ExternalProcess ns-3 module

Version 1.0.3

Emanuele Giona

2024-06-03

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 Unix named pipes
- 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

This module supports both ns-3 build systems (namely *Waf*, used until version 3.36, 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` directory

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

3. Configure & build ns-3

   From ns-3.36 and later versions (CMake)

   ```
   ./ns3 configure
   ./ns3 build
   ```

   Versions prior ns-3.36 (Waf)

```
./waf configure
./waf build
```

3 Usage

3.1 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>(
    "ProcessLauncher", StringValue("<path/to/executable>"),
    "ProcessExtraArgs", StringValue("<optional CLI arguments to
        the executable>"),
    "CrashOnFailure", BooleanValue(true),
    "ThrottleWrites", MilliSeconds(0),
    "ThrottleReads", MilliSeconds(0)
);
```

   - ns-3 attribute `ProcessLauncher` is **mandatory** at the time of invoking `ExternalProcess::Create(void)`: it should contain the path to an existing executable file (*e.g.* bash script or similar).
   - ns-3 attribute `ProcessExtraArgs` is *optional* and it represents a single string containing additional CLI arguments to the external process (*default:* empty string – not even passed to the executable).
   - ns-3 attribute `CrashOnFailure` is *optional* and it specifies whether to raise a fatal exception upon failure detection of the external process (*default:* **`true`**).
   - ns-3 attribute `ThrottleWrites` is *optional* and it specifies whether and, eventually, the amount of time to wait between a `Read()` and a subsequent `Write()` (*default:* 0 ms – no throttling).
   - ns-3 attribute `ThrottleReads` is *optional* and it specifies whether and, eventually, the amount of time to wait between a `Write()` and a subsequent `Read()` (*default:* 0 ms – no throttling).

> Throttling may be useful whenever the external process in not able to stay on par with ns-3's speed in reading/writing from/to named pipes.

- ns-3 attribute `ReadHangsTimeout` is *optional* and it specifies whether and, eventually, the amount of time to consider a simulation hanged on an empty-read loop (*default:* 0 ms – no timeout).

2. Execution of the external process

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

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

3. Communication *towards* the external process

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

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.

```
void ExternalProcess::Teardown(pid_t childPid);
```

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

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.

3.2 Programs compatible with `ExternalProcess`

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

- At least 2 CLI arguments must be supported:
  1. Input (*i.e.* ns3-to-proc) named pipe (**required**)
  2. Output (*i.e.* proc-to-ns3) named pipe (**required**)
  3. Additional arguments (single string consisting of `ProcessExtraArgs` attribute value, *optional*)
- Leverage named pipes blocking operations
  - Open and close named pipes at need
  - In this way, the external process will be waiting for commands and/or data from the ns-3 simulation without requiring any synchronization mechanisms
- Properly handle the following messaging prefixes:

```
// Macros for process messaging
#define MSG_KILL "PROCESS_KILL"
#define MSG_READY "PROCESS_READY"
#define MSG_WRITE_BEGIN "NS3_WRITE_BEGIN"
#define MSG_WRITE_END "NS3_WRITE_END"
#define MSG_READ_BEGIN "NS3_READ_BEGIN"
#define MSG_READ_END "NS3_READ_END"
```

  In particular, `MSG_KILL`, `MSG_WRITE_BEGIN`, and `MSG_WRITE_END` and sent by the ns-3 simulation towards the external process via `Write`s. `MSG_READY`, `MSG_READ_BEGIN`, and `MSG_READ_END` should be sent by the external process towards the ns-3 simulation in the following cases:
  - `MSG_READY`: as soon as the program is initialized and ready to receive ns-3 commands and data
  - `MSG_READ_BEGIN` and `MSG_READ_END`: the program should enclose any of its output within these two message prefixes for a correct interpretation by the ns-3 simulation

3.3 API Documentation

3.3.1 Macros

General utility

```
#define CURRENT_TIME Now().As(Time::S)
#define PIPE_TRAIL_IN "pipe_proc_to_ns3"
#define PIPE_TRAIL_OUT "pipe_ns3_to_proc"
```

Interprocess communication

```
#define MSG_KILL "PROCESS_KILL"
#define MSG_READY "PROCESS_READY"
#define MSG_WRITE_BEGIN "NS3_WRITE_BEGIN"
#define MSG_WRITE_END "NS3_WRITE_END"
#define MSG_READ_BEGIN "NS3_READ_BEGIN"
#define MSG_READ_END "NS3_READ_END"
```

3.3.2 Global functions

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

Function to use in the watchdog thread.

Arguments: - (IN) `arg` Pointer to process failure policy (True = crash on failues).

Returns: - No return (constant nullptr).

3.3.3 Global variables

```
//!< Map associating PID to instances that spawned them.
static std::map<pid_t, ExternalProcess*> g_runnerMap;

//!< Watchdog thread for checking running instances.
static pthread_t g_watchdog;

//!< Flag indicating the exit condition for the watchdog thread.
static bool g_watchdogExit = false;

//!< Mutex for runner map and exit condition for the watchdog thread.
static pthread_mutex_t g_watchdogMutex = PTHREAD_MUTEX_INITIALIZER;
```

3.3.4 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 named pipes. Streams regarding communication channels are opened and closed at need, in order to leverage named pipes' blocking on empty reads, thus avoiding busy waits. The external process should operate in the same way as well.

Attributes

```
"ProcessLauncher"
```

Absolute path to the process launcher script.

- Default value: `StringValue("")`
- Accesses: `ExternalProcess::m_processLauncher`

```
"ProcessExtraArgs"
```

String containing additional arguments to process launcher script.

- Default value: `StringValue("")`
- Accesses: `ExternalProcess::m_processExtraArgs`

```
"CrashOnFailure"
```

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

- Default value: `BooleanValue(true)`
- Accesses: `ExternalProcess::m_crashOnFailure`

```
"ThrottleWrites"
```

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

- Default value: `TimeValue(MilliSeconds(0))`
- Accesses: `ExternalProcess::m_throttleWrites`

```
"ThrottleReads"
```

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

- Default value: `TimeValue(MilliSeconds(0))`
- Accesses: `ExternalProcess::m_throttleReads`

```
"ReadHangsTimeout"
```

Timeout for preventing a simulation from hanging on empty reads; only applied for consecutive reads only.

- Default value: `TimeValue(MilliSeconds(0))`
- Accesses: ExternalProcess::m_readHangsTimeout

Public API

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

Terminates all external processes spawned during this simulation.

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

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

Default constructor.

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

Default destructor.

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

Registers this type.

Returns: - The TypeId.

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

Creates a side process given a launcher script.

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

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

Retrieves whether the side process is running or not.

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

```
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()`.

```
void ExternalProcess::Teardown(pid_t childPid);
```

Performs process teardown operations (e.g. deleting named pipes, etc.).

Arguments: - (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
    .
```

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

Writes a string as a line to the output named pipe (ns-3 –> process).

Arguments: - (IN) `str` String to write to the named pipe. - (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.

Warning: - This operation may be blocking.

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

Reads a line from the input named pipe (process –> ns-3) and returns it as a string.

Arguments: - (OUT) `str` String read from the named pipe (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.

Warning: - 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