ExternalProcess ns-3 module

Version 2.1.0

Emanuele Giona

2024-10-03

Contents

1 Introduction

`ExternalProcess` is a simple ns-3 module to facilitate running external processes within ns-3 simulations.

Aim & Features

- Custom program execution as a process parallel to ns-3 simulations
- Parallel process is started and kept alive until required
- Bi-directional communication with processes based on TCP sockets
- Multiple parallel processes supported (each with own `ExternalProcess` instance)
- Non-interfering with Unix signals: watchdog thread for process supervision (Thanks @vincenzosu)

Note: This module is *currently NOT* intended for processes that have carry out operations asynchronously to the ns-3 simulation.

2 Installation guidelines

2.1 Requirements

This module has been developed on Ubuntu 18.04 LTS and Ubuntu 22.04 LTS with installed ns-3 versions respectively being ns-3.33 and ns-3.40.

System requirements:

- Default ns-3 requirements (more details)
- OS supporting POSIX standard with GNU extensions
  - Dependencies: `pthread_tryjoin_np()` (more details)
- Boost libraries **1.66 or later** (more details)
  - Dependencies: ASIO (`io_context`), Boost.System
  - Please ensure that ns-3 is built against the correct version of the library *prior* to installing `ExternalProcess`

2.2 Installation

This module supports both ns-3 build systems (namely *Waf*, used until version 3.35, and *CMake*, from 3.36 onwards), and the following instructions apply to either.

1. Download or clone the contents of this repository

```
git clone https://github.com/emanuelegiona/ns3-ext-process.git
```

2. Enter the cloned directory and copy the `ext-process` directory into your ns-3 source tree, under the `src` or `contrib` directory

```
cd ns3-ext-process
cp -r <path/to/ns3/installation>/contrib/
```

3. Configure & build ns-3

   From ns-3.36 and later versions (CMake)

```
cd <path/to/ns3/installation>
./ns3 configure
./ns3 build
```

   Versions up to ns-3.35 (Waf)

```
cd <path/to/ns3/installation>
./waf configure
./waf build
```

2.3 Test your installation

This module includes an ns-3 test suite named `ext-process`, which also doubles as usage example.

The external process is a simple echo TCP *client* implemented in Python.

Test requirements

- Correct path to the external process's launcher script in file `ext-process/test/ext-process-test-suite.cc`:

```
// Path to the launcher script handling external process's
   execution
const std::string g_launcherPath = "<path/to/ns3/installation>/
   contrib/ext-process/launcher-py.sh";
```

- Python 3

  Note that this requirement is only necessary due to implementation of `ext-process/echo.py`, *i.e.* the external process used in this example. Processes leveraging binaries of a different nature may not enforce this requirement.

Launching the test

The test suite can be executed by prompting the following commands into a terminal:

```
cd <path/to/ns3/installation>
./test.py -n -s ext-process
```

A correct installation would present the following output:

```
[0/1] PASS: TestSuite ext-process
1 of 1 tests passed (1 passed, 0 skipped, 0 failed, 0 crashed, 0
   valgrind errors)
```

2.4 Compatibility across OS and ns-3 versions

`ExternalProcess` is tested in development environments as well as within Docker images.

Current version has successfully passed tests in the following environments:

- Ubuntu 18.04, ns-3.35 (Docker)
  - Note: `libboost` is available in version 1.65.1; for testing purposes only, it has been upgraded to version 1.74 via unofficial repository `ppa:mhier/libboost-latest` (more details)
- Ubuntu 20.04, ns-3.40 (Docker)
- Ubuntu 22.04, ns-3.41 (Docker)

More details on Docker images used at this page.

3 Usage

3.1 Modes of use

`ExternalProcess` supports several modes of use that can be combined depending on application requirements:

3.1.1 Side process mode

This aspect concerns how the process is being executed w.r.t. an `ExternalProcess` instance.

- **Locally-launched process**

  This kind of processes represent the ones where `ExternalProcess::Create` fully controls the creation and termination of a side process starting from the given launcher path.

  Setting attribute `Launcher` to a valid path results in the creation of a side process of this kind.

- **Full-remote process**

  > Default behavior

  This kind of processes represent the ones where an `ExternalProcess` instance only sets up the communication channel with an independent side process.

  Setting attribute `Launcher` to the empty string (`""`) results in the interaction with a side process of this kind.

3.1.2 Communication roles

This aspect concerns the role taken by an `ExternalProcess` instance within the TCP communication channel.

This behavior is self-explanatory and set via attribute `TcpRole` to a valid value of `enum ExternalProcess::TcpRole`, as described below:

- **SERVER**

  > Default behavior

  The created instance acts as TCP server, accepting connections from the side process (TCP client).

- **CLIENT**

  The created instance acts as TCP client, connecting to the side process (TCP server).

3.1.3 Attention

Initial remarks for new users:

1. In **full-remote process** mode, attribute `Port` must be specified explicitly regardless of communication role.

2. When combining **full-remote process** mode and **CLIENT** role for ns-3, both `Address` and `Port` must be specified explicitly.
3. In **SERVER** role for ns-3, attribute `Address` will be ignored regardless of side process mode.

More details can be found below.

Migration from previous versions:

1. New features are fully retro-compatible with v2.0.0.
2. Default side process mode is now **full-remote processes**.

   Invalid paths in attribute `Launcher` are interpreted as a **locally-launched process** mode and will result in a failure.
3. Default communication role is still **SERVER**.

3.2 Handbook

This section will briefly present how to use `ExternalProcess` for your ns-3 simulation scripts or modules.

1. Creation of an `ExternalProcess` instance

```
Ptr<ExternalProcess> myExtProc = CreateObjectWithAttributes<
    ExternalProcess>(
"TcpRole", UintegerValue(ExternalProcess::TcpRole::SERVER),   //
    Optional: TCP role of this instance (default: server)
"Launcher", StringValue("<path/to/executable>"),              //
    Optional: empty-string indicates a full-remote process (i.e. no
     launcher needed); it is mandatory for locally-launched
    processes however
"CliArgs", StringValue("--attempts 10 --debug True"),         //
    Optional: CLI arguments for launcher script (depend on the
    executable)
"Address", StringValue("127.0.0.1"),                          //
    Optional: ignored for locally-launched processes; it is
    mandatory for full-remote processes when CLIENT role is
    selected for ns-3 however
"Port", UintegerValue(0),                                     //
    Optional: default value (0) lets the OS pick a free port
    automatically; it is mandatory != 0 for full-remote processes
    however
```

```
7  "Timeout", TimeValue(MilliSeconds(150)),                    //
       Optional: enables timeout on socket operations (e.g. accept/
       connect, write, read)
8  "Attempts", UintegerValue(10),                              //
       Optional: enables multiple attempts for socket operations (only
        if timeout is non-zero)
9  "TimedAccept", BooleanValue(true),                          //
       Optional: enables timeout on socket accept operations (see
       above, 'Attempts')
10 "TimedWrite", BooleanValue(true),                           //
       Optional: enables timeout on socket write operations (see above
       , 'Attempts')
11 "TimedRead", BooleanValue(true)                             //
       Optional: enables timeout on socket read operations (see above,
        'Attempts')
12 );
```

Explanation:

- ns-3 attribute `TcpRole` is *optional* and it represents the TCP role taken by the `ExternalProcess` instance; available values are defined in `enum ExternalProcess::TcpRole`.
- ns-3 attribute `Launcher`: for locally-launched processes, it is **mandatory** at the time of invoking `ExternalProcess::Create(void)`, and it should contain the path to an existing executable file (*e.g.* bash script or similar); a full-remote process can be specified by assigning the empty string (`""`) to this attribute.

  > Non-empty strings result in attempting to create locally-launcher processes; invalid paths will yield a creation failure.

- ns-3 attribute `CliArgs` is *optional* and it represents a single string containing additional CLI arguments to the external process (*default:* empty string `""`).

  > The first argument passed to every executable is the TCP port used for communication (see below); the string hereby provided should contain arguments and their respective values considering that whitespace is used as a delimiter when splitting to tokens.

  > *e.g.* the string `"--attempts 10 --debug True"` results in 4 additional tokens passed to the executable: `--attempts`, `10`, `--debug`, and `True`.

> No CLI arguments are passed to full-remote processes, per definition: `ExternalProcess` does not handle launching the process thus no argument can be passed in this manner.

- ns-3 attribute `Port` is *optional* and it represents the port to use to accept communications via a TCP socket (*default:* `0`, *i.e.* allows the OS to pick a free port).

  > This value is automatically passed as the **first parameter** to your executable. The external process is thus expected to set up a TCP client and connect to IP address `127.0.0.1:<Port>` or `localhost:<port>`.

  > Automatic port selection **is not** supported for full-remote processes.

- ns-3 attributes `TimedAccept`, `TimedWrite`, and `TimedRead` change the default behavior (*i.e.* blocking socket operations, also indefinitely) into a using a timeout and repeated attempts; respectively, they refer to socket's `accept()` (used in `ExternalProcess::Create()`), `write()`, and `read()`.
- ns-3 attributes `Timeout` and `Attempts` allow customization of socket operations using timeout and repeated attempts.

  > Note: the same settings are used throughout socket operations; in necessity of specifying different settings for each operation, users may change these values using `Object::SetAttribute()`, which is allowed during ongoing simulations too.

2. Execution of the external process

```
1 bool ExternalProcess::Create(void);
```

   The return value should be checked for error handling in unsuccessful executions.

3. Communication *towards* the external process

```
1 bool ExternalProcess::Write(
2 const std::string &str,
3 bool first = true,
4 bool flush = true,
5 bool last = true
6 );
```

   This function sends `str` to the external process, with remaining arguments enabling some degree of optimization (*e.g.* in a series of `Write`s, only flushing at the last one).

4. Communication *from* the external process

```
bool ExternalProcess::Read(std::string &str, bool &hasNext);
```

This function attempts to read `str` from the external process: `str` should be ignored if the return value of this `Read` equals **`false`**. If multiple `Read`s are expected, the `hasNext` value indicates whether to continue reading or not, proving useful to its usage as exit condition in loops.

5. Termination of an external process

Deletion of an `ExternalProcess` instance automatically takes of this task, but it is possible to explicitly perform it at any point of the simulation.

```
bool ExternalProcess::Teardown(void);
bool ExternalProcess::Teardown(pid_t childPid);     // Discouraged
```

The `childPid` value may be obtained from the same `ExternalProcess` instance by invoking the `ExternalProcess::GetPid(void)` function, as implemented by function `ExternalProcess::Teardown(void)`.

> Note: while `ExternalProcess::Teardown(void)` is thread-safe with the rest of `ExternalProcess` mechanisms, `ExternalProcess::Teardown(pid_t childPid)` is not; users are advised to terminate external processes only through the first function, invoked on the appropriate `ExternalProcess` object. More details are provided below.

6. Termination of all external processes (*e.g.* simulation fatal errors)

In order to prevent external processes from living on in cases of `NS_FATAL_ERROR` being invoked by the simulation, it is possible to explicitly kill all processes via a static function.

```
static void ExternalProcess::GracefulExit(void);
```

Being a static function, there is no need to retrieve any instance of `ExternalProcess` for this instruction.

3.3 Programs compatible with `ExternalProcess`

In order to properly execute external processes via this module, the following considerations should be taken:

- At least 1 CLI argument must be supported:
    1. TCP port for socket connection (**required**)

2. Any number of additional arguments; see discussion above for attribute `CliArgs` (*optional*)

- Socket connectivity as a client (when `ExternalProcess` acts as server, see discussion above around attribute `TcpRole`; reversed roles are supported as well)
  - The TCP socket should be opened at the beginning of execution and kept alive throughout the process lifetime
- Properly handle the following messaging prefixes:

```
// Macros for process messaging
#define MSG_KILL "PROCESS_KILL"
#define MSG_DELIM "<ENDSTR>"
#define MSG_EOL '\n'
```

  In particular, `MSG_KILL` is sent by the ns-3 simulation towards the external process via `Write`s and indicates a gentle request to terminate execution. Ignoring this message results in abrupt termination using a `kill()` syscall. `MSG_DELIM` and `MSG_EOL` are used for implementing a line-based socket communication s.t.:
  - `MSG_DELIM` represents a delimiter between multiple tokens within the same line;
  - `MSG_EOL` represents the end-of-line delimiter, used any time a line is intended to be flushed through the socket.

3.4 API Documentation

3.4.1 Macros

General utility

```
#define CURRENT_TIME Now().As(Time::S)
```

Retrieves the current simulation time; typically used for logging.

Interprocess communication (IPC)

```
#define MSG_KILL "PROCESS_KILL"
```

Represents a gentle request to terminate process execution. If properly handled by the external process, it may be useful to save any temporary data before exiting.

```
#define MSG_DELIM "<ENDSTR>"
```

Represents a delimiter between multiple tokens contained within the same line sent through the socket. This delimiter is added between any pair of strings resulting from invoking `ExternalProcess::Write()` consecutively twice.

Example

```
Ptr<ExternalProcess> ep = [...];
ep->Create();
[...]
ep->Write("str1", true, false, false);
ep->Write("str2", false, false, true);
[...]
```

The effective string sent through the socket will be exactly 1, with value: `str1<ENDSTR>str2\n`.

```
#define MSG_EOL '\n'
```

Represents a end-of-line delimiter, used any time a line is intended to be flushed through the socket.

Example

```
Ptr<ExternalProcess> ep = [...];
ep->Create();
[...]
ep->Write("str1", true, false, false);
ep->Write("str2", false, true, false);
ep->Write("strX", false, false, false);
ep->Write("strY", false, false, true);
[...]
```

The effective strings sent through the socket will be exactly 2, with values:

- `str1<ENDSTR>str2\n`, and
- `strX<ENDSTR>strY\n`.

3.4.2 Global functions

```
void* WatchdogFunction(void* arg);
```

Function to use in the watchdog thread (POSIX). Its usage is typically transparent to the user.

```
void* AcceptorFunction(void* arg);
```

Function to use in acceptor threads (POSIX). Its usage is typically transparent to the user.

```
void* WriterFunction(void* arg);
```

Function to use in writer threads (POSIX). Its usage is typically transparent to the user.

```
void* ReaderFunction(void* arg);
```

Function to use in reader threads (POSIX). Its usage is typically transparent to the user.

3.4.3 Global variables

```
static WatchdogSupport g_watchdogArgs;
```

Represents the global variable holding arguments for the watchdog thread.

```
static pthread_t g_watchdog;
```

Watchdog thread for checking running instances.

```
static bool g_watchdogExit = false;
```

Flag indicating the exit condition for the watchdog thread.

```
static std::map<pid_t, ExternalProcess*> g_runnerMap;
```

Map associating PID to instances that spawned them.

```
static pthread_mutex_t g_watchdogExitMutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutex for exit condition for the watchdog thread.

```
static pthread_mutex_t g_watchdogMapMutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutex for runner map for the watchdog thread.

```
static pthread_mutex_t g_watchdogTeardownMutex =
    PTHREAD_MUTEX_INITIALIZER;
```

Mutex for accessing ExternalProcess::Teardown() from multiple threads.

```
static pthread_mutex_t g_gracefulExitMutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutex for accessing ExternalProcess::GracefulExit() from multiple threads.

3.4.4 Struct: WatchdogSupport

```
struct WatchdogSupport {
  bool m_initialized = false;                          //!< Flag
     indicating whether the support variabled is initialized.
  ExternalProcess::WatchdogData* m_args = nullptr;     //!< Pointer to
     the watchdog arguments to use.
};
```

3.4.5 Class: ExternalProcess

```
class ExternalProcess: public Object {};
```

Class for handling an external side process interacting with ns-3.

This class creates a new process upon initialization and sets up communication channels via a TCP socket. Employing a leader/follower classification of roles, ns-3 can act either as leader or follower, with the external process taking the complementary role in this relationship. As such, the 'TcpRole' attribute is self-explanatory, with the default behavior being set to SERVER. With this configuration, objects of this class shall set up the TCP server, with the client necessarily implemented by the external process.

A watchdog thread is spawned upon the first instance of this class being created, running until the last ExternalProcess object goes out of scope. This thread periodically checks whether external processes are still alive by means of their PID. Watchdog settings may be customized via attributes 'CrashOnFailure' and 'WatchdogPeriod'.

By default, socket operations are blocking, possibly for an indefinite time. Attributes 'TimedAccept', 'TimedWrite', and 'TimedRead' change the behavior of respective socket operations – i.e. accept() / connect(), write(), and read() – into using a timeout and, if configured, repeated attempts. Attributes 'Timeout' and 'Attempts' allow customization of such behavior, but are applied to any enabled timed operation equally.

> All socket operations are blocking, even in their timed versions. Asynchronous mode using callbacks is out of the scope of the current implementation.

Attributes `ExternalProcess` supports the following attributes within ns-3 `Object`'s attribute system.

```
"TcpRole"
```

TCP role implemented by this instance.

- Default value: `UintegerValue((uint8_t)TcpRole::SERVER)`
- *Optional*

```
"Launcher"
```

Absolute path to the side process launcher script; if empty, a full-remote external process is expected

- Default value: `StringValue("")`
- *Optional*
- **Locally-launched process:** must be a non-empty string representing a valid path within the filesystem
- **Full-remote process:** the empty string (`""`) indicates starting a full-remote process

```
"CliArgs"
```

String containing command-line arguments for the launcher script; tokens will be split by whitespace first.

- Default value: `StringValue("")`
- *Optional*
- **Locally-launched processes:** command-line arguments specified here are passed *after* the TCP port number the external process has to use for communicating with ns-3
- **Full-remote processes:** no CLI argument is passed to the process

```
"CrashOnFailure"
```

Flag indicating whether to raise a fatal exeception if the external process fails.

- Default value: `BooleanValue(true)`
- *Optional*

```
"WatchdogPeriod"
```

Time period spent sleeping by the watchdog thread at the beginning of the PID checking loop; lower values will allow detection of process errors quicker, longer values greatly reduce busy waits.

- Default value: `TimeValue(MilliSeconds(100))`
- *Optional*

- Range: min = `MilliSeconds(1)`, max = `Minutes(60)`
- The first instance of `ExternalProcess` invoking `Create()` will set the watchdog thread with this value; any subsequent `ExternalProcess` instance will not affect the watchdog polling period while there is a live watchdog thread

```
"GracePeriod"
```

Time period spent sleeping after killing a process, potentially allowing any temporary data on the process to be stored.

- Default value: `TimeValue(MilliSeconds(100))`
- *Optional*
- Range: min = `MilliSeconds(0)`

```
"Address"
```

IP address for communicating with external process; this is mandatory for ns-3 in CLIENT role and full-remote process in SERVER role.

- Default value: `StringValue("")`
- *Optional*
- **Full-remote processes** and **ns-3 CLIENT role**: an explicit value must be provided

```
"Port"
```

Port number for communicating with external process; if 0, a free port will be automatically selected by the OS.

- Default value: `UintegerValue(0)`
- *Optional*
- **Full-remote processes**: a non-zero value must be provided and it must match with the port used by the remote process

```
"Timeout"
```

Maximum waiting time for socket operations (e.g. accept); if 0, no timeout is implemented.

- Default value: `TimeValue(MilliSeconds(0))`
- *Optional*

- Range: min = `MilliSeconds(0)`

```
"Attempts"
```

Maximum attempts for socket operations (e.g. accept); only if a non-zero timeout is specified.

- Default value: `UintegerValue(1)`
- *Optional*
- Range: min = 1

```
"TimedAccept"
```

Flag indicating whether to apply a timeout on socket accept() / connect (), implementing 'Timeout' and 'Attempts' settings; only if a non-zero timeout is specified.

- Default value: `BooleanValue(false)`
- *Optional*

```
"TimedWrite"
```

Flag indicating whether to apply a timeout on socket write(), implementing 'Timeout' and 'Attempts' settings; only if a non-zero timeout is specified.

- Default value: `BooleanValue(false)`
- *Optional*

```
"TimedRead"
```

Flag indicating whether to apply a timeout on socket read_until(), implementing 'Timeout' and 'Attempts' settings; only if a non-zero timeout is specified.

- Default value: `BooleanValue(false)`
- *Optional*

```
"ThrottleWrites"
```

Minimum time between a read and a subsequent write; this delay is applied before writing.

- Default value: `TimeValue(MilliSeconds(0))`
- *Optional*
- Range: min = `MilliSeconds(0)`

```
"ThrottleReads"
```

Minimum time between a write and a subsequent read; this delay is applied before reading.

- Default value: `TimeValue(MilliSeconds(0))`
- *Optional*
- Range: min = `MilliSeconds(0)`

Public API

```
enum TcpRole {
  SERVER,   //!< This instance acts as a TCP server, with the client
     implemented by the external process.
  CLIENT    //!< This instance acts as a TCP client, with the server
     implemented by the external process.
};
```

Represents the role of this instance in the TCP communication with an external process.

> See attribute 'TcpRole'.

```
struct WatchdogData {
  bool m_crashOnFailure;    //!< [in] Flag indicating whether to raise
     a fatal exeception if the external process fails.
  Time m_period;            //!< [in] Time period spent sleeping by the
      watchdog thread at the beginning of the PID checking loop.
};
```

Represents the argument for a watchdog thread. Its usage is typically transparent to the user.

```
struct BlockingArgs {
  pthread_t* m_threadId = nullptr;      //!< [inout] Pointer to the
     blocking thread ID to set to -1.
  bool* m_exitNormal = nullptr;         //!< [inout] Pointer to flag
     indicating normal exit from thread.
  pthread_mutex_t* m_mutex = nullptr;   //!< [in] Pointer to the mutex
     to use.
  pthread_cond_t* m_cond = nullptr;

  /** [Constructor] */
};
```

Represents additional arguments for implementing BlockingSocketOperation() in thread functions. Its usage is typically transparent to the user.

```
enum EP_THREAD_OUTCOME {
  FATAL_ERROR = -1,   //!< A fatal error has occurred during thread
     execution (e.g. invalid arguments, unexpected exceptions).
  SUCCESS = 0,        //!< Thread execution resulted in a successful
     completion of the function.
  FAILURE = 1         //!< Thread execution resulted in a failed
     completion of the function (non-fatal error).
};
```

Represents the thread outcome status in the execution of AcceptorFunction(), ConnectorFunction(), WriterFunction(), and ReaderFunction(). Its usage is typically transparent to the user.

```
struct AcceptorData {
  int* m_threadOutcome = nullptr;                          //!< [out]
     Pointer to int value representing thread outcome (-1: fatal error,
      0: success, 1: failure).
  boost::asio::ip::tcp::acceptor* m_acceptor = nullptr;   //!< [in]
     Pointer to boost::asio acceptor.
  boost::asio::ip::tcp::socket* m_sock = nullptr;         //!< [in]
     Pointer to boost::asio socket.
  boost::system::error_code* m_errc = nullptr;            //!< [out]
     Pointer to boost::system error code.
  BlockingArgs* m_blockingArgs = nullptr;                 //!< [in]
     Pointer to additional args for blocking operations, if provided;
     if nullptr, timed operation is assumed.

  /** [Constructor] */
};
```

Represents the argument for an acceptor thread (i.e. implemented by AcceptorFunction). Its usage is typically transparent to the user.

```
struct ConnectorData {
    int* m_threadOutcome = nullptr;
                                                              //!< [
       out] Pointer to int value representing thread outcome (-1: fatal
        error, 0: success, 1: failure).
    boost::asio::ip::tcp::socket* m_sock = nullptr;
                                                 //!< [in] Pointer to
       boost::asio socket.
    boost::asio::ip::basic_resolver_results<boost::asio::ip::tcp>*
       m_endpoints = nullptr;   //!< [in] Pointer to boost::asio
       endpoints resolved from IP:PORT pair.
```

```
5     boost::system::error_code* m_errc = nullptr;
                                                    //!< [out] Pointer to
          boost::system error code.
6     BlockingArgs* m_blockingArgs = nullptr;
                                                          //!< [in]
         Pointer to additional args for blocking operations, if provided;
          if nullptr, timed operation is assumed.
7
8     /** [Constructor] */
9 };
```

Represents the argument for a connector thread (i.e. implemented by ConnectorFunction). Its usage is typically transparent to the user.

```
1 struct WriterData {
2   int* m_threadOutcome = nullptr;                   //!< [out] Pointer
       to int value representing thread outcome (-1: fatal error, 0:
       success, 1: failure).
3   boost::asio::ip::tcp::socket* m_sock = nullptr;   //!< [in] Pointer
       to boost::asio socket.
4   boost::asio::mutable_buffer* m_buf = nullptr;     //!< [in] Pointer
       to boost::asio::streambuf to write data from.
5   boost::system::error_code* m_errc = nullptr;      //!< [out] Pointer
       to boost::system error code.
6   BlockingArgs* m_blockingArgs = nullptr;           //!< [in] Pointer
       to additional args for blocking operations, if provided; if
       nullptr, timed operation is assumed.
7
8   /** [Constructor] */
9 };
```

Represents the argument for a writer thread (i.e. implemented by WriterFunction). Its usage is typically transparent to the user.

```
1 struct ReaderData {
2   int* m_threadOutcome = nullptr;                   //!< [out] Pointer
       to int value representing thread outcome (-1: fatal error, 0:
       success, 1: failure).
3   boost::asio::ip::tcp::socket* m_sock = nullptr;   //!< [in] Pointer
       to boost::asio socket.
4   boost::asio::streambuf* m_buf = nullptr;          //!< [out] Pointer
       to boost::asio::streambuf to read data to.
5   boost::system::error_code* m_errc = nullptr;      //!< [out] Pointer
```

```
        to boost::system error code.
6   BlockingArgs* m_blockingArgs = nullptr;          //!< [in] Pointer
        to additional args for blocking operations, if provided; if
        nullptr, timed operation is assumed.
7
8   /** [Constructor] */
9 };
```

Represents the argument for a reader thread (i.e. implemented by ReaderFunction). Its usage is typically transparent to the user.

```
1 static void ExternalProcess::GracefulExit(void);
```

Terminates all external processes spawned during this simulation.

> This function should be invoked whenever `NS_FATAL_ERROR` is used, preventing external processes to remain alive despite no chance of further communication.

> This function is thread-safe.

```
1 ExternalProcess::ExternalProcess();
```

Default constructor.

```
1 virtual ExternalProcess::~ExternalProcess();
```

Default destructor.

```
1 static TypeId ExternalProcess::GetTypeId(void);
```

Registers this type.

Returns:

- The TypeId.

```
1 bool ExternalProcess::Create(void);
```

Creates a TCP-based interface for communicating with a side process. Upon providing a launcher script via attribute ‘Launcher’, a local process is created, otherwise full-remote operation is supported too.

Returns:

- True if the creation has been successful, False otherwise.

> This operation may be blocking.

> Functionality changes depending on 'TcpRole' attribute value: SERVER (default) yields this instance acting as TCP server, whereas CLIENT yields this instance acting as TCP client.

```
1 bool ExternalProcess::IsRunning(void) const;
```

Retrieves whether the side process is running or not.

Returns:

- True if the side process is running, False otherwise.

```
1 pid_t ExternalProcess::GetPid(void) const;
```

Retrieves the PID of the side process.

Returns:

- The PID of the side process previously set up via `ExternalProcess::Create()`.

```
1 bool ExternalProcess::Teardown(void);
```

Performs process teardown operations using the result of `ExternalProcess::GetPid()`.

Returns:

- True if this external process is no longer tracked by the watchdog, False otherwise.

> This function is thread-safe.

```
1 bool ExternalProcess::Teardown(pid_t childPid);
```

Performs process teardown operations.

Parameters:

- `[in] childPid` PID of the child process associated with this teardown procedure. If different than `-1`, it will send a `SIGKILL` signal to the provided PID.

Returns:

- True if this external process is no longer tracked by the watchdog, False otherwise.

> This function is **NOT thread-safe**: a lock shall be acquired on `g_watchdogTeardownMutex` previously to the invocation of this function.

> Its usage is discouraged; users should invoke `ExternalProcess::Teardown(void)` instead, on the instance associated to the process intended to be shutdown.

```
1 bool ExternalProcess::Write(const std::string &str, bool first = true,
     bool flush = true, bool last = true);
```

Writes a string to the external process through the socket.

Parameters:

- `[in]` `str` String to write.
- `[in]` `first` Whether the string is the first of a series of writes (Default: true).
- `[in]` `flush` Whether to flush after writing this string or not (Default: true).
- `[in]` `last` Whether the string is the last of a series of writes (Default: true).

Returns:

- True if the operation is successful, False otherwise.

> This operation may be blocking.

```
1 bool ExternalProcess::Read(std::string &str, bool &hasNext);
```

Reads a string from the external process through the socket.

Parameters:

- `[out]` `str` String read (if return is True; discard otherwise).
- `[out]` `hasNext` Whether there is going to be a next line or not.

Returns:

- True if the operation is successful, False otherwise.

> This operation may be blocking.

4 Citing this work

If you use the module in this repository, please cite this work using any of the following methods:

APA

```
Giona, E. ns3-ext-process [Computer software]. https://doi.org/10.5281/
    zenodo.8172121
```

BibTeX

```
@software{Giona_ns3-ext-process,
author = {Giona, Emanuele},
doi = {10.5281/zenodo.8172121},
license = {GPL-2.0},
title = {{ns3-ext-process}},
url = {https://github.com/emanuelegiona/ns3-ext-process}
}
```

Bibliography entries generated using Citation File Format described in the CITATION.cff file.

5 License