# Posing Interactive Contest Problems in the ICPC Scoring Model

Ed Skochinski
edsko@sbcglobal.net

Jefery Roberts
jroberts7@socal.rr.com

Marc Furon
marcf@dslextreme.com

ACM ICPC Southern California Region
P. O. Box 8634
Calabasas, California 91372-8634

## ABSTRACT

The Southern California region has successfully posed interactive contest problems and fit them into the existing ICPC scoring model. The problems are interactive in that they require the contestant solution to interact with a judge-provided "server" program that responds to the output produced by the contestant's program. This paper discusses the server implementation, the scoring model, and the development of interactive contest problems.

All interaction between the contestant "client" and judge "server" programs is implemented via standard input and output over named pipes. Contestants are provided with the executable server program (without the judges' configuration file) to use for development, along with a simple protocol definition and code samples for buffer flushing. The responses produced by the server vary based on the output previously produced by the contestant's program. Standard ICPC judging responses are mapped to possible contestant program behaviors. The server must be prepared to handle anything a contestant program might do. We describe a Linux-based C language routine that implements I/O blocking, mandatory buffer flushing, and timeout detection in a secure and robust manner.

The region has posed a backtracking problem presented as a maze and a deductive logic puzzle as a human-vs-computer "Clue" (Cluedo) game. Each of these problems was solved by several teams and was well received in post-contest surveys. We offer guidance in selecting and specifying interactive problems based on our past experience.

## Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

## Keywords

ACM ICPC, programming contest, problem pedagogy, interactive problems, problem development

## 1. INTRODUCTION

Typical ACM International Collegiate Programming Contest problems involve unidirectional data flow in a batch model: one or more test cases are provided in a text input stream that is fed to a program that performs some computations and produces a text output stream. The output from a program submitted for judging is either compared with the expected output or otherwise evaluated for correctness. The result of the evaluation is returned to the submitter as one of several standard messages.[1] This model supports many problems that use a variety of algorithms—but still limits the problem scope to those that can fit the batch model.

There has been interest in expanding the contest to cover additional problem types beyond the batch input/batch output model for many years. The Visual Challenge, Java Challenge, and Parallel (Java) Challenge were conducted at recent ICPC Finals to experiment with problems that involve direct (visual) user interaction or a tournament model. Tournament problems require a team's program to compete directly with programs from other teams in a given environment, interacting with that environment via a provided API.

One issue that arises with these types of problems is how to score them within the ICPC. Several informal proposals have been made, from counting visual problems as regular problems but awarding credit minutes to solutions that go beyond the minimum requirements, to a two-round model where visual problem or tournament performance is a preliminary round that counts a to-be-determined percentage of a final ranking.

The Southern California region has developed a different class of problems that require the contestant program to interact with a judge-supplied server program. This allowed the region to pose problems that require strategy design and implementation or dynamic adaptation, which are not easily tested with unidirectional data flow. By exploiting interprocess communications, these classes of problems can be asked—expanding the scope of potential problems, and doing so in a way that can be fit into the existing ICPC scoring model with minimal impact. Two such problems have been

posed: a maze traversal in 2004, and a "Clue"[1] player in 2006.[2]

Such interactive problems involve writing not only a problem description and judges' solution, but also the server program to converse with the client. Within the problem description, contestants are given instructions how to mimic the server manually (test their client without the presence of a server), how to invoke the server processes manually (testing their intuitive understanding of the problem), and finally how to pit their client program against the actual server process. The client-server interaction takes place by using standard input and output over Linux named pipes—making the process language independent and easy for contestants to implement and test.

In the following sections we discuss the actual interprocess communication implementation, the techniques used to make sure the judges' server process is robust in the event a contestant program exhibits unexpected behavior, considerations for developing interactive problems, the integration of interactive problems into the ICPC scoring model, and our experiences with interactive problems actually posed in previous Southern California regional contests.

## 2. INTERACTIVE PROBLEM IMPLEMENTATION

### 2.1 The use of named pipes

The Southern California region uses standard input to communicate judges' test cases to contestant programs, and those programs produce output for evaluation on standard output. This custom dates as far back as the late 1970's, from the early mainframe contests, to those held on minicomputers, and finally PC-based contests. Through FORTRAN, Pascal, C, C++, and Java, this line-oriented paradigm persists today on Linux platforms. Contestants might additionally have to read from a named file, but such a file normally contains data other than the test cases.

The choice to use named pipes was a natural progression. Standard input and output work nearly identically for pipes, terminal I/O, and files. Named pipes do not suffer the security and network configuration concerns of socket daemons, and they offer language independence over APIs. The setup and invocation of named pipes are conceptually easy; the contestant can execute the individual programs as standalone line-oriented applications or as a true client-server pair. The additional programming burden for using named pipes is nothing more than simple buffer flushing.

### 2.2 Implementing named pipes

Linux named pipes are unidirectional, and two pipes are required to form bidirectional communication. There is a happy coincidence of using standard input and output; they are inherently pipable. With a clever use of piping, only one named pipe is necessary to complete the feedback loop. We chose to pipe the output of the client to the input of the server, leaving the named pipe to convey server output back to the client.

[1]"Clue" is a trademark of Hasbro, Inc.

Because the reader of a Linux pipe blocks until data is available, choosing the initiating process is crucial to prevent deadlock. To mimic classic client-server communication, the server was assigned the passive role, blocking on the read of its own standard input. This is the first significant difference between interactive and unidirectional problems. The contestant's client must start the dialog by writing to standard output. This works well because the contestant's program must typically query the problem configuration. For both client and server, standard input and output are relegated to conversation only. The configuration (the actual test case) moves to a supplementary file passed as an argument to the server. The general form of a client-server dialog is:

```
mkfifo server_to_client
client < server_to_client | \
   server configuration_file > server_to_client
```

In our environment, contestant C and C++ programs compile to the default executable `a.out`. Executing `a.out` is as simple as specifying a qualified path to the linked file. The following command sequence is very close to the actual script used in production judging:

```
mkfifo server_to_client
./a.out < server_to_client | \
   /path/to/server configuration_file \
   transcript_file > server_to_client
```

### 2.3 Java implementation considerations

Java generally produces a class file matching the name of each non-nested class definition. Furthermore, a monolithic piece of Java source code can produce many individual .class files. Even worse, when a class is declared as public, its source file name must match its class name. Rather than having to determine which class file contains a *main()* method, we restrict the syntax of contestants' submissions:

- No public classes.
- The class containing method *main()* must be named `aout`. Do not name the source file `aout.java`.

Once the restrictions are observed, we can take pseudorandom file names and compile to a working directory. We know that the class containing the *main()* method is `aout.class`. Invoking the Java client requires:

```
mkfifo server_to_client
java -cp . aout < server_to_client | \
   /path/to/server configuration_file \
   transcript_file > server_to_client
```

The Eclipse IDE inserts "public" where it deems appropriate. The contest environment provides a compilation script that the contestants can use to build their executable just as the judges do. This script randomizes the source file name to expose the presence of public classes and enforce the file name restriction.

**Table 1: Buffer flush code samples**

| | |
|---|---|
| C | `fputs("1\n",stdout); fflush(stdout);` |
| C++ | `cout << "1\n" << flush;` |
| Java | `System.out.println("1");`<br>`System.out.flush();` |

## 2.4 Presenting the problem to the contestant

Even without interactive problems, we describe the line-oriented nature of our execution environment to the contestants prior to the warmup problem. This familiarizes the contestants with the concepts of standard input and output, I/O redirection, and in particular, how to compile a program the same way the judges do. When interactive problems are present in the problem set, contestants are informed prior to the warmup, with additional instruction provided. The problem writeups themselves are explicitly identified as interactive programs.

Perhaps the biggest area of concern is buffer flushing. Typical terminal I/O flushes buffers upon issuing an end-of-line sequence. Named pipes behave more like traditional files, flushing buffers only on end-of-file or buffer full. At the pre-contest environment discussion, after the warmup problem, and within the problem description, example code is presented for each language. This explicit buffer flushing permits the back-and-forth dialog to proceed. Table 1 shows the code samples used in the maze problem.

Within the problem description, it is important to mention that the client process initiates the dialog. Basing one half of the conversation on standard input and output lets the client or server operate in isolation. This behavior is documented in the problem description as development or debugging tips. For example, a contestant could execute the compiled client program

```
./a.out
```

or

```
java -cp . aout
```

The client program initiates the dialog, and the user responds by typing server responses at the terminal. This invocation is useful for presenting ad hoc test cases to the client.

### 2.4.1 Contestant problem testing

We supply scripts for running the server process with and without the contestant's client. In our case, the command is test$n$, where $n$ is the problem number. The test$n$ command has two mandatory arguments and one optional argument. The two-argument form specifies a configuration file and a transcript file (more on the transcript file later):

```
testn configuration_file transcript_file
```

For the two-argument case, the contestant is expected to play the part of the client. The problem description includes instructions on how to obtain a sample configuration file for use by the contestants. When useful, the description contains a transcript based upon the sample configuration. The script essentially invokes the following for the user:

```
/path/to/server -i configuration_file \
transcript_file
```

By our second interactive problem, we added an interactive option, -i, to the server. This option produces a prompt to signal the user whose turn it is to talk. This provides the judges with the ability to converse directly with the server using a terminal shell during problem development, without having the server terminate abruptly in response to typographic errors. The -i option disables client timeouts. It also relaxes the response to protocol violations when executed as a stand-alone program. The server indicates a protocol error, ignores the input, and waits for the user to reissue the line.

When the test$n$ script is used with three arguments, it takes the form:

```
testn configuration_file transcript_file \
source_file
```

The source file name contains a language-specific suffix (.c, .C, .java) to assist the script in properly executing the client. The source code is compiled, and the mkfifo and language-appropriate client invocations are executed. The three-argument case has no human interaction; it pits an automated client against an automated server. The server produces a formatted transcript; it echoes the client input (errors and all) and juxtaposes that with the server responses. The transcript file is the only account of what transpired between client and server.

## 2.5 Designing the protocol

When describing an interactive problem, a formal protocol must be defined. The protocol should be simple enough to be implemented easily within the confines of a time-limited competition. The following general guidelines have been observed in the Southern California region with success.

*Use a terse, line-oriented, character protocol.* When executing the client or server in isolation, a user is more likely to prefer line-oriented input and output. The simple ability to backspace and correct typographical errors prevents user frustration. Terse commands keep the typing to a minimum. When client and server are executed together, the server produces a transcript file, which itself is human readable. Because the protocol has not been the focus of the problem, text interaction and human readability are valuable for understanding and debugging.

*Keep the protocol simple.* There is programming overhead associated with interactive problems, and the whole point is to expand the problem space. A simple protocol makes processing the dialog easier; the problem can focus on the peculiarities of interaction rather than debugging the protocol. Writing the server process was much easier without the burden of complicated I/O.

# 3. IMPLEMENTING THE SERVER

Interactive problems expand the problem space, but they come at a cost. In addition to writing a description, producing thorough test cases and coding a solution, the judges must write a server program. Producing a quality server roughly doubled the problem development effort.

The configuration itself incurred some lengthy and tedious coding. The server process reads the configuration file and the client elicits information by dialog. However, the server is exposed to the contestants, who are expected to formulate their own test cases. If incorrectly formatted or ill-conditioned configuration data is tolerated, it could result in misleading behavior or even abort the server. A significant part of our server code reads the configuration and diagnoses errors.

Despite the usefulness of running the client or server in isolation, the contentants will eventually need to pair their programs with the server. To record the interaction, the transcript file provides a record of client input vs. server response. The transcript is a simple line-oriented text file with client input aligned at beginning-of-line. Server responses have horizontal tabs preceding the response. Pattern matching tools such as *grep* and *awk* can be used easily to separate the individual components of the dialog. During problem development, the judges may alter (that is, fix) the server behavior. Client output extracted from a useful dialog transcript can be fed back to the server to verify correct operation:

```
grep '^[^\t]' old_transcript_file \
   > client_to_server
/path/to/server -i configuration_file \
   new_transcript_file < client_to_server
```

A useful behavior of our interactive problems is almost fully automated judging. Unless the client aborts outright, it is the server, not a human judge, that decides which standard response is issued. This has two important implications: care must be taken in specifying the relationship between client bevahior and judgments, and the server must survive any client behavior.

## 3.1 Mapping client behavior to standard ICPC judging responses

Even with only two interactive problems posed thus far, it has become apparent that mapping client behavior to standard judged responses may vary according to the problem being asked. For the maze problem, we mapped most diagnostics peculiar to interaction, especially protocol violations, to RUNTIME ERROR. This gave judges the freedom to include explanatory text to accompany the response. Spurious client output after escape from the maze was diagnosed as PRESENTATION ERROR. WRONG ANSWER was reserved for failing to escape from the maze within the transaction limit.

For the Clue game, it was already clear that the protocol specification had to be very exacting. Observing that there would be no human participation in determining the judges' response, a traditional output formatting error that would elicit a PRESENTATION ERROR could not occur. This freed up PRESENTATION ERROR to represent dialog protocol violations. Winning the game of Clue requires a player to claim a deduced piece of information. When the claim was wrong, the judgment was WRONG ANSWER.

Within the scripted shell environment of the Southern California region, the server ended up issuing the standard response as a single line to standard error. The response was forwarded to the users. This is our current mapping of standard judging responses:

*SYNTAX ERROR.* No change; this is a compilation error. Early during the development of the maze problem, it was tempting to use SYNTAX ERROR to denote a protocol violation. Because it was necessary to specify and compile the source file to determine how to invoke the client (`java aout` vs. `a.out`), SYNTAX ERROR could not be overloaded meaningfully.

*RUNTIME ERROR.* No change; this is a runtime abort from the client program. A client runtime error will likely cause the client-to-server pipe to close. This may cause the server to diagnose a PRESENTATION ERROR. Issuing the appropriate judgment involves surveying the execution log for system-generated error messages that would supersede the server's diagnosis.

*TIME LIMIT EXCEEDED.* This message becomes overloaded. It still represents traditional runaway CPU. There are two more conditions that it represents: I/O response timeouts (see "Battle hardening the server") and exceeded limits on the number of client-server transactions.

The server prevents deadlock by waiting for a configurable amount of wall-clock time to elapse before giving up on the client process. Timeout problems are generally caused either by runaway CPU or the client's confusion about whose turn it is to talk.

A less obvious use of this message is to identify that a submission has exceeded the transaction limit specified in the problem description. Transaction limits prevent brute force or aimless solutions. For the maze, the transaction limit was expressed as a number of movement attempts and was based upon the size of the maze. Clue had a fixed limit of queries, 43 to be exact. This seemingly arbitrary limit was based upon the judges' test cases, much the same way a combinatoric problem may be limited in its sample space.

In our experience, exceeding the transaction count has been the most likely cause of TIME LIMIT EXCEEDED messages. Contestants already have access to the production server for resolving handshaking problems. Our interactive problems posed so far have not been computationally intensive. Receiving TIME LIMIT EXCEEDED for an interactive problem probably means that the client code is not making efficient queries from the responses returned by the server.

*WRONG ANSWER.* For a problem that must make some claim or assertion, an incorrect claim should be judged as WRONG ANSWER. When success depends simply upon achieving a desired state, failure to obtain that state within specified limits might be judged better as TIME LIMIT EX-

CEEDED.

*PRESENTATION ERROR.* Due to the automated judging of the client-server dialog, the traditional "formatting error" indicated by a presentation error has no meaning. This judgment is free for reassignment as a response to protocol errors. See SYNTAX ERROR for further reasons why this is the appropriate message.

*ACCEPTED.* No change; this means that the problem was solved. However, the server must make one last read of standard input, detecting end-of-file before issuing an ACCEPTED judgment. Any spurious client-to-server communication results in PRESENTATION ERROR.

## 3.2 Battle hardening the server

To ensure the full and fair judging of a contestant's client program, the server must not abort of its own accord. The Southern California judging environment has the ability to perform multiple executions of the contestants' programs, one per test case if desired. Each time a client is executed, a brand new server is started—this is necessary given the invocation method detailed above. A one-shot server does not need the zealous attention to memory management and state re-initialization that a persistent server requires. That being said, the server must still remain viable up to the transaction limit.

The introduction of user input to a judge-supplied program dictates a secure, constrained buffer input routine. Despite a line-oriented protocol, the routine must still react to run-on, improperly terminated, or nonexistent input. We developed a C function *ubgets()* that satisfies each of these requirements.

The *ubgets()* function is modeled on the standard *fgets()* function in that it does buffer length checking, always NUL-terminates the string in the buffer, and indicates a complete line by including the terminating newline in the buffer. Unlike the *fgets()* function, it uses unbuffered I/O, always reads from standard input, returns the length of the string read instead of a pointer to the string, and takes a timeout value.

The *ubgets()* function takes three parameters: a pointer to a character buffer, the length of the buffer, and a timeout in milliseconds. A timeout of 0 reads whatever is ready to be input without waiting. A negative timeout is used to disable timeouts and wait as long as it takes to get input. The function returns the number of bytes read into the buffer (not counting the terminating NUL), or $-1$ if there was a disconnect or end-of-file.

Because Linux does not provide a read function with a built-in timeout, the heart of *ubgets()* is a call to the system *poll()* function. The *poll()* function waits for I/O, subject to a timeout, but does not do any I/O itself. If *poll()* times out, *ubgets()* returns whatever it has already copied into the character buffer. If *poll()* indicates that input is waiting, *ubgets()* reads up to the end of the buffer (less one byte for the NUL termination). If the last character read is a newline, *ubgets()* adds a NUL termination and returns the length of the string in the buffer. If the last character is not a newline and the buffer is not full, *ubgets()* adjusts its pointers and counters for the next read and updates the timeout, since the partial input might have taken part of the time. It then goes back to *poll()* again. If the buffer is full, *ubgets()* NUL-terminates it and returns the length of the string in the buffer.

The *ubgets()* function robustly hides the messiness of input with timeout while providing all of the information the server needs to determine the state of the input. If *ubgets()* returns $-1$, the client closed the connection or exited. If it returns a 0, the input timed out without reading anything. If it returns a positive number but the last character in the string is not a newline, the input timed out, unless the number is one less than the length of the buffer, which indicates that the input is longer than the buffer. If the last character in the string is a newline, a line was read successfully.

The *ubgets()* function protects the server from malformed client input. It is a low-level routine; however, all protocol requirements must still be enforced by the server.

## 3.3 Exposing the server to the user

Providing contestants access to the server and a sample configuration file is akin to providing sample input for unidirectional problems. It allows the contestants to understand the I/O specification and the nature of the problem being posed. Just as sample input is not intended to be an all-inclusive set of test cases, the sample configuration file should be built with similar consideration. A sample transcript behaves much like sample input.

We recommend that interactive problem authors choose a sample dialog that clarifies the types of transactions, but does not offer up the cleverest set of queries possible. Leave the work of developing relevant test cases to the contestants. In the words of the regional judges, "This is a programming contest, not a coding contest."

A reasoning server (one that "fights back" against the client) displays some intelligence and as a side effect exposes the judges' insight into the problem. Maze was not intelligent; its choice as our first offering was influenced equally by its familiar concept and the ease of implementing the server. It was a neutral responder, returning information about the maze geometry move by move. The Clue server responded on behalf of the "other players" in the game, and had to be defensive enough to force the client to adapt and formulate insightful queries. One particularly intelligent behavior was that when given the option, the server would respond in a fashion that revealed no new information, just like a skilled human Clue player might do. Displaying too much intelligence might allow contestants to deduce what is in the judges' test cases.

# 4. INTERACTIVE PROBLEM SELECTION

Online algorithms make a natural choice for interactive problems. Often heuristics are used, making an optimal solution infeasible. We have found that a looser, pass-fail criterion works better and is easier to judge. Limiting the number of transactions helps to impose a mimimum level of sophistication on the client. If an optimal solution is required, we

contend that a traditional unidirectional problem is superior; any strategies quickly resolve to standard algorithms.

Game playing is another good class of problems. The server can enforce the rules of the game impartially and still impose the resource limitations of the contest. Thus far, program acceptance has been judged by the attainment of a certain state (escape from a maze) or the assertion of deduced facts. Although we have not yet required true competitive defeat of an opponent, we would likely limit it to client vs. server ("human" vs. computer). We have not yet formulated team-versus-team problems within the interactive problem space because victory over an opponent is a different concept than acceptance of a solution (but see JAST below).

One problem type that is actively under consideration in our region is simulation. The server imposes the "physics" of a mathematical model, and the contestants interact in the world of the model. In a sense, the maze could be classified as a simulation.

## 5. RELATED WORK

Muranushi et. al. have developed JAST (JAva and friends' Simulation Tournament).[3] JAST implements an infrastructure for a tournament environment, where the competing programs communicate with the tournament executive using text standard input and output over named pipes. The contestants are therefore not constrained by any particular programming language or API—unlike the Java Challenge, the contestant program can be written in any language offered by the environment. JAST deals with the same robustness and security concerns as our server does, the primary difference being that JAST must make sure that adverse behavior of one contestant program does not affect either the JAST executive or the other contestant programs running in the tournament.

## 6. CONCLUSION

Interactive programs present both practical implementation issues as well as user interface considerations. The client-server model chosen expanded the possible problem space, but it doubled the effort required to pose such a problem. The code to ensure a secure, robust server constituted the majority of the judges' code; the actual responsive part of the program was relatively straightforward. The combination of Linux and C calls limited our server implementation to C or C++, but did not restrict the contestants' choice of programming languages.

Named pipes were chosen as the interprocess communication method for ease of implementation and programmer impact. The only extra code required by a contestant for an interactive dialog was explicit buffer flushing. The use of standard input and output for communication allowed both client and server to execute in isolation—a valuable tool for debugging.

We chose very simple line-oriented text protocols to reduce the complexity of I/O and to focus on the interactive nature of the problems. An exact protocol specification let us code the server process to act as a judge, but this required explicit mapping of client behavior to the standard ICPC judging responses.

The Southern California region expects to pose more interactive problems, such as game playing and simulation, favoring problems that do not require optimal algorithms or solutions. If an optimal solution is required, we contend that a traditional unidirectional problem is superior because any programming strategies quickly resolve to standard algorithms.

In post-contest reviews, the interactive problems have been received well by those teams that solved them. While the judges did not consider the interactive problems to be the hardest ones in the set, solutions were few. Ten teams out of 65 solved the maze problem of 2004; five other teams submitted unsuccessful attempts. Only five teams out of 68 solved the Clue game posed in the 2006 contest; six other teams submitted unsuccessful attempts.

## 7. REFERENCES


[1] ACM ICPC. ACM ICPC World Finals Rules. `http://icpc.baylor.edu/icpc/finals/About.htm`, June 2007.

[2] ACM ICPC Southern California Region. Past Problem Sets. `http://www.socalcontest.org`, November 2007.

[3] T. Muranushi et. al. JAST Users Manual. ACM ICPC Competitive Learning Symposium, March 2007.