Olympiads in Informatics, 2015, Vol. 9, 3–14
DOI: http://dx.doi.org/10.15388/ioi.2015.01

libinteractive: A Better Way to Write Interactive Tasks

Luis Héctor CHÁVEZ
omegaUp
Hacienda de Coaxamalucan 138, Col. Hda. de Echegaray Naucalpan
Estado de México, México CP 53300
e-mail: lhchavez@omegaup.com

Abstract. Interactive tasks are currently written as a set of language-dependent shims and libraries that are linked against the contestant's code to produce a single executable. This implies that task writers often need to generate three or four different libraries that need to be tested separately, for C/C++, Pascal and Java. Furthermore, the libraries must be written with care to avoid cheating, since it is possible for contestants to access the memory and opened files of the whole process. *libinteractive* solves these problems by defining an interface description language that is used to automatically generate shims in all IOI-approved languages in a way that is easily sandboxed; and a fast, portable interprocess communication mechanism that allows complete separation of the task writer and contestant code in different processes. This simplifies task creation and testing, making all tasks compatible with any future approved languages.

Keywords: interactive tasks, performance, omegaup, sandbox, security.

1. Introduction

Since its introduction in IOI 2010 until IOI 2014, all interactive tasks are distributed as a small package that contain a few files that the contestant can download to their machine, modify, compile, and validate their solution against a small set of provided inputs. The package contents are typically as follows:

- A source file created and tested by the task writer that reads task information from a file, interacts with the contestant's code through a series of well-established functions or procedures, and then either writes a text version of the contestant's response or a verdict of the solution.
- A header file that contains the function or procedure prototypes that can be included from the contestant's code.
- A template version of what the contestant is expected to implement.
- Optionally, some scripts that can be invoked to automatically compile, link, and execute the whole program and run it against some of the inputs. These scripts are usually written for the Unix shell `sh`.

All task code is language-dependent, so all previously created tasks have one version for each supported language, and they all need to be created manually by the task writer. Moreover, since the task's and contestant's code are both executed in the same process, there is no security guarantee whatsoever, and the programs must be coded defensively and obfuscated to prevent contestants from obtaining direct access to the input file or in-memory structures. The task files usually come with instructions to run and test the program, since the way they are created changes from contest to contest and is not defined anywhere or standardized. Some previous IOI tasks used some RPC mechanisms, like Regions from IOI 2009, but were done in an ad-hoc fashion. Given that the Microsoft Windows family of operating systems has between 88% (Statista, 2014) and 91% (NetMarketShare, 2014) of market share in desktop computers, the vast majority of students cannot easily test their solutions on the operating system they most likely have access to once the competition is over and they wish to train for the next one.

One solution to all the above problems, and the one implemented by *libinteractive*, is to run the code provided by the task writer and the contestant in separate processes that communicate through Remote Procedure Calls or RPC, which is a technique that makes executing code on a different process semantically similar or equivalent to calling a local function (Birrel, 1984). The separate processes can now be written in potentially different programming languages, allowing task writers to only provide a program in one language and allowing the remote procedure call machinery to perform translation on the fly. The rest of the ancillary code and scripts described above can be generated by a compiler by providing a formal description of the interface in an Interface Definition Language, or IDL (OMG, 1991).

libinteractive creates a standard, multi-platform, language agnostic, secure, and mainly transparent solution to describe, compile, run, and validate interactive tasks. This paper is structured in the following way: The second section briefly explores the state of the art in RPC libraries and automatic code generation through the *libinteractive* IDL. The third section describes the architecture and design of *libinteractive*, including a platform-specific optimization for Linux to significantly improve the performance of task execution. The fourth section explores the performance characteristics of the *libinteractive* RPC. The fifth section concludes with the results obtained so far, and points out further directions for future expansion.

2. RPC Mechanisms

The concept of RPC and automatic code generation has been around since the early 1980s, and since then several platforms have been created to serve different needs. They usually fall into one of the following two categories:

- Language/platform-specific RPC: most modern programming languages include an easy way to perform remote calls in a way that is syntactically equivalent to performing a regular function call. Java supports Remote Method Invocation, or RMI. Microsoft's C# and the rest of the languages supported by the Common Language Runtime allows for 9 different interprocess communication technolo-

gies, including COM. These solutions are well integrated into the language and platform they run on, but do require some extra code to be written, exceptions to be handled carefully by the consumer, and are not easy to consume outside of their respective languages or platforms.

- Service-oriented RPC: the main goal of these solutions is to very quickly define the interface of a service that then can be consumed through a network. The way this is done is by defining schemas in which objects are encoded into messages, which are passed around through the network. OMG's CORBA, Google's Protocol Buffers and GRPC, Facebook's (now Apache's) Thrift, and Apache's Avro are some popular service-oriented RPC platforms. All of these solutions are platform-neutral and have good performance on client/service architectures where the cost of constructing and transporting messages is negligible compared to the actual service they facilitate.

None of the solutions found had the right balance of transparency to the programmers, performance, and security. Language-specific RPCs had the best support for transparency and ease of use, but are not necessarily as performant as we wanted and were not easily portable to other environments. Some of the existing solutions being able to achieve a very low overhead for RPC calls and enabling throughputs up to several thousand messages per second, but required a multithreaded, fully asynchronous programming model, which does not work well for interactive tasks which are inherently synchronous. More highly performant platforms, like the LIMAX Disruptor even require a different programming paradigm to achieve their goals. *libinteractive* was created as a completely transparent, secure, and relatively performant RPC code generator and library.

3. Architecture and Design

libinteractive, much like any other RPC system, consists of three core components: an Interface Definition Language, a compiler that can convert IDL files into code and metadata, and the actual RPC mechanism used to communicate between processes and signal them. An optional component is provided to improve throughput when running on a Linux system: a kernel module that reduces the overhead of the RPC mechanism without compromising its security and the isolation between processes.

3.1. *The Libinteractive IDL*

The interface definition language chosen by *libinteractive* is based heavily on Web IDL (W3C, 2012), developed by the W3C as a way to express interfaces in JavaScript/ECMAScript and then later used by all web browser vendors in their own documentation. Its syntax resembles Java and allows for attributes to describe properties of various elements of the interface.

The building blocks of a *libinteractive* IDL file are the interface blocks, which describes what procedures or functions are implemented by which of the processes, which

are written just like interfaces in Java. The type system is similar to Java's but in order to better support multiple languages, there are a few restrictions on the types, which closely match the C language's semantics and limitations. There are six primitive types that can be used as parameter or return types: `bool, char, short, int, long`, and `float`. Procedures use the special return type `void`. Single- and multi-dimensional arrays of any of the primitive types can also be used for parameter types given that all their dimensions except possibly the first are compile-time constants. In the cases where an array dimension is variable, it must be passed in as a parameter that comes before in the parameter list, and it must have a Range attribute describing the lower and upper bounds of the value of that number in order to calculate the maximum size in bytes of the array. For instance, the task Parrots from IOI 2011 (Fakcharoenphol, 2011), can be described with the following IDL snippet:

```
interface Main {
    void send([Range(0, 65535)] int n);
    void output([Range(0, 255)] int n);
};

interface encoder {
    void encode([Range(0, 64)] int N, int[N] M);
};

interface decoder {
    void decode([Range(0, 64)] int N, [Range(0, 320)]
int L, int[L] X);
};
```

In the above example, for the `encoder.encode` procedure, it is known that the array `M` can have up to `N` elements, which in turn must be an integer between 0 and 64. This information is used to perform runtime parameter validation, as well as simplifying the protocol and memory management. By convention, the first interface in the IDL file is called `Main`, and it represents the program that the Task writer has created. The rest of the interfaces will be run in separate processes using the functions and procedures implemented by the contestant. This means that *libinteractive* supports isolating an arbitrary number of processes. `Main` is allowed to call the functions and procedures of the other interfaces, and the interfaces can call the functions and procedures of `Main`, but not any member of other interfaces. This is done to both simplify the design of *libinteractive* as well as to avoid cheating. The full syntax and semantics of the *libinteractive* IDL can be found on the project's documentation website[1].

3.2. *The libinteractive Compiler*

Once the task writer describes the contract between the main process and the one implemented by the contestant in the libinteractive IDL, the compiler can be used to generate

[1] `https://omegaup.com/libinteractive/`

all the files needed for contestants to compile, run, and test their solution against a set of predefined sample inputs. For a given platform and programming language, the compiler generates four sets of files:

1. A platform-dependent script that contains all the commands needed to compile and run the task: a Makefile, used in Linux/Mac OS X/Unix environments with the `make` command; or a .BAT file for Windows, that can be invoked directly as `run.bat`.
2. Language-dependent header files that can be included in both the task writer and contestant's programs, that expose the interface(s) that can be called.
3. Language-dependent utility functions that handle the serialization/deserialization of the parameters and handle all RPC invocation and signalling.
4. A platform-dependent run driver, written in C, which prepares the environment for the processes, executes them and prints out anything written to standard output/standard error. In Windows, the driver also orchestrates all the RPC communication between all processes.

Once the files are generated, they are packaged into a .zip file that contestants can download together with general instructions and examples.

One of the design goals of *libinteractive* is that it should be possible, if both the task writer and the contestant's source files are written in the same language, to compile them both into the same executable. This means that there is no additional syntax or unfamiliar semantics to be learned in order to write a *libinteractive* task or a solution for it.

The compiler is written in Scala, and is typically invoked as a standalone Java application, but it can also be used as a library. omegaUp (Chávez, 2014) leverages this and invokes the library to validate uploaded tasks, as well as generating all files that are to be consumed by contestants for all platforms and languages ahead of time.

3.3. *Execution Flow*

Programs compiled with libinteractive perform some initialization before handing control over to either the task writer's or the contestant's code. In this initialization phase, the RPC transports are created and prepared for communication, and then control is transferred to the normal program entrypoint (int main() in C/C++, public static void main(String[] args) in Java) for the Main process. Non-Main processes proceed to wait until Main interacts with them. Once execution reaches a point where a call to a function on another interface is made, all function parameters are serialized into a message in a transport-specific way, and is sent to the other process. The caller then waits until the callee acknowledges having finished execution of the call. The original caller then resumes execution and the callee goes back to waiting for a message. The acknowledgement itself is another message that might contain a return value, so all processes will be sending messages and waiting for a reply until one process terminates. If the process that terminates is Main, it is treated as a normal termination, otherwise it is an abnormal termination and the execution is treated as a failure.

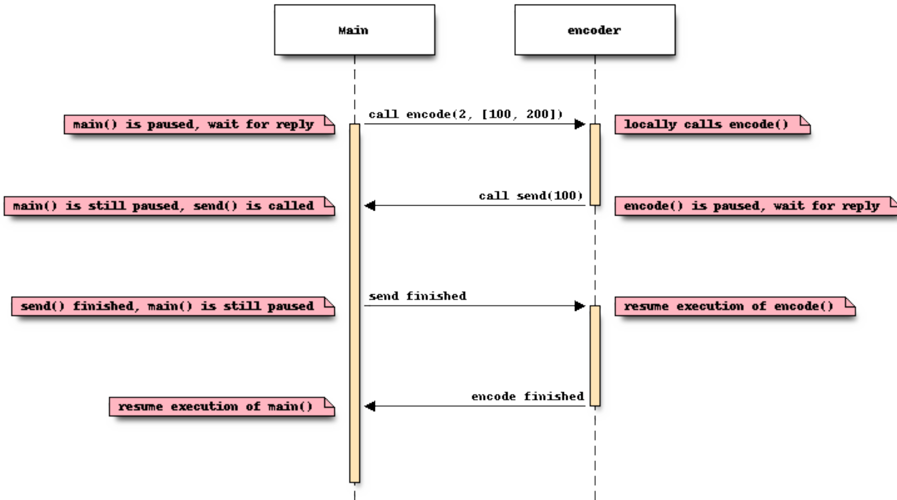

Fig. 1. Sequence diagram for message interaction in Parrots from IOI 2011.

One important thing to mention is that when a process is waiting for a message reply, a method call message can be received. This enables nested calls to be performed between the processes, as shown in Fig. 1.

3.4. *RPC Transports and Mechanisms*

The compiler can abstract away all of the lower-level details so that both task writers and contestants do not need to worry about exactly how the RPC calls are made, and in fact it is possible to choose from two available mechanisms: a cross-platform one that uses named pipes (Vaught, 1997) that contestants can use on their machines to test their solutions, and one designed for Linux based on shared memory (Stevens, 2003) that offers much lower roundtrip latency, but is not as portable and requires a Linux kernel module as described in the next subsection.

3.4.1. *Named Pipes*

Named pipes are available for all modern platforms, and they are simply streams of bytes that have endpoints in different processes. One process has the endpoint that can be written to and the other one reads from its endpoint. Pipes usually have blocking reads, which means that when one process reads from the pipe before it has any data in it, it will wait until the other process writes to it. This makes signalling easy, since each process is either doing computation or waiting for the other one make or return a call. The message encoding is also very simple, since the IDL mandates that all para-meters have a fixed size on compile time, or its size can be derived with only parameters that are already available: the binary representation of each parameter is written to the stream in a format compatible with C, which means that the memory of each parameter is copied directly

into the pipe one after the other. Arrays are serialized in row-major order. In order to distinguish between the different functions available to each interface, each function is assigned a unique, random 32-bit integer id during compilation, which is then prepended to each message before the parameter list. A 4-byte random cookie is appended to each message in the stream and then validated on the reply to avoid replay attacks. The actual data that goes through the pipe would then be similar to the one in Fig. 2.

Each interface pair has a pair of named pipes, one for outgoing messages and the other for incoming messages. Every time a message is received by one of the interfaces, it locally invokes the procedure or function and then sends the result of the function or a simple acknowledgement in case of procedures to the other pipe in the pair so that the caller can resume execution.

Named pipes do have one downside, that is shared amongst several of the RPC mechanisms outlined in the second section: they need to copy all the data into the pipe in one process, into the kernel, out of the kernel, and then out of the pipe for the other process. Typically this is done very fast for small messages shorter than 4 kb, but it becomes slower the larger the messages are. Also, since writes to a pipe are designed to not block unless the kernel buffer that receives the data from the pipe is full, the writing process will regain execution until it issues a read to the other pipe in the pair in order to wait for the response from the other process, which wastes a small amount of time.

3.4.2. *Shared Memory*

There is a second RPC mechanism available in all modern operating systems: shared memory. Two or more processes can request the operating system to allocate a flat memory location that can be accessed by all processes simultaneously[2]. Typically, shared memory is used together with a signalling mechanism that lets the other process know when it is safe to read from the shared memory without reading garbage, enforcing processes to take turns reading and writing the shared memory area. Given that the memory is never touched by the kernel, the cost of copying data around is greatly reduced. The overhead of the RPC call is now dominated by the cost of making a context switch between the processes, and can change depending on the signalling mechanism used. Semaphores and mutexes are the fastest ones available in typical Linux installs. One downside to using these synchronization primitives is that they are not easy to sandbox, since they require some extra system calls and access to a broader part of the filesystem

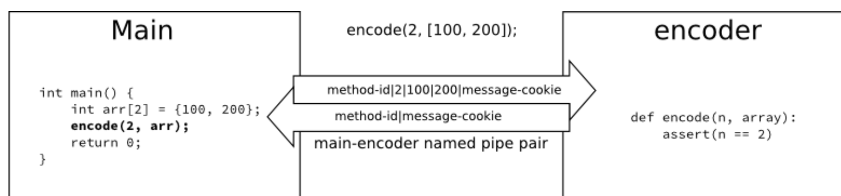

Fig. 2. Binary protocol for the named pipe backend.

[2] There are some caveats regarding cache coherence and consistency of the shared memory.

to be allowed in order to work properly. Programmers should also code very defensively when using them, since it is possible to deadlock if the process that currently holds the semaphore or mutex dies unexpectedly.

In *libinteractive*, since all messages have either fixed sizes, or contain range attributes to know the maximum size of any arrays passed as parameters, it is possible calculate an upper bound on message sizes on compile time. This information is used to implement a simple slab allocator (Bonwick, 1994) that allows for individual memory areas that are deallocated to be placed in a per-message-size linked list and quickly reused to avoid fragmentation. Since each interface handles its own shared memory, the number of message sizes has an upper bound on n, the number of functions and procedures in each interface. Given that most tasks are designed in a way that there is a constant amount of function call nesting across interfaces, at any given point in time there are at most $O(n)$ live messages and so the per-interface shared memory size is bounded. The generated code automatically manages all memory allocations needed, as well as validation of the input parameters and handling of all error conditions, so the whole process is transparent to that task writers and contestants.

Since the contestant's process also needs to modify the internal data structures of the allocator, all allocator calls are validated for consistency and the process aborts if it detects any modifications.

3.5. *transact Linux Kernel Module*

We must recognize that there is an unavoidable amount of overhead that is introduced by any RPC system. Most platforms deal with that through parallel processing, but interactive tasks are inherently serial and one process must wait for the other to respond before continuing. Most of this overhead comes from the context switching that the operating system must perform in order to stop running one thread/process and run another in a way that all processes are isolated from each other, but even that can be optimized: libinteractive also includes an optional Linux kernel module called transact that can provide significantly lower context switching cost in the scenario where there are exactly two processes switching back and forth from each other.

transact provides a fast, simple, file-based synchronization mechanism between exactly two processes that only uses the four most basic Linux system calls: `open` to acquire the lock, `write` to signal which of the two processes is the one owned by the task writer, `read` to make the context switch, and `close` to signal the other process that the current process is done with the lock and will shut down. Since the kernel manages the data structures that back the locks, it is possible to atomically force a context switch and yield control to the other process, reducing the overhead up to 20%. It is also resistant to deadlocks, since once a process is shut down, the kernel automatically closes all open files and will signal the other process that the other endpoint has died. By using transact and blocking thread creation at the sandbox level, it is possible to guarantee that there is exactly one process in each pair running at any point in time, so concurrent modification to the shared memory area is not possible.

Using transact is not a requirement for using *libinteractive*, but it improves performance significantly and allows to better measure the amount of time that the contestant's code is actually using to solve the task instead of being wasted in waiting for the kernel or serializing messages.

3.6. *Sandbox Compatibility*

libinteractive was designed with omegaUp's minijail sandbox in mind, which uses seccomp-bpf to do system call filtering, so it had to avoid using dangerous syscalls. It only relies on open, close, read, and write with named pipes transport, and additionally uses mmap when using transact. To also avoid having to relax filesystem sandboxing, a mode was added where all files that are to be shared among processes are all contained in a separate directory that can be mounted with read-write privileges (and additionally dev permissions with transact) in all sandboxed containers. This also makes it compatible with isolate (Mareš, 2012), the sandbox currently being used in IOI.

4. Performance Analysis

In a default, 64-bit install of Ubuntu Linux on a single-core virtual machine running on a AMD Opteron 4171 HE (used in some Windows Azure datacenters), we have measured that a well-written blocking RPC call has a wall-time context switch overhead of roughly 4.5–5.9 microseconds, depending on the RPC mechanism used to make the call. User time overhead (the CPU time spent executing the contestant's program exclusively) is much lower, on the order of around 0.5 microseconds. With the transact module, it is possible to lower both the user-time and wall-time overhead by 25%. Fig. 3 shows both user- and wall-time overheads for different message sizes.

Running interactive programs in multi-core machines is not recommended with *libinteractive*, since it makes the wall-time overhead significantly larger: over 10x in AMD processors. The operating system needs to ensure memory coherence between all processor cores and caches, so in the case where execution is transferred from one core to another and there is a data dependency between them, an additional synchronization step must be performed that further increases latency. Intel processors are also affected, but not as dramatically. If multi-core machines are to be used, it is possible to force all processes to run in the same CPU core by forcing their affinity, giving the same results as single-core environments.

Regarding variability of measurements, for small messages (<100 bytes), the user time overhead can vary up to ±0.4 microseconds per call in the worst case when both contestant and problemsetter processes perform negligible amounts of processing and the RPC costs dominate. The use of transact makes both the overhead and variability of measurements lower. Fig.4 shows a boxplot with the distribution of user-times for a 16-byte message.

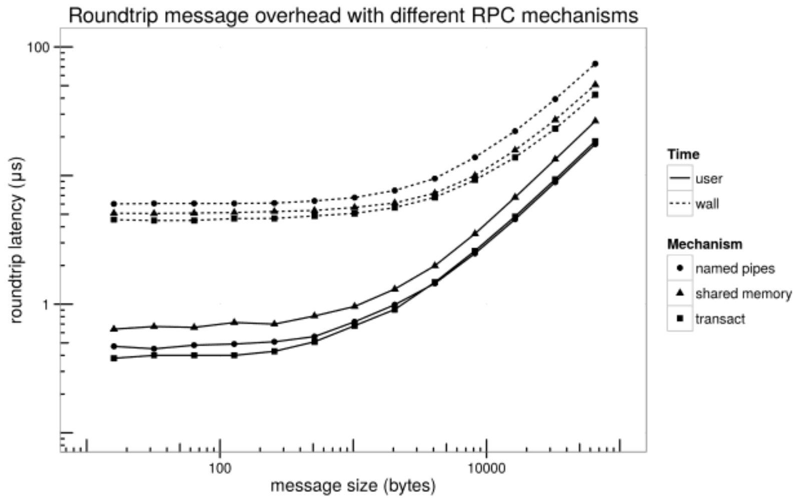

Fig. 3. User- and wall-time overheads for different message sizes.

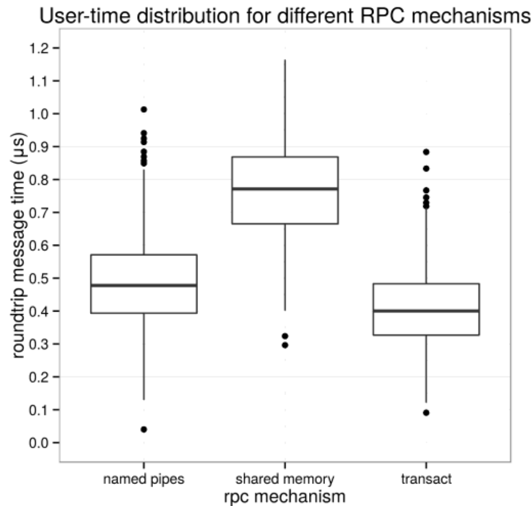

Fig. 4. User-time distribution.

One example of a good task for *libinteractive* is IOI 2013's Cave (Pouly, 2013), which has a limit of 70,000 messages per case and both contestant and problemsetter processes do non-trivial computation between each message. Both source code files were unmodified and were compiled in both a single binary and as *libinteractive* programs with the transact signalling mechanism. The single binary using the official solution finished in 23.91 s, with a user-time of 23.29 s. The *libinteractive* binaries finished in 38.19 s (+59.72%), with a user-time for the contestant process only of 20.14 s (-13.53%).

Despite the wall-time overhead, the user-time measurement was lower since all the processing related to reading the input file and performing validation was not accounted for. In general, tasks where the problemsetter code needs to do significant processing tended to fare better when run under *libinteractive*, whereas tasks with higher number of roundtrip calls tended to fare worse.

5. Conclusion

libinteractive is an excellent option to write interactive tasks that do not require a huge amount of roundtrips since it only requires a single source file in one language to be able to interact with the contestant's code. It is also platform-independent which allows contestants to practice writing solutions in the operating system they have access to. It was also designed to be sandbox friendly, and is compatible with both minijail and isolate. All code has been released through GitHub under the BSD license (except the transact kernel module which has a GPL license to match the Linux kernel's license)[3].

There are still a few things that we would like to do to improve the user friendliness of *libinteractive*, like IDE integration. Finding ways to further reduce the RPC overhead, especially on Intel processors, and supporting more data types like strings and structs/records are also high on the list.

Acknowledgments

The rest of the omegaUp development team and volunteers, especially Ethan Jiménez for his feedback during beta testing.

References

Birrel, A.D., Nelson, B.J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1).

Bonwick, J. (1994). The slab allocator: an object-caching kernel memory allocator. *In USENIX Summer*, 1994, 87–98.

Chávez, L.H., González, A., Ponce, J. (2014). omegaUp: cloud-based contest management system and training platform in the Mexican olympiad in informatics. *Olympiads in Informatics*, 8, 169–178.

Fakcharoenphol, J. (2011). Parrots. *The 23rd International Olympiad in Informatics*.

Mareš, M., Blackham, B. (2012). A new contest sandbox. *Olympiads in Informatics*, 6, 100–109.

NetMarketShare – Desktop Operating System Market Share (2014).
http://www.netmarketshare.com/operating-system-market-share.aspx?qprid

[3] https://github.com/omegaup/

=10&qpcustomd=0&qpsp=2014&qpnp=2&qptimeframe=Y
Statista – Global market share held by operating systems Desktop PCs from January 2012 to December 2014 (2014). http://www.statista.com/statistics/218089/global-market-share-of-windows-7/
Stevens, R. (2003). *UNIX Network Programming*, Vol. 2, Second Edition: Interprocess Communications. Prentice Hall, 303–323.
Object Management Group (1991). The Common Object Request Broker: Architecture and Specification.
Pouly, A., Charguéraud, A. (2013). Caves. *The 25th International Olympiad in Informatics.*
Vaught, A. (1997). Introduction to named pipes. *Linux Journal*, #41.
W3C – Web IDL (2012). http://www.w3.org/TR/WebIDL/

L.H. Chávez is an ACM-ICPC world finalist (2010) and has a bachelor's degree in computer science (2011) from Tecnológico de Monterrey, Campus Querétaro. He has been involved in several efforts to improve the state of programming contests in Mexico since 2007, and is one of the co-founders of omegaUp. He is currently employed at Google in the Chrome team and is also studying towards a MSc in computer science from Stanford.