Serial-TCP/IP bridge

Petr Zemek
xzemek02@stud.fit.vutbr.cz

2009-03-06

Abstract

This document describes design and implementation of a serial-TCP/IP bridge, implemented for our school course named "Data Communications, Computer Networks and Protocols". It is a concurrent TCP/IP server written in C++ that accepts incoming client connections and resends received messages (text/binary) to the serial port. It also resends messages from the serial port to all connected clients. The bridge is fully configurable via a simple text configuration file.

Keywords

Serial port, TCP/IP, bridge, concurrent server, C++.

1 Introduction

Regardless of the lack of the serial port [7] on most of the current computers for masses, it is still useful – mostly because of its simplicity. For example, it is used to update firmware on various consumer devices, for microcontrollers programming and connecting various devices like mice, printers and modems to a computer. It is also used for network devices configuration, like routers.

One typically communicates with the serial device directly through the serial port via a locally installed computer terminal emulator, which has the disadvantage that one has to be physically working with the terminal. But what if one needs to configure some router remotely from the Internet? This is the situation where the serial-TCP/IP bridge comes handy.

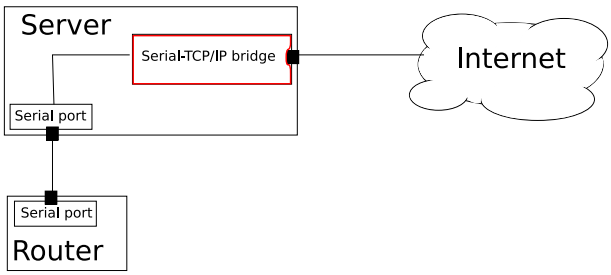

Figure 1: The role and position of the serial-TCP/IP bridge.

Figure 1 shows how it works. If there is a server on which a serial-TCP/IP bridge is running, one can connect to that bridge via `telnet` or `netcat` from the Internet and communicate with the serial device remotely. The bridge resends everything from the client to the serial port and when there is a message from the serial device, it resends it to the client.

The following sections describe design and implementation of such a serial-TCP/IP bridge. Program usage is also included.

2 Project design

This section describes the project design. First there is an overall view on the classes used in the project, then the concurrent behaviour of the server is discussed and finally the way how errors are handled is given.

2.1 Overall design

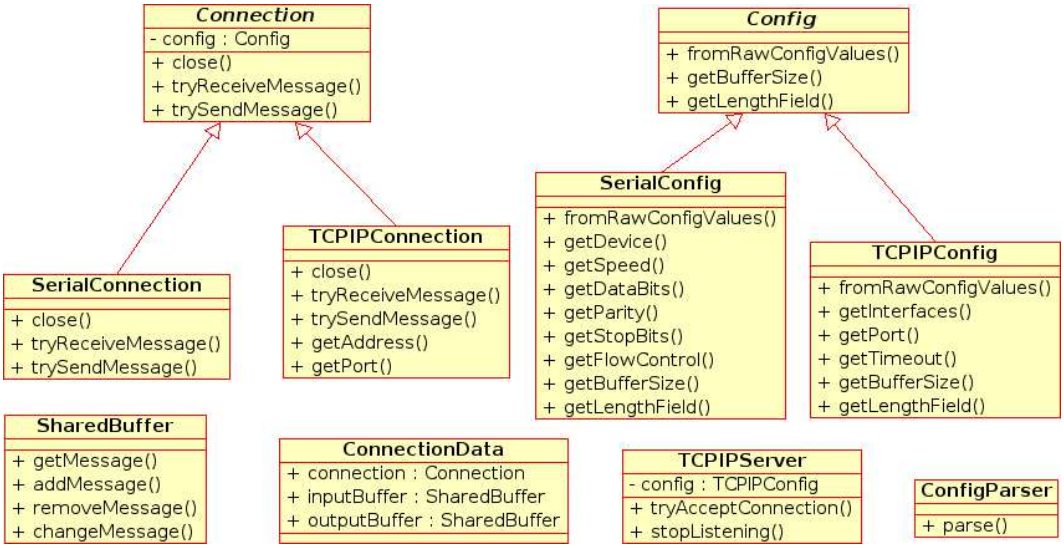

Figure 2: Overall class diagram.

Figure 2 shows the overall class diagram. The `ConfigParser` class is used to parse a bridge configuration file into an object (see the implementation part) which is then used to create instances of two configuration classes: the `SerialConfig`, which is used to setup the serial port and the `TCPIPConfig`, which is used to configure the TCP/IP part of the bridge. This part is represented by the `TCPIPServer` class, which accepts client connections to the bridge, which are represented by the `TCPIPConnection` class. The `ConnectionData` class is used to store a connection (along with other information) with input and output buffer, which are represented by the `SharedBuffer` class.

2.2 Concurrent server behaviour

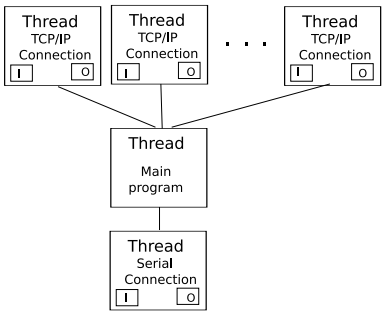

Figure 3: How the concurrent behaviour is achieved.

One part of the assignment was that the bridge must be concurrent, i.e. it has to be able to serve more than one client at a time. To achieve this, I decided to implement a threaded bridge, so it has a main thread which accepts client connections and creates one thread per connection. Figure 3 depicts this kind of a solution.

Note that there is also a separate thread for the communication with the serial port. The main thread also transfers messages between connection buffers, so when there is a message in the serial connection output buffer, it copies that message to each client connections input buffer. The same actions are taken in case of a new message in a client connections output buffer (the message is transferred to the serial connections input buffer).

2.3 Error handling

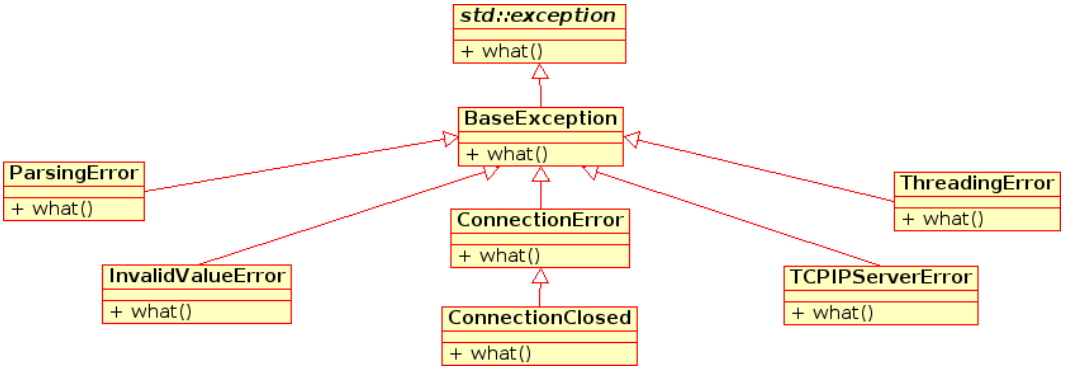

Figure 4: Exceptions hierarchy.

As for the error handling, I decided to use exceptions and I created a separate exception for each area of problems (connection-related problems, threading problems etc.). Figure 4 shows the exception classes hierarchy in detail.

3 Project implementation

The bridge is implemented in C++ using the standard C library and C++98 library (with STL) with GNU and POSIX extensions. The implementation details of all project parts follows.

3.1 Main program

When the bridge is started, the passed configuration file is parsed by the `ConfigParser` class. Since the grammar for the configuration file is regular, I implemented it as a simple state machine. The parsed configuration file is stored into an associative array (option name ⇒ raw option value) and this array is then used co create configurations for the serial part and the TCP/IP part. Each configuration class (`SerialConfig` and `TCPIPConfig`) gets from the array only options that are relevant to that class purpose, parses them, checks their validity (there are several restrictions, like the port number must be a natural number between 1 and 65535) and stores them. If some option is missing, default values are used instead.

If both configurations are valid, instances of the `SerialConnection` and `TCPIPServer` classes are created and the server starts accepting connections. When there is a new connection, the server creates an instance of the `TCPIPConnection` class and puts it into an instance of the `ConnectionData` class, which is then stored into the connection data container. The acceptance is nonblocking, so the bridge is also able to transfer messages between buffer as was described in section 2.2, which is (apart from the connection acceptance part) the second most important part of its job (it is fast so it does not cause any problems with new connections).

When there is an error or a request to stop the server (by sending a signal to the bridge, like `SIGTERM`), it breaks the main loop, correctly ends all connections and exits.

3.2 TCP/IP part

The TCP/IP part of the bridge (client connections acceptance, settings and communication) is using UNIX sockets and was implemented according to [5].

To start listening and accepting connections, the `TCPIPServer` class uses functions `socket()` and `fcntl()` (to create and setup a socket), `setsockopt()` (to set port reusing), `bind()` (to bind the address to the socket), `listen()` (to create a client connection queue) and `accept()` (to accept an iccoming client connection). The IP address of a network interface address is get by using the `ioctl()` function. To make all operations nonblocking, `fcntl()` is used.

The `TCPIPConnection` class uses functions `send()` and `recv()` to send messages and receive them. Messages are stored into a buffer of a size that is specified in the TCP/IP part of the configuration.

3.2.1 Extensions, restrictions and things to note

The configuration option `length_field` can be maximally 4 bytes, because there are is no equivalent of the function `ntohl()` for a number with more bytes than 4.

Connection timeout is reseted not only when a message is received, but also if a message is sent (to allow large data transfers).

3.3 Serial part

The serial part of the bridge (serial port settings and communication) was implemented according to [3], [2] and [6].

The `SerialConnection` class uses functions `open()` (to open the serial device), `tcgetattr()` and `tcsetattr()` (to set the serial port). To make all operations nonblocking, `fcntl()` is again used. Messages are received from the serial port by calling `read()` and sent by calling `write()`.

3.3.1 Extensions, restrictions and things to note

Because there seems to be no way how to set 1.5 stop bits on Linux, this number of stop bits is *not* supported. Mark and space parity is set using an undocumented flag `CMSPAR` [4].

3.4 Threading part

As for the implementation of the threading behaviour, POSIX threads were used and [1] was used as the main information resource.

After a connection is created (does not matter whether it is a TCP/IP connection or a serial port connection), a handler (new thread) for that connection is created using the `pthread_create()` function. There is a loop in that handler, in which the thread checks whether the connection should be closed (expired connection timeout; uses `time()` to get the current time). If so, then it closes that connection and signals to the main thread that this connection should be removed from the connections container and exits. After that, it checks whether there are is a message to be received and if so, it receives it and put it into the output buffer. If there are any messages in the input buffer, it sends the first one (only one message is sent in a single loop pass).

Note that both buffers (input and output) can be accessed from the connection handler and also from the main thread, so the operations on these buffers have to be synchronized. To achieve this, functions `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()` and `pthread_mutex_destroy()` are used. To ensure proper message removal from both buffers, messages are stored in a pair (message, ID) and a message can be removed from the buffer by its ID.

4 Program usage

First you need to compile the program, so run `make` in the project directory. After the compilation you can start the server by running:

```
./bridge -f CONF_FILE_PATH
```

where `CONF_FILE_PATH` is a configuration file path. An example of the configuration file is provided in the project directory (`bridge.cfg`). The running server can be stopped by pressing `Ctrl+C` (when the server is running on foreground) or sending a `SIGINT` or `SIGTERM` signal (when the server is running on background) with `kill -SIGTERM ‘pidof bridge‘`.

If the server was configured properly, you should be able to connect to it by using `nc` (netcat; `nc interface port`) or simply `telnet` (`telnet interface port`).

5 Conclusion

All specifications from the assignment were observed and satisfied during the project development. Main parts were designed considering possible future development and extensibility, like adding new configuration options, adding new methods of data receiving or sending and lowering the CPU usage. The application was successfully tested on the faculty server `merlin` (CentOS 5, 2.6.16, x86-64) and on my two systems (Debian 5.0, 2.6.28, x86-64 and Kubuntu 8.04, 2.6.24, x86-32). Unit testing suites (63 tests in total) for some classes were written using the `CPPUnit` library and run regularly. Memory leaks were also taken into account and tested with `valgrind` (`<http://valgrind.org/>`) and no memory leaks were discovered (even in case of an error).

Used libraries

Standard C library and C++98 library (with STL) with GNU and POSIX extensions
CPPUnit for unit testing (<http://cppunit.sourceforge.net/>)

Metrices

Source files: 35
Source lines of code: 4029 (without empty lines)
Executable file size: 1.1 MB (GNU/Linux x86-64, g++ 4.3.2)

References

[1] Barney, B.: POSIX Threads Programming. 2009, [Online; visited 2009-03-06].
URL <https://computing.llnl.gov/tutorials/pthreads/>

[2] Iannella, A.: Linux Serial Port Programming Mini-Howto. March 1997, [Online; visited 2009-03-06].
URL <http://slackware.osuosl.org/slackware-3.3/docs/mini/Serial-Port-Programming>

[3] Lawyer, D. S.; Hankins, G.: Serial HOWTO. December 2008, [Online; visited 2009-03-06].
URL <http://tldp.org/HOWTO/Serial-HOWTO.html>

[4] Lochmatter, T.: Linux and MARK/SPACE Parity. 2008, [Online; visited 2009-03-06].
URL <http://www.lothosoft.ch/thomas/libmip/markspaceparity.php>

[5] Richard, S. W.: *UNIX Network Programming: Networking APIs: Sockets and XTI*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997, ISBN 013490012X.

[6] Sweet, M. R.: Serial Programming Guide for POSIX Operating Systems. 1999, [Online; visited 2009-03-06].
URL <http://www.c-program.com/pdf/serialPort_Programming_c.pdf>

[7] Serial port. In *Wikipedia - The Free Encyclopedia*, WIKIMEDIA Foundation, March 2009, [Online; visited 2009-03-06].
URL <http://en.wikipedia.org/wiki/Serial_port>