Constructing a Tester for Checking Student Protocol Implementations

Andrey Shabaldin, *Tomsk State University, Tomsk*

***Abstract*—The paper is devoted to constructing a tester for checking student implementations of application-layer protocols. There are two testing modes for checking a protocol implementation: the active testing and the passive testing. We automate the testing process in both modes for checking student protocol implementations through Internet programming labs. Test set that contains test cases for detecting most frequent student faults, is derived for active testing of server protocol implementations. For passive testing of client protocol implementations, the paper uses the invariant approach. The approach is illustrated by a part of Dynamic Host Configuration Protocol (DHCP). The structure of the tester is also described.**

***Index Terms*—Active testing, computer applications, passive testing, protocols.**

I. Introduction

THE problem of testing software protocol implementations is well known. In this paper, we are concerned about testing student application-layer protocol implementations in the context of Internet programming labs.

The well known approach to protocol testing is formal model based testing. Finite transition systems are widely used as such formal models [1]. The reason is that given a protocol, it is easy enough to specify the set of states, the input and output alphabets, transition and output functions [2]. However it is nearly to impossible to use a finite transition model when checking data flow or authentication processes that also have to be checked during the protocol conformance testing.

For this reason, when deriving test cases for a given protocol we use another approach: specifying and classifying most common faults in student protocol implementations. Differently from the formal model based approach, this approach does not use formal methods for test derivation and correspondingly, the complete test coverage cannot be guaranteed. However, such tests are useful for detecting faults that are typical for students.

It should be noticed that an implementation of most application-layer protocols can be divided into two independent parts: the client part and the server part and both of them should be tested. In order to test the server part the active testing can be used, since the server part, as a system under test can be completely controlled and observed by the tester. In this case, the system under test should be turned off from normal operation conditions, and this is not always possible for real applications. For example, it is hard to apply active testing for checking the client part of a protocol implementation. In this case, passive testing is used [3]. Passive testing is performed by observing (and then analyzing the input-output behavior of a system under test) without interference in the system operating.

In this paper, we describe a tester that is used for automated testing of student protocol implementations.

Andrey Shabaldin, e-mail: psi@sah.tomsk.ru.
Tomsk State University, 36 Lenin Prospekt, Tomsk, 634050, Russia.

II. Short Lab Description

Each student should implement a client or a server part of one of application-layer protocols listed below. Here we note that usually it is not necessary for a student to implement the complete protocol specification; only a protocol part is given to implement. The protocol implementations should be based on the sockets library. A student is given one of the following protocols for implementing.

--POP3 – Post Office Protocol, version 3 [4]. This protocol is used to fetch mail messages from a user mail drop on a mail server. Commands that have to be implemented by a student are USER, PASS, STAT, LIST, RETR and QUIT.

--SMTP – Simple Mail Transfer Protocol [5]. This protocol is used to exchange mail messages between mail servers and to forward messages from a user to another mail server. A student software implementation must “know” the following commands: HELO, MAIL, RCPT and DATA.

--TFTP – Trivial File Transfer Protocol [6]. According to the lab task, a TFTP protocol implementation must be able to process read and write requests only for files located in the implementation working directory.

--Time is the protocol for time and date synchronization [7]. According to the lab task, a client implementation requests the time from the server, retrieves the time and displays it in the format ‘DD/MM/YYYY HH:MM:SS’.

--HTTP – Hypertext Transfer Protocol [8]. This is the fundamental protocol for World Wide Web (WWW). HTTP controls hypertext data transmission between hosts. According to the lab task, a server implementation has to process requests for transmission of html-documents and image files (in jpeg, gif and png format) over the HTTP. A software implementation must “know” the GET method and must me able to convert an URL string into a local path.

--NNTP – Network News Transfer Protocol [9]. This protocol is used for the distribution, inquiry, retrieval, and posting of news articles. News articles are stored in a central database, giving subscribers the ability to select only those articles they wish to read. A student software implementation must be familiar with the following commands: LIST, QUIT, STAT, ARTICLE.

--Hostname [10]. The function of this protocol is to deliver machine-readable name/address information describing networks, hosts, and domains, within the Internet environment. A student software implementation must "know" the HNAME and HADDR commands.

III. Specifying The Most Frequent Faults in Student Protocol Implementations

There can be specified most frequent faults that may occur when implementing any protocol. Correspondingly it is possible to generate some test cases that can check any protocol implementation according to lab tasks. Some of such test cases are enumerated below.

1) An input sequence that is not specified in the specification is applied as an input to a software implementation. Correct implementation of the program has to produce an error message as the output to such a sequence. Respectively, a corresponding test case may contain any binary or text sequence that is meaningless for the protocol.
2) There is no input for a long time. A fault free protocol implementation waits for a message for a fixed time, and then must break the connection producing the timeout error.
3) A buffer overflow fault is the one that is most hard-to-detect, since the buffer overflow does not always imply a program failure. This error can only be detected when it changes the program output behavior.

Here we note that frequent implementation faults also significantly depend on an implemented protocol; i.e. there are faults that are specific for each application-layer protocol. For each protocol, we analyze such faults and construct test cases to detect specific and common faults in a protocol implementation.

IV. Active Testing of the Server Part

We now consider the situation when we need to check the server part of an application-layer protocol implementation. Below we introduce necessary definitions.

A test sequence for an application-layer protocol implementation is a finite sequence of "request/expected_response" pairs. *Request* is a sequence of input symbols applied to an implementation, after which the response is generated. *Response* is a sequence of output symbols, which is generated by the implementation as the response to a request. *An expected response* is the response generated by a fault free implementation. *A test for an application-layer protocol implementation* is a finite set of test sequences.

In order to check the server part of a protocol implementation we apply the request sequence of the first "request/expected_response" pair to an implementation under test (IUT), and the output sequence is then observed. If the response of the IUT matches the expected response, then the next pair of a test sequence is processed. If an unexpected output is observed then the IUT *fails* a test sequence, i.e., the IUT has a fault.

Thus, to test the server part of an implementation we have to somehow emulate the behavior of the client part, applying requests to the server and analyzing its responses. This can be done either manually or automatically. In the manual mode the client part can be emulated by using *telnet* utility: requests are typed manually from keyboard, displayed responses are checked manually against the expected responses. If there is no match, the IUT has a fault.

The process can be simplified by using the tester software, which automatically applies test sequences and checks if a response is an expected one. The interaction between the tester and a server implementation is shown in Figure 1.

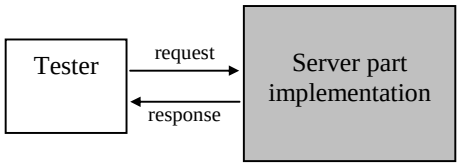

Fig. 1. Interaction between the tester and a server implementation.

It should be noted that the server behavior depends only on requests from the tester, i.e. the server part is completely controlled by the tester over the network connection. For this reason, it is possible to use the active testing when checking the server part of an implementation almost for all application-layer protocols. However, the active testing mode is not suitable when checking the client part of an application-layer protocol implementation, since the tester cannot completely control the client part over the network connection. The client part functioning significantly depends on the user activity and is almost independent of a server output. Therefore, it is necessary to check the client part through the user and network interfaces.

Since none of protocol specifications includes the user interface description, a programmer is free to implement the interface according to his own preferences. Subsequently there is no uniform user interface for each protocol implementation. Therefore a tester cannot apply control actions to the user interface of the implementation. A solution can be to test the client part of an application-layer protocol implementation using passive testing technique.

V. Passive Testing of the Client Part

The main idea of passive testing is to detect faults by studying the input-output behavior without interference into

the system normal operation. In our case, the system is the pair of the client and server parts working together. We assume that the server component is tested first and thus, is assumed to be fault free. In this case, any fault detected during passive testing is implied by a fault of the client part. As shown in Figure 2, the tester does not interfere into the interaction of two components; it only studies the input-output behavior of the client part.

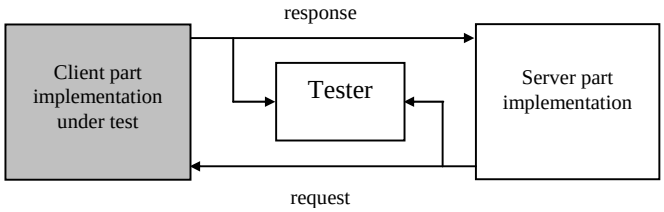

Fig. 2. Interaction between the tester and a client implementation under test.

In this paper, for passive testing we use a so-called *invariant*-based approach proposed in [11].

Invariant is a formal representation of some properties extracted from the protocol specification. The invariant approach for passive testing analyses the input-output sequence of an IUT (request/response sequence of the client implementation) in order to check whether this sequence matches invariants. An *invariant* is usually composed of two distinct parts: *a preamble* (*a postabmle*) and *a test*. A preamble (a postamble) is the part that needs to be common with the input-output sequence to launch a test. We define several types of invariants.

Output invariants allow us to express properties such as "after such symbol(s), we must have such output symbol". To use them, it is necessary to read the input-output sequence from left to right and to search for a pattern, which is used as a preamble before the test is launched.

Input invariants allow us to express properties such as "before such symbol(s), we must have such input symbol". To use them, it is necessary to read the input/output sequence from right to left and to search for a pattern, which is used as *postamble* before launching the test.

As an example of constructing input and output invariants from the protocol specification we consider the control part of the client part of application-layer Dynamic Host Configuration Protocol (DHCP) [12]. The DHCP diagram is shown in Figure 3. Nodes of the diagram correspond to states; while edges of the diagram, labeled with the "request/expected_response" pairs, correspond to transitions between states. As usual, S_0 is the initial state; *timeout* is a pseudo-input that shows that maximum waiting time for a server response has expired.

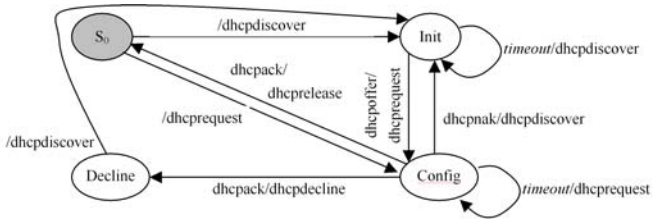

Fig. 3. The DHCP diagram.

The Dynamic Host Configuration Protocol (DHCP) is responsible for passing configuration information to hosts on a TCP/IP local network. DHCP consists of two components: a protocol for delivering host-specific configuration parameters from a DHCP server to a host and a mechanism for allocation of network addresses to hosts. DHCP uses a client-server model as the underlying model, where the designated DHCP servers allocate network addresses and deliver configuration parameters to dynamically configured hosts. Configuration parameters may include an IP address, a lease duration, a network gateway address, a name server address etc. The connection is initiated by a client that broadcasts a *dhcpdiscover* message for its local physical subnet. The server may respond with a *dhcpoffer* message that includes an available network address and other configuration parameters in DHCP message. The client broadcasts a *dhcprequest* message that may include options specifying desired configuration values. The server stores the configuration parameters of the requesting client in the database and responds with the *dhcpack* message containing the configuration parameters. If the selected server is unable to satisfy the *dhcprequest* message (e.g., the requested network address has already been allocated), the server responds with the *dhcpnak* message. After receiving a *dhcpack* message a client performs the final check on parameters. If the client detects an error, it sends the *dhcpdecline* message to the server and restarts the configuration process.

The set of output invariants of length 1 constructed from the specification of DHCP-client is shown in Figure 4. The set of input invariants of length 1 is shown in Figure 5.

```
–         / (dhcpdiscover | dhcprequest)
dhcpack   / (dhcpdecline | dhcprelease)
dhcpoffer / dhcprequest
dhcpnak   / dhcpdiscover
timeout   / (dhcpdiscover | dhcprequest)
```

preamble test

Fig. 4. The set of output invariants of length 1 constructed from the specification of DHCP-client.

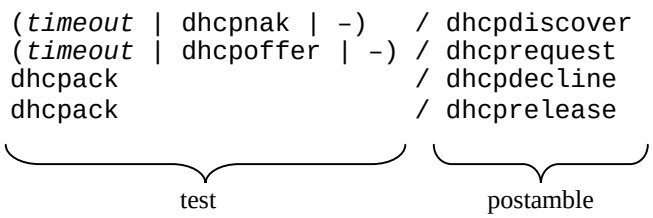

Fig. 5. The set of input invariants of length 1 build from the specification of DHCP-client.

During the testing process we observe the input-output sequence of a client implementation. If a pattern in the sequence that matches a preamble (postamble) of an input (output) invariant is found, then we check that the next (previous) symbols of the sequence match the test. If there is a match, then the testing process is continued. If there is no match, then an IUT has a fault. In this case, the testing process is stopped and a student is asked to correct the fault.

Generally, the length of preamble (postamble) in the output (input) invariant may be greater than one. For example, invariants of length 2 can also be constructed for the diagram in Figure 3. An example an output invariant {*request*1/*response*1, *request*2/?} of length 2 and an input invariant {*request*1/*response*1, ?/*response*2} can be considered. In Figure 6 the set of output invariants of length 2 for DHCP-client is shown.

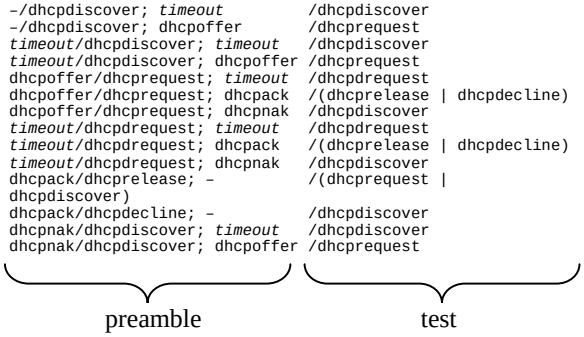

Fig. 6. The set of output invariants of length 2 constructed from the specification of DHCP-client.

VI. The Tester Structure

In our implementation, a tester is the software tool for active testing of the server part of an application-layer protocol and for passive testing of the client part. The tester structure is shown in Figure 7.

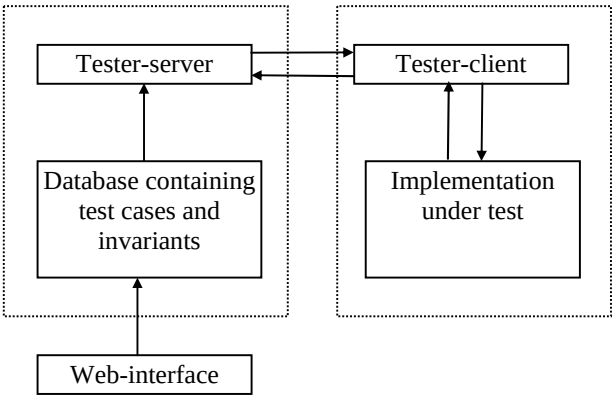

Fig. 7. The tester structure.

Test sequences and invariants are stored in the MySQL database. It is possible to add, edit or delete test sequences and invariants to/from the database using the web-interface. Since test sequences and invariants are sequences of pairs "request/response", both of them can be stored using the same data structure. We use regular expressions to express sequences of requests and responses. Regular expressions were chosen for the following reason.

During the active testing of the server part of a protocol implementation several correct server responses to the same request can occur. For example, the correct response to POP3 command "USER …" must consist of obligatory part "+OK" after which any arbitrary string is possible. In particular, the set of correct responses includes: "+OK Welcome, user!", "+OK Welcome!" and simply "+OK". Since it is impossible to enumerate all possible correct responses we have to elaborate a rule when a response is correct.

As for passive testing, requests and responses of a real implementation have parameters. For high quality testing, it is necessary to check not only the control part of the client part of a protocol implementation (the command order correctness), but also the correctness of data flow that includes parameters going with commands. To check parameters we additionally need a set of rules for each invariant. Such rules allow to check whether parameters that go with the response (request) are correct.

For protocols that transfer textual data the rules may be written as regular expressions.

A regular expression [13] is a pattern that describes or matches a set of strings, which are written according to certain syntax rules. Regular expressions are usually used to give a concise description of a set, without having to list all elements.

According to formal language theory the basic regular expressions contain only concatenation, alternation '|', grouping '()' and quantification '*' operators. These constructions can be combined to form arbitrarily complex expressions. However, the syntax of basic regular is not quite easy-to-use. For example if we need to match any symbol (including the empty symbol) one or more time, we need to explicitly point out all possible symbols from the alphabet and

put the '*' after the list, such as (*a*|*b*|*c*|*d*…)*. For this reason, in our labs the extended regular expression syntax [14] is used. For example, the set containing three strings "+OK Welcome, user!", "+OK Welcome!" and "+OK" can be described by the extended regular expression "^\+OK(|.*)\r\n$" where '^' denotes to search for the match in the start of the string, '\' is the escape symbol. Since the character '+' is used as special symbol it has to be escaped if it is meant literally. The operator '()' defines a block, '|' is the alternation operator. The expression '.' matches any single character, '.*' corresponds to zero or more copies of any single character, '\r\n' are carriage return and line feed symbols respectively, '$' means the end of a string.

The tester is the distributed application consisting of two interacting modules: the tester-client and the tester-server. The tester-server shares the same computer with the database containing invariants and test sequences. The tester-server handles requests from the tester-client to the database. The tester-client works on the user side, and provides the graphical user interface for controlling the testing process. One tester-server can process requests from many tester-clients simultaneously.

The client-server architecture of the tester is preferable for the following reasons:

1) load balancing: all operations about database interaction are handled by the tester-server, all other operations are handled by the tester-client;
2) all changes in test sequences or invariants are coming from the web-interface directly into the database, there is no need to make any changes at the client side;
3) the access to the tester-client does not mean the access to test sequences and invariants; thus, the probability of unauthorized modification of the tester database is significantly reduced.

VII. The Database Structure

The structure of the tester database is shown in Figure 8.

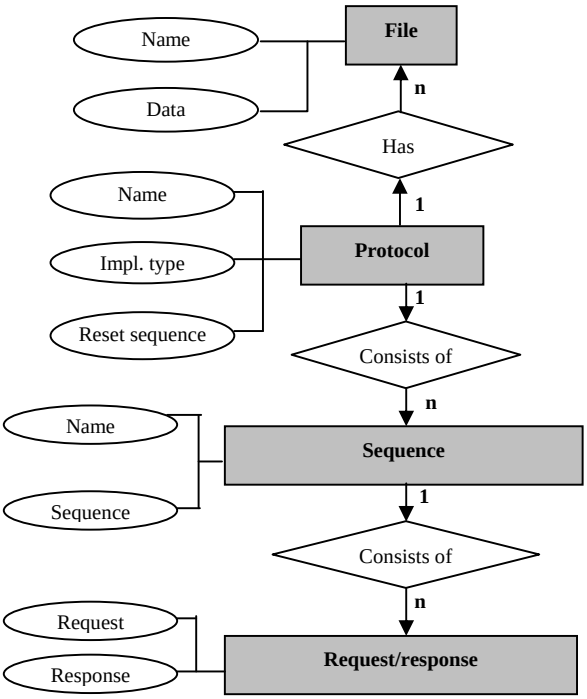

Fig. 8. ER-diagram of the database.

Table "File" contains the initial files. For active testing these files contain initial data which are needed for the successful work of a server implementation, i.e. the table has the file with usernames and passwords for POP3, the mail drop file for SMTP etc. For passive testing, these files contain *server modules* for the interaction with the client part of an implementation.

A server module is an implementation of the server part of an application-layer protocol, which is needed for the interaction with the client part of an IUT. The server module is a dll-library, which is loaded and executed by the tester-client during the passive testing process.

Table "Protocol" contains information about protocols for which the given test sequences and/or invariants can be used. If "Type" field has 'C' value, then invariants can be used. If the value of "Type" field is 'S', then test sequences can be used. Here we note that table "Protocol" may include both 'C'-type and 'S'-type records for a single protocol.

For each protocol there exist several rows in the "Sequence" table, which correspond to test sequences for the server part or to the set of invariants for the client part of an implementation. The "Content" field of the table "Sequence" contains the list of identifiers of "Request/response" table rows.

Each row of "Request/response" table contains the pair of regular expressions. They are "request/expected_response" pairs for active testing, and "preamble/test" or "test/postamble" pairs for passive testing.

VIII. Using the Tester to Check the Server Part of an Application-layer Protocol

The sequence of actions when performing the active testing is as follows.

1) A user runs the tester-client application. In Figure 9, the main window when running the tester-client is shown.
2) The tester-client connects to the tester-server and downloads the list of available protocols for active testing.
3) The user selects a protocol using the item (5). Then the user puts a port number into (6), which his implementation listens to. After this, the user points out the location of an IUT by selecting directory (7).
4) After filling all required fields the user may press the "Start" button (9). The tester-client connects to the tester-server again, downloads initial files and puts them into the specified in the field (7) directory.
5) When the tester-client is ready the message appears telling the user that he/she may run his implementation for testing. A user is not allowed to run his/her implementation for testing before the tester-server downloads initial files. The reason is the server part implementation could try to open initial files directly after starting, before these files are ready. It may cause an error.
6) After a user runs his implementation for testing the tester-client downloads test sequences and the testing process starts. The tester sends requests to an IUT, gets responds and checks them whether they are expected. Since there are many test sequences the testing process is time consuming. The progress bar (16) shows the testing progress. If all test sequences were applied to the server part of a tested implementation and all responds were correct, the IUT is said to *pass* the test. Otherwise, the IUT *fails* the test and a corresponding message is shown.

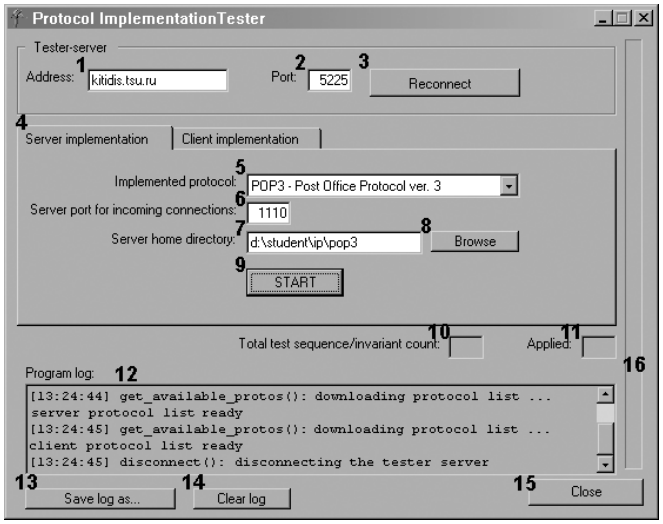

Fig. 9. The main window of tester program.

IX. Using the Tester to Check the Client Part of an Application-layer Protocol

In some way the passive testing is more complex than the active testing. The required steps of the passive testing are described below.

1) A user starts the tester-client application (Figure 10).
2) The tester-client connects to the tester-server and downloads the list of available protocols for the passive testing. The tester-server address and the port are specified in fields (1) and (2).
3) The user points out that he would like to test the client part of an implementation by selecting the "Client implementation" (4) and the client protocol list becomes available. Here we note that it is necessary to correctly specify the port to which the client implementation should be connected. The server module will listen to incoming connections to "loopback" interface (127.0.0.1) on the port specified in (18). Thus, an implementation under test should be configured for these address and port.
4) After the start button is pressed, the tester-client downloads the server module for a selected protocol and runs it. The server module dll-file is copied into the temporary directory and will be deleted when the testing is over. If the server module runs successfully then the set of invariants for the selected protocol is downloaded. Afterwards the tester-client opens the port specified in (18) and waits for the client to be connected. If there is no connection within 20 seconds, the testing process is stopped and the timeout error message is shown.
5) The tester examines input-output behavior of a client implementation during the interaction with the server-module. The "request/response" sequence, which is observed during the testing process, is checked against the set of invariants. If there is the match then the testing process is continued until the client is connected with the server. If there is no match for some part of the observed "request/response" sequence, then the IUT fails the test, i.e. the IUT has a fault. When the client disconnects from the tester, the tester waits for reconnection for 20 seconds. If no reconnection has happened the IUT passes the test. Here we note that a student who controls the client part during the test process has to follow some scenario. This scenario should contain step-by-step instructions that aim to reveal all specific features of the protocol implementation. Otherwise, there is a small guarantee that only correct implementations pass the test. The scenario should be included in the lab descriptions.

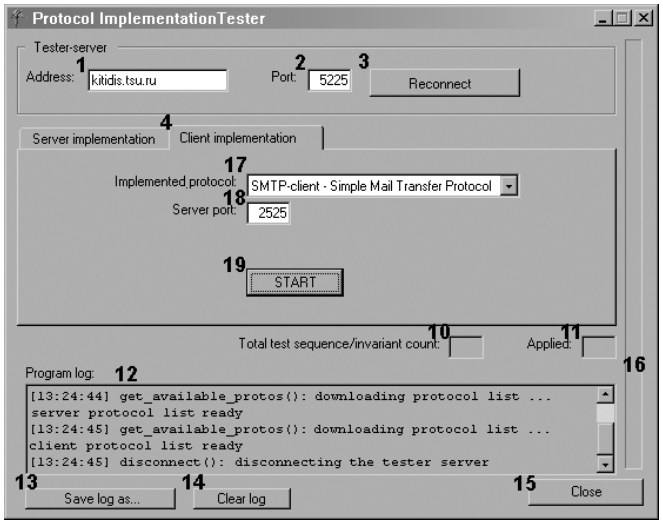

Fig. 10. The main window of tester program.

X. Conclusion

A tester proposed in this paper is an application for computer-aided checking of student implementations of application-layer protocols. Both active and passive testing modes are used in the tester. For active testing we do not use a formal model based approach when deriving test sequences. The latter are derived based on the analysis of most frequent faults, which are made by students when implementing protocols. The passive testing is based on a so-called invariant approach. The invariants are derived based on the FSM model. The tester is implemented and has been used in labs on INTERNET programming by teachers as well as by students for testing their implementations during the developing process.

References

[1] N. Yevtushenko, A. Petrenko, “Test derivation method for an arbitrary deterministic automaton,” Automatic Control and Computer Sciences, Allerton Press Inc., USA, no. 5, 1990.

[2] S. Seol, M. Kim, S. T. Chanson, “Interoperability Test Generation for Communication Protocols based on Multiple Stimuli Principle,” Proceedings of the IFIP 14th Inter. Conf. TestCom2002, pp. 151–169.

[3] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, A. John, “Passive testing and applications to network management,” Proceedings of the 1997 International Conference on Network Protocols (ICNP '97), 1997, p. 113.

[4] J. Myers, M. Rose. (May 1996). Post Office Protocol – Version 3 [Online]. Available: http://www.ietf.org/rfc/rfc1939.txt.

[5] J. Klensin. (April 2001). Simple Mail Transfer Protocol [Online]. Available: http://www.ietf.org/rfc/rfc2821.txt.

[6] K. Sollins. (July 1992). The TFTP Protocol (Revision 2) [Online]. Available: http://www.ietf.org/rfc/rfc1350.txt.

[7] J. Postel, K. Harrenstien. (May 1983). Time Protocol [Online]. Available: http://www.ietf.org/rfc/rfc868.txt.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. (June 1999). Post Hypertext Transfer Protocol – HTTP/1.1 [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt.

[9] C. Feather. (October 2006). Network News Transfer Protocol (NNTP) [Online]. Available: http://www.ietf.org/rfc/rfc3977.txt.

[10] K. Harrenstien, M.K. Stahl, E.J. Feinler. (October 1985). Hostname Server [Online]. Available: http://www.ietf.org/rfc/rfc953.txt.

[11] A. Cavalli, C. Gervy, S. Prokopenko. “New approaches for passive testing using an Extended Finite State Machine specification,” Journal of Information and Software Technology 45, 2003, pp. 837–852.

[12] R. Droms. (March 1997). Dynamic Host Configuration Protocol [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt.

[13] S.C. Kleene, *Representation of events in nerve nets and finite automata*. Automata Studies. Princeton, 1956.

[14] PCRE – Perl Compatible Regular Expressions [Online]. Available: http://pcre.org/.