Refefree

Protocol Version 1

Wolfgang Hess

July 7, 2005

Contents

Thanks to Markus Moll, Sebastian Kanthak and Martin Girschick for their helpful comments on this protocol.

1 Bitstream Protocol

1.1 Sockets

Communication between the server and its clients is done via *TCP sockets*. The server SHOULD support *SSL* connections. This can be done by listening to an arbitrary *local* port and using the `stunnel` program to implement *SSL*.

1.2 Server Ports

The server MUST be configurable to listen to the *default port* 27251, accepting at least logins from *contestants*. The server SHOULD be configurable to listen to *multiple ports*, of which some may accept *only judges*, *only contestants* or both. A client MUST be configurable to connect to any port. Both clients and servers SHOULD use the *default port* if the user does not specify otherwise. If separate ports are used for contestant and judge connections, port 27252 is recommended for accepting the judges.

1.3 Communication Blocks

Communication is done a *message* at a time. Each message is contained in its own *block*, which has a *header* containing the length of the data section followed by the data itself. The length is encoded in 10 bytes as a string, i. e. as a left-aligned decimal integer padded with spaces representing a 32-bit value. This is followed by as many bytes of data as stated in the header.

size	*data*	*size*	*data*	*size*	*data* ...

Each block contains the data as *text* in one or more *lines* which are terminated by a single LF, i. e. by a value `0x0a`. To allow some `telnet` clients to connect, a server SHOULD strip most received data from CRs before further processing, i. e. characters with a value of `0x0c` should be removed. Received source code is an exception to this rule. Most lines are encoded in *ASCII* unless specified otherwise. Lines MUST NOT contain any control characters. If *flags* are written in a single line, flags are seperated by space, i. e. by values `0x20`. Flags only contain ASCII characters, and they SHOULD be seperated by adding a single space as a suffix to every flag, including the last one. The first line contains the *protocol code* for this block. Identification lines, e. g. login names and passwords, are encoded in *UTF-8*.

1.4 Handling Exceptions

Receiving *protocol codes* or *flags* that are unknown to the implementation is not an error, unless specified otherwise. Whenever this case occurs, unknown parts MUST simply be silently ignored. This is used to enable extensions of the protocol. However, a message that is understood but malformed is an error.

Currently until the client has successfully logged in, message sent from the client other than `login_request` are considered an error. Servers that do accept other messages at that stage MUST advertise this fact in the *server flags* as will be specified then.

If an error is detected the connection is simply closed. The host closing the connection SHOULD send an *error message* just before the connection is closed if it is possible. The format of this message is the *protocol code* `error` on the first line, followed by an error text on the next line or lines.

1.5 Synchronous Communication

Communication is done in order, i. e. requests MUST be answered in the same order as they are received. This makes it possible to synchronize a connection by simply sending a *heartbeat request*, because at the time the reply is received, all requests made before must have been answered completely.

1.6 Entities

Throughout this protocol description several *entities* are considered. *Contestants* and *judges* are identified by their login name and both are referred to as *clients*. They all are *persistent* independent of specific network connections.

2 Server Protocol

A server MUST support three states `before`, `running` and `after`. `before` is the state before the start of the contest, clients may connect but submissions are not accepted. `running` is the state once the contest is started. In this state the clock advances until the contest is over and the state is changed to `after`. In the state after the contest logins are not accepted anymore and no submissions may be made.

A server SHOULD support a fourth state `paused` which is the same as `running` except that the clock is stopped.

The server MUST support some method to start a contest.

2.1 The First Message

After a client has established a new connection the server sends the first message and then waits for the client to login or close the connection. The message includes four lines:

1. the protocol code `hello`
2. the server identification, e. g. `Overlord Server 1.0` for the first version of the reference implementation, *UTF-8*
3. the identification of the contest; this is a completely arbitrary identification string, a client SHOULD show this string to the user, *UTF-8*
4. the last line contains the *server flags*

2.2 Server Flags

Currently these *server flags* are defined:

`contestants` the server accepts logins of contestants at this port.

`judges` the server accepts logins of judges at this port.

2.3 Login Request

After the connection to the server has been established the client has to decide whether to login or not. If the login fails, the server closes the connection, if it succeeds the server sends a welcome message. The *login message* is quite simple and contains four lines:

1. the protocol code `login_request`
2. the *login flags*, `contestant` if contestant logins, `judge` if judge logins are supported
3. the login name; note that login names of all clients, i. e. judges and contestants, have a single common namespace, *UTF-8*
4. the password, *UTF-8*

2.4 A Warm Welcome

Logins that pass the password authentication are welcomed by the server by a welcome message after which the login procedure has ended and normal communication starts. The *welcome message* has these three lines:

1. the protocol code `login_welcome`
2. the account name, an arbitrary string naming the contestant, e. g. for use in a scoreboard, *UTF-8*
3. the *connection flags*

2.5 Connection Flags

The following *connection flags* are defined:

`contestant` the server has accepted the connected client as a contestant.

`judge` the server has accepted the connected client as a judge.

`notifies` the server supports *submission notification* and will notify this client.

`status` the server includes some of its status in heartbeat answers

`autoconfig` the server supports the protocol extension for *automatic client configuration*.

`scoreboard` the server supports the extension for showing a *scoreboard*.

`clarifications` the server supports the extension for *clarification requests*.

`pause` the judge client can use the *pause toggle* message.

`serverstatus` the judge client can use the *server status* extension.

2.6 Heartbeats

To enable clients and servers to *synchronize* connections and to test whether a connection is *alive*, the *heartbeat protocol* MUST be implemented by all clients and servers.

After a server has sent the *welcome message* it CAN sent a *heartbeat request* over the corresponding connection at any point in time. Likewise

a client CAN send *heartbeat requests* after the *login message*. A heartbeat request contains a single line with the protocol code `heartbeat_request`.

A heartbeat request MUST be answered immediately by a *heartbeat reply* message. This message contains the protocol code `heartbeat_whoomp` in a single line, or a status message using the same protocol code if the server announced this.

2.7 Server Status

The current state of the server and the elapsed time in minutes can be queried if the server included the `status` flag in the connection flags.

In this case the server MUST answers to *heartbeat requests* with its status in a message that contains three lines:

1. the protocol code `heartbeat_whoomp`
2. the current state, one of `before`, `running`, `paused`, `after`
3. the number of elapsed minutes
4. the duration of the contest in minutes

While all clients can get the above mentioned status, the *server status* extension enables judge clients to receive a more complete status of the server. If this extension is supported, the `serverstatus` flag is included in the connection flags of a connecting judge client. To request the status, a message containing a single line with the protocol code `serverstatus_request` is sent by the judge, whereupon the server MUST answer with a message with the protocol code `serverstatus_reply` on the first line, followed by an *implementation-defined* status report. This SHOULD be an XML structure including at least the compile and runtime configuration, the contest time and state.

2.8 Pause Toggle

If the server sets the `pause` connection flag, it MUST support the `paused` state in addition to `before`, `running` and `after`.

In this case a *judge client* can send a message to change the server state that contains the following two lines:

1. the protocol code `pause_toggle`
2. the new state, one of `before`, `running`, `paused`, `after`

3 Submission Protocol

When a contestant is connected to the server, *submitting* solutions and *polling* for the results MUST be supported.

A server which supports *notifications* has the corresponding *connection flag* set and sends results whenever the status of a submission of this contestant changes. Usually this is a notification when the code is received by the server and another notification after the submission has been judged. When a submission gets rejudged, an additional notification is sent. If a connection has the notification flag set and results for the contestant already exist, the server MUST send these results immediately after login. Servers SHOULD support notifications.

3.1 Submission

When solutions are submitted by a contestant, the message is split into a *header* and the *source code*. Even if the server does strip CRs in the received data, the source code MUST be copied as is. The header is a normal message and source code is the rest of the message block. The header contains the following lines:

1. the protocol code `submission_submit`
2. the problem identification, *UTF-8*
3. the programming language identification, *UTF-8*

3.2 Language Identifications

The identification string for a programming language SHOULD be the file extensions of source code files of this language, if this is possible. Clients and servers SHOULD use the *default languages* as default if not configured otherwise and MUST be configurable to use these languages. For the three *default languages* the identification is defined as follows:

`c` C

`cc` C++

`java` Java

3.3 Polling for Results

Contestants can poll the current results of all their submissions. The message used contains a single line with the protocol code `submission_results`.

The answer may contain new results that are marked as such, and a notifying server will treat these results as notified, even though the notification was part of the answer to a polling request.

If the client needs to know when the complete answer has been received, it can guard the request with a *heartbeat*.

3.4 Getting Results

Each result message contains the result of a single submission. This is used to notify contestants and to answer the polling request described above. It contains the following lines:

1. the protocol code `submission_result`
2. the submission number, an integer
3. the submission time in minutes since the contest started, an integer
4. the problem identification, *UTF-8*
5. the programming language identification, *UTF-8*
6. `notifies` if this message is a notification, empty otherwise
7. the state of this submission, either `new`, `unjudged`, `rejected`, `accepted` or `ignored`
8. an explanation for the judgement, empty for the `new` state, *UTF-8*

3.5 Judgements

The following judgement states are used throughout this protocol:

`new` the state of the judgement directly after the submission

`unjudged` the state of a judgement that has been withdrawn and awaits rejudging; it is otherwise treated the same as `new`

`rejected` the submission has been judged as *wrong*

`accepted` the submission has been judges as *correct*

`ignored` the submission is ignored; this means the submission is treated as if it were never submitted

There is a standard set of judgements, of which only the first is for accepted, the others are for incorrect submissions:

- `Correct`
- `Presentation error`
- `Wrong answer`
- `Time limit exceeded`
- `Run-time error`
- `Compilation error`
- `Contact staff`

4 Automatic Client Configuration

When a server supports the extension for *automatic client configuration* it informs a client by sending the `autoconfig` connection flag. Immediately after the *welcome message* and before any other messages, the *configuration messages* are sent. They all share a single protocol code `login_autoconfig`. After sending the configuration data, a *completion message* must be sent. The server MUST send a *complete* configuration to support the extension.

4.1 Programming Languages

For every *programming language* supported by the server, a message is send, containing the following four lines:

1. the protocol code `login_autoconfig`
2. the configuration type `language`
3. the programming language identification, *UTF-8*
4. the programming language name, to be shown to the user, *UTF-8*

4.2 Problem Set

For every *problem* in the *problem set* for the contest, a message is send, containing the following four lines:

1. the protocol code `login_autoconfig`
2. the configuration type `problem`
3. the problem identification, *UTF-8*
4. the problem name, to be shown to the user, *UTF-8*

4.3 Completion Message

The *completion message* consists of these two lines:

1. the protocol code `login_autoconfig`
2. the configuration type `completed`

5 The Scoreboard

Servers that support the *scoreboard extension* advertise so in their connection flags by sending the flag `scoreboard`. These servers MUST answer scoreboard requests.

5.1 Requests

Scoreboard requests can be sent by any connected client and contain only a line with the protocol code `scoreboard_request`.

5.2 Replies

A server supporting the extension MAY *deny access* to the scoreboard, e. g. to implement a period at the end of the contest when the scoreboard is not shown anymore or to answer requests while a scoreboard has not been computed. In this case the reply is a single line containing the `scoreboard` protocol code.

Otherwise the protocol code `scoreboard` is the first line and is followed by the most current scoreboard the server has computed. This output is *implementation-defined* although it SHOULD be possible to use the data for display using a standard *web browser*. When the *scoreboard duration* is over

and the scoreboard is no longer updated, the server SHOULD answer the request using the *last computed scoreboard*.

5.3 Scores

This section describes the commonly used scoring system.

For each problem and contestant the number of submissions until and including the first accepted submission are counted. If a problem was solved successfully, the time in minutes from the beginning of the contest until the first successful submission is taken. For each contestant a penalty is computed as the sum of times consumed for the solved problems and 20 penalty minutes for each rejected submission before an accepted submission of the same problem.

A contestant solving more problems ranks higher, and, if two contestants solved the same number of problems, the one with the lower penalty.

This means, that for determining the ranking, submissions for problems that a contestant already solved can be ignored. Likewise submissions by a contestant for a problem, that this contestant ultimately fails to solve, can be ignored.

6 Judge Protocol

If the server accepts judge connections at one or multiple ports, it MUST set the *judges* server flag when welcoming a new connection at these ports. When a judge logs in, the welcome message MUST have the *judge* connection flag set.

When the *connection flag* for *submission notification* `notifies` is set, the server MUST inform all *connected judge clients* with a notification whenever the state of a submission is changed. When notifications are used, the server MUST send a list of all submissions after login. This is similar to the contestant case. Thus, using notifications, a connected judge client will have up-to-date information about all submissions without polling.

Judges can use some protocol codes in the same way as contestants do, like heartbeats, automatic configuration and scoreboard requests, if they are supported by the server. What differs significantly is the submission protocol. Here the role of the judge is to *query the submissions* that have been made, and *fetch* and *judge* them.

6.1 Polling for Submissions

Judges can poll a complete list of all submissions. The message used contains a single line with the protocol code `submission_list`.

Like in the contestant client case, some of the answers may be marked as notifications, although they are part of this polling request, and thus are not notified separately.

If the client needs to know when the complete answer has been received, it can guard the request with a *heartbeat*.

6.2 Getting Submission Information

Each submission notification message contains the information about one submission. This is used to notify the judges and to answer the polling request described above. It contains the following lines:

1. the protocol code `submission_notify`
2. the submission number, an integer
3. the login name of the submitting contestant, *UTF-8*
4. the submission time in minutes since the contest started, an integer
5. the problem identification, *UTF-8*
6. the programming language identification, *UTF-8*
7. `notifies` if this message is a notification, empty otherwise
8. the login name of the judge, if already judged, or empty, *UTF-8*
9. the state of this submission, either `new`, `unjudged`, `rejected`, `accepted` or `ignored`
10. an explanation for the judgement, empty for the `new` state, *UTF-8*
11. the *lock string* for the judgements, typically `locked` when anyone holds the lock, empty if unlocked

6.3 Fetching a Submission Source

A judge can fetch a submission source by sending a message in the following format:

1. the protocol code `submission_fetch`
2. the submission number, an integer

When receiving such a message the server SHOULD try to obtain a *lock* on this submission, so that no other judges can fetch the same submission while it is being judged. The answer that is sent back contains a header with the following lines, which is followed by the *source code* of the submission:

1. the protocol code `submission_source`
2. the submission number, an integer
3. the string `success`

If the request fails, e. g. because the lock is already held by another judge the server answers with a message containing only these lines:

1. the protocol code `submission_source`
2. the submission number, an integer
3. the string `failure`

6.4 Judging a Submission

After a judge has successfully *fetched* the source code to a submission this submission can be judged. To do this the judge sends a message in the following format:

1. the protocol code `submission_judge`
2. the submission number, an integer
3. the state of this submission, either `unjudged`, `rejected`, `accepted`, `ignored` or empty
4. an explanation for the judgement, empty if the state is empty, *UTF-8*

When a judge sends this message, the server MUST *release a taken lock* for the corresponding submission. When a judge sends an empty string as state, the judgement is not changed, but a lock is released.

7 Clarifications

A server which supports *clarifications* sets the `clarifications` *connection flag* and uses several messages similar to the *submission handling*. Clarifications can be *requested* by *both* contestant and judge clients. They are *answered* by a judge who can decide to answer them for the *requesting client only* or for *all* clients.

If a server supports clarifications and has the `notifies` connection flag set, it must also notify clarifications. This is done similar to the notification for submissions. When a new connection is made, the state of all clarifications for this client, i. e. requests that this client made and requests that were answered to everyone, MUST be sent after login using the `clarification_reply` message. In addition to that, judge clients receive a `clarification_notify` and a `clarification_notify2` message for each request. Whenever the state of a clarification changes the questioner, or if the request was answered to everybody all connected clients, MUST be notified. Likewise every judge has to be notified using both notification messages for every change in the state of a request.

7.1 Requesting a Clarification

A clarification request can be sent by contestants and judges. It is split in a *header* and the *question*, like the source code in a submission, and the question MUST be copied as is. The header contains the following two lines:

1. the protocol code `clarification_request`
2. the problem identification, *empty* if it is a *general question*, *UTF-8*

7.2 Polling for Clarification Requests

Judges can poll a list of all submitted clarification requests. The message used for this purpose is the protocol code `clarification_list` on a single line.

The answer contains both two messages, `clarification_notify` and `clarification_notify2`, for every clarification request. These messages can contain *notifications* in which case these clarifications are not notified again separately.

To know when the request has been completely answered, *heartbeats* can be used.

7.3 Getting Clarification Request Information

Each notification *to a judge* about a clarification request and its state consists of *two* messages that are sent one after the other. The first message contains the request information that will remain unchanged. It is split in a *header* which is followed by the question. The header contains the following lines:

1. the protocol code `clarification_notify`
2. the request number, an integer
3. the login name of the questioner, *UTF-8*
4. the problem identification, *empty* if it is a *general question*, *UTF-8*
5. the submission time of the request in minutes since the contest started, an integer

This message is always directly followed by the current state of the clarification which is split into two parts as well, a header and the answer. The header has the following format:

1. the protocol code `clarification_notify2`
2. the request number, an integer
3. the login name of the judge, if already answered, or empty, *UTF-8*
4. the state of this request, either `unanswered`, `questioner` or `everyone`
5. the *answer time* in minutes since contest start when the last change was made, or *empty if none*
6. `notifies` if this message is a notification, empty otherwise
7. the *lock string* for the answer, typically `locked` when a lock is held, empty if unlocked

7.4 Getting Answers

Clients can poll the current clarification status of all their own questions and answers to everyone. This is done by sending a single line message with the protocol code `clarification_replies`.

The answer my contain *notifications* in which case the clarifications will not be notified again. The answer can be guarded by a *heartbeat* if desired.

For every clarification a separate message is sent containing a header followed by the answer to the clarification. The header has the following format:

1. the protocol code `clarification_reply`
2. the request number, an integer
3. the problem identification, *empty* if it is a *general question*, *UTF-8*
4. the state of this request, either `unanswered`, `questioner` or `everyone`
5. `notifies` if this message is a notification, empty otherwise

7.5 Locking a Clarification Request

If a judge wants to answer a clarification request, the request has to be locked first, which is done using this message format:

1. the protocol code `clarification_lock`
2. the request number, an integer

When receiving such a message the server SHOULD try to obtain a *lock* on this request, so that no other judges can lock the same request while it is being answered. The reply by the server has this form:

1. the protocol code `clarification_locked`
2. the request number, an integer
3. either `success` or `failure`

If the server answers *failure* the request made by the judge has no effect.

7.6 Answering a Clarification Request

After a judge has successfully *locked* the request, it can be answered. For that purpose the judge sends a message with these lines as a header, followed by the answer:

1. the protocol code `clarification_answer`
2. the request number, an integer

3. the state of this request, either `unanswered`, `questioner`, `everyone` or empty

When a judge sends this message, the server MUST *release a taken lock* for the corresponding request. When a judge sends an empty string as state, the answer is not changed, but a lock is released. The answer is empty in this case.

If a judge wants to answer the question to the *questioner only*, it has to use the state `questioner`, likewise if the judge likes everyone to receive the answer, the state `everyone` is necessary.

8 File System Structure

The server writes all data necessary to stop and restart the server to disk. These files SHOULD be human readable. Submission related data is stored in files with names of the form `n.purpose`, where n is the zero-based decimal number of the submission, and *purpose* is a string describing the content of the file. The format to use for these files is specified herein. The following protocols can either be used directly or by the server on behalf of a client.

8.1 Submissions

When a submission is received by the server three files are created: `n.source`, `n.judgement` and `n.submission`. The first one contains the source code that was submitted. The last one MUST be created *atomically* as the last of the three, and contains the following lines:

1. the identification of the *contestant who submitted* this code, *UTF-8*
2. the identification of the *problem*, for which this was submitted, *UTF-8*
3. the identification of the *language* the code was submitted in, *UTF-8*
4. the *submission time* in minutes from the beginning of the contest

To atomically create this file it SHOULD be created as a temporary file and moved into place. Using this approach, the server MUST cope with a rename that *failed due to NFS*, e. g. by verifying the result with a `stat` call.

These two files, `n.submission` and `n.source`, MUST NOT be changed.

8.2 Judgements

Judgements are saved as files named `n.judgement` and can be created by *external processes* as well as by the server itself. All changes to these files have to be done *atomically*. The contents are on separate lines:

1. the identification of the judge, *UTF-8*, or empty if none
2. the judgement itself, either `new`, `unjudged`, `rejected`, `accepted` or `ignored`
3. an explanation for the judgement, empty for the `new` state, *UTF-8*

When a submission is received, a judgement file is created containing the initial state.

As described above atomic creation of the file can be done by *renaming* an already written temporary file to the correct name, keeping in mind that *NFS* might need to be handled. Renaming also handles *atomic replacement* of the file, if the submission is being *rejudged*. The contents of a file MUST NOT be changed.

8.3 Locking

To prevent several judges to interfere with each other while judging a submission, a locking mechanism is implemented. Judges only change the *judgement files* when they hold a *lock*. For this purpose a judge creates a *temporary file* and *links* it to `n.lock`. If this succeeds this judge can, still atomically, change the *judgement* as long as the lock is held. The lock is released by unlinking the *lock file*. If the *linking* of the lock does not succeed, i. e. the *temporary file* is still linked only once, the judge *is not allowed* to change the judgement.

8.4 Notifications

If notifications are supported, the server should send out notifications whenever a judgement is made, either as a first judgement to a submission (the usual case) or when a submission is rejudged. The following section describe a possible implementation.

Notifications that have already been sent out to the contestant are remembered using `n.notified` and `n.jnotified` file names. To test whether a contestant was already notified, the number of hard links of the `n.notified` file are checked. Therefore the judgement files MUST only be hard linked once, i. e. *moving* them into place is acceptable, just *linking* them is *not*.

When a new judgement exists its corresponding file is created with a single hard link. If there exists an old judge notification file `n.jnotified` it has to be removed and is replaced by *linking* the judgement to `n.jnotified`, thus resulting in two hard links. When the is hard linked twice, an old notification file `n.notified` must be removed as is replaced by *linking* the *judge notification* `n.jnotified` to `n.notified` resulting in the final number of three hard links.

When the submission is *rejudged* the hard link count drops to one and another notification is sent.

Likewise changes of the lock file `n.lock` are notified by a file named `n.lock.notified`. If this file *exists* the last notified state was that the *lock is taken*, and if it *does not exist* the notification said the *lock is not taken*. *Only judges* are notified about a changes in the lock state.

8.5 Clarifications

The files used for *clarifications* are similar to those for submissions. They are of the form `Cn.purpose`, with n being the number of the clarification request and *purpose* used to discriminate between the different files of a single clarification request.

When the server receives a clarification request, a file `Cn.clarification` is created *atomically* after an *answer file* in the *unanswered state* has been created. The clarification file contains the following lines followed by the question:

1. the identification of the *questioner who submitted* this request, *UTF-8*
2. the problem identification, *empty* if it is a *general question*, *UTF-8*
3. the *time* in minutes from the beginning of the contest

The *answer file* has the name `Cn.answer` and contains the following lines followed by the answer, if it exists:

1. the identification of the judge, *UTF-8*, or empty if none
2. the request state, `unanswered`, `questioner` or `everyone`
3. the *answer time* in minutes since contest start when the last change was made, or *empty if none*

The *answer file* is similar to the *judgements* and can be changed atomically. The locking mechanism is exactly the same and the lock files are prefixed with `C`.

If *notifications* are supported, several additional files are created. There are some differences to the submission notifications. For every answer two notification files `Cn.onotified` and `Cn.notified` are created using the methods for the `n.jnotified` and `n.notified` respectively. The former file is created when all connected judges and, in case the answer was for *everyone*, all connected clients except the questioner are notified. The latter file is only created when the questioner is notified. *Lock notifications* are sent to the judges as before, using `Cn.lock.notified` files.

A Protocol Codes

A.1 Basics

`hello` Used by the server to greet new connections.

`login_request` First message of a client with login name and password.

`login_welcome` Sent by the server to confirm a successful login.

`login_autoconfig` Sent by the server to automatically configure a client.

`heartbeat_request` Requests a heartbeat for connection synchronisation.

`heartbeat_whoomp` The answer to a heartbeat request.

`error` Error messages sent before closing a connection.

`scoreboard_request` Requests a scoreboard message from the server.

`scoreboard` Answers (or not) a scoreboard request.

`pause_toggle` Request by a judge to change the current contest state.

`serverstatus_request` Request by a judge to get the server status.

`serverstatus_reply` Reply to a server status request.

A.2 Submissions

`submission_submit` Used to submit a solution.

`submission_results` Used by a contestant to fetch all submission result.

`submission_result` Used to inform a contestant about a submission result.

`submission_notify` Used to inform a judge about a (new) submission.

`submission_list` Used by a judge to fetch a list of all submissions.

`submission_fetch` Used by a judge to fetch a submission.

`submission_source` Used to inform a judge about submission source codes.

`submission_judge` Used to judge (or "unfetch") a submission.

A.3 Clarifications

`clarification_request` Used by a client (contestant or judge) to request a clarification.

`clarification_replies` Used by a client to fetch all clarifications.

`clarification_reply` Used to inform a client about a clarification.

`clarification_notify` Used to inform a judge about a (new) request.

`clarification_notify2` Used to inform a judge about a (new) answer.

`clarification_list` Used by a judge to fetch all requests.

`clarification_lock` Used by a judge to lock a request.

`clarification_locked` Used to inform a judge about a locked request.

`clarification_answer` Used to answer (or unlock) a request.