
// then receive the data, and finally send an acknowledgement to the
// squidlet for the data
// Return true if it could receive the result data, false else
bool SquadReceiveTaskResult(
             Squad* const that,
  SquadRunningTask* const runningTask) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (runningTask == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'runningTask' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Declare a variable to memorize if we have received the result
  bool receivedFlag = false;

  // Declare a variable to memorize the size in byte of the result data
  size_t sizeResultData = 0;

  // Shortcuts
  SquidletInfo* squidlet = runningTask->_squidlet;
  SquidletTaskRequest* task = runningTask->_request;

  // Make sure the buffer to receive the task is empty
  if (task->_bufferResult != NULL) {
    free(task->_bufferResult);
    task->_bufferResult = NULL;
  }

  // Try to receive the size of the reply from the squidlet
  // and give up immediately
  if (SocketRecv(&(squidlet->_sock), sizeof(size_t),
    (char*)&sizeResultData, 0)) {

    // If we could get the size it means the result is ready
    if (sizeResultData > 0) {

      // Update history
      SquadPushHistory(that,
        "received the size of result from squidlet:");
      SquadPushHistorySquidletInfo(that, squidlet);

      // Send the acknowledgement of received size of result
      char ack = 1;
      int flags = 0;
      (void)send(squidlet->_sock, &ack, sizeof(char), flags);

      // Update history
      SquadPushHistory(that,
        "send ack of received size of result data to squidlet:");
      SquadPushHistorySquidletInfo(that, squidlet);

      // Allocate memory for the result data
      task->_bufferResult = PBErrMalloc(TheSquidErr, sizeResultData + 1);
      memset(task->_bufferResult, 0, sizeResultData + 1);

      // Wait to receive the result data with a time limit proportional
      // to the size of result data
      int timeOut = 5 + (int)round((float)sizeResultData / 100.0);

      // If we coudln't received the result data
      if (!SocketRecv(&(squidlet->_sock), sizeResultData,
        task->_bufferResult, timeOut)) {
```

```
        // Free the memory allocated to the result buffer
        free(task->_bufferResult);
        task->_bufferResult = NULL;

        // Update history
        SquadPushHistory(that,
          "couldn't received result data from squidlet:");
        SquadPushHistorySquidletInfo(that, squidlet);
        SquadPushHistory(that,"waited for %ds", timeOut);

      } else {

        // Set the flag to memorized we have received the result
        receivedFlag = true;

        // Send the acknowledgement of received result
        (void)send(squidlet->_sock, &ack, 1, flags);

        // Update history
        SquadPushHistory(that, "received result data from squidlet:");
        SquadPushHistorySquidletInfo(that, squidlet);
        SquadPushHistory(that,"size result data %d", sizeResultData);

      }

    // Else, we couldn't get the size, it means the result is not
    // ready yet
    } else {

      // Update history
      SquadPushHistory(that,
        "received a null size of result from squidlet:");
      SquadPushHistorySquidletInfo(that, squidlet);

    }
  }

  // Return the flag memorizing if we have received the result
  return receivedFlag;
}

// Request the execution of a task on a squidlet for the squad 'that'
// Return true if the request was successfull, false else
bool SquadSendTaskOnSquidlet(
                Squad* const that,
         SquidletInfo* const squidlet,
  SquidletTaskRequest* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
```

```
  }
#endif

  // Request the execution of the task by the squidlet
  bool ret = SquadSendTaskRequest(that, task, squidlet);

  // If the request was successfull
  if (ret == true) {

    // Send the task's data to the squidlet
    ret = SquadSendTaskData(that, squidlet, task);

    // If we could send the task's data
    if (ret == true) {

      // Create a new running task and add it to the set of running tasks
      SquadRunningTask* runningTask =
        SquadRunningTaskCreate(task, squidlet);
      GSetAppend((GSet*)SquadRunningTasks(that), runningTask);

      // Update history
      SquadPushHistory(that, "created running task:");
      SquadPushHistorySquadRunningTask(that, runningTask);

    // Else, we couldn't send the task data
    } else {

      // Update history
      SquadPushHistory(that, "couldn't send data to squidlet:");
      SquadPushHistorySquidletInfo(that, squidlet);

    }

  // Else, the request of execution wasn't successfull
  } else {

    // Update history
    SquadPushHistory(that, "task refused by squidlet:");
    SquadPushHistorySquidletInfo(that, squidlet);

  }

  // Return the result
  return ret;
}

// Step the Squad 'that', i.e. tries to affect the remaining tasks to
// available Squidlets and check for completion of running tasks.
// Return the GSet of the completed SquadRunningTask at this step
// Non blocking, if there is no task to compute or no squidlet
// available, and no task completed, do nothing and return an empty set
GSetSquadRunningTask SquadStep(
  Squad* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Create the set of completed tasks
  GSetSquadRunningTask completedTasks = \
```

```
  GSetSquadRunningTaskCreateStatic();

// If there are running tasks
if (SquadGetNbRunningTasks(that) > 0L) {

  // Declare a flag to manage the removing of tasks during the loop
  // on running tasks
  bool flag = false;

  // Loop on running tasks
  GSetIterForward iter =
    GSetIterForwardCreateStatic((GSet*)SquadRunningTasks(that));
  do {

    // Reinit the flag to manage the removing of tasks during the loop
    flag = false;

    // Get the running tasks
    SquadRunningTask* runningTask = GSetIterGet(&iter);

    // Request the result for this task
    bool complete = SquadReceiveTaskResult(that, runningTask);

    // If the task is complete
    if (complete == true) {

      // Update history
      SquadPushHistory(that, "completed task:");
      SquadPushHistorySquadRunningTask(that, runningTask);

      // Post process the completed task
      SquadProcessCompletedTask(that, runningTask);

      // Put back the squidlet in the set of squidlets
      GSetAppend((GSet*)SquadSquidlets(that), runningTask->_squidlet);

      // Add the task to the set of completed tasks
      GSetAppend(&completedTasks, runningTask);

      // Remove the task from the running tasks
      flag = GSetIterRemoveElem(&iter);

    // Else, the task is not complete
    // If we've been waiting too long for this task
    } else if (time(NULL) - runningTask->_startTime >
      runningTask->_request->_maxWaitTime) {

      // Update history
      SquadPushHistory(that, "gave up task:");
      SquadPushHistorySquadRunningTask(that, runningTask);

      // Put back the squidlet in the set of squidlets
      GSetAppend((GSet*)SquadSquidlets(that), runningTask->_squidlet);
      runningTask->_squidlet = NULL;

      // Put back the task to the set of tasks
      SquadTryAgainTask(that, runningTask->_request);
      runningTask->_request = NULL;

      // Remove the task from the running tasks
      flag = GSetIterRemoveElem(&iter);
```

```
      // Free memory
      SquadRunningTaskFree(&runningTask);

    // Else, the task is not complete and we can wait more for it
    } else {

      // Update history
      SquadPushHistory(that, "waiting for task:");
      SquadPushHistorySquadRunningTask(that, runningTask);

    }

  } while (flag || GSetIterStep(&iter));

}

// If there are tasks to execute and available squidlet
if (SquadGetNbRemainingTasks(that) > 0L &&
  SquadGetNbSquidlets(that) > 0L) {

  // Declare a flag to manage the removing of tasks during the loop
  // on running tasks
  bool flag = false;

  // Loop on squidlets
  GSetIterForward iter =
    GSetIterForwardCreateStatic((GSet*)SquadSquidlets(that));
  do {

    // Reinit the flag to manage the removing of tasks during the loop
    flag = false;

    // Get the squidlet
    SquidletInfo* squidlet = GSetIterGet(&iter);

    // Get the next task to complete
    SquidletTaskRequest* task = GSetPop((GSet*)SquadTasks(that));

    // If there is a task to complete
    if (task != NULL) {

      // Request the task on the squidlet
      bool ret = SquadSendTaskOnSquidlet(that, squidlet, task);

      // If the squidlet accepted the task
      if (ret == true) {

        // Remove the squidlet from the available squidlet
        flag = GSetIterRemoveElem(&iter);

      // Else, the squidlet refused the task or the data couldn't be
      // sent
      } else {

        // Put back the task in the set
        GSetPush((GSet*)SquadTasks(that), task);
      }
    }
  } while (flag || GSetIterStep(&iter));
}

// Update the TextOMeter if necessary
```

```
  if (SquadGetFlagTextOMeter(that) == true) {
    SquadUpdateTextOMeter(that);
  }

  // Return the set of completed tasks
  return completedTasks;
}

// Process the completed 'task' with the Squad 'that' after its
// reception in SquadStep()
void SquadProcessCompletedTask(
             Squad* const that,
  SquadRunningTask* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Call the appropriate function based on the type of the task
  switch (task->_request->_type) {
    case SquidletTaskType_Dummy:
      break;
    case SquidletTaskType_Benchmark:
      break;
    case SquidletTaskType_PovRay:
      SquadProcessCompletedTask_PovRay(that, task->_request);
      break;
    case SquidletTaskType_ResetStats:
      // Nothing to do
      break;
    case SquidletTaskType_EvalNeuranet:
      break;
    default:
      break;
  }

  // Update the stats about the squidlets
  SquidletInfoUpdateStats(task->_squidlet, task->_request);
}

// Update the statitics of the SquidletInfo 'that' with the result of
// the 'task'
void SquidletInfoUpdateStats(
         SquidletInfo* const that,
  SquidletTaskRequest* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
```

```
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a variable to decode the JSON data from the completed task
  JSONNode* jsonResult = JSONCreate();

  // Decode the JSON data from the completed task
  bool ret = JSONLoadFromStr(jsonResult, task->_bufferResult);

  // If we could decode the JSON
  if (ret == true) {

    // Get the properties
    JSONNode* propNbAcceptedConnection = \
      JSONProperty(jsonResult, "nbAcceptedConnection");
    JSONNode* propNbAcceptedTask = \
      JSONProperty(jsonResult, "nbAcceptedTask");
    JSONNode* propNbRefusedTask = \
      JSONProperty(jsonResult, "nbRefusedTask");
    JSONNode* propNbFailedReceptTaskData = \
      JSONProperty(jsonResult, "nbFailedReceptTaskData");
    JSONNode* propNbFailedReceptTaskSize = \
      JSONProperty(jsonResult, "nbFailedReceptTaskSize");
    JSONNode* propNbSentResult = \
      JSONProperty(jsonResult, "nbSentResult");
    JSONNode* propNbFailedSendResult = \
      JSONProperty(jsonResult, "nbFailedSendResult");
    JSONNode* propNbFailedSendResultSize = \
      JSONProperty(jsonResult, "nbFailedSendResultSize");
    JSONNode* propNbFailedReceptAck = \
      JSONProperty(jsonResult, "nbFailedReceptAck");
    JSONNode* propNbTaskComplete = \
      JSONProperty(jsonResult, "nbTaskComplete");
    JSONNode* propTimeToProcessMs = \
      JSONProperty(jsonResult, "timeToProcessMs");
    JSONNode* propTimeWaitedTaskMs = \
      JSONProperty(jsonResult, "timeWaitedTaskMs");
    JSONNode* propTimeWaitedAckMs = \
      JSONProperty(jsonResult, "timeWaitedAckMs");
    JSONNode* propTemperature = \
      JSONProperty(jsonResult, "temperature");
    JSONNode* propTimeTransferSquidSquad = \
      JSONProperty(jsonResult, "timeTransferSquidSquadMs");

    // If all the properties are present
    if (propNbAcceptedConnection != NULL &&
      propNbAcceptedTask != NULL &&
      propNbRefusedTask != NULL &&
      propNbFailedReceptTaskData != NULL &&
      propNbFailedReceptTaskSize != NULL &&
      propNbSentResult != NULL &&
      propNbFailedSendResult != NULL &&
      propNbFailedSendResultSize != NULL &&
      propNbFailedReceptAck != NULL &&
      propNbTaskComplete != NULL &&
      propTimeToProcessMs != NULL &&
      propTimeWaitedTaskMs != NULL &&
      propTimeWaitedAckMs != NULL &&
      propTemperature != NULL &&
      propTimeTransferSquidSquad != NULL) {
```

```
// Update the stats with the received info from the Squidlet
SquidletInfoStats* stats =
  (SquidletInfoStats*)SquidletInfoStatistics(that);
stats->_nbAcceptedConnection =
  atol(JSONLblVal(propNbAcceptedConnection));
stats->_nbAcceptedTask =
  atol(JSONLblVal(propNbAcceptedTask));
stats->_nbRefusedTask =
  atol(JSONLblVal(propNbRefusedTask));
stats->_nbFailedReceptTaskData =
  atol(JSONLblVal(propNbFailedReceptTaskData));
stats->_nbFailedReceptTaskSize =
  atol(JSONLblVal(propNbFailedReceptTaskSize));
stats->_nbSentResult =
  atol(JSONLblVal(propNbSentResult));
stats->_nbFailedSendResult =
  atol(JSONLblVal(propNbFailedSendResult));
stats->_nbFailedSendResultSize =
  atol(JSONLblVal(propNbFailedSendResultSize));
stats->_nbFailedReceptAck =
  atol(JSONLblVal(propNbFailedReceptAck));
stats->_nbTaskComplete =
  atol(JSONLblVal(propNbTaskComplete));

// If its not the first completed task
if (stats->_nbTaskComplete > 1) {

  // Update the statistics about time
  float timeToProcessMs =
    atof(JSONLblVal(propTimeToProcessMs));
  if (stats->_timeToProcessMs[0] > timeToProcessMs) {
    stats->_timeToProcessMs[0] = timeToProcessMs;
  }
  if (stats->_nbTaskComplete <= SQUID_RANGEAVGSTAT) {
    stats->_timeToProcessMs[1] =
      (stats->_timeToProcessMs[1] *
      (float)(stats->_nbTaskComplete - 1) +
      timeToProcessMs) /
      (float)(stats->_nbTaskComplete);
  } else {
    stats->_timeToProcessMs[1] =
      (stats->_timeToProcessMs[1] *
      (float)(SQUID_RANGEAVGSTAT - 1) +
      timeToProcessMs) /
      (float)SQUID_RANGEAVGSTAT;
  }
  if (stats->_timeToProcessMs[2] < timeToProcessMs) {
    stats->_timeToProcessMs[2] = timeToProcessMs;
  }

  float timeWaitedTaskMs =
    atof(JSONLblVal(propTimeWaitedTaskMs));
  if (stats->_timeWaitedTaskMs[0] > timeWaitedTaskMs) {
    stats->_timeWaitedTaskMs[0] = timeWaitedTaskMs;
  }
  if (stats->_nbTaskComplete <= SQUID_RANGEAVGSTAT) {
    stats->_timeWaitedTaskMs[1] =
      (stats->_timeWaitedTaskMs[1] *
      (float)(stats->_nbTaskComplete - 1) +
      timeWaitedTaskMs) /
      (float)(stats->_nbTaskComplete);
  } else {
```

```
    stats->_timeWaitedTaskMs[1] =
      (stats->_timeWaitedTaskMs[1] *
      (float)(SQUID_RANGEAVGSTAT - 1) +
      timeWaitedTaskMs) /
      (float)SQUID_RANGEAVGSTAT;
  }
  if (stats->_timeWaitedTaskMs[2] < timeWaitedTaskMs) {
    stats->_timeWaitedTaskMs[2] = timeWaitedTaskMs;
  }

  float timeWaitedAckMs =
    atof(JSONLblVal(propTimeWaitedAckMs));
  if (stats->_timeWaitedAckMs[0] > timeWaitedAckMs) {
    stats->_timeWaitedAckMs[0] = timeWaitedAckMs;
  }
  if (stats->_nbTaskComplete <= SQUID_RANGEAVGSTAT) {
    stats->_timeWaitedAckMs[1] =
      (stats->_timeWaitedAckMs[1] *
      (float)(stats->_nbTaskComplete - 1) +
      timeWaitedAckMs) /
      (float)(stats->_nbTaskComplete);
  } else {
    stats->_timeWaitedAckMs[1] =
      (stats->_timeWaitedAckMs[1] *
      (float)(SQUID_RANGEAVGSTAT - 1) +
      timeWaitedAckMs) /
      (float)SQUID_RANGEAVGSTAT;
  }
  if (stats->_timeWaitedAckMs[2] < timeWaitedAckMs) {
    stats->_timeWaitedAckMs[2] = timeWaitedAckMs;
  }

  float temperature =
    atof(JSONLblVal(propTemperature));
  if (stats->_temperature[0] > temperature) {
    stats->_temperature[0] = temperature;
  }
  if (stats->_nbTaskComplete <= SQUID_RANGEAVGSTAT) {
    stats->_temperature[1] =
      (stats->_temperature[1] *
      (float)(stats->_nbTaskComplete - 1) +
      temperature) /
      (float)(stats->_nbTaskComplete);
  } else {
    stats->_temperature[1] =
      (stats->_temperature[1] *
      (float)(SQUID_RANGEAVGSTAT - 1) +
      temperature) /
      (float)SQUID_RANGEAVGSTAT;
  }
  if (stats->_temperature[2] < temperature) {
    stats->_temperature[2] = temperature;
  }

  float timeTransferSquidSquadMs =
    atof(JSONLblVal(propTimeTransferSquidSquad));
  SquidletInfoStatsUpdateTimeTransfer(
    stats, timeTransferSquidSquadMs, 1);

// Else, this is the first completed task
} else {
```

```
        float timeToProcessMs =
          atof(JSONLblVal(propTimeToProcessMs));
        stats->_timeToProcessMs[0] = timeToProcessMs;
        stats->_timeToProcessMs[1] = timeToProcessMs;
        stats->_timeToProcessMs[2] = timeToProcessMs;

        float timeWaitedTaskMs =
          atof(JSONLblVal(propTimeWaitedTaskMs));
        stats->_timeWaitedTaskMs[0] = timeWaitedTaskMs;
        stats->_timeWaitedTaskMs[1] = timeWaitedTaskMs;
        stats->_timeWaitedTaskMs[2] = timeWaitedTaskMs;

        float timeWaitedAckMs =
          atof(JSONLblVal(propTimeWaitedAckMs));
        stats->_timeWaitedAckMs[0] = timeWaitedAckMs;
        stats->_timeWaitedAckMs[1] = timeWaitedAckMs;
        stats->_timeWaitedAckMs[2] = timeWaitedAckMs;

        float temperature =
          atof(JSONLblVal(propTemperature));
        stats->_temperature[0] = temperature;
        stats->_temperature[1] = temperature;
        stats->_temperature[2] = temperature;

        float timeTransferSquidSquadMs =
          atof(JSONLblVal(propTimeTransferSquidSquad));
        stats->_timeTransferSquidSquadMs[0] = timeTransferSquidSquadMs;
        stats->_timeTransferSquidSquadMs[1] = timeTransferSquidSquadMs;
        stats->_timeTransferSquidSquadMs[2] = timeTransferSquidSquadMs;

      }

    }

  }

}

// Process the completed Pov-Ray 'task' with the Squad 'that'
void SquadProcessCompletedTask_PovRay(
                Squad* const that,
  SquidletTaskRequest* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare variables to decode the JSON data from the request
  // and the completed task
  JSONNode* jsonRequest = JSONCreate();
  JSONNode* jsonResult = JSONCreate();

  // Decode the JSON data from the request and the completed task
  bool ret = JSONLoadFromStr(jsonResult, task->_bufferResult);
```

```
ret &= JSONLoadFromStr(jsonRequest, task->_data);

// If we could decode the JSON
if (ret == true) {

  // Get the necessary properties
  JSONNode* propResultImg = JSONProperty(jsonRequest, "outTga");
  JSONNode* propWidth = JSONProperty(jsonRequest, "width");
  JSONNode* propHeight = JSONProperty(jsonRequest, "height");
  JSONNode* propTga = JSONProperty(jsonResult, "tga");
  JSONNode* propTop = JSONProperty(jsonResult, "top");
  JSONNode* propLeft = JSONProperty(jsonResult, "left");
  JSONNode* propRight = JSONProperty(jsonResult, "right");
  JSONNode* propBottom = JSONProperty(jsonResult, "bottom");

  // If the necessary properties were present
  if (propTga != NULL && propTop != NULL && propLeft != NULL &&
    propRight != NULL && propBottom != NULL && propResultImg != NULL) {

    // Load the result image
    GenBrush* resultImg =
      GBCreateFromFile(JSONLblVal(propResultImg));

    // If the result image doesn't exists
    if (resultImg == NULL) {

      // Create the result image
      VecShort2D dim = VecShortCreateStatic2D();
      VecSet(&dim, 0, atoi(JSONLblVal(propWidth)));
      VecSet(&dim, 1, atoi(JSONLblVal(propHeight)));
      resultImg = GBCreateImage(&dim);
      GBSetFileName(resultImg, JSONLblVal(propResultImg));
    }

    // Load the fragment
    GenBrush* fragment =
      GBCreateFromFile(JSONLblVal(propTga));

    // If we could load the fragment
    if (fragment != NULL) {

      // Crop the relevant portion of the image
      // Pov-Ray has its coordinate system origin at the top left of
      // the image, while GenBrush has its own at the bottom left
      // Pov-Ray starts counting at 1, so the top left is (1,1)
      VecShort2D dim = VecShortCreateStatic2D();
      VecSet(&dim, 0,
        atoi(JSONLblVal(propRight)) -
        atoi(JSONLblVal(propLeft)) + 1);
      VecSet(&dim, 1,
        atoi(JSONLblVal(propBottom)) -
        atoi(JSONLblVal(propTop)) + 1);
      VecShort2D posLR = VecShortCreateStatic2D();
      VecSet(&posLR, 0, atoi(JSONLblVal(propLeft)) - 1);
      VecSet(&posLR, 1,
        atoi(JSONLblVal(propHeight)) -
        atoi(JSONLblVal(propBottom)));

      // Add the fragment to the result image
      GBCopyFragment(fragment, resultImg, &posLR, &posLR, &dim);

      // Save the result image
```

```
        GBRender(resultImg);

        // Free memory
        GBFree(&fragment);

      // Else, we couldn't load the fragment
      } else {

        // Update history
        SquadPushHistory(that, "Couldn't read the fragment (%s)",
          GenBrushErr->_msg);
      }

      // Free memory
      GBFree(&resultImg);

      // Delete the fragment
      char* cmd = PBErrMalloc(TheSquidErr,
        sizeof(char) * (1 + strlen(JSONLblVal(propTga)) + 3));
      sprintf(cmd, "rm %s", JSONLblVal(propTga));
      int ret = system(cmd);
      free(cmd);
      (void)ret;

    } else {

      SquadPushHistory(that,
        "Can't preprocess the Pov-Ray task (invalid data)");

    }

    // Free memory
    if (jsonResult != NULL)
      JSONFree(&jsonResult);
    if (jsonRequest != NULL)
      JSONFree(&jsonRequest);
  }
}

// Set the flag memorizing if the TextOMeter is displayed for
// the Squad 'that' to 'flag'
void SquadSetFlagTextOMeter(
  Squad* const that,
    const bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Create a copy of the flag
  bool effectiveFlag = flag;

  // If the requested flag is different from the current flag;
  if (that->_flagTextOMeter != effectiveFlag) {

    // If the user requested to turn on the textometer and the
    // textometer is currently off
    if (effectiveFlag && that->_textOMeter == NULL) {
```

```
      // Create the TextOMeter
      char title[] = "Squad";
      int width = SQUAD_TXTOMETER_LENGTHLINEHISTORY + 1;
      int height = SQUAD_TXTOMETER_NBLINEHISTORY +
        SQUAD_TXTOMETER_NBTASKDISPLAYED + 4;
      that->_textOMeter = TextOMeterCreate(title, width, height);

      // If we couldn't create the TextOMeter
      if (that->_textOMeter == NULL) {

        // Force the flag to false
        effectiveFlag = false;
      }
    }

    // If the user requested to turn off the textometer and the
    // textometer is currently on
    if (!effectiveFlag && that->_textOMeter != NULL) {

      // Terminate the TextOMeter
      TextOMeterFree(&(that->_textOMeter));
    }

    // Memorize the current status of the TextOMeter
    that->_flagTextOMeter = effectiveFlag;
  }
}

// Add one line to the history of messages for the TextOMeter
// 'msg' is truncated if it doesn't fit in one line of history
// If the TextOmeter is not turned on, do nothing
// Variadic function with the same signature as printf family
void SquadPushHistory(
  Squad* const that,
         char* msg,
               ...) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (msg == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'msg' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // If the TextOMeter is not turned on
  if (SquadGetFlagTextOMeter(that) == false) {

    // Do nothing
    return;
  }

  // Loop on each line of the history except the last one
  for (int iLine = 0; iLine < SQUAD_TXTOMETER_NBLINEHISTORY - 1;
    ++iLine) {

    // Copy the following line into the current line to
    // 'move up' the history by one step
```

```
    strcpy(that->_history[iLine], that->_history[iLine + 1]);
  }

  // Increment the counter of lines written in history
  ++(that->_countLineHistory);

  // Write the new line at the end of history, ensuring there is no
  // overflow
  sprintf(that->_history[SQUAD_TXTOMETER_NBLINEHISTORY - 1],
    "[%06u] ", that->_countLineHistory);
  va_list ap;
  va_start(ap, msg);
  vsnprintf(that->_history[SQUAD_TXTOMETER_NBLINEHISTORY - 1] + 9,
    SQUAD_TXTOMETER_LENGTHLINEHISTORY - 11, msg, ap);
  va_end(ap);

  // Force a line return at the end of the line in history
  unsigned long len =
    MIN(SQUAD_TXTOMETER_LENGTHLINEHISTORY - 2, strlen(
    that->_history[SQUAD_TXTOMETER_NBLINEHISTORY - 1]));
  that->_history[SQUAD_TXTOMETER_NBLINEHISTORY - 1][len] = '\n';
  that->_history[SQUAD_TXTOMETER_NBLINEHISTORY - 1][len + 1] = '\0';

  // Update the TextOMeter
  SquadUpdateTextOMeter(that);
}

// Add the result of SquidletInfoPrint(squidlet) to the history
// of messages for the TextOMeter
// The output of SquidletInfoPrint is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquidletInfo(
               Squad* const that,
  const SquidletInfo* const squidlet) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'squidlet' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a buffer to memorize the result of the SquidletInfoPrint
  char buffer[SQUAD_TXTOMETER_LENGTHLINEHISTORY - 1];

  // Open a stream on the buffer
  FILE* stream = fmemopen(buffer, sizeof(buffer), "w");

  // Print the task
  SquidletInfoPrint(squidlet, stream);

  // Close the stream
  fclose(stream);

  // Push the buffer in history
  SquadPushHistory(that, buffer);
```

```
}

// Add the result of SquidletTaskRequestPrint(request) to the history
// of messages for the TextOMeter
// The output of SquidletTaskRequest is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquidletTaskRequest(
                      Squad* const that,
  const SquidletTaskRequest* const request) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (request == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'request' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a buffer to memorize the result of the
  // SquidletTaskRequestPrint
  char buffer[SQUAD_TXTOMETER_LENGTHLINEHISTORY - 1];

  // Open a stream on the buffer
  FILE* stream = fmemopen(buffer, sizeof(buffer), "w");

  // Print the task
  SquidletTaskRequestPrint(request, stream);

  // Close the stream
  fclose(stream);

  // Push the buffer in history
  SquadPushHistory(that, buffer);
}

// Add the result of SquadRunningTaskPrint(request) to the history
// of messages for the TextOMeter
// The output of SquadRunningTaskPrint is truncated if it doesn't fit in
// one line of history
// If the TextOmeter is not turned on, do nothing
void SquadPushHistorySquadRunningTask(
                    Squad* const that,
  const SquadRunningTask* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a buffer to memorize the result of the SquadRunningTaskPrint
```

```
  char buffer[SQUAD_TXTOMETER_LENGTHLINEHISTORY - 1];

  // Open a stream on the buffer
  FILE* stream = fmemopen(buffer, sizeof(buffer), "w");

  // Print the task
  SquadRunningTaskPrint(task, stream);

  // Close the stream
  fclose(stream);

  // Push the buffer in history
  SquadPushHistory(that, buffer);
}

// Refresh the content of the TextOMeter attached to the
// Squad 'that'
void SquadUpdateTextOMeter(const Squad* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (that->_textOMeter == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that->_textOMeter' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Clear the TextOMeter
  TextOMeterClear(that->_textOMeter);

  // Declare a buffer to send text to the TextOMeter
  char buffer[SQUAD_TXTOMETER_LENGTHLINEHISTORY + 1];

  // Print the header
  sprintf(buffer, SQUAD_TXTOMETER_FORMAT1,
    SquadGetNbRunningTasks(that), SquadGetNbRemainingTasks(that),
    SquadGetNbSquidlets(that));
  TextOMeterPrint(that->_textOMeter, buffer);

  // Print the history
  for (int iLine = 0; iLine < SQUAD_TXTOMETER_NBLINEHISTORY; ++iLine) {
    TextOMeterPrint(that->_textOMeter, that->_history[iLine]);
  }

  // Print the tasks header
  sprintf(buffer, SQUAD_TXTOMETER_TASKHEADER);
  TextOMeterPrint(that->_textOMeter, buffer);

  // Print the running tasks
  int iLine = 0;
  if (SquadGetNbRunningTasks(that) > 0) {
    GSetIterForward iter = GSetIterForwardCreateStatic(
      (GSet*)SquadRunningTasks(that));
    do {
      SquadRunningTask* task = GSetIterGet(&iter);
      if (task != NULL) {
        char bufferTask[SQUAD_TXTOMETER_LENGTHLINEHISTORY - 12];
        FILE* stream = fmemopen(
          bufferTask, sizeof(bufferTask), "w");
```

```
      SquadRunningTaskPrint(task, stream);
      fclose(stream);
      sprintf(buffer, SQUAD_TXTOMETER_FORMATRUNNING, bufferTask);
    } else {
      buffer[0] = '\0';
    }
    TextOMeterPrint(that->_textOMeter, buffer);
    ++iLine;
  } while (GSetIterStep(&iter) &&
    iLine < SQUAD_TXTOMETER_NBTASKDISPLAYED);
}

// Print the remaining tasks
if (SquadGetNbRemainingTasks(that) > 0 &&
  iLine < SQUAD_TXTOMETER_NBTASKDISPLAYED) {
  GSetIterForward iter = GSetIterForwardCreateStatic(
    (GSet*)SquadTasks(that));
  do {
    SquidletTaskRequest* task = GSetIterGet(&iter);
    if (task != NULL) {
      char bufferTask[SQUAD_TXTOMETER_LENGTHLINEHISTORY - 12];
      FILE* stream = fmemopen(
        bufferTask, sizeof(bufferTask), "w");
      SquidletTaskRequestPrint(task, stream);
      fclose(stream);
      sprintf(buffer, SQUAD_TXTOMETER_FORMATQUEUED, bufferTask);
    } else {
      buffer[0] = '\0';
    }
    TextOMeterPrint(that->_textOMeter, buffer);
    ++iLine;
  } while (GSetIterStep(&iter) &&
    iLine < SQUAD_TXTOMETER_NBTASKDISPLAYED - 1);
}

// If there are more tasks than space to print them all
if (iLine == SQUAD_TXTOMETER_NBTASKDISPLAYED - 1) {

  // Print a mark to specify there are non displayed tasks
  sprintf(buffer, "...\n");
  TextOMeterPrint(that->_textOMeter, buffer);

// Else, there are remaining space to display more tasks
} else {

  // Fill in the remainnig space with empty lines
  sprintf(buffer, "\n");
  for (; iLine < SQUAD_TXTOMETER_NBTASKDISPLAYED; ++iLine) {
    TextOMeterPrint(that->_textOMeter, buffer);
  }
}

// Add an empty line at the bottom of the TextOMeter
sprintf(buffer, "\n");
TextOMeterPrint(that->_textOMeter, buffer);

// Flush the content of the TextOMeter
TextOMeterFlush(that->_textOMeter);
}

// Check all the squidlets of the Squad 'that' by processing a dummy
// task and display information about each one on the file 'stream'
```

```
// Return true if all the tasks could be performed, false else
bool SquadCheckSquidlets(
  Squad* const that,
   FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a variable to memorize the result
  bool res = true;

  // Declare variables to create a dummy task request
  char* buffer = "{\"v\":\"0\"}";
  time_t maxWait = 5;

  // If there are squidlets
  if (SquadGetNbSquidlets(that) > 0) {

    // Loop on the squidlets
    GSetIterForward iter =
      GSetIterForwardCreateStatic(SquadSquidlets(that));
    do {

      // Get the squidlet
      SquidletInfo* squidlet = GSetIterGet(&iter);

      // Display info about the squidlet
      SquidletInfoPrint(squidlet, stream);
      fprintf(stream, "\n");

      // Create a dummy task
      unsigned long id = 0;
      unsigned long subId = 0;
      SquidletTaskRequest* task = SquidletTaskRequestCreate(
        SquidletTaskType_Dummy, id, subId, buffer, maxWait);
      GSetAppend((GSet*)SquadTasks(that), task);

      // Memorize the start time
      struct timeval start;
      gettimeofday(&start, NULL);

      // Request the execution of the dummy task on the squidlet
      bool ret = SquadSendTaskOnSquidlet(that, squidlet, task);

      // Memorize the time to send the request
      struct timeval timeToSend;
      gettimeofday(&timeToSend, NULL);

      // If the request was not successfull
      if (!ret) {

        // Set the flag and display a message on the stream
        res = false;
```

```
        fprintf(stream, "\tThe request for a dummy task failed.\n");

      // Else, the request was successfull
      } else {

        // Get the running tasks
        SquadRunningTask* runningTask =
          GSetPop((GSet*)SquadRunningTasks(that));

        // Loop until the task ends
        bool flagStop = false;
        while (!flagStop && time(NULL) - runningTask->_startTime <=
          runningTask->_request->_maxWaitTime) {

          // If the task is completed
          if (SquadReceiveTaskResult(that, runningTask)) {

            // Get the time to process
            struct timeval timeToProcess;
            gettimeofday(&timeToProcess, NULL);

            // Stop the loop
            flagStop = true;

            // Process the result
            SquidletTaskRequest* request = runningTask->_request;
            fprintf(stream, "\tRequest for dummy task succeeded.\n");
            fprintf(stream, "\t%s\n", request->_bufferResult);
            unsigned long delayToSendms =
              (timeToSend.tv_sec - start.tv_sec) * 1000 +
              (timeToSend.tv_usec - start.tv_usec) / 1000;
            unsigned long delayToProcessms =
              (timeToProcess.tv_sec - timeToSend.tv_sec) * 1000 +
              (timeToProcess.tv_usec - timeToSend.tv_usec) / 1000;
            fprintf(stream,
              "\tdelay to send: %lums, delay to process: %lums\n",
              delayToSendms, delayToProcessms);
          }
        }

        // If we got out of the loop without the flag raising, it means
        // we gave up on time
        if (!flagStop) {
          fprintf(stream, "\tGave up due to time limit.\n");
        }

        // Free memory
        runningTask->_request = NULL;
        runningTask->_squidlet = NULL;
        SquadRunningTaskFree(&runningTask);
      }
    } while (GSetIterStep(&iter));
  }

  // Return the result
  return res;
}

// Run the benchmark with the squad 'that' and output the result on
// the file 'stream'
// The benchmark consists of executing benchmark tasks with several
// size and number of loop, first locally and then on the squidlet
```

```
// (which must have been loaded before calling this function)
void SquadBenchmark(
  Squad* const that,
   FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  fprintf(stream, "-- Benchmark started --\n");

  // Declare parameters of the benchmark
  // The benchmark will last 'lengthTest' per pair of 'maxSizePayload'
  // and 'nbMaxLoop'
  // 'maxSizePayload' is the max size of the set to be sorted by the
  // Squidlet. The size starts at 9 and increment geometrically by 10
  // 'nbMaxLoop' is the max number of time the set is sorted by the
  // Squidlet for one given task. The number starts at 1 and increment
  // geometrically by 2.
  float lengthTest = 240000.0; // ms
  size_t maxSizePayload = 900;
  int nbMaxLoop = 1024;

  // Header for the results
  char* header = "nbLoopPerTask\tnbBytePayload\tCompleted\tExpected\n";

  // If the squad has no squidlet, it means we are running the benchmark
  // on the local device for comparison
  if (SquadGetNbSquidlets(that) == 0) {

    // Display info
    fprintf(stream, "Execution on local device:\n");
    fprintf(stream, "%s", header);

    // Loop on the payload
    for (size_t sizePayload = 9;
      sizePayload <= maxSizePayload; sizePayload *= 10) {

      // Create a dummy buffer with the size of the payload
      // The content of the buffer is actually set up by the benchmark
      // function on the squidlet
      char* buffer = PBErrMalloc(TheSquidErr, sizePayload + 1);
      memset(buffer, ' ', sizePayload);
      buffer[sizePayload] = '\0';

      // Loop on nbLoop
      for (int nbLoop = 1; nbLoop <= nbMaxLoop; nbLoop *= 2) {

        // Variable to memorize the starting time
        struct timeval stop, start;
        gettimeofday(&start, NULL);

        // Variabel to memorize the number of completed tasks and
        // final time
```

```
      unsigned long nbComplete = 0;
      float deltams = 0;

      // Loop on the duration of the test
      do {

        // Execute the benchmark function
        TheSquidBenchmark(nbLoop, buffer);

        // Increment the number of completed task
        ++nbComplete;

        // Get the current time
        gettimeofday(&stop, NULL);
        deltams = (float)(stop.tv_sec - start.tv_sec) * 1000.0 +
          (float)(stop.tv_usec - start.tv_usec) / 1000.0;

      } while (deltams < lengthTest);

      // Calculate the exact nb of tasks completed in the delay of
      // the test
      float nbTaskExpected = lengthTest / deltams * (float)nbComplete;

      // Print results
      fprintf(stream, "%04d\t%08u\t%lu*%f/%f\t%.6f\n",
        nbLoop, sizePayload, nbComplete, lengthTest, deltams, nbTaskExpected);
      fflush(stdout);
    }

    // Free memory
    free(buffer);
  }

// Else the squad has at least one squidlet, it means we execute the
// benchmark on TheSquid
} else {

  // Display info
  fprintf(stream, "Execution on TheSquid:\n");
  fprintf(stream, "%s", header);

  // Variables to create the tasks and manage the loops
  time_t maxWait = 10000;
  unsigned int id = 0;
  bool flagStop = false;

  // Loop on payload size
  for (size_t sizePayload = 9; !flagStop &&
    sizePayload <= maxSizePayload; sizePayload *= 10) {

    // Loop on nbLoop
    for (int nbLoop = 1; !flagStop && nbLoop <= nbMaxLoop;
      nbLoop *= 2) {

      // Reset the stats of all the squidlets
      SquadRequestAllSquidletToResetStats(that);

      // Variable ti measure time of execution
      struct timeval stop, start;
      gettimeofday(&start, NULL);
      float deltams = 0.0;
```

```
// Loop during lengthTest seconds
do {

  // Create benchmark tasks, twice as many as squidlets to ensure
  // there is always task ready to send in the SquadStep loop
  while (SquadGetNbRunningTasks(that) +
    SquadGetNbRemainingTasks(that) <
    2 * SquadGetNbSquidlets(that)) {

    SquadAddTask_Benchmark(that, id++, maxWait, nbLoop,
      sizePayload);

  }

  // Step the Squad
  GSetSquadRunningTask completedTasks = SquadStep(that);

  // Loop on co,pleted tasks
  while (GSetNbElem(&completedTasks) > 0L) {

    // Get the completed task
    SquadRunningTask* completedTask = GSetPop(&completedTasks);
    SquidletTaskRequest* task = completedTask->_request;

    // If the task failed
    if (strstr(task->_bufferResult,
      "\"success\":\"1\"") == NULL) {

      // Display info and stop the benchmark
      SquidletTaskRequestPrint(task, stdout);
      fprintf(stream, " failed !!\n");
      fprintf(stream, "%s\n", task->_bufferResult);
      flagStop = true;
    }

    // Free memory
    SquidletTaskRequestFree(&task);
    SquadRunningTaskFree(&completedTask);
  }

  // Get the time
  gettimeofday(&stop, NULL);
  deltams = (float)(stop.tv_sec - start.tv_sec) * 1000.0 +
    (float)(stop.tv_usec - start.tv_usec) / 1000.0;

} while (!flagStop && deltams < lengthTest);

// If there are Squidlets which are not running at this point
if (SquadGetNbSquidlets(that) > 0) {

  // Loop on free Squidlets
  GSetIterForward iter =
    GSetIterForwardCreateStatic(SquadSquidlets(that));
  do {

    // Get the Squidlet
    SquidletInfo* squidlet = GSetIterGet(&iter);

    // Update the timePerTask of the Squidlets
    SquidletInfoStats* stats =
      (SquidletInfoStats*)SquidletInfoStatistics(squidlet);
    stats->_timePerTask =
```

```
      deltams / (float)(stats->_nbTaskComplete);

  } while (GSetIterStep(&iter));
}

// Flush the remaining tasks to let only the remaining Squidlet
// finish their task and avoid starting new ones
while (SquadGetNbRemainingTasks(that) > 0) {
  SquidletTaskRequest* task = GSetPop(&(that->_tasks));
  SquidletTaskRequestFree(&task);
}

// While there are currently running tasks
while (!flagStop && SquadGetNbRunningTasks(that) > 0) {

  // Get the completed tasks
  GSetSquadRunningTask completedTasks = SquadStep(that);

  // Measure time
  gettimeofday(&stop, NULL);
  deltams = (float)(stop.tv_sec - start.tv_sec) * 1000.0 +
    (float)(stop.tv_usec - start.tv_usec) / 1000.0;

  // Loop on completed tasks
  while (GSetNbElem(&completedTasks) > 0L) {

    // Get the completed task
    SquadRunningTask* completedTask = GSetPop(&completedTasks);
    SquidletTaskRequest* task = completedTask->_request;

    // If the task failed
    if (strstr(task->_bufferResult,
      "\"success\":\"1\"") == NULL) {

      // Display info and stop the benchmark
      SquidletTaskRequestPrint(task, stdout);
      fprintf(stream, " failed !!\n");
      fprintf(stream, "%s\n", task->_bufferResult);
      flagStop = true;

    // Else, the task succeeded
    } else {

      // Update the timePerTask
      SquidletInfo* squidlet = completedTask->_squidlet;
      SquidletInfoStats* stats =
        (SquidletInfoStats*)SquidletInfoStatistics(squidlet);
      stats->_timePerTask =
        deltams / (float)(stats->_nbTaskComplete);
    }

    // Free memory
    SquidletTaskRequestFree(&task);
    SquadRunningTaskFree(&completedTask);
  }
}

// Get the time when all the tasks have finished
gettimeofday(&stop, NULL);

// Display the stats of all the squidlets
//SquadPrintStatsSquidlets(that, stream);
```

```
        // Decalre a variable to memorize the extrapolated total
        // number of completed tasks at the current time
        float nbTaskComplete = 0.0;

        // Calculate the delay from the start of this run
        deltams = (float)(stop.tv_sec - start.tv_sec) * 1000.0 +
          (float)(stop.tv_usec - start.tv_usec) / 1000.0;

        // Loop on the Squidlets
        GSetIterForward iter =
          GSetIterForwardCreateStatic(SquadSquidlets(that));
        do {

          // Calculate the total nb of completed tasks at the time of
          // completion
          SquidletInfo* squidlet = GSetIterGet(&iter);
          SquidletInfoStats* stats =
            (SquidletInfoStats*)SquidletInfoStatistics(squidlet);
          nbTaskComplete += deltams / stats->_timePerTask;
        } while (GSetIterStep(&iter));

        // Calculate the exact nb of tasks completed in the delay of
        // the test
        float nbTaskExpected = lengthTest / deltams * nbTaskComplete;

        // Display the results
        fprintf(stream, "%04d\t%08u\t%f*%f/%f\t%.6f\n",
          nbLoop, sizePayload, nbTaskComplete, lengthTest,
          deltams, nbTaskExpected);
        fflush(stream);
      }
    }
  }

  // Display info
  fprintf(stream, "-- Benchmark ended --\n");

}

// Put back the 'task' into the set of task to complete of the Squad
// 'that'
// Failed tasks (by timeout due to there 'maxWait' in
// SquadAddTask_xxx() or by failure code from the squidlet in the
// result data) are automatically put back into the set of task to
// complete
void SquadTryAgainTask(
               Squad* const that,
  SquidletTaskRequest* const task) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (task == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'task' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
```

```
  // Ensure the result buffer is empty
  if (task->_bufferResult != NULL) {
    free(task->_bufferResult);
    task->_bufferResult = NULL;
  }

  // Put back the task in the set of task to complete
  GSetAppend((GSet*)SquadTasks(that), task);
}

// Print the statistics about the currently available Squidlets of
// the Squad 'that' on the 'stream'
void SquadPrintStatsSquidlets(
  const Squad* const that,
        FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // If there are currently available squidlets
  if (SquadGetNbSquidlets(that) > 0) {

    // Loop on the squidlets
    GSetIterForward iter =
      GSetIterForwardCreateStatic(SquadSquidlets(that));
    do {

      // Get the squidlet
      SquidletInfo* squidlet = GSetIterGet(&iter);

      // Print the stats about this squidlet
      fprintf(stream, " --- ");
      SquidletInfoPrint(squidlet, stream);
      fprintf(stream, " --- \n");
      SquidletInfoStatsPrintln(SquidletInfoStatistics(squidlet), stream);
    } while (GSetIterStep(&iter));
  }
}

// -------------- Squidlet

// ================= Global variable ==================

// Variable to handle the signal Ctrl-C
bool Squidlet_CtrlC = false;

// ================ Functions implementation ====================

// Handler for the signal Ctrl-C
void SquidletHandlerCtrlC(
  const int sig) {
  // Don't use the signal
  (void)sig;
```

```
  // Raise the flag to memorize we have received the signal Ctrl-C
  Squidlet_CtrlC = true;
  // Display info
  time_t intTime = time(NULL);
  char* strIntTime = ctime(&intTime);
  printf("Squidlet :  !!! Interrupted by Ctrl-C !!! %s", strIntTime);
  fflush(stdout);
}

// Handler for the signal SIGPIPE
// Catch the signal to avoid the Squidlet dying over a socket failure
void SquidletHandlerSigPipe(
  const int sig) {
  // Don't use the signal
  (void)sig;
  // Display info
  time_t intTime = time(NULL);
  char* strIntTime = ctime(&intTime);
  printf("Squidlet :  !!! Received SIGPIPE !!! %s", strIntTime);
  fflush(stdout);
}

// Return a new Squidlet listening to the IP adress 'addr' and port
// 'port'
// If 'addr' equals 0, select automatically the first network address
// of the host
// If 'port' equals -1, select automatically one available between
// THESQUID_PORTMIN and THESQUID_PORTMAX
Squidlet* SquidletCreateOnPort(
  const uint32_t addr,
       const int port) {

  // Allocate memory for the squidlet
  Squidlet* that = PBErrMalloc(TheSquidErr, sizeof(Squidlet));

  // Open the socket
  that->_fd = socket(AF_INET, SOCK_STREAM, 0);

  // If we couldn't open the socket
  if (that->_fd == -1) {

    // Free memory and return null
    free(that);
    sprintf(TheSquidErr->_msg, "socket() failed");
    return NULL;
  }

  // Set the timeout for sending and receiving on this socket to
  // THESQUID_ACCEPT_TIMEOUT seconds
  struct timeval tv;
  tv.tv_sec = THESQUID_ACCEPT_TIMEOUT;
  tv.tv_usec = 0;
  int reuse = 1;
  bool ret = (setsockopt(that->_fd, SOL_SOCKET, SO_SNDTIMEO,
    (char*)&tv, sizeof(tv)) != -1);
  ret &= (setsockopt(that->_fd, SOL_SOCKET, SO_RCVTIMEO,
    (char*)&tv, sizeof(tv)) != -1);
  ret &= (setsockopt(that->_fd, SOL_SOCKET, SO_REUSEADDR,
    &reuse,sizeof(int)) != -1);

  // If we couldn't set the timeout, free memory and return null
  if (ret == false) {
```

```
  // Free memory and return null
  close(that->_fd);
  free(that);
  sprintf(TheSquidErr->_msg, "setsockopt() failed");
  return NULL;
}

// Get the hostname
ret = (gethostname(that->_hostname, sizeof(that->_hostname)) != -1);

// If we couldn't get the hostname
if (ret == false) {

  // Free memory and return null
  close(that->_fd);
  free(that);
  sprintf(TheSquidErr->_msg, "gethostname() failed");
  return NULL;
}

// Get the info about the host
that->_host = gethostbyname(that->_hostname);

// If we couldn't get the host info
if (that->_host == NULL) {

  // Free memory and return null
  close(that->_fd);
  free(that);
  sprintf(TheSquidErr->_msg, "gethostbyname() failed");
  return NULL;
}

// Init the port and socket info
memset(&(that->_sock), 0, sizeof(struct sockaddr_in));
that->_sock.sin_family = AF_INET;
if (addr != 0)
  that->_sock.sin_addr.s_addr = addr;
else
  that->_sock.sin_addr.s_addr = *(uint32_t*)(that->_host->h_addr_list[0]);
if (port != -1)
  that->_port = port;
else
  that->_port = THESQUID_PORTMIN;
that->_sock.sin_port = htons(that->_port);

// If the port is not specified by the user
if (port == -1) {

  // Loop on the ports until we find one availalble
  while (bind(that->_fd, (struct sockaddr *)&(that->_sock),
    sizeof(struct sockaddr_in)) == -1 &&
    that->_port < THESQUID_PORTMAX) {
    ++(that->_port);
    that->_sock.sin_port = htons(that->_port);
  }

  // If we couldn't find an available port
  if (that->_port == THESQUID_PORTMAX) {

    // Free memory and return null
```

```
      close(that->_fd);
      free(that);
      sprintf(TheSquidErr->_msg, "bind() failed");
      return NULL;
    }

  // Else if the port is specified by the user, try to bind only on
  // that port
  } else {

    // If we couldn't bind the socket on the requested port
    if (bind(that->_fd, (struct sockaddr *)&(that->_sock),
      sizeof(struct sockaddr_in)) == -1) {

      // Free memory and return null
      close(that->_fd);
      free(that);
      sprintf(TheSquidErr->_msg, "bind() failed");
      return NULL;
    }
  }

  // Start listening through the socket
  if (listen(that->_fd, THESQUID_NBMAXPENDINGCONN) == -1) {

    // If we can't listen through the socket, close it, free memory and
    // return null
    close(that->_fd);
    free(that);
    sprintf(TheSquidErr->_msg, "listen() failed");
    return NULL;
  }

  // Init the PID
  that->_pid = getpid();

  // Init the socket for reply
  that->_sockReply = -1;

  // Set the handler to catch the signal Ctrl-C
  signal(SIGINT, SquidletHandlerCtrlC);

  // Init the stream for output
  that->_streamInfo = NULL;

  // Init the variables for statistics
  SquidletResetStats(that);

  // Init the properties for neuranet evaluation task
  that->_datasetPath = NULL;
  that->_dataset = GDataSetVecFloatCreateStatic();

  // Return the new squidlet
  return that;
}

// Free the memory used by the Squidlet 'that'
void SquidletFree(
  Squidlet** that) {

  // If the pointer is null there is nothing to do
  if (that == NULL || *that == NULL)
```

```
    return;

  // Close the socket
  close((*that)->_fd);

  // Close the socket for the reply if it is opened
  if ((*that)->_sockReply != -1)
    close((*that)->_sockReply);

  // Free memory
  GDataSetVecFloatFreeStatic(&((*that)->_dataset));
  free(*that);
  *that = NULL;
}

// Print the PID, Hostname, IP and Port of the Squidlet 'that' on the
// file 'stream'
// Example: 100 localhost 0.0.0.0:3000
void SquidletPrint(
  const Squidlet* const that,
            FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (stream == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'stream' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Print the info on the stream
  fprintf(stream, "%d %s %s:%d", SquidletGetPID(that),
    SquidletHostname(that), SquidletIP(that), SquidletGetPort(that));
}

// Reset the statistics of the Squidlet 'that'
void SquidletResetStats(
  Squidlet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  that->_nbAcceptedConnection = 0;
  that->_nbAcceptedTask = 0;
  that->_nbRefusedTask = 0;
  that->_nbFailedReceptTaskData = 0;
  that->_nbFailedReceptTaskSize = 0;
  that->_nbSentResult = 0;
  that->_nbFailedSendResult = 0;
  that->_nbFailedSendResultSize = 0;
  that->_nbFailedReceptAck = 0;
  that->_nbTaskComplete = 0;
  that->_timeToProcessMs = 0;
  that->_timeWaitedTaskMs = 0;
```

```
  that->_timeWaitedAckMs = 0;

}

// Wait for a task request to be received by the Squidlet 'that'
// Return the received task request, or give up after
// THESQUID_ACCEPT_TIMEOUT if there was no request and return a task
// request of type SquidletTaskType_Null
SquidletTaskRequest SquidletWaitRequest(
  Squidlet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Declare a variable to memorize the reply to the request
  char reply = THESQUID_TASKREFUSED;

  // Declare a buffer to receive the payload of the connection
  SquidletTaskRequest taskRequest;

  // Initialize the type of th received task to SquidletTaskType_Null
  taskRequest._type = SquidletTaskType_Null;

  // Declare a variable to memorize the info about the incoming
  // connection
  struct sockaddr_in incomingSock;
  socklen_t incomingSockSize = sizeof(incomingSock);

  // Make sure the socket for reply is closed
  if (that->_sockReply != -1) {
    close(that->_sockReply);
    that->_sockReply = -1;
  }

  // Extract the first connection request on the queue of pending
  // connections if there was one. If there are none wait for
  // one during THESQUID_ACCEPT_TIMEOUT seconds and then give up
  that->_sockReply = accept(that->_fd, (struct sockaddr *)&incomingSock,
    &incomingSockSize);

  // If we could extract a pending connection
  if (that->_sockReply >= 0) {

    // Update the number of accepted connection
    ++(that->_nbAcceptedConnection);

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that), " : accepted connection\n");
    }

    // Set the timeout for sending and receiving on the
    // extracted socket to THESQUID_PROC_TIMEOUT sec
    struct timeval tv;
    tv.tv_sec = THESQUID_PROC_TIMEOUT;
    tv.tv_usec = 0;
    int reuse = 1;
    bool ret = (setsockopt(that->_sockReply, SOL_SOCKET, SO_SNDTIMEO,
      (char*)&tv, sizeof(tv)) != -1);
```

```
ret &= (setsockopt(that->_sockReply, SOL_SOCKET, SO_RCVTIMEO,
  (char*)&tv, sizeof(tv)) != -1);
ret &= (setsockopt(that->_sockReply, SOL_SOCKET, SO_REUSEADDR,
  &reuse, sizeof(int)) != -1);

// If we couldn't set the timeout
if (ret == false) {

  // Refuse the task
  reply = THESQUID_TASKREFUSED;

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that), " : setsockopt failed\n");
  }

// Else, we could set the timeout
} else {

  // Receive the task type, give up after
  // THESQUID_PROC_TIMEOUT seconds
  ret = SocketRecv(&(that->_sockReply), sizeof(SquidletTaskType),
    (char*)&taskRequest, THESQUID_PROC_TIMEOUT);

  // If we could receive the task type
  if (ret == true) {

    // The task is accepted
    reply = THESQUID_TASKACCEPTED;

    // Update the number of accepted task
    ++(that->_nbAcceptedTask);

    // Update the statistics
    if (that->_nbTaskComplete > 0) {
      struct timeval now;
      gettimeofday(&now, NULL);
      that->_timeWaitedTaskMs =
        (now.tv_sec - that->_timeLastTaskComplete.tv_sec) * 1000 +
        (now.tv_usec - that->_timeLastTaskComplete.tv_usec) / 1000;
    }

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : received task type %d\n", taskRequest._type);
    }

  // Else, we couldn't receive the task type
  } else {

    // Refuse the task and ensure the task type is equal to
    // SquidletTaskType_Null
    taskRequest._type = SquidletTaskType_Null;
    reply = THESQUID_TASKREFUSED;

    // Update the number of refused task
    ++(that->_nbRefusedTask);

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
```

```
            " : couldn't receive task type\n");
        }
      }

      // Send the reply to the task request
      int flags = 0;
      ret = (send(that->_sockReply, &reply, sizeof(reply), flags) != -1);

      // If we couldn't send the reply
      if (ret == false) {

        // If we couldn't send the reply, do not process the task
        taskRequest._type = SquidletTaskType_Null;

        if (SquidletStreamInfo(that)){
          SquidletPrint(that, SquidletStreamInfo(that));
          fprintf(SquidletStreamInfo(that),
            " : couldn't sent reply to task request %d\n", reply);
        }

      // Else, we could send the reply
      } else {

        if (SquidletStreamInfo(that)){
          SquidletPrint(that, SquidletStreamInfo(that));
          fprintf(SquidletStreamInfo(that),
            " : sent reply to task request %d\n", reply);
        }
      }
    }
  }

  // Return the received task request
  return taskRequest;
}

// Process the task request 'request' with the Squidlet 'that'
void SquidletProcessRequest(
             Squidlet* const that,
  SquidletTaskRequest* const request) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (request == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'request' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // If the task is of type null
  if (request->_type == SquidletTaskType_Null) {

    // Nothing to do
    return;
  }

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
```

```
  fprintf(SquidletStreamInfo(that), " : process task\n");
}

// Declare a variable to memorize the size in byte of the input data
size_t sizeInputData = 0;

// Declare a buffer for the result data
char* bufferResult = NULL;

// Wait to receive the size of the input data with a time limit of
// THESQUID_WAITDATARECEPT_TIMEOUT seconds
bool ret = SocketRecv(&(that->_sockReply), sizeof(size_t),
  (char*)&sizeInputData, THESQUID_WAITDATARECEPT_TIMEOUT);

// If we could receive the data size
if (ret == true) {

  // Declare a buffer for the raw input data
  char* buffer = NULL;

  // If there are input data
  if (sizeInputData > 0) {

    if (SquidletStreamInfo(that)) {
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : received size task data %d\n", sizeInputData);
    }

    // Allocate memory for the input data
    buffer = PBErrMalloc(TheSquidErr, sizeInputData + 1);
    memset(buffer, 0, sizeInputData + 1);

    // Wait to receive the input data with a time limit proportional
    // to the size of input data
    int timeLimit = THESQUID_WAITDATARECEPT_TIMEOUT + \
      (int)round((float)sizeInputData / 100.0);
    ret = SocketRecv(&(that->_sockReply), sizeInputData, buffer,
      timeLimit);

    // If we couldn't receive the data
    if (ret == false) {

      // Free memory
      free(buffer);
      buffer = NULL;

      // Update the number of failed reception of data
      ++(that->_nbFailedReceptTaskData);

      if (SquidletStreamInfo(that)){
        SquidletPrint(that, SquidletStreamInfo(that));
        fprintf(SquidletStreamInfo(that),
          " : couldn't receive task data\n");
      }

    // Else, we could receive the data
    } else {

      if (SquidletStreamInfo(that)){
        SquidletPrint(that, SquidletStreamInfo(that));
        if (strlen(buffer) > 50) {
```

```
          char tmp[4];
          tmp[0] = buffer[47];
          tmp[1] = buffer[48];
          tmp[2] = buffer[49];
          tmp[3] = buffer[50];
          buffer[47] = ' ';
          buffer[48] = '.';
          buffer[49] = '.';
          buffer[50] = '\0';
          fprintf(SquidletStreamInfo(that),
            " : received task data %s\n", buffer);
          buffer[47] = tmp[0];
          buffer[48] = tmp[1];
          buffer[49] = tmp[2];
          buffer[50] = tmp[3];
        } else {
          fprintf(SquidletStreamInfo(that),
            " : received task data %s\n", buffer);
        }
      }
    }
  }

  // If we could receive the expected data
  if (sizeInputData > 0 && buffer != NULL) {

    // Process the request according to the request type
    // and store the result into bufferResult
    switch (request->_type) {
      case SquidletTaskType_Dummy:
        SquidletProcessRequest_Dummy(that, buffer, &bufferResult);
        break;
      case SquidletTaskType_Benchmark:
        SquidletProcessRequest_Benchmark(that, buffer, &bufferResult);
        break;
      case SquidletTaskType_PovRay:
        SquidletProcessRequest_PovRay(that, buffer, &bufferResult);
        break;
      case SquidletTaskType_ResetStats:
        SquidletProcessRequest_StatsReset(that);
        break;
      case SquidletTaskType_EvalNeuranet:
        SquidletProcessRequest_EvalNeuranet(that, buffer,
          &bufferResult);
        break;
      default:
        break;
    }

    // Free memory
    free(buffer);
  }

// Else we couldn't receive the data size
} else {

  // Update the number of failed reception of data
  ++(that->_nbFailedReceptTaskSize);

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
```

```
        " : couldn't receive data size\n");
    }

  }

  // If bufferResult is not null here it means there is a result
  // to be sent
  if (bufferResult != NULL) {

    // Send the result
    SquidletSendResultData(that, bufferResult);

    // Free memory
    free(bufferResult);

  }

  // Update the time when we last processed a request to calculate
  // later the time between two processing
  gettimeofday(&(that->_timeLastTaskComplete), NULL);

}

// Send the result 'bufferResult' of the processing of a task
// by the Squidlet 'that'
void SquidletSendResultData(
             Squidlet* const that,
           const char* const bufferResult) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (bufferResult == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'bufferResult' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Get the current time for statistics
  struct timeval start;
  gettimeofday(&start, NULL);

  // Send the result data size
  int flags = 0;
  size_t len = strlen(bufferResult);
  bool ret = (send(that->_sockReply,
    (char*)&len, sizeof(size_t), flags) != -1);

  if (ret == true) {

    // Update stats
    struct timeval stop;
    gettimeofday(&stop, NULL);
    that->_timeTransferSquidSquadMs =
      (stop.tv_sec - start.tv_sec) * 1000 +
      (stop.tv_usec - start.tv_usec) / 1000;
    that->_timeTransferSquidSquadMs /= (float)len;

    if (SquidletStreamInfo(that)){
```

```
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
    " : sent result size\n");
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
    " : wait for acknowledgement from squad\n");
}

// Receive the acknowledgement
char ack = 0;
int waitDelayMaxSec = 60;
struct timeval start;
gettimeofday(&start, NULL);
ret = SocketRecv(&(that->_sockReply), sizeof(char), &ack,
  waitDelayMaxSec);

// If we could receive the acknowledgment
if (ret == true) {

  // Update the statistics
  struct timeval now;
  gettimeofday(&now, NULL);
  that->_timeWaitedAckMs =
    (now.tv_sec - start.tv_sec) * 1000 +
    (now.tv_usec - start.tv_usec) / 1000;

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : received acknowledgement from squad\n");
  }

  // Send the result
  ret = (send(that->_sockReply, bufferResult, len, flags) != -1);

  // If we could send the result
  if (ret == true) {

    // Update the number of successfully sent result
    ++(that->_nbSentResult);

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : sent result %s\n", bufferResult);
    }

  // Else, we couldn't send the result
  } else {

    // Update the number of unsuccessfully sent result
    ++(that->_nbFailedSendResult);

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : couldn't send result %s\n", bufferResult);
    }
  }

// Else, we couldn't receive the acknowledgement
} else {
```

```
    // Update the number of unsuccessfully received acknowledgment
    ++(that->_nbFailedReceptAck);

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : couldn't receive acknowledgement\n");
    }
  }

// Else, we couldn't send the result size
} else {


  // Update the number of unsuccessfully sent result size
  ++(that->_nbFailedSendResultSize);

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : couldn't send data size\n");
  }
}

if (SquidletStreamInfo(that)){
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
    " : wait for acknowledgement from squad\n");
}

// Receive the acknowledgment of reception of result
// Give up after THESQUID_PROC_TIMEOUT seconds
char ack = 0;
ret = SocketRecv(&(that->_sockReply), 1, &ack,
  THESQUID_PROC_TIMEOUT);

// If we could receive the acknowledgement
if (ret == true) {

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : received acknowledgement from squad\n");
  }

// Else, we couldn't receive the acknowledgement
} else {

  // Update the number of unsuccessfully received acknowledgement
  ++(that->_nbFailedReceptAck);

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : couldn't receive acknowledgement from squad\n");
  }
}

if (SquidletStreamInfo(that)){
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
```

```
      " : ready for next task\n");
  }

}

// Process a dummy task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_Dummy(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (buffer == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'buffer' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a variable to memorize if the process has been successful
  bool success = false;

  // Declare a variable to memorize the result of processing
  int result = 0;

  // Allocate memory for the result
  *bufferResult = PBErrMalloc(TheSquidErr, THESQUID_MAXPAYLOADSIZE);
  memset(*bufferResult, 0, THESQUID_MAXPAYLOADSIZE);

  // Start measuring the time used to process the task
  that->_timeToProcessMs = 0;
  struct timeval start;
  gettimeofday(&start, NULL);

  // Process the data
  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : process dummy task %s\n", buffer);
  }

  // Decode the input from JSON
  JSONNode* json = JSONCreate();
  bool ret = JSONLoadFromStr(json, buffer);

  // If we could decode the input
  if (ret == true) {

    // Get the value to process
    JSONNode* prop = JSONProperty(json, "v");

    // If the value is present
    if (prop != NULL) {
```

```
    // Convert the value from string to int
    int v = atoi(JSONLblVal(prop));

    // Process the value
    result = v * -1;

    // Sleep for v seconds
    sleep(v);

    // Set the flag for successfull process
    success = true;

    // Update the time used to process the task
    struct timeval now;
    gettimeofday(&now, NULL);
    that->_timeToProcessMs =
      (now.tv_sec - start.tv_sec) * 1000 +
      (now.tv_usec - start.tv_usec) / 1000;

    // Update the number of completed tasks
    ++(that->_nbTaskComplete);

    // Prepare the result data as JSON
    JSONNode* jsonResult = JSONCreate();
    float temperature = SquidletGetTemperature(that);
    char temperatureStr[10] = {'\0'};
    sprintf(temperatureStr, "%.2f", temperature);
    JSONAddProp(jsonResult, "temperature", temperatureStr);
    char successStr[2] = {'\0'};
    sprintf(successStr, "%d", success);
    JSONAddProp(jsonResult, "success", successStr);
    char resultStr[10] = {'\0'};
    sprintf(resultStr, "%d", result);
    JSONAddProp(jsonResult, "v", resultStr);

    // Append the statistics data
    SquidletAddStatsToJSON(that, jsonResult);

    // Convert the JSON to a string
    bool compact = true;
    ret = JSONSaveToStr(jsonResult, *bufferResult,
      THESQUID_MAXPAYLOADSIZE, compact);
    if (ret == false) {
      sprintf(*bufferResult,
        "{\"success\":\"0\",\"temperature\":\"0.0\","
        "\"err\":\"JSONSaveToStr failed\"}");
    }

  // else the value is not present
  } else {

    sprintf(*bufferResult,
      "{\"success\":\"0\",\"temperature\":\"0.0\","
      "\"err\":\"Invalid input\"}");
  }

// Else, we couldn't decode the input
} else {

  sprintf(*bufferResult,
    "{\"success\":\"0\",\"temperature\":\"0.0\","
    "\"err\":\"JSONLoadFromStr failed\"}");
```

```
  }

  // Free memory
  JSONFree(&json);
}

// Append the statistical data about the squidlet 'that' to the JSON
// node 'json'
void SquidletAddStatsToJSON(
  const Squidlet* const that,
        JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (json == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'json' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Declare a variable to convert numbers into string
  // Hopefully numbers won't have more than 99 digits
  const int bufferSize = 99;
  char buffer[bufferSize];

  // Convert numbers into string and add the JSON property for each one
  sprintf(buffer, "%lu", that->_nbAcceptedConnection);
  JSONAddProp(json, "nbAcceptedConnection", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbAcceptedTask);
  JSONAddProp(json, "nbAcceptedTask", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbRefusedTask);
  JSONAddProp(json, "nbRefusedTask", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbFailedReceptTaskSize);
  JSONAddProp(json, "nbFailedReceptTaskSize", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbFailedReceptTaskData);
  JSONAddProp(json, "nbFailedReceptTaskData", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbSentResult);
  JSONAddProp(json, "nbSentResult", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbFailedSendResult);
  JSONAddProp(json, "nbFailedSendResult", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbFailedSendResultSize);
  JSONAddProp(json, "nbFailedSendResultSize", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbFailedReceptAck);
```

```
  JSONAddProp(json, "nbFailedReceptAck", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_nbTaskComplete);
  JSONAddProp(json, "nbTaskComplete", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_timeToProcessMs);
  JSONAddProp(json, "timeToProcessMs", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_timeWaitedTaskMs);
  JSONAddProp(json, "timeWaitedTaskMs", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%lu", that->_timeWaitedAckMs);
  JSONAddProp(json, "timeWaitedAckMs", buffer);
  memset(buffer, 0, bufferSize);

  sprintf(buffer, "%.3f", that->_timeTransferSquidSquadMs);
  JSONAddProp(json, "timeTransferSquidSquadMs", buffer);
  memset(buffer, 0, bufferSize);
}

// Process a benchmark task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_Benchmark(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Start measuring the time used to process the task
  that->_timeToProcessMs = 0;
  struct timeval start;
  gettimeofday(&start, NULL);

  // Declare a variable to memorize if the process has been successful
  bool success = false;

  // Variable to memorize the result of the benchmark function
  int result = 0;

  // Declare a variable to store the error message if any
  char errMsg[THESQUID_MAXPAYLOADSIZE] = {'\0'};

  // Decode the input from JSON
  JSONNode* json = JSONCreate();
  bool ret = JSONLoadFromStr(json, buffer);

  // If we could decode the JSON
  if (ret == true) {
```

```
  // Get the values to process
  JSONNode* propNb = JSONProperty(json, "nb");
  JSONNode* propPayloadSize = JSONProperty(json, "payloadSize");

  // If the value were in the JSON
  if (propNb != NULL && propPayloadSize != NULL) {

    // Convert the value from string to int
    int nb = atoi(JSONLblVal(propNb));

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : run benchmark\n");
    }

    // Run the benchmark function
    result = TheSquidBenchmark(nb,
      JSONLblVal(propPayloadSize));

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that),
        " : benchmark complete\n");
    }

    // Set the flag for successfull process
    success = true;

  // Else, at least one value was missing
  } else {

    // Update the error message
    sprintf(errMsg, "missing data (nb or v)");

    if (SquidletStreamInfo(that)){
      SquidletPrint(that, SquidletStreamInfo(that));
      fprintf(SquidletStreamInfo(that), " : missing data (nb or v)\n");
    }
  }

// Else, we couldn't decode the JSON
} else {

  // Update the error message
  sprintf(errMsg, "couldn't load json (%s)", JSONErr->_msg);

  if (SquidletStreamInfo(that)){
    SquidletPrint(that, SquidletStreamInfo(that));
    fprintf(SquidletStreamInfo(that),
      " : couldn't load json %s\n", buffer);
  }
}

// Free memory
JSONFree(&json);

// Update the time used to process the task
struct timeval now;
gettimeofday(&now, NULL);
that->_timeToProcessMs =
  (now.tv_sec - start.tv_sec) * 1000 +
```

```
    (now.tv_usec - start.tv_usec) / 1000;

  // Update the number of completed tasks if it was successfull
  if (success == true) {
    ++(that->_nbTaskComplete);
  }

  // Prepare the result data as JSON
  JSONNode* jsonResult = JSONCreate();
  float temperature = SquidletGetTemperature(that);
  // This software is not guaranteed to run under temperature having
  // more than 7 digits, you've be warned !
  char temperatureStr[10] = {'\0'};
  sprintf(temperatureStr, "%.2f", temperature);
  JSONAddProp(jsonResult, "temperature", temperatureStr);
  char successStr[2] = {'\0'};
  sprintf(successStr, "%d", success);
  JSONAddProp(jsonResult, "success", successStr);
  // Result is an int, 10 digits will be fine
  char resultStr[10] = {'\0'};
  sprintf(resultStr, "%d", result);
  JSONAddProp(jsonResult, "v", resultStr);
  JSONAddProp(jsonResult, "err", errMsg);

  // Append the statistics data
  SquidletAddStatsToJSON(that, jsonResult);

  // Convert the JSON to a string and store it in the result buffer
  *bufferResult = PBErrMalloc(TheSquidErr, THESQUID_MAXPAYLOADSIZE);
  memset(*bufferResult, 0, THESQUID_MAXPAYLOADSIZE);
  bool compact = true;
  ret = JSONSaveToStr(jsonResult,
    *bufferResult, THESQUID_MAXPAYLOADSIZE, compact);
  if (ret == false) {
    sprintf(*bufferResult,
      "{\"success\":\"0\",\"temperature\":\"0.0\","
      "\"err\":\"JSONSaveToStr failed\"}");
  }

}

// Process a Pov-Ray task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_PovRay(
    Squidlet* const that,
  const char* const buffer,
             char** bufferResult) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif
  // Declare a variable to memorize if the process has been successful
  bool success = false;

  // Display info
  if (SquidletStreamInfo(that)) {
```

```
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
    " : process Pov-Ray task %s\n", buffer);
  fflush(SquidletStreamInfo(that));
}

// Decode the input from JSON
JSONNode* json = JSONCreate();
bool ret = JSONLoadFromStr(json, buffer);

// If we could decode the JSON
if (ret == true) {

  // Get the arguments
  JSONNode* propIni = JSONProperty(json, "ini");
  JSONNode* propTga = JSONProperty(json, "tga");
  JSONNode* propTop = JSONProperty(json, "top");
  JSONNode* propLeft = JSONProperty(json, "left");
  JSONNode* propBottom = JSONProperty(json, "bottom");
  JSONNode* propRight = JSONProperty(json, "right");

  // If all the arguments are presents
  if (propIni != NULL && propTga != NULL && propTop != NULL &&
    propLeft != NULL && propBottom != NULL && propRight != NULL) {

    // Create the Pov-Ray command
    // povray +SC<left> +SR<top> +EC<right> +ER<bottom> +O<tga>
    //   +FT -D <ini>
    char cmd[500];
    sprintf(cmd,
      "povray %s +SC%s +SR%s +EC%s +ER%s +O%s +FT -D",
      JSONLblVal(propIni),
      JSONLblVal(propLeft),
      JSONLblVal(propTop),
      JSONLblVal(propRight),
      JSONLblVal(propBottom),
      JSONLblVal(propTga));

    // Execute the Pov-Ray command
    int ret = system(cmd);

    if (ret == 0) {
      // Set the flag for successfull process
      success = true;
    }
  }
}

// Update the number of completed tasks if it was successfull
if (success == true) {
  ++(that->_nbTaskComplete);
}

// Prepare the result data as JSON
*bufferResult = PBErrMalloc(TheSquidErr, THESQUID_MAXPAYLOADSIZE);
memset(*bufferResult, 0, THESQUID_MAXPAYLOADSIZE);
char successStr[2] = {'\0'};
sprintf(successStr, "%d", success);
JSONAddProp(json, "success", successStr);
float temperature = SquidletGetTemperature(that);
// This software is not guaranteed to run under temperature having
// more than 7 digits, you've be warned !
```

```
  char temperatureStr[10] = {'\0'};
  sprintf(temperatureStr, "%.2f", temperature);
  JSONAddProp(json, "temperature", temperatureStr);

  // Append the statistics data
  SquidletAddStatsToJSON(that, json);
  ret = JSONSaveToStr(json,
    *bufferResult, THESQUID_MAXPAYLOADSIZE, true);
  if (ret == false) {
    sprintf(*bufferResult,
      "{\"success\":\"0\",\"temperature\":\"0.0\","
      "\"err\":\"JSONSaveToStr failed\"}");
  }

  // Free memory
  JSONFree(&json);

}

// Process a stats reset task request with the Squidlet 'that'
void SquidletProcessRequest_StatsReset(
    Squidlet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Reset the stats
  SquidletResetStats(that);
}

// Process a neuranet evaluation task request with the Squidlet 'that'
// The task request parameters are encoded in JSON and stored in the
// string 'buffer'
// The result of the task are encoded in JSON format and stored in
// 'bufferResult' which is allocated as necessary
void SquidletProcessRequest_EvalNeuranet(
    Squidlet* const that,
  const char* const buffer,
            char** bufferResult) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
  if (buffer == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'buffer' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Declare a variable to memorize if the process has been successful
  bool success = false;

  // Allocate memory for the result
  *bufferResult = PBErrMalloc(TheSquidErr, THESQUID_MAXPAYLOADSIZE);
  memset(*bufferResult, 0, THESQUID_MAXPAYLOADSIZE);
```

```
// Start measuring the time used to process the task
that->_timeToProcessMs = 0;
struct timeval start;
gettimeofday(&start, NULL);

// Process the data
if (SquidletStreamInfo(that)){
  SquidletPrint(that, SquidletStreamInfo(that));
  fprintf(SquidletStreamInfo(that),
    " : process neuranet evaluation task %s\n", buffer);
}

// Decode the input from JSON
JSONNode* json = JSONCreate();
bool ret = JSONLoadFromStr(json, buffer);

// If we could decode the input
if (ret == true) {

  // Get the dataset path
  JSONNode* propDataset = JSONProperty(json, "dataset");

  // Get the working dir path
  JSONNode* propWorkingDir = JSONProperty(json, "workingDir");

  // Get the entity ids
  JSONNode* propIds = JSONProperty(json, "nnids");
  VecLong* nnids = NULL;
  VecDecodeAsJSON(&nnids, propIds);

  // Get the best value
  JSONNode* propBest = JSONProperty(json, "best");

  // Get the category in dataset
  JSONNode* propCat = JSONProperty(json, "cat");

  // If all the values are present
  if (propDataset != NULL &&
    propWorkingDir != NULL &&
    propIds != NULL &&
    propBest != NULL &&
    propCat != NULL) {

    // Convert the ids, cat and best value from string
    long cat = atol(JSONLblVal(propCat));
    float bestVal = atof(JSONLblVal(propBest));

    // If the dataset in argument is different from the last one used
    if (that->_datasetPath == NULL ||
      strcmp(that->_datasetPath, JSONLblVal(propDataset)) != 0) {

      // Free the current dataset
      GDataSetVecFloatFreeStatic(&(that->_dataset));

      // Load the requested dataset
      that->_dataset = GDataSetVecFloatCreateStaticFromFile(
        JSONLblVal(propDataset));

      // Memorize the path
      if (that->_datasetPath != NULL)
        free(that->_datasetPath);
```

```
  that->_datasetPath = strdup(JSONLblVal(propDataset));

}

// If we could load the dataset
if (GDSGetSizeCat(&(that->_dataset), cat) > 0) {

  // Declare a variable to memorize the values
  VecFloat* values = VecFloatCreate(VecGetDim(nnids));

  // Set the flag for successfull process by default
  success = true;

  // Loop on the NeuraNet to evaluate
  for (int iNN = 0; iNN < VecGetDim(nnids); ++iNN) {

    // Load the Neuranet
    NeuraNet* nn = NULL;
    char nnFilename[100];
    sprintf(nnFilename, "nn%ld.json", VecGet(nnids, iNN));
    char* pathNN = PBFSJoinPath(
      JSONLblVal(propWorkingDir),
      nnFilename);
    FILE* fpnn = fopen(pathNN, "r");
    free(pathNN);
    if (fpnn != NULL && NNLoad(&nn, fpnn)) {

      // Close the file pointer to the Neuranet definition
      // file
      fclose(fpnn);

      // Create the inputs and outputs vector from the
      // dimension of the NeuraNet
      // The sample values must be ordered as follow
      // <i0, i1, ..., in, o0, o1, ..., om>
      VecShort* inputs = VecShortCreate(NNGetNbInput(nn));
      for (unsigned int i = VecGetDim(inputs); i--;)
        VecSet(inputs, i, i);
      VecShort* outputs = VecShortCreate(NNGetNbOutput(nn));
      for (unsigned int i = VecGetDim(outputs); i--;)
        VecSet(outputs, i, VecGetDim(inputs) + i);

      // Run the evaluation of the neuranet on the dataset
      VecSet(
        values,
        iNN,
        GDSEvaluateNN(
          &(that->_dataset),
          nn,
          cat,
          inputs,
          outputs,
          bestVal));

      // Free memory
      VecFree(&inputs);
      VecFree(&outputs);
      NeuraNetFree(&nn);

    // Else, we couldn't load the Neuranet
    } else {
```

```
      // Set the successfull flag to false
      success = false;

    }
  }

  if (success == true) {

    // Update the time used to process the task
    struct timeval now;
    gettimeofday(&now, NULL);
    that->_timeToProcessMs =
      (now.tv_sec - start.tv_sec) * 1000 +
      (now.tv_usec - start.tv_usec) / 1000;

    // Update the number of completed tasks
    ++(that->_nbTaskComplete);

    // Prepare the result data as JSON
    JSONNode* jsonResult = JSONCreate();
    float temperature = SquidletGetTemperature(that);
    char temperatureStr[10] = {'\0'};
    sprintf(temperatureStr, "%.2f", temperature);
    JSONAddProp(jsonResult, "temperature", temperatureStr);
    char successStr[2] = {'\0'};
    sprintf(successStr, "%d", success);
    JSONAddProp(jsonResult, "success", successStr);
    JSONAddProp(jsonResult, "nnids", propIds);

    JSONNode* jsonValues = VecEncodeAsJSON(values);
    JSONAddProp(jsonResult, "v", jsonValues);

    // Append the statistics data
    SquidletAddStatsToJSON(that, jsonResult);

    // Convert the JSON to a string
    bool compact = true;
    ret = JSONSaveToStr(jsonResult, *bufferResult,
      THESQUID_MAXPAYLOADSIZE, compact);
    if (ret == false) {
      sprintf(*bufferResult,
        "{\"success\":\"0\",\"temperature\":\"0.0\","
        "\"err\":\"JSONSaveToStr failed\"}");
    }

  // Else, we couldn't load the neuranet
  } else {

    sprintf(*bufferResult,
      "{\"success\":\"0\",\"temperature\":\"0.0\","
      "\"err\":\"Invalid neuranet\"}");
  }

  // Free memory
  VecFree(&values);

// Else, the dataset could not be loaded or was empty
} else {

  sprintf(*bufferResult,
    "{\"success\":\"0\",\"temperature\":\"0.0\","
    "\"err\":\"Invalid dataset\"}");
```

```
      }

    // else the value is not present
    } else {

      sprintf(*bufferResult,
        "{\"success\":\"0\",\"temperature\":\"0.0\","
        "\"err\":\"Invalid input\"}");
    }

    // Free memory
    VecFree(&nnids);

  // Else, we couldn't decode the input
  } else {

    sprintf(*bufferResult,
      "{\"success\":\"0\",\"temperature\":\"0.0\","
      "\"err\":\"JSONLoadFromStr failed\"}");
  }

  // Free memory
  JSONFree(&json);
}

// Return the temperature of the squidlet 'that' as a float.
// The result depends on the architecture on which the squidlet is
// running. It is '0.0' if the temperature is not available
float SquidletGetTemperature(
  const Squidlet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'that' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // The Squidlet's info are not used
  (void)that;

#if BUILDARCH == 0
  return 0.0;
#endif

#if BUILDARCH == 1
  // Declare a variable to pipe the shell command
  FILE* fp = NULL;

  // Run the command and pipe its output
  fp = popen("vcgencmd measure_temp", "r");

  // If we could execute the command
  if (fp != NULL) {

    // Declare a variable to store the output
    char output[100] = {0};

    // Read the output, expected to be as:
    // temp=42.8'C
    while (fgets(output, sizeof(output), fp) != NULL);
```

```
    // Close the pipe
    pclose(fp);

    // Remove the line return and two last characters
    if (strlen(output) > 0)
      output[strlen(output) - 3] = '\0';

    // Convert the output to a float
    float t = 0.0;
    sscanf(output + 5, "%f", &t);

    // Return the result
    return t;

  // Else, the command failed
  } else {

    // Return the default result
    return 0.0;
  }
#endif

#if BUILDARCH == 2
  // Declare a variable to pipe the shell command
  FILE* fp = NULL;

  // Run the command and pipe its output
  fp = popen("vcgencmd measure_temp", "r");

  // If we could execute the command
  if (fp != NULL) {

    // Declare a variable to store the output
    char output[100] = {0};

    // Read the output, expected to be as:
    // temp=42.8'C
    while (fgets(output, sizeof(output), fp) != NULL);

    // Close the pipe
    pclose(fp);

    // Remove the line return and two last characters
    if (strlen(output) > 0)
      output[strlen(output) - 3] = '\0';

    // Convert the output to a float
    float t = 0.0;
    sscanf(output + 5, "%f", &t);

    // Return the result
    return t;

  // Else, the command failed
  } else {

    // Return the default result
    return 0.0;
  }
#endif
}
```

```
// -------------- TheSquid

// ================ Functions implementation ====================

// Function for benchmark purpose
// Create a set of null element of size equals to the size of the
// buffer in the task data, and sort it 10 times the 'nbLoop' in the
// task data, using each time diferrent sorting values for each element
int TheSquidBenchmark(
                int nbLoop,
  const char* const buffer) {

  // Variable to memorize the dummy result
  int res = 0;

  // Loop on sample code
  for (int iLoop = 0; iLoop < nbLoop; ++iLoop) {
    for (unsigned int scaling = 10; scaling--;) {
      GSet set = GSetCreateStatic();
      for(unsigned long i = strlen(buffer); i--;) {
        GSetPush(&set, NULL);
        set._head->_sortVal = (float)(i + scaling + iLoop);
      }
      GSetSort(&set);
      res = (int)round(set._head->_sortVal);
      GSetFlush(&set);
    }
  }

  // Return the dummy result
  return res;
}

// Function to receive in blocking mode 'nb' bytes of data from
// the socket 'sock' and store them into 'buffer' (which must be big
// enough). Give up after 'maxWait' seconds.
// Return true if we could read all the requested byte, false else
bool SocketRecv(short* sock, unsigned long nb, char* buffer,
  const time_t maxWait) {
#if BUILDMODE == 0
  if (sock == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'sock' is null");
    PBErrCatch(TheSquidErr);
  }
  if (buffer == NULL) {
    TheSquidErr->_type = PBErrTypeNullPointer;
    sprintf(TheSquidErr->_msg, "'buffer' is null");
    PBErrCatch(TheSquidErr);
  }
#endif

  // Open the socket in reading mode
  FILE* fp = fdopen(*sock, "r");

  // Declare a pointer to the next received byte and initialize it
  // on the first byte of the result buffer
  char* freadPtr = buffer;

  // Declare a pointer to the byte after the last received byte
  char* freadPtrEnd = freadPtr + nb;
```

```
  // Declare variables to memorize the start time and elapsed time
  time_t startTime = time(NULL);
  time_t elapsedTime = 0;

  // While we haven't received all the requested bytes and the time
  // limit is not reached
  do {

    // Try to read one more byte, if successful moves the pointer to
    // the next byte in the result buffer
    ssize_t nbReadByte = fread(freadPtr, 1, nb, fp);
    if (nbReadByte > 0) {
      freadPtr += nbReadByte;
    }

    // Update the elapsed time
    elapsedTime = time(NULL) - startTime;

  } while (freadPtr != freadPtrEnd && maxWait != 0 &&
    elapsedTime < maxWait && !Squidlet_CtrlC);

  // Duplicate the socket to avoid it being killed when we close the
  // stream used to read the incoming bytes
  *sock = dup(*sock);

  // Close the stream
  fclose(fp);

  // Return the success/failure code
  if (freadPtr != freadPtrEnd) {
    return false;
  } else {
    return true;
  }

}
```

6.2 squidlet.c

```
// -------------- squidlet.c ---------------

// Include third party libraries
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Include own libraries
#include "thesquid.h"

// Main function for the Squidlet executable
int main(int argc, char** argv) {

  // Declare and initialise variables to process arguments
  int port = -1;
  uint32_t ip = 0;
  char* outputFilePath = NULL;

  // Loop on the arguments to process the prior arguments
  for (int iArg = 0; iArg < argc; ++iArg) {
```

```
  // -ip <a.b.c.d>
  if (strcmp(argv[iArg], "-ip") == 0 && iArg < argc - 1) {

    // Decode the IP adress to which the Squidlet is attached
    ++iArg;
    unsigned int v[4] = {0, 0, 0 ,0};
    int ret = sscanf(argv[iArg], "%d.%d.%d.%d",
      v, v + 1, v + 2, v + 3);

    // If we couldn't decode the IP
    if (ret == EOF) {
      fprintf(stderr, "Failed to decode the IP adress.\n");
      return 1;
    }

    // Encode the IP into a 4 bytes integer
    for (int i = 0; i < 4; ++i) {
      ((unsigned char*)(&ip))[i] = v[i];
    }

  }

  // -port <port>
  if (strcmp(argv[iArg], "-port") == 0 && iArg < argc - 1) {

    // Decode the value of the port on which the Squidlet is listening
    ++iArg;
    port = atoi(argv[iArg]);

  }

  // -stream <stdout | file path>
  if (strcmp(argv[iArg], "-stream") == 0 && iArg < argc - 1) {

    // Decode the output stream
    ++iArg;
    outputFilePath = argv[iArg];

  }

  // -help
  if (strcmp(argv[iArg], "-help") == 0) {

    // Display the help message and quit
    printf("squidlet [-ip <a.b.c.d>] [-port <port>] ");
    printf("[-stream <stdout | file path>] [-temp] [-help]\n");
    return 0;

  }
}

// Create the squidlet
Squidlet* squidlet = SquidletCreateOnPort(ip, port);

// If we couldn't create the Squidlet
if (squidlet == NULL) {
  fprintf(stderr, "Failed to create the squidlet\n");
  fprintf(stderr, "TheSquidErr: %s\n", TheSquidErr->_msg);
  fprintf(stderr, "errno: %s\n", strerror(errno));
  return 2;
}
```

```
// Display info about the Squidlet:
// <pid> <hostname> <ip>:<port>
printf("Squidlet : ");
SquidletPrint(squidlet, stdout);
printf("\n");

// Set the output stream to stdout by default
FILE* stream = stdout;

// If the user provided an output stream
if (outputFilePath != NULL) {

  // If the user requested the standard output stream
  if (strcmp(outputFilePath, "stdout") == 0) {

    // Set the output stream of the Squidlet to the standard output
    SquidletSetStreamInfo(squidlet, stdout);

  // else if the user requested a different stream
  } else if (outputFilePath != NULL) {

    // Open the requested stream
    stream = fopen(outputFilePath, "w");

    // If we couldn't open the requested stream
    if (stream == NULL) {
      fprintf(stderr, "Failed to open the file %s\n", outputFilePath);
      return 3;
    }

    // Set the output stream of the Squidlet to the requested stream
    SquidletSetStreamInfo(squidlet, stream);

  }

// Else, the user hasn't requested any output stream
} else {

  // Turn off the output stream of the Squidlet
  SquidletSetStreamInfo(squidlet, NULL);

}

// Loop on the arguments to process the posterior arguments
for (int iArg = 0; iArg < argc; ++iArg) {

  // -temp(erature)
  if (strcmp(argv[iArg], "-temp") == 0) {

    // Get the temperature of the Squidlet
    float temperature = SquidletGetTemperature(squidlet);

    // Display the temperature
    fprintf(stream, "Squidlet : temperature: %f\n", temperature);

  }
}

// Set the handler for SIGPIPE
signal(SIGPIPE, SquidletHandlerSigPipe);
```

```
  // Loop until the Squidlet is killed by Ctrl-C:
  // kill -INT <squidlet's pid>
  do {

    // Wait for a request
    SquidletTaskRequest request = SquidletWaitRequest(squidlet);

    // Process the received request
    SquidletProcessRequest(squidlet, &request);

  } while (!Squidlet_CtrlC);

  // Free memory
  SquidletFree(&squidlet);

  // Sayonara
  fprintf(stream, "Squidlet : ended\n");

  // Close the output stream if necessary
  if (stream != NULL && stream != stdout)
    fclose(stream);

  // Return success code
  return 0;
}
```

6.3 squad.c

```
// -------------- squidlet.c ---------------

// Include third party libraries
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Include own libraries
#include "thesquid.h"

// Main function for the Squad executable
int main(int argc, char** argv) {

  // Init the random generator
  srand(time(NULL));

  // Declare and initialise variables to process arguments
  char* tasksFilePath = NULL;
  char* squidletsFilePath = NULL;
  bool flagTextOMeter = false;
  unsigned int freq = 1;

  // Loop on the arguments to process the prior arguments
  for (int iArg = 0; iArg < argc; ++iArg) {

    // -freq <delay in second between step>
    if (strcmp(argv[iArg], "-freq") == 0 && iArg < argc - 1) {

      // Memorize a pointer to the path to the task file
      ++iArg;
      freq = atoi(argv[iArg]);
```

```
  }

  // -tasks <path to tasks file>
  if (strcmp(argv[iArg], "-tasks") == 0 && iArg < argc - 1) {

    // Memorize a pointer to the path to the task file
    ++iArg;
    tasksFilePath = argv[iArg];

  }

  // -squidlets <path to squidlets file>
  if (strcmp(argv[iArg], "-squidlets") == 0 && iArg < argc - 1) {

    // Memorize a pointer to the path to the squidlet file
    ++iArg;
    squidletsFilePath = argv[iArg];

  }

  // -verbose
  if (strcmp(argv[iArg], "-verbose") == 0) {

    // Set teh fag to display messages in a TextOMeter
    flagTextOMeter = true;

  }

  // -help
  if (strcmp(argv[iArg], "-help") == 0) {

    // Display the help message and quit
    printf("squad [-squidlets <path/to/squidlets/config.file | ");
    printf("'{\"_squidlets\":[{\"_name\":\"name\",");
    printf("\"_ip\":\"a.b.c.d\",\"_port\":\"port\"}]}'> ");
    printf("[-verbose] [-tasks <path to tasks file>] ");
    printf("[-freq <delay in second between step, default: 1>] ");
    printf("[-check] [-benchmark] [-help]\n");
    return 0;

  }
}

// Create the squad
Squad* squad = SquadCreate();

// If we couldn't create the squad
if (squad == NULL) {

  fprintf(stderr, "Squad: Failed to create the squad\n");
  return 1;

// Else, we could create the squad
} else {

  printf("Squad : started\n");

}

// If the user has provided a squidlet configuration file
if (squidletsFilePath != NULL) {
```

```
  // Declare a variable to memorize if we could lod the squidlet
  // configuration
  bool retLoadSquidlets = false;

  // If the Squidlet configuration is inlined
  if (squidletsFilePath[0] == '{') {

    // Load the content of the squidlets file
    retLoadSquidlets = SquadLoadSquidletsFromStr(squad, squidletsFilePath);

  // Else, the provided argument must really be a file path
  } else {
    // Open the squidlets file
    FILE* squidletsFile = fopen(squidletsFilePath, "r");

    // If we couldn't open the squidlets file
    if (squidletsFile == NULL) {

        // Print an error message
        fprintf(stderr, "Squad: Couldn't open the squidlets file: %s\n",
          squidletsFilePath);
        fprintf(stderr, "errno: %s\n", strerror(errno));

        // Free memory
        SquadFree(&squad);

        // Stop here
        return 2;

    // Else, we could open the squidlets file
    } else {

      // Load the content of the squidlets file
      retLoadSquidlets = SquadLoadSquidlets(squad, squidletsFile);

      // Close the squidlets file
      fclose(squidletsFile);

    }

  }

  // If we couldn't load the squidlets configuration
  if (retLoadSquidlets == false) {

    // Print an error message
    fprintf(stderr,
      "Squad: Couldn't load the squidlets config file %s\n",
      squidletsFilePath);
    fprintf(stderr, "TheSquidErr: %s\n", TheSquidErr->_msg);

    // Free memory
    SquadFree(&squad);

    // Stop here
    return 3;

  }

}
```

```
// Set the TextOMeter accordingly to the -verbose argument
SquadSetFlagTextOMeter(squad, flagTextOMeter);

// Loop on the arguments to process the posterior arguments
for (int iArg = 0; iArg < argc; ++iArg) {

  // -check
  if (strcmp(argv[iArg], "-check") == 0) {

    // Check that the squidlets described in the squidlets file
    // are up and running
    bool resCheckSquidlets = SquadCheckSquidlets(squad, stdout);

    // If at least one of the squidlets is not responding
    if (resCheckSquidlets == false) {

      // Free memory
      SquadFree(&squad);

      // Stop here
      return 4;

    }

  }

  // -benchmark
  if (strcmp(argv[iArg], "-benchmark") == 0) {

    // Run the standard benchmark on the loaded squidlets
    SquadBenchmark(squad, stdout);

  }

}

// If the user has provided a tasks file
if (tasksFilePath != NULL) {

  printf("Squad: Executing task file: %s\n", tasksFilePath);

  // Open the task file
  FILE* tasksFile = fopen(tasksFilePath, "r");

  // If we couldn't open the tasks file
  if (tasksFile == NULL) {

      // Print an error message
      fprintf(stderr, "Squad: Couldn't open the tasks file: %s\n",
        tasksFilePath);
      fprintf(stderr, "errno: %s\n", strerror(errno));

      // Free memory
      SquadFree(&squad);

      // Stop here
      return 5;

  // Else, we could open the tasks file
  } else {

    // Load the task file
```

```
bool retLoadTasks = SquadLoadTasks(squad, tasksFile);

// If we couldn't load the tasks file
if (retLoadTasks == false) {

  // Print an error message
  fprintf(stderr, "Squad: Couldn't load the tasks file %s\n",
    tasksFilePath);
  fprintf(stderr, "TheSquidErr: %s\n", TheSquidErr->_msg);

  // Free memory
  SquadFree(&squad);
  fclose(tasksFile);

  // Stop here
  return 6;
}

// Close the tasks file
fclose(tasksFile);

// Loop as long as there are task to complete
while (SquadGetNbTaskToComplete(squad) > 0) {

  // Sleep between each step of the Squad
  sleep(freq);

  // Step the squad and get the completed tasks at this step
  GSetSquadRunningTask completedTasks = SquadStep(squad);

  // While there are completed tasks
  while (GSetNbElem(&completedTasks) > 0) {

    // Pop the completed task
    SquadRunningTask* completedTask = GSetPop(&completedTasks);
    SquidletTaskRequest* task = completedTask->_request;

    // Display the completed task
    printf("Squad : ");
    SquidletTaskRequestPrint(task, stdout);

    // If the task has succeeded
    if (SquidletTaskHasSucceeded(task)) {

      printf(" succeeded\n");

    // Else, the task has failed
    } else {

      printf(" failed !!\n");

      // Put the task back into the set of tasks to complete
      completedTask->_request = NULL;
      SquadTryAgainTask(squad, task);

    }

    // Free the completed task
    SquadRunningTaskFree(&completedTask);

  }
```

```
      }

    }

  }

  // Free memory
  SquadFree(&squad);

  printf("Squad : ended\n");

  // Return success code
  return 0;
}
```

7 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: pbmake_wget main squidlet squad

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=thesquid
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$

freeports:
echo "\nThe list of used ports is given by\nsudo netstat -tunlep | grep LISTEN\nand\ncat /etc/services\nalso, possibl

squidlet: \
squidlet.o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) squidlet.o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG) -o squidle

squidlet.o: \
$($(repo)_DIR)/squidlet.c \
```

```
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/s

valgrind_squidlet :
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./squidlet

squad: \
squad.o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) squad.o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG) -o squad

squad.o: \
$($(repo)_DIR)/squad.c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/s

valgrind_squad :
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./squad

lsPortListeners:
lsof -n | grep LISTEN
```

8 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include "thesquid.h"

void UnitTestSquad() {
  Squad* squad = SquadCreate();
  if (squad == NULL) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquadCreate failed");
    PBErrCatch(TheSquidErr);
  }
  FILE* fp = fopen("unitTestSquad.json", "r");
  if (fp == NULL || SquadLoadSquidlets(squad, fp) == false) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquadLoad failed");
    PBErrCatch(TheSquidErr);
  }
  fclose(fp);
  SquadFree(&squad);
  if (squad != NULL) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquadFree failed");
    PBErrCatch(TheSquidErr);
  }
  printf("UnitTestSquad OK\n");
}

void UnitTestSquadCheckSquidlets() {
  const int nbSquidlet = 2;
  int squidletId = -1;
```

```
int port[2] = {9000, 9001};
pid_t pidSquidlet[2];
char buffer[100];
// Create the squidlet processes
for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
  int pid = fork();
  if (pid == 0) {
    squidletId = iSquidlet;
    break;
  } else {
    pidSquidlet[iSquidlet] = pid;
  }
}
if (squidletId != -1) {

  // In a squidlet process

  Squidlet* squidlet = SquidletCreateOnPort(0, port[squidletId]);
  if (squidlet == NULL) {
    printf("Failed to create the squidlet #%d\n", squidletId);
    printf("errno: %s\n", strerror(errno));
  }
  sprintf(buffer, "unitTestDummySquidlet%d.log", squidletId);
  FILE* stream = fopen(buffer, "w");
  SquidletSetStreamInfo(squidlet, stream);
  do {
    SquidletTaskRequest request = SquidletWaitRequest(squidlet);
    SquidletProcessRequest(squidlet, &request);
  } while (!Squidlet_CtrlC);
  SquidletFree(&squidlet);
  fclose(stream);
  exit(0);
} else {

  // In the squad process

  // Create the Squad
  Squad* squad = SquadCreate();
  if (squad == NULL) {
    printf("Failed to create the squad\n");
    printf("errno: %s\n", strerror(errno));
  }
  // Automatically create the config file
  FILE* fp = fopen("unitTestDummy.json", "w");
  char hostname[256];
  gethostname(hostname, sizeof(hostname));
  struct hostent* host = gethostbyname(hostname);
  char* ip = inet_ntoa(*((struct in_addr*)host->h_addr_list[0]));
  fprintf(fp, "{\"_squidlets\":[");
  for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
    fprintf(fp,
      "{\"_name\":\"UnitTestDummy\"\"_ip\":\"%s\",\"_port\":\"%d\"}",
      ip, port[iSquidlet]);
    if (iSquidlet < nbSquidlet - 1)
      fprintf(fp, ",");
  }
  fprintf(fp, "]}");
  fclose(fp);
  // Load the info about the squidlet from the config file
  fp = fopen("unitTestDummy.json", "r");
  SquadLoadSquidlets(squad, fp);
  fclose(fp);
```

```
    // Wait to be sure the squidlets are up and running
    sleep(2);
    // Check the squidlets
    bool res = SquadCheckSquidlets(squad, stdout);
    // Kill the child process
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      if (kill(pidSquidlet[iSquidlet], SIGINT) < 0) {
        printf("Couldn't kill squidlet %d\n", pidSquidlet[iSquidlet]);
      }
    }
    // Wait for the child to be killed
    sleep(2);
    // Free memory
    SquadFree(&squad);
    if (res) {
      printf("UnitTestSquadCheckSquidlets OK\n");
    } else {
      printf("UnitTestSquadCheckSquidlets NG\n");
    }
    fflush(stdout);
  }
}

void UnitTestSquidlet() {
  Squidlet* squidlet = SquidletCreate();
  if (squidlet == NULL) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquidletCreate failed");
    PBErrCatch(TheSquidErr);
  }
  SquidletPrint(squidlet, stdout);
  printf("\n");
  float temperature = SquidletGetTemperature(squidlet);
  printf("squidlet temperature: %f\n", temperature);
  SquidletFree(&squidlet);
  if (squidlet != NULL) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquidletFree failed");
    PBErrCatch(TheSquidErr);
  }
  printf("UnitTestSquidlet OK\n");
}

void UnitTestDummy() {
  const int nbSquidlet = 2;
  int squidletId = -1;
  int port[2] = {9000, 9001};
  int nbRequest = 3;
  pid_t pidSquidlet[2];
  char buffer[100];
  // Create the squidlet processes
  for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
    int pid = fork();
    if (pid == 0) {
      squidletId = iSquidlet;
      break;
    } else {
      pidSquidlet[iSquidlet] = pid;
    }
  }
  if (squidletId != -1) {
```

```
    // In a squidlet process

    Squidlet* squidlet = SquidletCreateOnPort(0, port[squidletId]);
    if (squidlet == NULL) {
      printf("Failed to create the squidlet #%d\n", squidletId);
      printf("errno: %s\n", strerror(errno));
    }
    sprintf(buffer, "unitTestDummySquidlet%d.log", squidletId);
    FILE* stream = fopen(buffer, "w");
    SquidletSetStreamInfo(squidlet, stream);
    printf("Squidlet #%d : ", squidletId);
    SquidletPrint(squidlet, stdout);
    printf("\n");
    do {
      SquidletTaskRequest request = SquidletWaitRequest(squidlet);
      SquidletProcessRequest(squidlet, &request);
    } while (!Squidlet_CtrlC);
    SquidletFree(&squidlet);
    fclose(stream);
    printf("Squidlet #%d ended\n", squidletId);
    fflush(stdout);
    exit(0);
  } else {

    // In the squad process

    // Create the Squad
    Squad* squad = SquadCreate();
    if (squad == NULL) {
      printf("Failed to create the squad\n");
      printf("errno: %s\n", strerror(errno));
    }
#if BUILDARCH == 0
    // Turn on the TextOMeter
    SquadSetFlagTextOMeter(squad, true);
#endif
    // Automatically create the config file
    FILE* fp = fopen("unitTestDummy.json", "w");
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    struct hostent* host = gethostbyname(hostname);
    char* ip = inet_ntoa(*((struct in_addr*)host->h_addr_list[0]));
    fprintf(fp, "{\"_squidlets\":[");
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      fprintf(fp,
        "{\"_name\":\"UnitTestDummy\"\"_ip\":\"%s\",\"_port\":\"%d\"}",
        ip, port[iSquidlet]);
      if (iSquidlet < nbSquidlet - 1)
        fprintf(fp, ",");
    }
    fprintf(fp, "]}");
    fclose(fp);
    // Load the info about the squidlet from the config file
    fp = fopen("unitTestDummy.json", "r");
    SquadLoadSquidlets(squad, fp);
    fclose(fp);
    // Wait to be sure the squidlets are up and running
    sleep(2);
    // Create all the tasks
    time_t maxWait = 5;
    for (unsigned long id = 0;
      id < (unsigned long)(nbRequest * nbSquidlet); ++id) {
```

```
      SquadAddTask_Dummy(squad, id, maxWait);
    }
    // Loop until all the tasks are completed or give up after 60s
    time_t startTime = time(NULL);
    bool flagStop = false;
    do {

      // Step the Squad
      GSetSquadRunningTask completedTasks = SquadStep(squad);
      sleep(1);
      while (GSetNbElem(&completedTasks) > 0L) {
        SquadRunningTask* completedTask = GSetPop(&completedTasks);
        SquidletTaskRequest* task = completedTask->_request;
        printf("squad : ");
        SquidletTaskRequestPrint(task, stdout);
        if (strstr(task->_bufferResult, "\"success\":\"1\"") == NULL) {
          printf(" failed !!\n");
          flagStop = true;
        } else {
          printf(" succeeded\n");
        }
        SquadRunningTaskFree(&completedTask);
      }

    } while (SquadGetNbTaskToComplete(squad) > 0L &&
      time(NULL) - startTime <= 60 && !flagStop);
    // Kill the child process
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      if (kill(pidSquidlet[iSquidlet], SIGINT) < 0) {
        printf("Couldn't kill squidlet %d\n", pidSquidlet[iSquidlet]);
      }
    }
    // Wait for the child to be killed
    sleep(2);
    // Free memory
    SquadFree(&squad);
    printf("Squad ended\n");
    printf("UnitTestDummy OK\n");
    fflush(stdout);
  }
}

void UnitTestPovRay() {
  int ret = system("povray ./testPov.ini +OtestPovRef.tga +FT -D");
  (void)ret;
  const int nbSquidlet = 2;
  int squidletId = -1;
  int port[2] = {9000, 9001};
  pid_t pidSquidlet[2];
  // Create the squidlet processes
  for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
    int pid = fork();
    if (pid == 0) {
      squidletId = iSquidlet;
      break;
    } else {
      pidSquidlet[iSquidlet] = pid;
    }
  }
  if (squidletId != -1) {

    // In a squidlet process
```

```
    Squidlet* squidlet = SquidletCreateOnPort(0, port[squidletId]);
    if (squidlet == NULL) {
      printf("Failed to create the squidlet #%d\n", squidletId);
      printf("errno: %s\n", strerror(errno));
    }
    printf("Squidlet #%d : ", squidletId);
    SquidletPrint(squidlet, stdout);
    printf("\n");
    do {
      SquidletTaskRequest request = SquidletWaitRequest(squidlet);
      SquidletProcessRequest(squidlet, &request);
    } while (!Squidlet_CtrlC);
    SquidletFree(&squidlet);
    printf("Squidlet #%d ended\n", squidletId);
    fflush(stdout);
    exit(0);
  } else {

    // In the squad process

    // Create the Squad
    Squad* squad = SquadCreate();
    if (squad == NULL) {
      printf("Failed to create the squad\n");
      printf("errno: %s\n", strerror(errno));
    }
#if BUILDARCH == 0
    // Turn on the TextOMeter
    SquadSetFlagTextOMeter(squad, true);
#endif
    // Automatically create the config file
    FILE* fp = fopen("unitTestPovRay.json", "w");
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    struct hostent* host = gethostbyname(hostname);
    char* ip = inet_ntoa(*((struct in_addr*)host->h_addr_list[0]));
    fprintf(fp, "{\"_squidlets\":[");
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      fprintf(fp,
        "{\"_name\":\"UnitTestPovRay\"\"_ip\":\"%s\",\"_port\":\"%d\"}",
        ip, port[iSquidlet]);
      if (iSquidlet < nbSquidlet - 1)
        fprintf(fp, ",");
    }
    fprintf(fp, "]}");
    fclose(fp);
    // Load the info about the squidlet from the config file
    fp = fopen("unitTestPovRay.json", "r");
    SquadLoadSquidlets(squad, fp);
    fclose(fp);
    // Wait to be sure the squidlets are up and running
    sleep(2);
    // Create the task
    time_t maxWait = 600;
    int id = 1;
    unsigned int sizeMinFragment = 100;
    unsigned int sizeMaxFragment = 1000;
    SquadAddTask_PovRay(squad, id, maxWait, "./testPov.ini",
      sizeMinFragment, sizeMaxFragment);
    // Loop until all the tasks are completed or giveup after 60s
    time_t startTime = time(NULL);
```

```
    bool flagStop = false;
    do {
      sleep(1);
      // Step the Squad
      GSetSquadRunningTask completedTasks = SquadStep(squad);
      while (GSetNbElem(&completedTasks) > 0L) {
        SquadRunningTask* completedTask = GSetPop(&completedTasks);
        SquidletTaskRequest* task = completedTask->_request;
        printf("squad : ");
        SquidletTaskRequestPrint(task, stdout);
        if (strstr(task->_bufferResult, "\"success\":\"1\"") == NULL) {
          printf(" failed !!\n");
          flagStop = true;
        } else {
          printf(" succeeded\n");
        }
        SquadRunningTaskFree(&completedTask);
      }

    } while (SquadGetNbTaskToComplete(squad) > 0L &&
      time(NULL) - startTime <= 60 && !flagStop);
    // Kill the child process
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      if (kill(pidSquidlet[iSquidlet], SIGINT) < 0) {
        printf("Couldn't kill squidlet %d\n", pidSquidlet[iSquidlet]);
      }
    }
    // Wait for the child to be killed
    sleep(2);
    // Compare the result to the reference
    GenBrush* result = GBCreateFromFile("./testPov.tga");
    GenBrush* ref = GBCreateFromFile("./testPovRef.tga");
    if (result == NULL || GBIsSameAs(result, ref) == false) {
      TheSquidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(TheSquidErr->_msg, "UnitTestPovRay failed");
      TheSquidErr->_fatal = false;
      //PBErrCatch(TheSquidErr);
      TheSquidErr->_fatal = true;
    } else {
      printf("UnitTestPovRay OK\n");
    }
    // Free memory
    SquadFree(&squad);
    GBFree(&result);
    GBFree(&ref);
  }
}

void UnitTestLoadTasks() {
  Squad* squad = SquadCreate();
  FILE* stream = fopen("./testLoadTasks.json","r");
  if (!SquadLoadTasks(squad, stream)) {
    TheSquidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(TheSquidErr->_msg, "SquadLoadTasks failed");
    PBErrCatch(TheSquidErr);
  }
  fclose(stream);
  SquadFree(&squad);
  printf("UnitTestLoadTasks OK\n");
}

void UnitTestEvalNeuranet() {
```

```
  const int nbSquidlet = 2;
  int squidletId = -1;
  int port[2] = {9000, 9001};
  pid_t pidSquidlet[2];
  char buffer[100];
  // Create the squidlet processes
  for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
    int pid = fork();
    if (pid == 0) {
      squidletId = iSquidlet;
      break;
    } else {
      pidSquidlet[iSquidlet] = pid;
    }
  }
  if (squidletId != -1) {

    // In a squidlet process

    Squidlet* squidlet = SquidletCreateOnPort(0, port[squidletId]);
    if (squidlet == NULL) {
      printf("Failed to create the squidlet #%d\n", squidletId);
      printf("errno: %s\n", strerror(errno));
    }
    sprintf(buffer, "unitTestEvalNeuranetSquidlet%d.log", squidletId);
    FILE* stream = fopen(buffer, "w");
    SquidletSetStreamInfo(squidlet, stream);
    printf("Squidlet #%d : ", squidletId);
    SquidletPrint(squidlet, stdout);
    printf("\n");
    do {
      SquidletTaskRequest request = SquidletWaitRequest(squidlet);
      SquidletProcessRequest(squidlet, &request);
    } while (!Squidlet_CtrlC);
    SquidletFree(&squidlet);
    fclose(stream);
    printf("Squidlet #%d ended\n", squidletId);
    fflush(stdout);
    exit(0);
  } else {

    // In the squad process

    // Create the Squad
    Squad* squad = SquadCreate();
    if (squad == NULL) {
      printf("Failed to create the squad\n");
      printf("errno: %s\n", strerror(errno));
    }
#if BUILDARCH == 0
    // Turn on the TextOMeter
    SquadSetFlagTextOMeter(squad, true);
#endif
    // Automatically create the config file
    FILE* fp = fopen("unitTestEvalNeuranet.json", "w");
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    struct hostent* host = gethostbyname(hostname);
    char* ip = inet_ntoa(*((struct in_addr*)host->h_addr_list[0]));
    fprintf(fp, "{\"_squidlets\":[");
    for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
      fprintf(fp,
```

```
    "{\"_name\":\"UnitTestEvalNeuranet\"\"_ip\":\"%s\",\"_port\":\"%d\"}",
    ip, port[iSquidlet]);
  if (iSquidlet < nbSquidlet - 1)
    fprintf(fp, ",");
}
fprintf(fp, "]}");
fclose(fp);
// Load the info about the squidlet from the config file
fp = fopen("unitTestEvalNeuranet.json", "r");
SquadLoadSquidlets(squad, fp);
fclose(fp);
// Wait to be sure the squidlets are up and running
sleep(2);

// Create all the tasks
time_t maxWait = 60;
float best = -1000.0;
int cat = 0;
VecLong* ids = VecLongCreate(2);
for (int id = 0; id < nbSquidlet * 2; ++id) {
  VecSet(ids, 0, id * 2);
  VecSet(ids, 1, id * 2 + 1);
  SquadAddTask_EvalNeuraNet(squad, id, maxWait,
    "./dataset.json", "./", ids, best, cat);
}
VecFree(&ids);

// Loop until all the tasks are completed or give up after 60s
time_t startTime = time(NULL);
bool flagStop = false;
do {

  // Step the Squad
  GSetSquadRunningTask completedTasks = SquadStep(squad);
  sleep(2);
  while (GSetNbElem(&completedTasks) > 0L) {
    SquadRunningTask* completedTask = GSetPop(&completedTasks);
    SquidletTaskRequest* task = completedTask->_request;
    printf("squad : ");
    SquidletTaskRequestPrint(task, stdout);
    if (strstr(task->_bufferResult, "\"success\":\"1\"") == NULL) {
      printf(" failed !!\n");
      flagStop = true;
    } else {
      printf(" succeeded\n");
    }
    SquadRunningTaskFree(&completedTask);
  }

} while (SquadGetNbTaskToComplete(squad) > 0L &&
  time(NULL) - startTime <= 60 && !flagStop);
// Kill the child process
for (int iSquidlet = 0; iSquidlet < nbSquidlet; ++iSquidlet) {
  if (kill(pidSquidlet[iSquidlet], SIGINT) < 0) {
    printf("Couldn't kill squidlet %d\n", pidSquidlet[iSquidlet]);
  }
}
// Wait for the child to be killed
sleep(2);
// Free memory
SquadFree(&squad);
printf("Squad ended\n");
```

```
    printf("UnitTestEvalNeuranet OK\n");
    fflush(stdout);
  }
}

void UnitTestAll() {
  UnitTestSquad();
  UnitTestLoadTasks();
  UnitTestSquadCheckSquidlets();
  UnitTestSquidlet();
  UnitTestDummy();
  UnitTestPovRay();
  UnitTestEvalNeuranet();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

9 Unit tests output

```
UnitTestSquad OK
UnitTestLoadTasks OK
Squidlet :  !!! Interrupted by Ctrl-C !!! Mon Sep  9 23:06:04 2019
UnitTestSquad OK
UnitTestLoadTasks OK
UnitTestDummy(127.0.1.1:9000)
Request for dummy task succeeded.
{"temperature":"0.00","success":"1","v":"0","nbAcceptedConnection":"1","nbAcceptedTask":"1","nbRefusedTask":"0","nbFa

delay to send: 0ms, delay to process: 38ms
UnitTestDummy(127.0.1.1:9001)
Request for dummy task succeeded.
{"temperature":"0.00","success":"1","v":"0","nbAcceptedConnection":"1","nbAcceptedTask":"1","nbRefusedTask":"0","nbFa

delay to send: 0ms, delay to process: 39ms
UnitTestSquadCheckSquidlets OK
5664 bayashi-Aspire-X1935 127.0.1.1:9000
squidlet temperature: 0.000000
UnitTestSquidlet OK
Squidlet #0 : 5671 bayashi-Aspire-X1935 127.0.1.1:9000
Squidlet :  !!! Interrupted by Ctrl-C !!! Mon Sep  9 23:06:22 2019
Squidlet #1 ended
5664 bayashi-Aspire-X1935 127.0.1.1:9000
squidlet temperature: 0.000000
UnitTestSquidlet OK
squad : Dummy(#0-0) {"v":"0"} succeeded
squad : Dummy(#1-0) {"v":"1"} succeeded
squad : Dummy(#2-0) {"v":"2"} succeeded
squad : Dummy(#3-0) {"v":"3"} succeeded
squad : Dummy(#4-0) {"v":"4"} succeeded
squad : Dummy(#5-0) {"v":"5"} succeeded
Squad ended
UnitTestDummy OK
Squidlet #0 : 5703 bayashi-Aspire-X1935 127.0.1.1:9000
```

```
Squidlet :  !!! Interrupted by Ctrl-C !!! Mon Sep  9 23:06:51 2019
Squidlet #0 ended
Squidlet #1 : 5704 bayashi-Aspire-X1935 127.0.1.1:9001
Squidlet :  !!! Interrupted by Ctrl-C !!! Mon Sep  9 23:06:51 2019
Squidlet #1 ended
squad : PovRay(#1-3) {"id":"1","subid":"3","ini":"./testPov.ini","tga":"testPov-00003.tga","top":"1","left":"251", (
squad : PovRay(#1-4) {"id":"1","subid":"4","ini":"./testPov.ini","tga":"testPov-00004.tga","top":"251","left":"251 (
squad : PovRay(#1-5) {"id":"1","subid":"5","ini":"./testPov.ini","tga":"testPov-00005.tga","top":"501","left":"251 (
squad : PovRay(#1-0) {"id":"1","subid":"0","ini":"./testPov.ini","tga":"testPov-00000.tga","top":"1","left":"1","b (
squad : PovRay(#1-1) {"id":"1","subid":"1","ini":"./testPov.ini","tga":"testPov-00001.tga","top":"251","left":"1", (
squad : PovRay(#1-2) {"id":"1","subid":"2","ini":"./testPov.ini","tga":"testPov-00002.tga","top":"501","left":"1", (
Squidlet #0 : 5812 bayashi-Aspire-X1935 127.0.1.1:9000
Squidlet :  !!! Interrupted by Ctrl-C !!! Mon Sep  9 23:07:02 2019
Squidlet #1 ended
squad : PovRay(#1-3) {"id":"1","subid":"3","ini":"./testPov.ini","tga":"testPov-00003.tga","top":"1","left":"251", (
squad : PovRay(#1-4) {"id":"1","subid":"4","ini":"./testPov.ini","tga":"testPov-00004.tga","top":"251","left":"251 (
squad : PovRay(#1-5) {"id":"1","subid":"5","ini":"./testPov.ini","tga":"testPov-00005.tga","top":"501","left":"251 (
squad : PovRay(#1-0) {"id":"1","subid":"0","ini":"./testPov.ini","tga":"testPov-00000.tga","top":"1","left":"1","b (
squad : PovRay(#1-1) {"id":"1","subid":"1","ini":"./testPov.ini","tga":"testPov-00001.tga","top":"251","left":"1", (
squad : PovRay(#1-2) {"id":"1","subid":"2","ini":"./testPov.ini","tga":"testPov-00002.tga","top":"501","left":"1", (
squad : EvalNeuranet(#0-0) {"id":"0","subid":"0","dataset":"./dataset.json","workingDir":"./","nnids":{"_dim":"2","_v
squad : EvalNeuranet(#1-0) {"id":"1","subid":"0","dataset":"./dataset.json","workingDir":"./","nnids":{"_dim":"2","_v
squad : EvalNeuranet(#2-0) {"id":"2","subid":"0","dataset":"./dataset.json","workingDir":"./","nnids":{"_dim":"2","_v
squad : EvalNeuranet(#3-0) {"id":"3","subid":"0","dataset":"./dataset.json","workingDir":"./","nnids":{"_dim":"2","_v
Squad ended
UnitTestEvalNeuranet OK
UnitTestAll OK
```