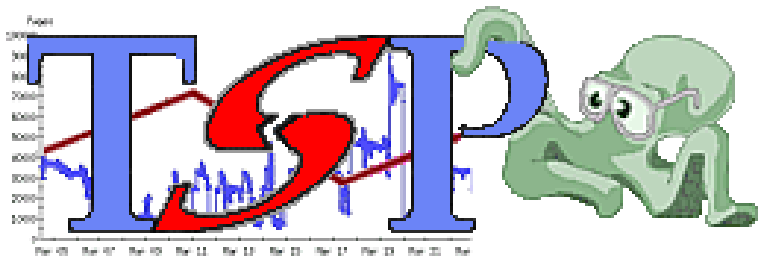
The Transport Sample Protocol: A Provider/Consumer programming Tutorial

Frederik DEWEERDT
frederik.deweerdt@gmail.com

Eric NOULARD
eric.noulard@gmail.com

Version 1.0 of October 20^{th}, 2006



Abstract

This document is a TSP programmer primer's guide. Using it you should be able to quickly understand what are the TSP objectives, how to install TSP on your system and how to use TSP within your application. In order to fully understand the document one should have a reasonable understanding of C language programming and basic knowledge of TCP/IP networked application.

Contents

List of Listings

List of Figures

List of Tables

1 What is TSP?

1.1 TSP principles

TSP stands for the Transport Sample Protocol. TSP is a sampling framework, mostly written in C and accessible in a wide variety of languages (Java, Ruby, Perl, Python) and platforms (Linux, OpenBSD, FreeBSD, Solaris, DEC OSF and Windows).

The aim of TSP is to provide an easy and straightforward way for programmers to sample data that lies within a running program. To achieve that goal, TSP provides two core components:

- The TSP provider. Plugged into the observed program, it's role is to *provide* the observed data to the outside world (TSP consumer) by embedding it in the TSP protocol.
- The TSP consumer negotiate with a TSP provider the data he wants to *consumer*. It is able to parse and understand the TSP protocol and display collected data it in some useful way.

This TSP principle is depicted on figure 1 on page 1. A TSP provider may be any application which

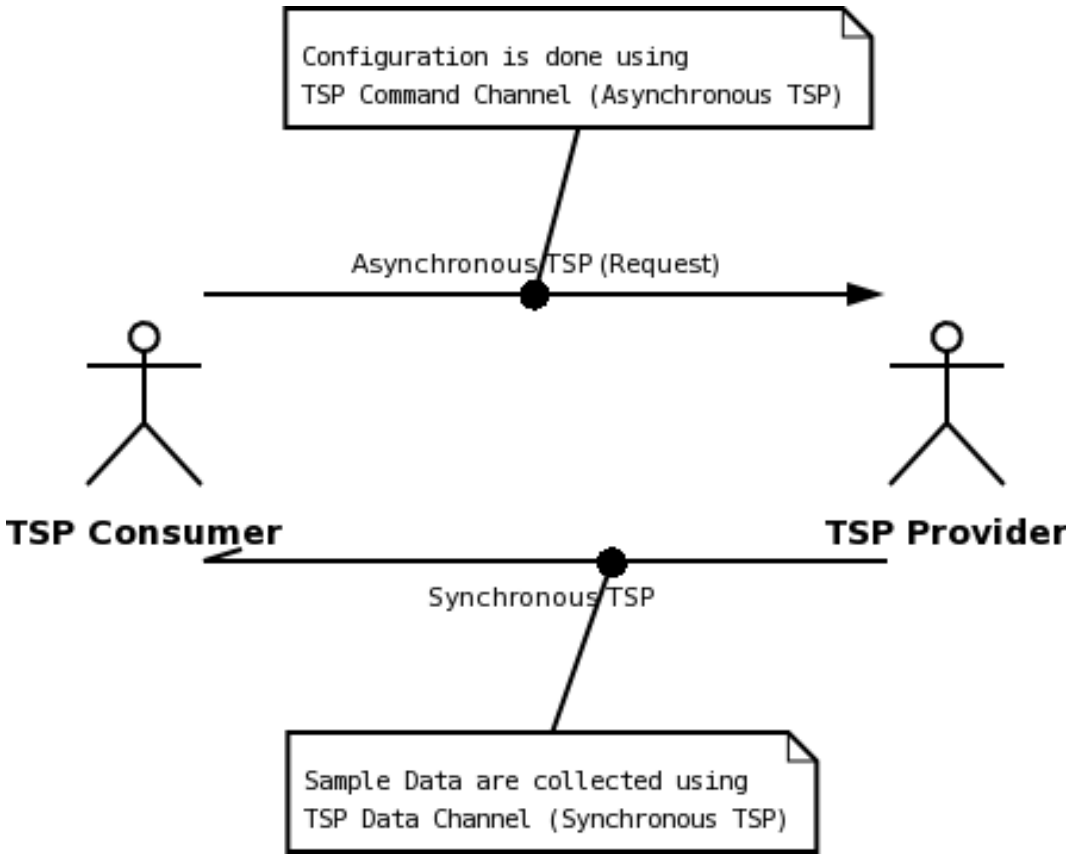


Figure 1: TSP Provider/Consumer principles

wants to expose any evolving data to the outside world in a easy, efficient and dynamic fashion. The evolving data provided by a TSP provider are called *TSP symbols*. A TSP consumer is an application which wants to get the evolving value TSP symbols in order to display or store those values. A typical TSP usage in the satellite test and integration domain is to have EGSE which are TSP providers and Graphical display which are TSP consumers.

Using the TSP software development kit you will be able to bring the simple efficiency of TSP into your application.

1.2 The TSP tools

TSP itself is both a Protocol and a Software Development Kit (SDK) including a set of ready-to-use tools such as sample file recorder, GUI graph display, Blackboard Library [Dew06] and others helpers tools and/or library. It is out of the scope of this document to describe them all, we just provide here on Table 1a synoptic list of TSP Tools which indicates their role and if those tools are available on Unix, Windows or other TSP supported platforms. In the following table the first column "**P/C/B**" indicates whether the TSP tools is on **P**rovider side, **C**onsumer side, or **B**oth sides. When only a specific unix platform (Solaris, Linux, FreeBSD, ...) is concerned it is indicated as such in the "Platform" column, otherwise Unix is given. If you want more detailed informations about TSP tools please consult [Tea06, §11 TSP Applications].

Table 1: TSP tools synoptic

P/C/B	Tool Name	Description	Platform
B	TSP Core	The TSP Core is the base TSP software module in C language. This is the mandatory module for building TSP Provider or Consumer in C. The TSP Core may be configured to use ONC-RPC or XML-RPC. XML-RPC channel is currently in alpha stage. Concerned TSP source locations: • `tsp/src/core/common` • `tsp/src/core/ctrl` • `tsp/src/core/ctrl_init` • `tsp/src/core/driver` • `tsp/src/core/include` • `tsp/src/core/rpc` • `tsp/src/core/misc_utils`	Unix Win32
P	Stubbed Server	The TSP Stubbed Server is a TSP provider which generates 1000 TSP tests symbols at 100Hz. It can viewed as faked simulator whose purpose is to be an example of TSP Provider side programming. Concerned TSP source locations: • `tsp/src/providers/stub`	Unix Win32
P	RT Stubbed	This is a variant of TSP Stubbed Server running on PC type machine under linux, which is driven by the RTC chip and use POSIX compliant realtime system interface. Concerned TSP source locations: • `tsp/src/providers/rt_stub`	Linux

▽ *table continues on next page* ▽

Table 1: TSP tools synoptic (continued)

P/C/B	Tool Name	Description	Platform
P	VX Stubbed	VXWorks specific version of TSP Stubbed Server. Concerned TSP source locations: • `tsp/src/providers/vxstub`	VxWorks
P	RES Reader	Binary RES file format (EADS-Astrium) reader. Concerned TSP source locations: • `tsp/src/providers/res_reader`	Unix
P	Gen Reader	Generic file reader. The generic file reader may read data file in different file format and provides symbols value as described by the file format handler. Concerned TSP source locations: • `tsp/src/providers/generic_reader`	Unix
P	BB Provider	Blackboard provider. Concerned TSP source locations: • `tsp/src/providers/bb_provider`	Unix VxWorks
C	Visu 3D	An experimental OpenGL consumer. Concerned TSP source locations: • `tsp/src/consumers/Visu3D`	Linux
C	Ascii Writer	A TSP consumer which may write to ascii files in different file format. Concerned TSP source locations: • `tsp/src/consumers/ascii_writer`	Unix
C	GDisp	A Graphical (GTK+1.2) TSP consumer which may display graphs and textual values. Concerned TSP source locations: • `tsp/src/consumers/gdisp`	Unix
C	Targa	A Sophisticated Graphical (GTK+1.2) TSP consumer which may display graphs and textual values. Using Targa one may build it's synoptic interactively and save/restore your sampling configuration. Concerned TSP source locations: • `tsp/src/consumers/gdisp+`	Unix

▽ *table continues on next page* ▽

Table 1: TSP tools synoptic (continued)

P/C/B	Tool Name	Description	Platform
C	Generic Consumer	The generic TSP consumer is a test consumer which offers command line options for sending any TSP Request to a TSP provider. sampling configuration. Concerned TSP source locations: • `tsp/src/consumers/generic`	Unix Win32
⊙ *end of table* ⊙			

1.3 Getting TSP

TSP is an Open Source software[1] one may get the TSP software at TSP home on Savannah [TSP]. The download section, `http://download.savannah.nongnu.org/releases/tsp/` contains source and binary release for different languages and platform.

[1] TSP license is LGPL `www.gnu.org/licenses/lgpl.html`

2 Installing TSP

TSP is an Open Source software so one may install TSP either from a pre-compiled binary installer of from the source archive using your favorite C compiler and some development tools. If you do not understand the difference between source installation and binary installation it means you certainly needs a binary installer. Binary installer comes as an executable ".exe" program on the Windows platform and as an RPM or [compressed] Tar archive on Unix platform.

2.1 Software Prerequisite

TSP needs some third party tools which needs to be installed before TSP. The main dependencies are:

1. Binary installation dependencies
   (a) a POSIX thread library
   (b) an ONC-RPC library and portmapper
2. Source installation dependencies
   (a) CMake build system [CMa]
   (b) a POSIX thread library
   (c) an ONC-RPC library and portmapper
   (d) NSIS Installer (Windows Platform Only) [NSI]

Since pre-requisite depends on the target platform (Linux, Windows, Solaris...), please read the appropriate specific installation instructions in the forthcoming section below.

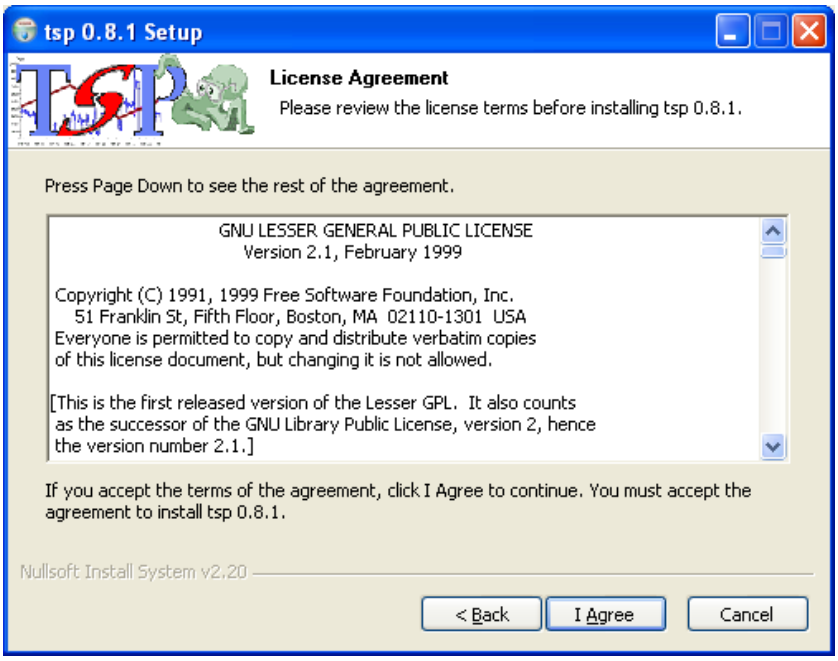
2.2 TSP binary installation

2.2.1 TSP binary installation for Windows

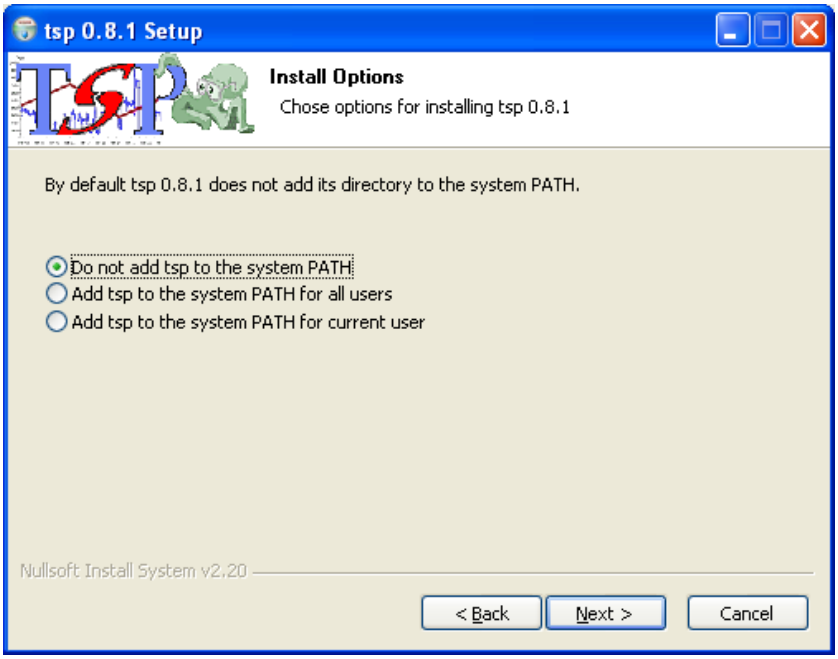
1. Get *tsp-<x.y.z>-Windows.exe* from http://download.savannah.nongnu.org/releases/tsp/

2. Execute the installer by doudble-clicking on the downloaded file. *You should have administrator privilege to perform a successful installation*

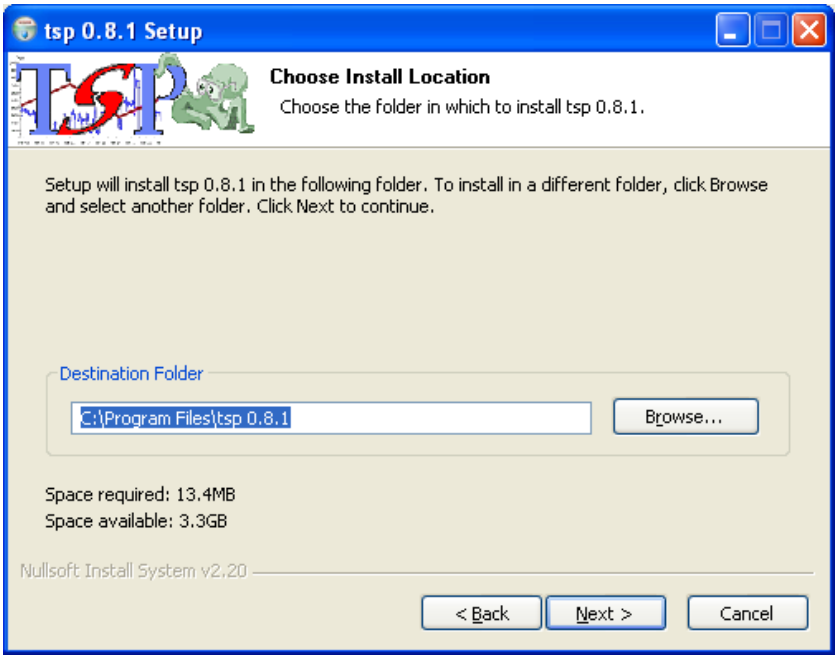
3. Accept the LGPL license policy (Open Source Software)



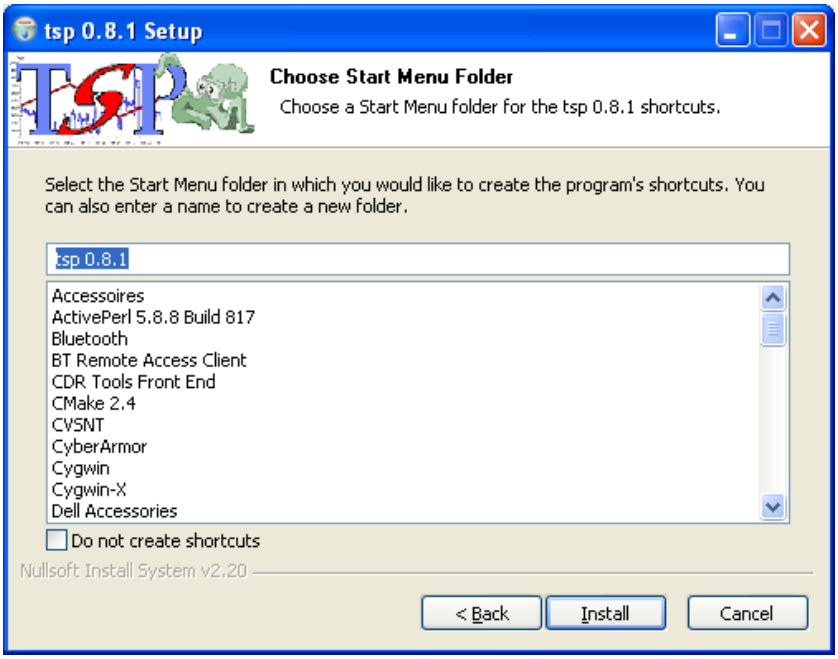
4. Choose whether you want system path to be modified for including TSP executable. If you choose "*Add tsp to the system PATH for all users*" every user of the system may launch TSP SDK excutable from any location. If unsure check "*Add tsp to the system PATH for all users*".



5. Chose TSP install location

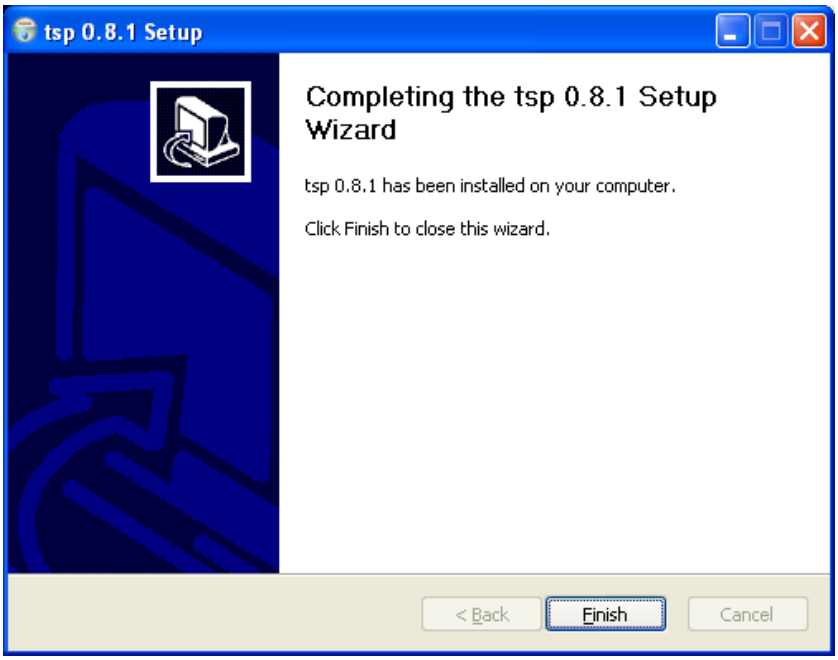


6. Chose TSP start menu folder name



7. The TSP for Windows installer will install prerequisite softwares and copy some DLL to system folder. This is not a choice but it may takes some time thus be patient...

8. TSP is now properly installed on your system



2.2.2 TSP binary installation for Unix

It is not an objective of the TSP Team to maintain and distribute binary packages for many Unix flavor (All Linux distribution, Solaris, DEC OSF...). So the favorite way of installing TSP on Unix is from source. Nevertheless, if your source installation does not go smooth you may ask for help on the TSP Developer mail list http://lists.nongnu.org/mailman/listinfo/tsp-devel.

2.3 TSP source installation

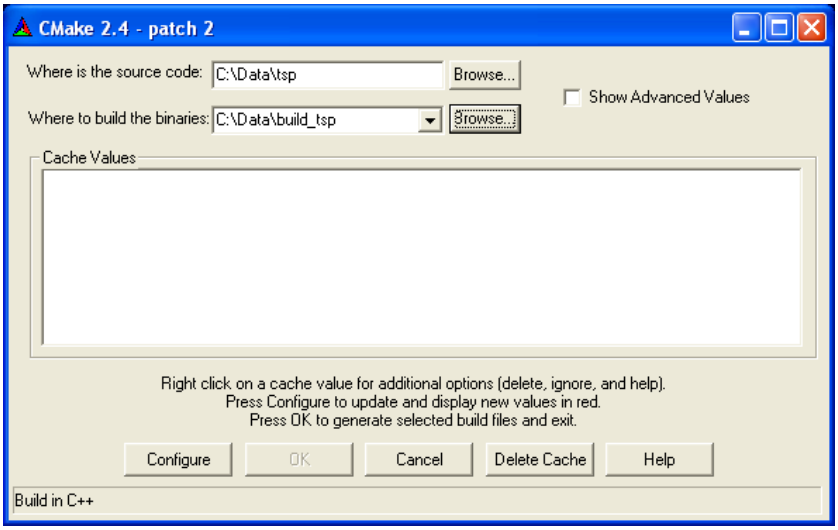
If you have made a TSP binary installation you may skip this section.

TSP can be downloaded as a source code *tar.gz* archive from http://download.savannah.nongnu.org/releases/tsp/. Note that the *tar.gz* source archive is as usable on the windows platform as it is on unix platforms[2]. The TSP source code is portable and configurable. The CMake TSP build system detect what may be compiled on the host platform and configure the source accordingly.

As TSP, since version 0.8.1, uses the CMake build system, CMake is a prerequisite for any TSP source installation. Please check that you have a properly installed CMake (see A.1) before reading on about source installation. TSP source are meant to be built using CMake out-of-source build feature. This means that the compiled binaries (object, libraries and executable) are produced in a *separate* tree from the source tree. You will see TSP with CMake source configuration example in the forthcoming sections.

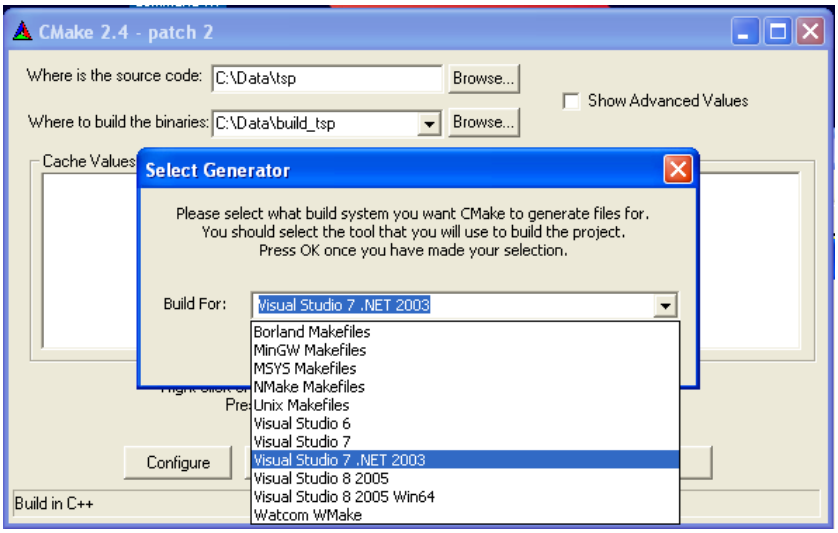
2.3.1 TSP source installation for Windows

1. Check you have the minimal prerequisite softwares installed:
   (a) CMake see A.1.
2. Get *tsp-<x.y.z>-Source.tar.gz* from http://download.savannah.nongnu.org/releases/tsp/
3. Unpack the archive at your favorite place. In the following screenshot the source location is `C:\Data\tsp` .
4. Run CMake and chose a build directory which is separate from source (this is called out of source build).

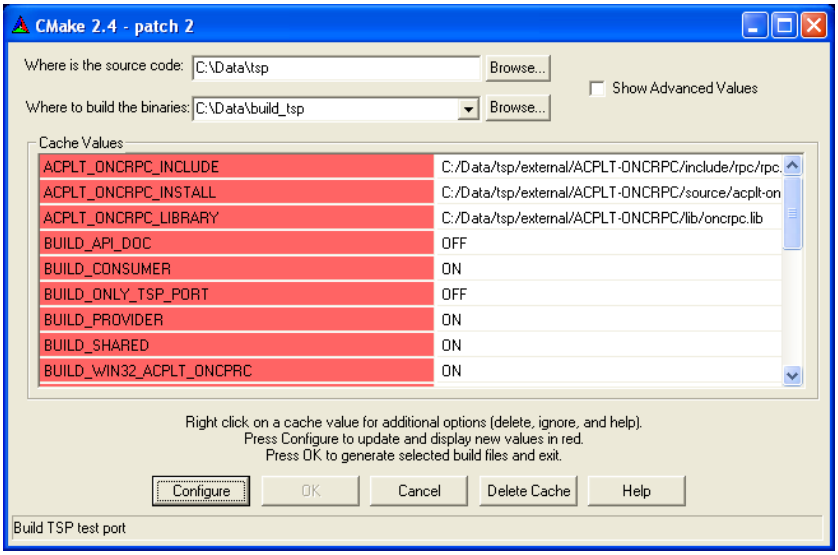


5. Click on *Configure*, CMake will ask you for which build tools he should generate files, for example "*Visual Studio 7 .Net 2003*":

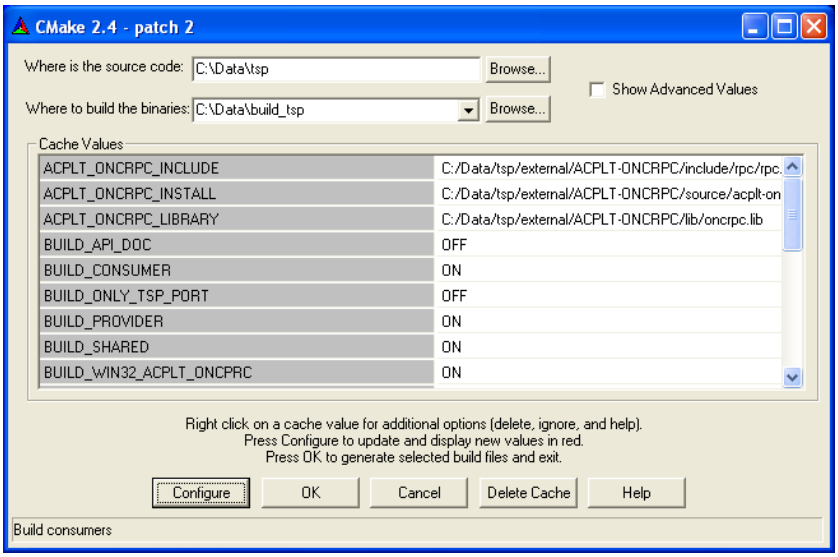
[2]Many Windows Zip softwares are able to extract *tar.gz* archives, see for example http://www.7-zip.org/



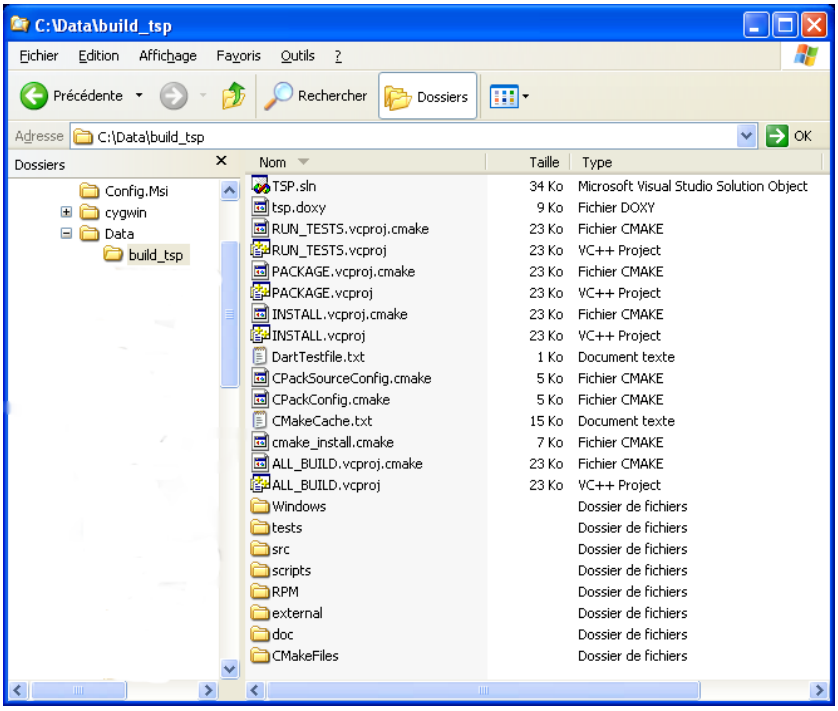
6. After the Generator is selected on you Clicked OK, CMake will do its first discover task and you should obtain something similar to:



7. Then click on *Configure* again in order to make CMake do its configuration task, and you get:



8. Finally click on *OK* in order to make CMake generate the project files. If you open the build directory you will see that you now have a "Microsoft Visual Studio 7 .Net" solution file `TSP.sln` which is ready to use.



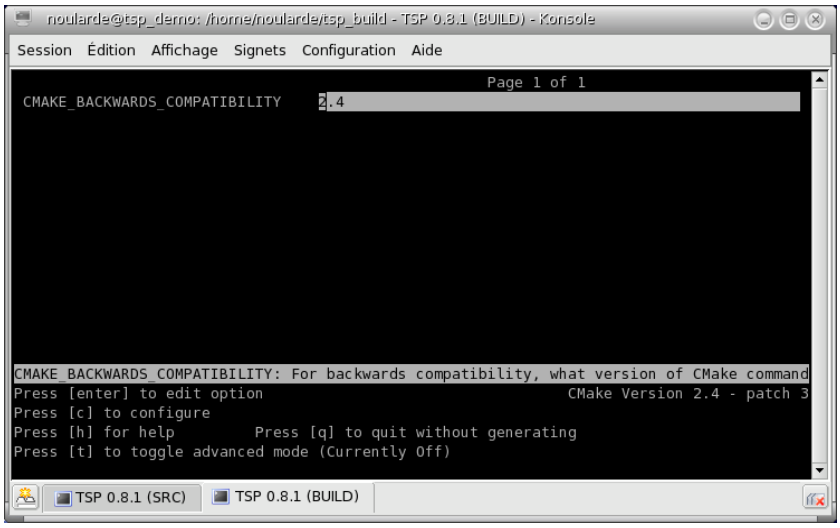
2.3.2 TSP source installation for Unix

The installation under Unix is straightforward, just deviating slightly from the standard `./configure; make ; make install` routine since we want an out of source build:

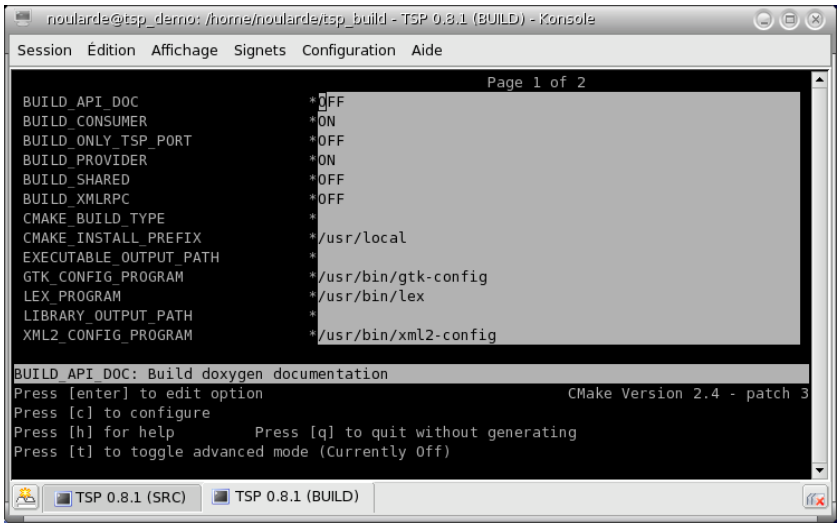
1. Check you have the CMake installed (see A.1).
2. Get *tsp-<x.y.z>-Source.tar.gz* from `http://download.savannah.nongnu.org/releases/tsp/`
3. Unpack the archive at your favorite place:

   `cd $HOME; tar zxvf tsp-<x.y.z>-Source.tar.gz` . The command should create a directory `$HOME\tsp-x.y.z-Sources` containing the whole TSP C SDK sources.
4. Create your build directory `cd $HOME; mkdir tsp_build` and change directory `cd tsp_build` .
5. Run CMake from within the build directory,
   - you may run the default configuration using the non interactive `cmake` command `cmake $HOME\tsp-x.y.z-Sources`
   - or you may run the interactive curse CMake interface `ccmake` which looks like the Windows interface in a full text version.

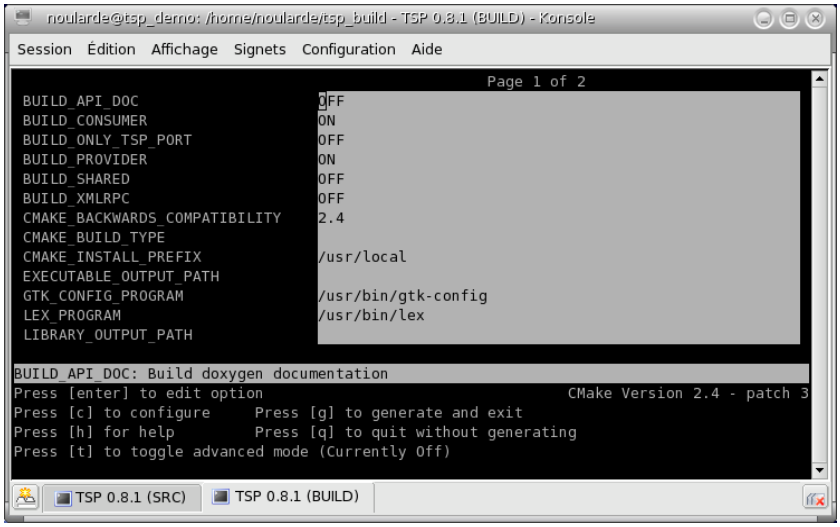
     (a) You run `ccmake $HOME\tsp-x.y.z-Sources` and get:



(b) Then you hit 'c' key for "configure" and make CMake do its first discover task and you get



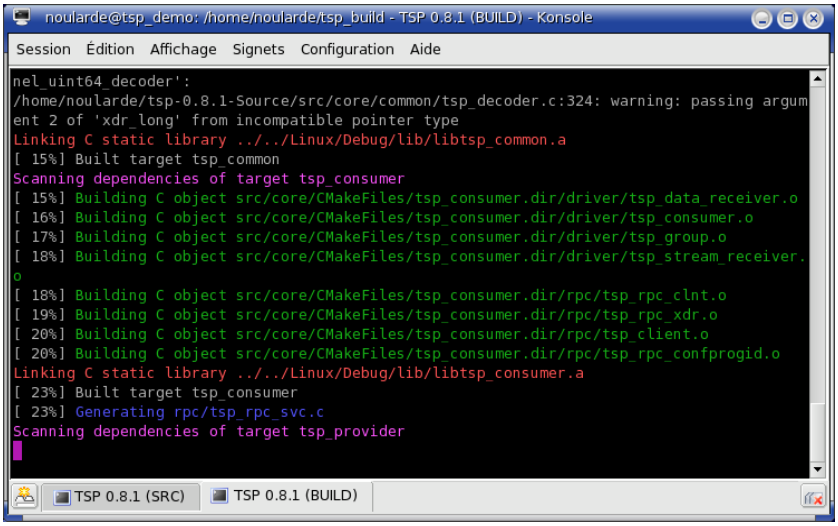
(c) Afterwards you hit 'c' key again for making CMake do its configuration work:



(d) Last step is to hit 'g' for making CMake generate the Makefiles. CCMake exits properly and you may proceed as if you had launch `cmake` (and not `ccmake`).

6. Launch the build command and wait for termination:

the command `make` will build TSP.

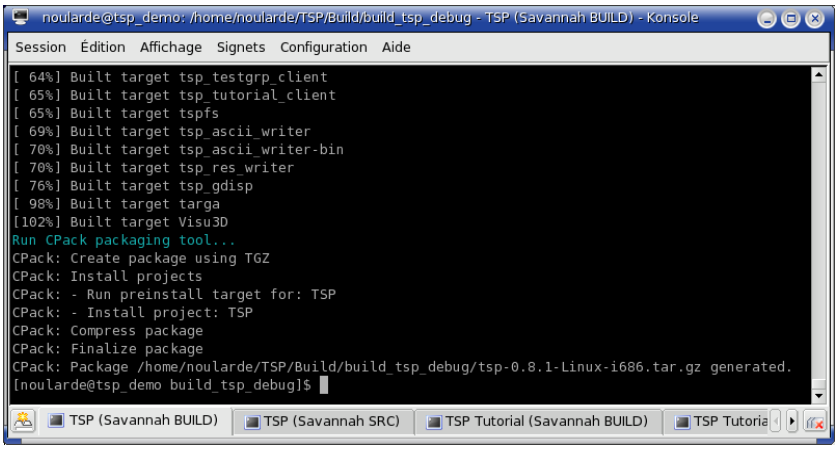


Note that `make` may eventually re-run the non-interactive `cmake` automatically. You should not worry about this.

7. Packaging TSP:

   Before installing TSP it's better to build a binary TSP package you will be able to deploy on every machine you need, just as you can do with the TSP for Windows installer.

   The command `make package` will build the binary TSP package.



   Will build a unix tar and gzip-compressed archive whose name depends on your machine architecture, in our example this leads us to: `tsp-0.8.1-Linux-i686.tar.gz` .

8. Install TSP

   Take the `tsp-<version>-<system>-<arch>.tar.gz` binary TSP package you have built in the previous step and *untar* the archive at the install place you want with the following command:

   ```
   tar zxvf tsp-0.8.1-Linux-i686.tar.gz
   ```

   This will produce a `tsp-0.8.1-Linux-i686` directory containing the TSP install directory tree:

```
tsp-0.8.1-Linux-i686
        |-- bin
        |-- include
        |-- lib
        `-- scripts
```

- `bin` contains binary executables,
- `include` contains public include files,
- `lib` contains libraries,
- `scripts` contains helper scripts and test files.

After that you may want to update your PATH to include `<path_to_tsp_install>/bin` .

2.3.3 TSP Source tree primer

A quick look at the TSP sources may be helpful in understanding and locating the TSP components:

```
<tspdir> $ tree -L 2
[...]
|-- src
|   |-- consumers
|   |-- core
|   |-- providers
[...]
```

The `core` directory contains the code implementing the core TSP functionalities: both the consumer and provider API are implemented there.

The `consumers` directory contains readily available TSP consumers coded by the TSP team. They target a wide range of uses, and are well beyond the scope of this document, it is recommended to refer to the *TSP Design & Programming Guide Document* [Tea06].

The `providers` directory contains TSP providers that might prove useful as reference for the future provider writer. In particular, the `src/providers/stub` directory contains the Stub Server provider.

3 Testing TSP installation

To make sure that we now have a working TSP installation on our system, we will proceed two small tests:

- the first test will check a TSP installation on a single host which may or may not be connected to a LAN,
- the second test will check a TSP installation on 2 hosts interconnected with a TCP/IP LAN.

3.1 Standalone TSP test (1 host)

This test simply consists in launching two TSP applications:

- one provider, the stub server and,

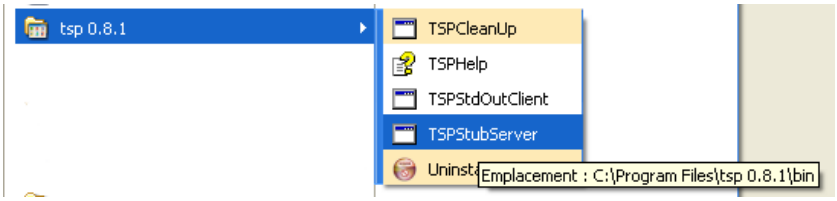
  The Stub Server is a test and tutorial TSP provider that generates TSP Symbols value at approximately 100Hz. It is used to test TSP installation and may be used to test new TSP consumers.

- one consumer the stdout consumer,

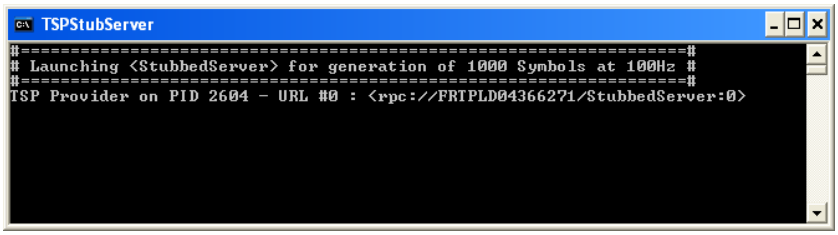
  The Stdout Client consulmer is a test and tutorial TSP consumer which may connect to any TSP provider and request a specified number of TSP symbols and print their ebolving values on standard output.

The screenshots shown hereafter are taken on a Windows system but you may run the same test on any TSP supported unix systems too. The test TSP applications used here may be launched from the TSP start menu on windows but we will give you the corresponding command line command and arguments usable both on Windows and on Unix platforms.

1. Launch the Stub Server, either with command line `tsp_stub_server` (Unix) or `tsp_stub_server.exe` (Windows) or even from the TSP start menu group:

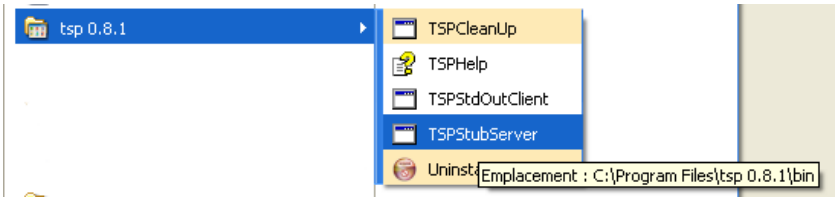
   

   Now the StubServer is running and wainting for a TSP consumer to connect. The StubServer console window should display something like that:

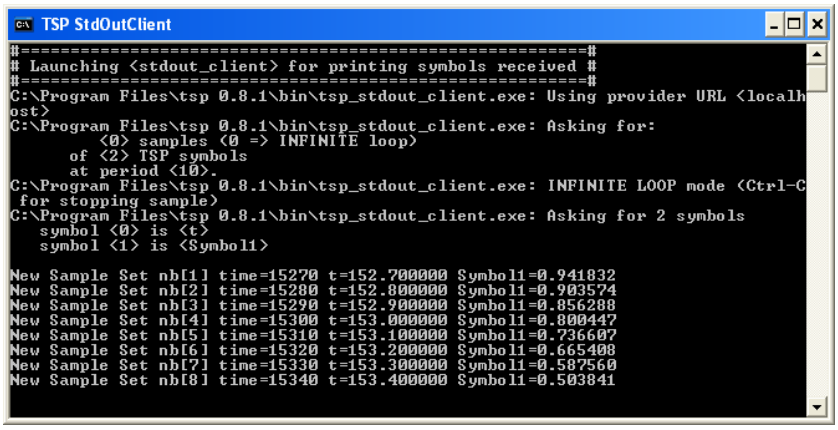
   

   If ever the StubServer is not able to start please check if you RPC Portmapper is running properly as described at §A.2.1.

2. Launch the Stdout Client, either with command line `tsp_stdout_client -p 10 -s 2 -n 0` (Unix) or `tsp_stdout_client.exe -p 10 -s 2 -n 0` (Windows) or even from the TSP start menu group:



Now the StdOut Client console window should display just like that:



The Stdout display TSP symbols values provided by the Stubbed Server. This is an infinite loop you may terminate by hitting Ctrl-C within the Stdout Client console window.

Now you should close both windows.

The console should not close, otherwise it means that something in the initialization went wrong. In case it does close, consider running the 'cmd.exe' program, change dir to the directory where the provider's binary lies, and launch it by typing `tsp_stub_server.exe` , you should be able to read an informative message.

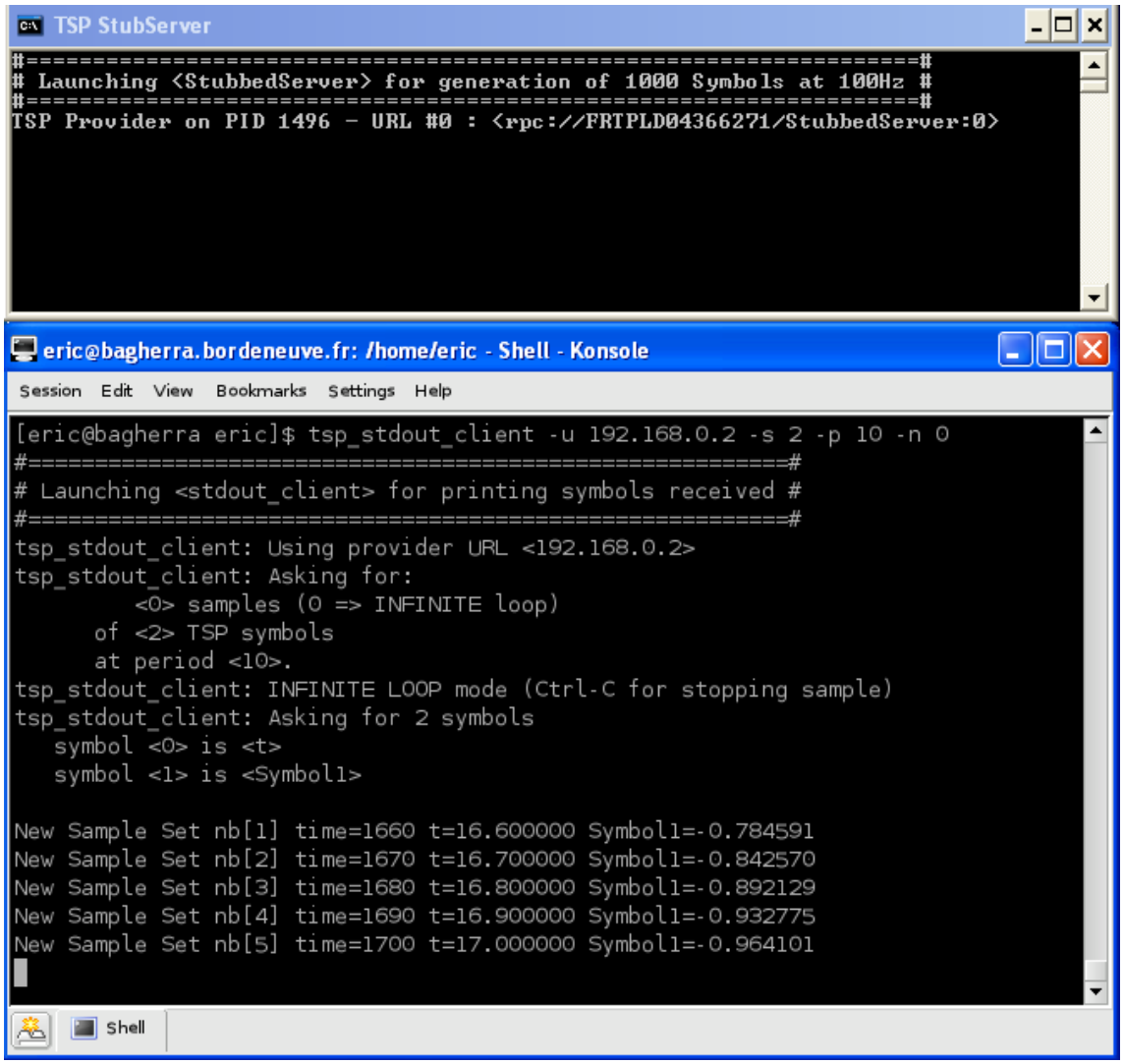
3.2 Networked TSP test (at least 2 machines)

TSP is meant to be used between several hosts exchanging data using the TSP protocol. When the standalone TSP is OK you may run the same test using 2 machines. You have to run the Stubbed Server just as before and to run the Stdout Client from a command line and providing the necessary network argument.

On our example the StubbedServer is run on a Windows box whose IP address is 192.168.0.2, so that the Stdout Client running on a Linux Box connected to the Windows box's network should be run with the following command line:

```
tsp_stdout_client.exe -u 192.168.0.1 -p 10 -s 2 -n 0
```

The corresponding screens shots are shown just below:



If the Stdout client cannot connect to the Stubbed Server:

1. Check TSP installation on each box by running the standalone test
2. Check the network connectivity between the provider box and the consumer box by trying network connectivity test like `ping` each other.
3. Check whether the provider box does not have some firewall software activated.

   If one of your host is a Linux box you may play with several graphical TSP consumers with your StubbedServer running. Note that some TSP consumers may not have been compiled on your systems if some development libraries (libxml2, gtk+1.2 etc...) have not been detected by CMake. In the `scripts` directory of your TSP installation you have some handy TSP consumer configuration files.

   - TSP GDisp:

     `tsp_gdisp -u 192.168.0.1 -x <TSP_install_dir>/scripts/stub_gdisp_config.xml`
     This should lead to something like:
   - TSP Ascii Writer:

     `tsp_ascii_writer -u 192.168.0.1 -x <TSP_install_dir>/scripts/stub_ascii_writer_co`

Check [Tea06, §11.2 TSP Consumers] if you want more instructions.

4 Building a TSP Provider

This section describes the necessary steps needed to add a provider to an existing program. For simplicity's sake, this program will consist on a simple loop incrementing two variables. Our job will be to make those variables available to a basic TSP consumer.

We will start by studying the original code, then we will had the necessary TSP hooks to make the code TSP aware. We will then use `targa`, which is a handy GTK+ TSP consumer[3], to display the data. To conclude, we will write a simple consumer to display the value of our variable to a console's screen.

4.1 The observed application

Listing 1 shows a sample simulator in action. It simply consists in a `simulation()` function (line 8), that runs as a thread. This function iterates 20000 times, incrementing the `test_variable1` (line 20) and decrementing the `test_variable2` (line 21), both of which, we will suppose, are of crucial importance for our project.

Listing 1: A simplified simulator

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  /*
6   * The pseudo simulation function
7   */
8  void *simulation(void *unused)
9  {
10         unsigned long test_variable1;
11         unsigned long test_variable2;
12
13         test_variable1 = 0;
14         test_variable2 = ~0UL;
15         /*
16          * The pseudo simulator main loop.
17          */
18         while (test_variable1 < 20000) {
19                 /* Update internal state of our simulator */
20                 test_variable1++;
21                 test_variable2--;
22                 /* wait for next simulator cycle */
23                 usleep(100000);
24         }
25         return NULL;
26 }
27
28 /*
29  * This the main entry point of our
30  * pseudo simulator
31  */
32 int main(int argc, char *argv[])
33 {
34         pthread_t simu_thread;
35         int ret;
36
37         printf("#\n");
38         printf("# Launching <Observed App>\n");
39         printf("#\n");
40
41         /* Create a thread which launches the simulation function */
42         ret = pthread_create(&simu_thread, NULL, simulation, NULL);
43
44         /* pthread_create retcode must not be NULL */
```

[3] You'll find the sources at `src/consumers/gdisp+`

```
45          assert(!ret);
46
47          /* Wait for the simulation termination by joining the
48           * simulation thread
49           */
50          pthread_join(simu_thread, NULL);
51
52          printf("#=== End ===#\n");
53          return 0;
54  }
```

4.2 Providerizing the program

Now that we identified the data to be *provided* by our program, we will proceed the necessary steps to make the simulator TSP-aware.

Let's recall from [Tea06] that being a TSP provider means being able to answer to TSP requests, those TSP requests are used between a TSP provider and a TSP consumer to negotiate the samples they will exchange. A typical TSP request sequence is shown on figure 2.

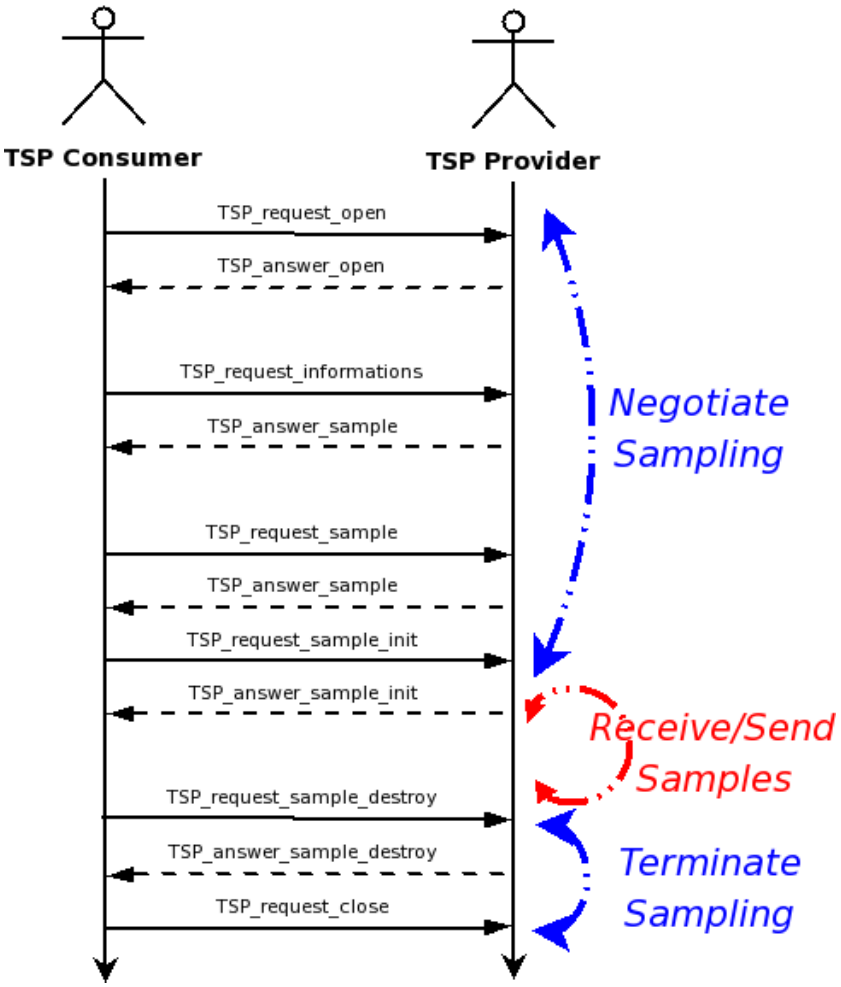


Figure 2: TSP typical sequence

The typical TSP sequence is simple:

1. Negotiate sampling configuration with the provider,

2. Start sampling and loop to receive samples,
3. Ask for sampling termination.

This may seems hard to implements but the TSP Library makes it really simple to do. The TSP Library will take care of handling the request/answer mecanism for us as soon as the application:

- Implements and register a object-oriented C callback object called the GLU.
- Calls some TSP API for initialization and termination,
- Tells the TSP library for sample update.

Figure 3 illustrate the layered aspect of TSP GLU interface.

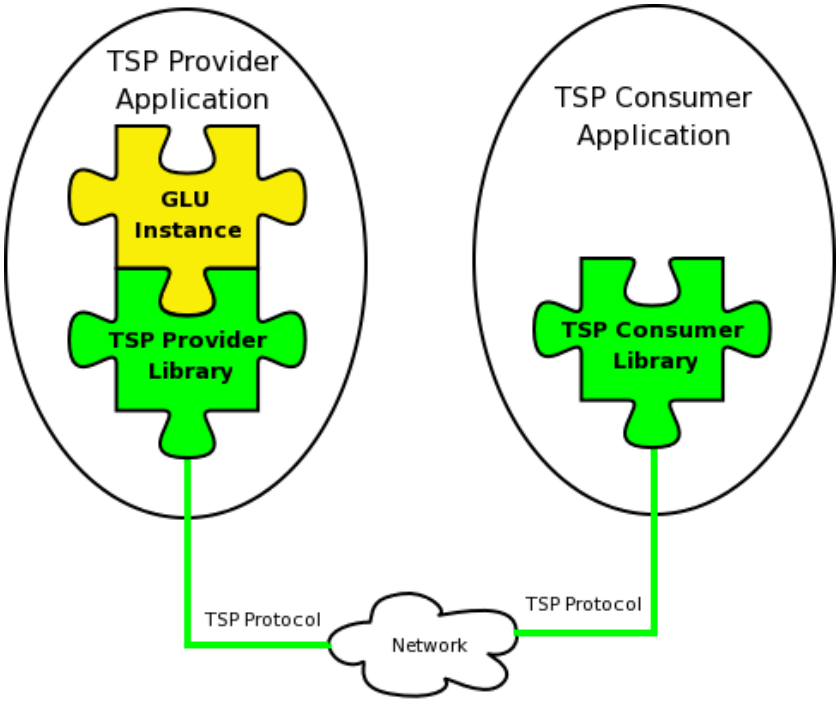


Figure 3: GLU vs TSP library

The TSP GLU object is a C structure which contains data and pointer to functions. We may not detail the whole structure content here but the main idea is that whenever the TSP library needs informations for filling-up TSP Answer to Consumer TSP request (available sample symbol list, name and type description of the symbols etc...), the TSP Library will call the GLU structured callback object our application has provided. Let's go for some source code now.

As expected, we will first include the needed TSP headers at the top of the source file. This is shown in listing 2.

Listing 2: Headers of the TSP aware simulator

```
1 /* .... */
2 #include <tsp_abs_types.h>       /* platform independant data types definition */
3 #include <tsp_provider_init.h>   /* provider init API */
4 #include <tsp_glu.h>             /* TSP GLU object definition and API */
5 #include <tsp_common.h>          /* TSP common structure manipulation API */
6 #include <tsp_datapool.h>        /* TSP provider datapool API */
7 /* .... */
```

Those headers contains the needed prototypes of the functions used in listing 3 in order to interface our application with the TSP Library. The listing 3 illustrates how to create a GLU object in our main application and register it to TSP library during its initialization. This example is taken from the Stubbed Server provider you may found in `tsp/src/provider/stub` the example has been slightly modified to makes it more simple and readable at first glance. You should read all the code and comments keeping in mind the 3 steps for programming a TSP provider:

1. Build your own GLU object structured callback,
2. Register the GLU object into TSP Provider library and initialize TSP,
3. Launch TSP provider request handler AKA `TSP_provider_run` .

Listing 3: TSP core initialization

```
1  /* declare my GLU object static variable */
2  static GLU_handle_t* stub_GLU = NULL;
3
4  /* Create the GLU object instance */
5  GLU_handle_t *STUB_GLU_create()
6  {
7          /*
8           * Create a default GLU object instance
9           */
10         GLU_handle_create(&stub_GLU,              /* pointer to pointer to GLU object     */
11                           "SampleTSPProvider",    /* Provider name                        */
12                           GLU_SERVER_TYPE_ACTIVE, /* my GLU can't wait it is ACTIVE       */
13                           100.0);                 /* my advertised base frequency (in Hz) */
14
15         /*
16          * Now we must provide GLU member functions
17          * which will be called by TSP provider library in order
18          * to build TSP answers to TSP consumer requests
19          */
20
21         stub_GLU->initialize = &STUB_GLU_init;     /* initialize GLU member function pointer */
22         stub_GLU->run        = &simulation;        /* main loop GLU member function pointer  */
23
24         /* provides get Sample Symbol Info List GLU member functions */
25         stub_GLU->get_ssi_list           = &STUB_GLU_get_ssi_list;
26
27         /* provides get Sample Symbol Extended Information from PGI member function pointer
28          * PGI = Provider Global Index
29          */
30         stub_GLU->get_ssei_list_fromPGI = &STUB_GLU_get_ssei_list_fromPGI;
31
32         return stub_GLU;
33 }
34
35 int main(int argc, char *argv[])
36 {
37         int ret;
38
39         printf("# Launching <Sample server>\n");
40
41         /* Create our structured GLU callbacks */
42         GLU_handle_t *GLU_stub = GLU_stub_create();
43
44         /* Initialize TSP Provider library and register OUR GLU object
45          * so that the TSP core knows it and is able
46          * to call appropriate callback GLU member functions.
47          */
48         if (TSP_STATUS_OK == TSP_provider_init(GLU_stub, &argc, &argv)) {
49
50                 /* configure TSP request handling SIMPLE mode */
51                 unsigned int flags = TSP_ASYNC_REQUEST_SIMPLE;
52
```

```
53                  /* TSP Request Handler will loop forever when started */
54                  flags |= TSP_ASYNC_REQUEST_BLOCKING;
55
56                  /*
57                   * Start TSP request handling loop
58                   * In this case the function will not return
59                   * until the program is interrupted (Ctrl-C).
60                   *
61                   * Provider run will:
62                   *    1- Call GLU->initialize()
63                   *    2- Start a thread running GLU->run()
64                   *    3- Start TSP request handler
65                   */
66                  if (TSP_STATUS_OK != TSP_provider_run(flags)) {
67                          return -1;
68                  }
69
70                  /* Terminate TSP Provider library */
71                  TSP_provider_end();
72
73                  /* * * NO TSP_xxx functions may be called after this call * * */
74          }
75          return 0;
76  }
```

We will review the functions and their role one by one hereafter, nevertheless keep in mind that the most *up to date* information is in the concerned headers sources files themselves. The TSP headers are documented using doxygen[4] structured comments, so that complete and browsable API documentation may be generated either in HTML format or CHM (Windows Help) format as illustrated at figure 4 on page 24. The TSP Windows help file is available through the TSP menu group. The root HTML index document may be found in `<TSP_INSTALL_DIR>/doc/api/html/index.html` and may be opened by any HTML Browser[5].

Now let's go further inside TSP provider API role and features:

- `TSP_provider_init(handle_t* theGLU, int* argc, char** argv[])`

  Initialize the TSP provider library and register `theGLU` structured callback. `argc` and `argv` are the classical arguments of a main program. If you don't have them you should fake them like this:

```
int argc = 1;
char** argv = 0;

argv = (char**)calloc(argc+1, sizeof(char*));

argv[0] = strdup("MyOwnProvider");
argv[1] = NULL;
```

- `TSP_provider_run(int spawn_mode)`

  Start TSP provider library. This will call the `GLU->initialize()` function and then launch the TSP request handler (ONC RPC request handler in the default case). The `spawn_mode` is a mask of OR-ed values:

[4] http://www.doxygen.org

[5] You may find an online version of TSP API documentation at http://www.ts2p.org/tsp/API_doc/html/index.html

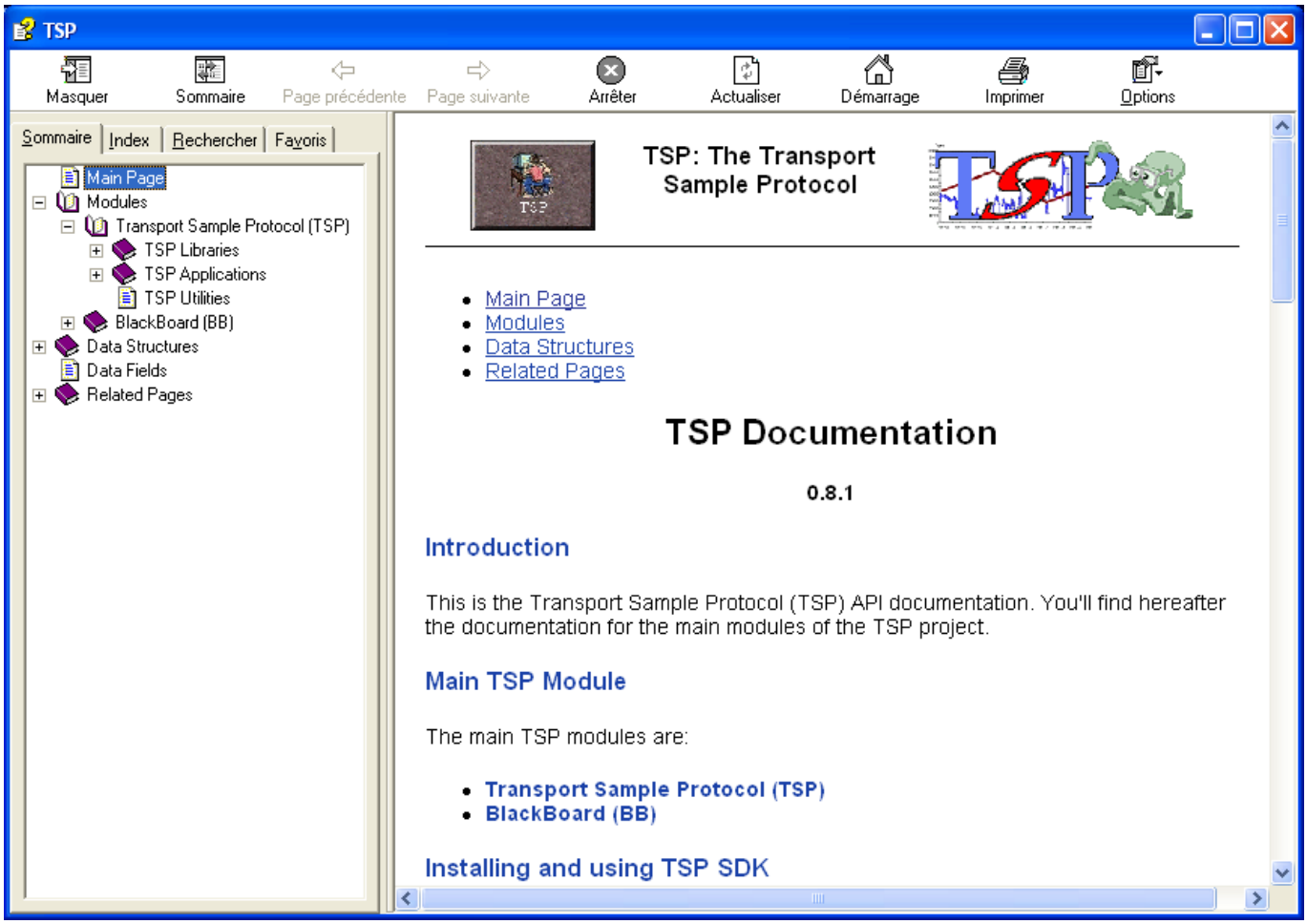


Figure 4: Windows Help TSP API documentation

- the asynchronous request mode. This will tell TSP if several and dynamically registered request handler should be used or not. For now only `TSP_ASYNC_REQUEST_SIMPLE` is supported since `TSP_ASYNC_REQUEST_DYNAMIC` is not implemented yet.
- the blocking mode. This indicates if the call to `TSP_provider_run` should block or not. When `TSP_ASYNC_REQUEST_NON_BLOCKING` mode is invoked a new thread is started and function returns, whereas when `TSP_ASYNC_REQUEST_BLOCKING` mode is requested function never return unless program receive a signal.

- `TSP_provider_end()`

  Finalize the TSP provider library, i.e. shut down TSP. No TSP calls may done after this call, not even to `TSP_provider_init` again.

Now we should have a look at the differents GLU mandatory member functions we have to implement:

- `GLU->initialize` shown in listing 4 must define what *TSP symbols* the providers will offer. The GLU should define a list of symbols to provide. This list is a `TSP_sample_symbol_info_list_t` structure containing `TSP_sample_symbol_info_t` elements. Those structures may be manipulated with `TSP_SSIList_xxx` and `TSP_SSI_xxx` API defined in `<tsp_common.h>` . The minimal information that should be provided for a TSP symbols is:

- its name, which is a human readable label associated to the data (its key),
- its provider global index, which a unique integer identifier used by TSP core to index the provided symbol,
- its TSP type (DOUBLE, FLOAT, INT8/16/32..., browse TSP API documentation for the complete `enum TSP_datatype_t` definition)
- its dimension, 1=scalar, >1 array of symbol. TSP only support rank 1 array.
- its period, which is how often the provider update the symbol value if unsure put 1 (see [Tea06] for detail on this).

Listing 4: GLU initialization function

```
[...]
static TSP_sample_symbol_info_list_t X_SSI_list;
[...]
int STUB_GLU_init(GLU_handle_t * this, int fallback_argc,
                  char *fallback_argv[])
{
        int32_t size;

        /* Initialize provided sample info list for 1 symbol */
        if (TSP_STATUS_OK!=TSP_SSIList_initialize(&X_SSI_list,1)) {
           return FALSE;
        }
        /*
         * Initialize a TSP_sample_symbol_info_t structure
         * which will hold metadata for 'test_variable'
         */
        TSP_SSI_initialize(TSP_SSIList_getSSI(X_SSI_list,0),
                           "test_variable",   /* name */
                           0,                 /* provider global index */
                           0,0,               /* pgridx,pgrank */
                           TSP_TYPE_DOUBLE,   /* type   */
                           1,                 /* dimension */
                           0,0,               /* offset, nelem */
                           1,                 /* period */
                           0);                /* phase */
        /* compute symbol memory size */
        size =
            X_sample_symbol_info.dimension * tsp_type_size[X_sample_symbol_info.type];

        /* This is not really needed here for now,
         * but let's register the biggest size we'll deal with
         */
        if (taille_max_symbol < size) {
                taille_max_symbol = size;
        }
        return TRUE;
}
```

- `GLU->get_ssi_list` shown in listing 5 should return the complete list of provided symbols. We simply provides the value of our `static` variable `X_SSI_list` previously initialized by `GLU->initialize()` .

Listing 5: GLU get Sample Symbol Information

```
int
STUB_GLU_get_ssi_list(GLU_handle_t* h_glu,TSP_sample_symbol_info_list_t* symbol_list)
{
        symbol_list->TSP_sample_symbol_info_list_t_len =
           X_SSI_list.TSP_sample_symbol_info_list_t_len;
        symbol_list->TSP_sample_symbol_info_list_t_val =
           X_SSI_list.TSP_sample_symbol_info_list_t_val;

        return TRUE;
}
```

- `GLU->run` which is our application updated `simulation` and shown in listing 6. `GLU->run` must be the main TSP provider update loop. It should feed the TSP datapool with sample values at the chosen provider pace.

Listing 6: GLU run

```
void *simulation(void *athis)
{
        /* my test variable */
        double test_variable;
        /* a GLU datapool item */
        glu_item_t *item;
        /* the athis is the pointer on the GLU object itself */
        GLU_handle_t* cthis  = (GLU_handle_t*) athis;
        int symbols_nb, *ptr_index;
        int temp;

        item = calloc(1, sizeof(*item));
        assert(item);
        /* Reserve enough memory for one symbol */
        item->raw_value=calloc(1,taille_max_symbol);
        assert(item->raw_value);

        test_variable = 0.0;
        while (1) {
                /*
                 * Reverse list of wanted items index
                 * The TSP Provider library maintains the list of
                 * of all symbols that are requested by connected TSP consumers.
                 * This is handy way for a provider to only update a reduced set
                 * of provided symbol.
                 * Using this scheme a provider may potentially offer
                 * a huge number of symbols while only effectively providing a few.
                 */
                TSP_datapool_get_reverse_list(cthis->datapool,&symbols_nb, &ptr_index);
                item->size = X_sample_symbol_info.dimension * tsp_type_size[X_sample_symbol_info.type];

                /* Export to the consumers at which _internal_ time, the data was sampled */
                item->time = my_time;

                /* Assign the new value to our variable */
                *((double*)item->raw_value) = test_variable;

                /*
                 * Enqueue the variable value, so that
                 * next commit will take into account the new value set
                 * separating PUSH from COMMIT (see later)
                 * ensure that TSP will provide a coherent set of sample
                 */
                TSP_datapool_push_next_item(cthis->datapool, item);

                /* Perform complex computations on our test variable */
                test_variable++;

                /*
                 * Commit ALL the variable's new values
                 * to the data pool
                 * so that TSP Library may send the whole set to consumer.
                 */
                TSP_datapool_push_commit(cthis->datapool,my_time, GLU_GET_NEW_ITEM);

                /* Increase the simulation's internal time reference */
                my_time++;
                tsp_usleep(TSP_USLEEP_PERIOD_US);
        }
        return NULL;
}
```

- `void TSP_datapool_get_reverse_list (TSP_datapool_t* datapool, int *nb, int **list)`

this function gets the symbol list of a given data pool. The data pool in question lies within the TSP provider we are connected to. As expected, the list's size and the list itself are returned in the `nb` and `list` arguments of the function.

- `int TSP_datapool_push_next_item (TSP_datapool_t* datapool, glu_item_t* item)`

  This function enqueues a glu_item, which is a TSP symbol value. Those value are kept in the provider datapool until `TSP_datapool_push_commit` is called.

- `int TSP_datapool_push_commit(TSP_datapool_t* datapool, time_stamp_t time_stamp, GLU_get_state_t state)`

  The commit that we've just refered to above. This informs the underlying TSP core that new data is ready to be sent to consumer side. The TSP core will then handle the delivery of the actual data to the consumer.

Let's summarize what we have done:
After writing our 3 GLU minimal member functions:

- `GLU->initialize` , see listing 4
- `GLU->run` , see listing 6
- `GLU->get_ssi_list` , see listing 5

and the main program initializing TSP provider lib (see listing 3) we have a functionnally running TSP provider offering a single TSP Symbol. TSP provider side programming has many more possibilities you may discover by reading more provider source code in `tsp/src/providers/stub` and others `tsp/src/providers/xxx` .

Keep in mind that documentation is *never* as accurate as source code itself. That's why TSP API documentation is extracted from directly from source code using Doxygen in order to make it available as soon as code is updated.

Moreover TSP is an Open Source project so you should not hesitate to ask questions on the TSP Development mailing list at http://lists.nongnu.org/mailman/listinfo/tsp-devel.

5 Building a TSP consumer

Now that we have handled the provider side aspect of TSP programming let's continue on the TSP Consumer side.

A TSP consumer is an application that wants to get TSP sample symbols informations and evolving values of a subset of the provided symbols. As already shown on figure 1 on page 1 and then more precisely on figure 2 on page 20 a TSP consumer *negotiate* with one or several TSP provider(s) the sample symbols value he wants to receive.

The typical TSP consumer/TSP provider negotiation sequence shown on figure 2 is recalled here:

1. Open a TSP Session (mandatory).

   Send the `TSP_request_open` the consumer will get a TSP session Id to be used in other TSP request calls.

2. Get Sample Information (optional).

   Using the previously obtained TSP session Id you may ask the provider for information regarding the symbols he may provide you.

   Send `TSP_request_informations` and/or `TSP_request_filtered_informations` . Using those requests the TSP consumer may get a (filtered) list of available TSP Symbols.

3. Request for Sample (mandatory).

   The TSP Consumer selects the list of TSP Sample Symbols he wants to get using their name, sampling period and phase. The consumer sends one or several `TSP_request_sample` until he gets an *OK* from the TSP provider. The provider may refuse the sample request for different reason:

   - one or several requested symbols are unknown,
   - specified period may not be satisfied,
   - number of active (i.e. sampling) TSP Session is exhausted,
   - provider specific reason.

4. Request for Sample Initialization (mandatory).

   When the last `TSP_request_sample` sent by the consumer is accepted by the provider, the consumer may send `TSP_request_sample_init` which tells the provider to allocate a socket for the consumer sampling session and be ready to send as soon as the consumer is connected. The `TSP_answer_sample_init` tells the consumer how to connect (IP address and socket port).

5. Read Sample (mandatory loop)

   As soon as the consumer is connected he only have to loop on calling `TSP_consumer_read_sample` for getting sample values.

6. Request for Sample Destroy (mandatory).

   Tells the provider to stop sending sample and to close the socket.

7. Request Close (mandatory).

   Tells the provider to close the TSP Session. No more TSP request may be sent using the previously obtained session Id.

We will show in the following section how to program a simple TSP consumer. Now if you only want to quickly get a consumer for testing your own TSP provider or to experiment with TSP you may skipped directly to §5.2.

5.1 Writing a simple consumer

We will shown in this specific section how to program a simple TSP consumer in C. This consumer will be able to ask for the first n sample symbols offered by any TSP provider. All example of code below are taken from `tsp/src/consumers/stdout` TSP consumer. The program have been slightly modified in order to ease understanding and presentation.

As usual the listing 7 shows the necessary headers you need for writing our TSP consumer application.

Listing 7: Headers of the TSP consumer application

```
/* .... */
#include <tsp_sys_headers.h>    /* platform independant data types definition */
#include <tsp_prjcfg.h>         /* TSP project config header                  */
#include <tsp_consumer.h>       /* TSP consumer API                           */
/* .... */
```

The TSP consumer library initialization and open session is shown at listing 8.

Listing 8: Initialize TSP consumer library and open TSP Session

```

    TSP_provider_t provider;

    /*
     * Initialize TSP consumer library
     */
    if (TSP_STATUS_OK!=TSP_consumer_init(&argc, &argv)) {
        retcode=1;
        return retcode;
    }

    /*
     * Connect to the TSP provider request handler using
     * a TSP URL
     */
    provider = TSP_consumer_connect_url(provider_url);

    /*
     * Check if we really found a provider using the URL
     */
    if (provider) {
        const char* info = TSP_consumer_get_connected_name(provider);
        printf("Found provider <%s>",info);
    }
    else {
        retcode = 3;
        return retcode;
    }

    /*
     * Now send the TSP Request Open
     */
    if (TSP_STATUS_OK!=TSP_consumer_request_open(provider, 0, NULL)) {
        return -1;
    }
```

Now we are ready for using our TSP Session. Listing 9 describe how to retrieve TSP Symbols Information from our TSP Session.

Listing 9: Request for Information on TSP Symbols

```
1
2     TSP_sample_symbol_info_t* aSSI = NULL;
3
4     /*
5      * Send TSP Request Information to the provider
6      * the TSP session is implicitly associated with the
7      * provider.
8      */
9     if (TSP_STATUS_OK!=TSP_consumer_request_information(provider)) {
10      return -1;
11    }
12
13    /*
14     * Retrieve provider information from provider Session object
15     */
16    information = TSP_consumer_get_information(provider);
17
18    /*
19     * The return information object contain the list
20     * of available symbols, aka a Sample Symbol Info List = SSI List.
21     */
22    printf("Provider is offering %d symbols on the provider",
23           TSP_SSIList_getSize(information->symbols));
24
25    /*
26     * Each element of the list is a SSI = Sample Symbol Info
27     * which is a structure containing several information:
28     *   name, dimension, type, minimal possible period, provider global index ...
29     * (consult API documentation to know more)
30     */
31    for (i = 0 ; i < TSP_SSIList_getSize(information->symbols), ++i) {
32        aSSI = TSP_SSIList_getSSI(information->symbols,i);
33        printf("Symbol <%s> has PGI <%d> and minimal possible period <%d>\n",
34               aSSI->name, aSSI->provider_global_index, aSSI->period);
35    }
```

Note that if do not want to be flooded with a list of 1000000 symbols coming from provider on may use the *filtered* request for information:

`TSP_consumer_request_filtered_information(provider)` .

See API documentation for the usage of the filtered request. Now that we have some information about available symbols we may build a TSP Request Sample for getting the first n symbols. This is shown by Listing 10.

Listing 10: Requesting selected symbols

```
1
2   /* declare the list of requested symbols */
3   TSP_sample_symbol_info_list_t  requested_symbols;
4   /*
5    * Request n symbols if n < number of available symbol
6    * Else Request number of available symbol.
7    */
8   int nb_symbols = n < TSP_SSIList_getSize(information->symbols) ?
9                       n :  TSP_SSIList_getSize(information->symbols);
10
11  /* Initialize requested symbols list */
12  TSP_SSIList_initialize(&requested_symbols,nb_symbols);
13
14  /*
15   * Now for each requested symbol
16   * we have to tell
17   *  - its name
18   *  - the requested period of sampling
19   */
```

```
20   for (i = 0 ; i < TSP_SSIList_getSize(requested_symbols) ; ++i) {
21     TSP_SSI_initialize_request_minimal(TSP_SSIList_getSSI(requested_symbols,i),
22                                        TSP_SSIList_getSSI(information->symbols,i)->name,
23                                        period);
24     printf("   symbol <%d> is <%s>\n",i,TSP_SSIList_getSSI(symbols,i)->name);
25   }
26
27   /* Now send Request Sample */
28   if (TSP_STATUS_OK!=TSP_consumer_request_sample(provider, &symbols)) {
29     return -1;
30   }
31
32   /* And finally ask for starting sampling process */
33   if(TSP_STATUS_OK!=TSP_consumer_request_sample_init(provider, 0, 0)) {
34     return -1;
35   }
```

Note that each `TSP_xxx` function returns a TSP status which is `TSP_STATUS_OK` on success and `TSP_STATUS_ERROR_xxx` on error. One should *always* check the returned code. For example a Provider may return `TSP_STATUS_ERROR_SYMBOLS` to a TSP request sample, which means that the request may not be satisfied because some sample symbol are unknown from provider. When this occurs you should check the `provider_global_index` of each sample symbol in the `symbols` list for a -1 value. Every symbol whose PGI is -1 is unknown from the provider, thus you may either remove those symbols from your request and send the updated request or request user action (TSP Consumer GUI). You may read the Ascii Writer code (located in `tsp/src/consumers/ascii_writer`) which implements a kind of "ignore unknown" symbols feature.

Now we can enter the loop for sample read and terminate sampling when we have received p samples. This is shown in Listing 11.

Listing 11: Consumer Sample loop

```
1
2     int new_sample;
3
4     /* A TSP sample as returned by read_sample */
5     TSP_sample_t sample;
6
7     /* The number of received sample set */
8     int n_received_sample = 0;
9
10    do {
11        if (TSP_STATUS_OK==TSP_consumer_read_sample(provider,&sample, &new_sample)) {
12
13            /* We have some sample to process */
14            if (new_sample) {
15                n_received_sample++;
16                printf("%s=%f\n",
17                       TSP_SSIList_getSSI(requested_symbols,
18                                          sample.provider_global_index)->name,
19                                          sample.uvalue.double_value);
20            }
21
22            /*
23             * we have not received any sample yet,
24             * wait a little time in order to avoid busy loop
25             */
26            else {
27                tsp_usleep(100*1000); /* gives time [10 Hz] for sample to come in */
28            }
29        else {
30            /* TSP sample read error */
31            return -1;
32        }
33    } while (n_received_sample < p);
34
35    /* End Sampling Process */
36    if (TSP_STATUS_OK!=TSP_consumer_request_sample_destroy(provider)) {
```

```
37          return -1;
38      }
39
40      /* Release memory */
41      TSP_SSIList_finalize(&requested_symbols);
```

Now we terminate TSP session and finalize TSP consumer library as shown in listing 12

Listing 12: Terminate TSP consumer

```
1
2           /*
3            * Terminate TSP Session
4            */
5           if (TSP_STATUS_OK!=TSP_consumer_request_close(provider)) {
6              return -1;
7           }
8
9           /*
10           * Disconnect from provider and End TSP consumer library.
11           */
12          TSP_consumer_disconnect_one(provider);
13          TSP_consumer_end();
```

We have reviewed how to program a simple TSP consumer using the TSP Consumer library. We have not handled some complex cases where symbols may be of differents types or how to request array or array slice. The main thing to remember is that all information you need to know about your sample symbol should be included in the updated list sample symbol information list you get from:

```
TSP_consumer_request_sample(provider, &symbols)
```

The interested reader should now have sufficient knowledge to read more TSP consumer codes by himself in the TSP source `tsp/src/consumers/xxx` in order to discover more complicated cases.

Again you should remember that TSP is an Open Source software with a living community living at https://savannah.nongnu.org/projects/tsp/. Do not hesitate to ask your tricky question on the developper mailing list: http://lists.nongnu.org/mailman/listinfo/tsp-devel.

5.2 Ready-to-use consumers

There is a growing numbers of ready-to-use TSP consumers, please check [Tea06, §11.2 TSP Consumers] for more informations on using ready-to-use TSP consumers. You may check the platform availabilty of each consumer first in table 1 on page 2 of this document. You may find information on each TSP Consumers application online directly at http://www.ts2p.org/tsp/API_doc/html/group__TSP__Consumers.html.

And again, you should ask for information on the mailing list: http://lists.nongnu.org/mailman/listinfo/tsp-devel.

A Installing prerequisite software

A.1 CMake

TSP uses cmake [CMa] as build system, CMake is used by many important opensource projects (KDE, MySQL, ...). This build system offers two crucial advantages regarding multi-platform build:

- CMake is multi-platform
- CMake supports various development environments
- CMake 2.4.x comes with two other very interesting tools
  - CPack which is a package generator tools, still in beta but used by TSP for generating Windows installer and Linux binary archive.
  - CTest which may be used to drive testing (use by TSP is under examination).

On Windows CMake has a graphical frontend that can be used to define build configuring variables: are we doing a win32 build, which binaries do we want to build, etc... CMake will generate the appropriate build files (Makefiles, Visual Studio project files, ...). On Linux there is a curse UI (`ccmake`) which offers the same functionnalities.

It is recommended to have a look at http://www.cmake.org/HTML/Documentation.html, to get familiar with this powerful and versatile set of tools.

A.2 ACPLT-ONCRPC

The Win32 TSP port use a package called ACPLT ONCPRC, which is a win32 port of the original Sun code. Since we had to recompile it with our target C compiler, the modified version of ACPLT ONCRPC has been shipped along with the TSP source in `tsp/external/ACPLT-ONCRPC/` . The TSP Team did send the patched source back to the original authors. The original project's home page is: http://www.plt.rwth-aachen.de/index.php?id=258

A.2.1 Verifying Portmap Service/Daemon

An RPC Server program should be able to register to the so-called *RPC Portmapper*. The RPC portmapper is generally a daemon on Unix and this is a Windows Service on the Windows platform. A TSP provider includes an RPC Server, so if you want your favorite TSP Provider to be able to start you should have an RPC portmapper up and running. On Windows, you can check that the RPC Portmapper is running by opening Control Panel/Administration Tools/Services in order to check that everything is behaving as expected. On Linux the command `service portmap status` (run as `root`) will show you the status of the RPC portmapper. For other platform please contact your system administrator for help on this subject.

A.3 PthreadsWin32

PthreadsWin32 is a software package developped and maintained by Red Hat Inc. Albeit it's thread implementation is not as fine grained that under Linux, unit tests showed a sufficient coverage of TSP needs. The project's home page is: http://sourceware.org/pthreads-win32/. In order to ease TSP source usage, the TSP source tree ship a version of PthreadWin32 which has been tested with TSP in `tsp/external/PthreadWin32` .

A.4 NullSoft Scriptable Install System

The Win32 TSP port use NullSoft Scriptable Install System (NSIS) [NSI] in order to produce the TSP for Windows Binary Installer. In fact we use CPack which has an NSIS Generator http://www.cmake.org/Wiki/CPack:Generator_Information. NSIS has an Open Source license and may be downloaded here: http://nsis.sourceforge.net/.

References

[CMa] CMake Homepage. https://www.cmake.org/.

[Dew06] Frederik Deweerdt. The blackboard: a debugging and reporting tool. Technical report, TSP Team, 2006. In preparation.

[NSI] NSIS Homepage. http://nsis.sourceforge.net/.

[Tea06] The TSP Team. The TSP Design & Programming Guide. Technical Report Rev. 1.0 for TSP v0.8.0, The TSP Team, 2006. Available at http://download.savannah.nongnu.org/releases/tsp/tsp_programming_guide-1.0.pdf.

[TSP] TSP Homepage at Savannah. https://savannah.nongnu.org/projects/tsp.