Bachelor thesis
Computer Science

Radboud University

Learning a State Diagram of TCP Using Abstraction

Author:
Ramon Janssen
s0711667

First supervisor/assessor:
Prof.dr. Frits W. Vaandrager
F.Vaandrager@cs.ru.nl

Second supervisor:
Msc. Fides Aarts
F.Aarts@cs.ru.nl

Second assessor:
dr. ir. Sicco Verwer
s.verwer@cs.ru.nl

August 22, 2013

Abstract

Techniques have been developed for automated generation of state diagrams for modelling network automata, by only observing the external behaviour of such an automaton. Although these techniques have been applied on simple protocols and on software simulations, very few complex real-world network interfaces have been modeled in this manner. For this thesis, these modelling techniques have been applied to TCP-interfaces by connecting an implementation of a learning algorithm to them over a network. In this way, the behaviour of TCP-implementations of four different operating systems has been modeled, and compared to the standard model.

Contents

1 Introduction

To find errors or unexpected behaviour of a network interface, having access to an abstract model is useful, as this makes model-based testing possible [4]. An abstract model such as a state diagram can assist in proving the correctness of a network interface, finding security flaws, or in studying the behaviour of a protocol. However, a formal model of a network interface may not be available: the corresponding model may not have been made or published. Also it may be uncertain whether a customly-made model is consistent with the actual implementation.

Effort has been made to tackle this problem. D. Angluin [3] has introduced the L* algorithm, which was extended by Niese [6] to model Mealy machines. This algorithm can be used to learn a state diagram of a device in an active way, by externally observing the behaviour: the learning algorithm sends input queries to the device, and analyses the returning output messages to generate a state diagram which is consistent with the given input and output. This can be applied to a network interface, also called the System Under Test (SUT), by connecting the learner to it. The learner may then send messages over the network and analyse the returning messages. As only external behaviour is observed, the SUT can be seen as a black box and the code used to implement it is not needed.

Such methods work well if the number of input symbols and states is not too large. However, many protocols make use of messages with parameters such as sequence numbers and flags. Also, state variables may be used to store information about the network connections. The number of possible symbols and states of such network interfaces may therefore be nearly infinite, making it impossible to learn the corresponding state diagram. F. Aarts et al. [2] describe a framework to describe such interfaces with a state diagram by introducing a mapper. The mapper reduces the numerous different values of parameters and variables to a small number of abstract values, so that the L* algorithm can be applied. By applying this framework to the ns-2 network simulator, they have been able to make models of the Transmission Control Protocol (TCP) and the Session Initiation Protocol (SIP).

This technique has not yet been applied on a real TCP-interface. For this thesis, a mapper is made and connected to a module that can send network packets. By using another computer on the network as a SUT, an actual implementation of a TCP-server has been modelled to show that the framework described can actually be applied in a non-simulated environment. Models of different operating systems have been made in this way, and the results are discussed.

Related work. In addition to software simulations of network protocols, the framework described by Aarts et al. has also been used to generate state diagrams of embedded control software [10] and banking cards [1].

Furthermore, Dawn Song et al. [5] have developed techniques to learn the state diagram of a communication protocol used to control botnets.

Organization. In the following section, the relevant parts of TCP will be briefly summarized. In section 3, a description of Mealy machines, the learner-setting and abstraction will be given. In section 4, the practical setup of the software used and of the network will be described. The obtained results will be presented in section 5, and conclusions and possibilities for future work can be found in sections 6 and 7.

2 Transmission Control Protocol

The transmission control protocol [7], or TCP, is a widely used network protocol which is in the internet protocol (IP) suite. It allows two devices to set up a connection (or possibly two software components on the same device), and send data in both directions over that connection. The protocol is designed to be reliable, and data sent over the connection will be received in the right order. In this section, the relevant parts of the protocol will be briefly summarized.

To use a TCP connection, a device needs to assign a port to it. This is an integer used to distinguish different connections, so that one device can set up multiple connections in parrallel. As both sides of the connection have an IP-address and a TCP-port, a single TCP-connection is defined by the four-tuple (source-IP, destination-IP, source-port, destination-port). This implies that a device can open multiple connections on one of it's ports, as long as the IP-addresses or ports of the remote end are different.

Every segment exchanged between the two devices contains (among others) the SYN, ACK and FIN control flags, an acknowledgement number (ack), and a sequence number (seq). These control flags are used to inform the other device about a change of state (e.g. open a connection, close a connection), or to request the other device to change state. The numbers seq and ack are used to verify that the segments are received in the right order and that no segments are lost. A segment in which the SYN-flag is set is also called a SYN-message; the same is true for other flags. The flow of TCP can be described with a state diagram such as in Figure 1, which is based on a state diagram in the official specifications.

Both devices communicating through TCP initially have a different role: one device acts as a server and the other as a client. Before the connection is set up, the server must be in the LISTEN state, waiting for a client to connect with it. Setting up a TCP-connection happens with a three-way handshake: the client sends a segment to the server in which only the SYN-flag is set. On receiving this message, the server responds with a segment in which only the SYN- and ACK-flags are set. Finally, the client acknowledges the server's synchronization with an ACK.

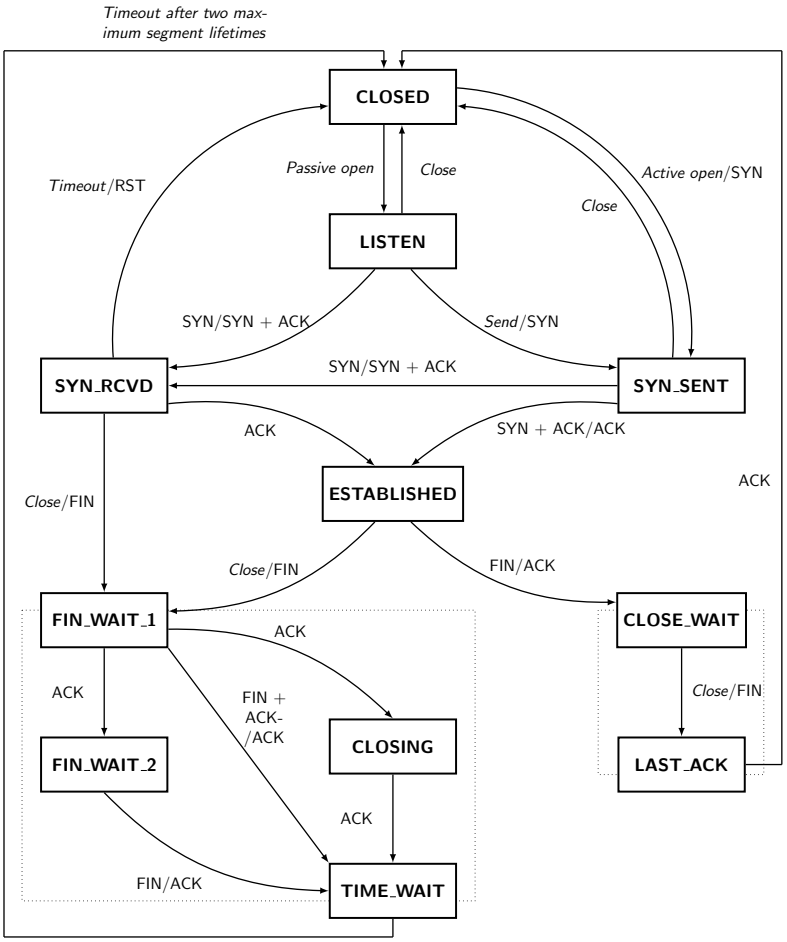


Figure 1: A state diagram describing TCP. Code to generate this model was retrieved from http://www.texample.net/tikz/examples/tcp-state-machine/. Copyright 2009 Ivan Griffin. Reprinted under the LaTeX Project Public License, version 1.3.

The connection is then ready for data transmission, by sending segments with a PSH-flag and data, which will be acknowledged. When either side has finished sending data, it sends a segment with a FIN-flag, after which it cannot send any more data. This can be acknowledged with a FIN+ACK if the other device also wants to close the connection, or with only an ACK if that device still wants to send data until it also sends a FIN. When the connection from both sides is closed, the corresponding four-tuple defining the connection will be blocked for an arbitrary period. This is to prevent a new connection being set up, while there might still be network packets incoming from the last connection. If anything goes wrong during communicating (e.g. an unexpected flag is set), either end may send a fragment

with the RST-flag set, which aborts the connection.

For this thesis, the states of the SUT that will be reached will be LISTEN, SYN_RCVD, ESTABLISHED and CLOSE_WAIT. These states are reachable by letting the SUT start in the LISTEN-state, and by sending fragments over the network. A SYN is sent to reach the SYN_RCVD state, an ACK is then sent to reach the ESTABLISHED state and a FIN is sent to reach the CLOSE_WAIT state. To reach other states, the learner would have to be able to send messages from the application layer of the SUT as well. Note that the connection cannot be closed entirely by a FIN, as this only closes the communication in one way. The application layer of the SUT would have to send a FIN as well to close the connection entirely.

The description and state diagram in Figure 1 do not fully describe the behaviour of a TCP-implementation. It is not described what response will be given if either side sends an unexpected segment, i.e. a segment for which there is no transition from its current state. A RST-message may be sent, the implementation may not respond at all, or the implementation may even show different behaviour.

The first segment sent by each device contains an arbitrary sequence number, usually randomly chosen. In all following segments, this sequence number is incremented, such that the correct order of the segments can be retrieved by the receiving device. Every segement also contains an acknowledgement number, which is set to the next sequence number expected from the other device. The acknowledgement number only needs to have this value if the ACK-flag is set, otherwise it is undefined.

3 Preliminaries

3.1 Mealy machines

The TCP interface will be modelled as a deterministic Mealy machine. In this section, a definition of such a Mealy machine will be given. This terminology is based on the terminology used by Aarts et al. [2]. A deterministic Mealy machine $\mathcal{M}$ is defined as $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where

- I and O are sets of input and output symbols, respectively,
- Q is the set of states, of which q_0 is an element. q_0 is the starting state,
- $\delta : Q \times I \to Q$ is the transition function,
- $\lambda : Q \times I \to O$ is the output function.

A transition $q' = \delta(q, i)$ with output $o = \lambda(q, i)$ can be interpreted in the following way: if the machine is in state q, it will accept input i. It will then return output o, and will change to state q'. This is also denoted as the

transition $q \xrightarrow{i/o} q'$. This definition assumes a deterministic Mealy machine; for every state and input, there is exactly one transition and output. A Mealy machine is finite if $|I|$ and $|Q|$ are finite.

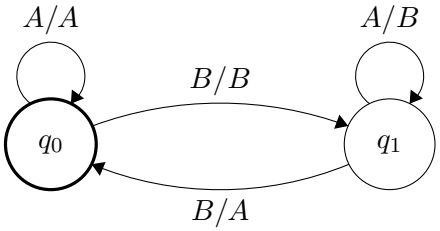


Figure 2: An example of a Mealy machine. The state with the thick edge is the starting state.

Figure 2 shows a graphical representation of a Mealy machine, which has the following properties:

- $I = O = \{A, B\}$
- $Q = \{q_0, q_1\}$, with q_0 as the starting state.
- $\delta(q_0, A) = q_0$
  $\delta(q_0, B) = q_1$
  $\delta(q_1, A) = q_1$
  $\delta(q_1, B) = q_0$
- $\lambda(q_0, A) = A$
  $\lambda(q_0, B) = B$
  $\lambda(q_1, A) = B$
  $\lambda(q_1, B) = A$

An input string is defined as $u \in I^*$, and an output string is defined as $s \in O^*$. Symbol ϵ is defined as the empty string. Transitions and output can be defined for strings in addition to single symbols:

$\delta(q, \epsilon) = q$
$\delta(q, ui) = \delta(\delta(q, u), i)$
$\lambda(q, \epsilon) = \epsilon$
$\lambda(q, ui) = \lambda(q, u)\lambda(\delta(q, u), i)$

When giving an input string to a Mealy machine, it simply uses all symbols in that string as inputs one-by-one. It makes the corresponding transitions and returns the string of all output symbols that were returned.

An observation is a tuple $(u, s) \in I^* \times O^*$ in which $|u| = |s|$. Every Mealy machine $\mathcal{M}$ with starting state q_0 has a set of observations $obs_{\mathcal{M}}$, defined by
$obs_{\mathcal{M}} = \{(u, s) \mid \lambda(q_0, u) = s\}$
This can be interpreted as that the observation (u, s) is made when u is given as an input string when the machine is in state q_0; s is returned by the machine. Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are observation equivalent if they have the same input alphabet I, and if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

For the Mealy machine in Figure 2, the input string $u = ABAB$ would return output string $s = ABBA$ from the starting state q_0. Therefore, $(ABAB, ABBA) \in obs_{\mathcal{M}_{example}}$

3.2 The learner

In the L* algorithm of Angluin [3], a learner is described which can learn deterministic finite automata. Niese [6] described how this can be used to learn a Mealy machine. To learn such a machine, it needs an *implementation* of that machine. An implementation of $\mathcal{M}$ is a device that accepts inputs in I, as well as a special *reset* input. It should also maintain a state, and have a well defined start state. When it is in the start state, it should give the same output sequence as $\mathcal{M}$ would on receiving an input sequence without a reset. When receiving a reset, it should return to its start state. A graphical model of the learner is given in Figure 3.

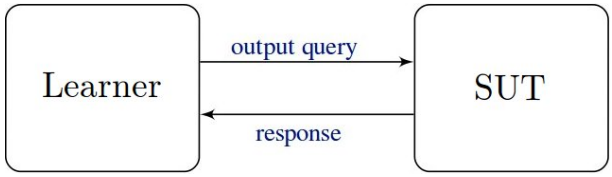


Figure 3: An overview of the learner and the SUT

The learner is connected to the implementation of $\mathcal{M}$ so it can send input to it and receive output from it. The learner will send *output queries* and *reset*s to the implementation, and observe the returning output. It will then give a hypothesis $\mathcal{H}$, which is the learned Mealy machine. $\mathcal{H}$ describes the observed behaviour of the device. If this hyphothesis is correct, the learned Mealy machine and the device are observation equivalent.

In the learner setting of Aarts et al. [2] an *oracle* for $\mathcal{M}$ is used to test whether a hypothesis $\mathcal{H}$ is correct. The oracle accepts $\mathcal{H}$, and will return *yes* if the hyphothesis is correct. If it is incorrect, it will return a *counterexample* which is an observation (u, s) made by the oracle, for which $\lambda_{\mathcal{H}}(q_0, u) \neq s$, meaning that the hypothesis and the device are not observation equivalent.

When a counterexample is returned, the learner will use this counterexample and send extra output queries to improve the hypothesis. The learner

will then generate a new hypothesis and the oracle will again test this new hypothesis. This process will be repeated until the learner receives a *yes*. When it does, the hypothesis is a model which describes the device.

The learner used in L* is implemented in a tool called LearnLib [8], which is used for this thesis. The SUT which will be modelled will be a TCP interface. LearnLib also contains an oracle which generates test sequences of queries and checks whether the hypothesis and the device give the same results. These test sequences are of fixed length and are generated by taking random input symbols. For the hypotheses found for this thesis, they are considered to be correct when 5000 test sequences of length 10 have been tested without finding a counterexample.

3.3 Abstraction

The learner setting as described is able to learn Mealy machines of devices if the number of input messages and states is not too large. Many protocols, including TCP, make use of parameters in messages. These parameters may be binary flags, strings or numbers in different domains, such as integers. If a message has parameters $p_1, p_2...$ from the domains $D_1, D_2...$ respectively, the symbols corresponding to that message will be in the domain $D_1 \times D_2 \times ....$

Network interfaces may also make use of state variables to store data. These variables cannot be stored in a Mealy machine. If a device has a set of states Q and state variables $v_1, v_2...$ with domains $D_1, D_2...$ respectively, an observation equivalent Mealy machine without state variables can be made by taking the set of states $Q' = Q \times D_1 \times D_2 \times ....$ These states contain all possible combinations of device states and state variables.

In this way of handling parameters and state variables, the number of different states and input symbols would become very large. With such an approach, the Mealy machines would be too large to use algorithms like L* or to be human-readable.

A solution to this problem proposed by Aarts et al. [2] is to map the concrete values of every parameter to a small domain of abstract values. This is done because the behaviour of the device may be the same for many different concrete values. In TCP, the *ack*-parameter is called *valid* if it is equal to the last *seq* that was received, plus one. Otherwise, it is *invalid*. Therefore, there are many different concrete values for the *ack* which are all *invalid*, and the device may return the same output for all these different *invalid* inputs. The possible values can thus be reduced from the large integer domain to the small domain of $\{valid, invalid\}$. If multiple parameters are used, one abstract input symbol exists for each combination of abstract values of those parameters.

The mapping from concrete to abstract values can depend on the values of the state variables of the device. In the example of the *ack*-parameter, the validity of the parameter depends on the *seq*-value of the last received

message. If a concrete value for a *ack*-parameter is valid in one message, it might be invalid in the next message as a new *seq*-value is received. To handle this, the mapping should also depend on state variables that can be updated with each input and output.

Such abstractions can reduce the number of input symbols from nearly infinite to a small number, so that an abstract representation of the device can be learned by L*, and can even be made human-readable. This abstraction is done by the mapper, which is placed between the learner and the device. The learner uses abstract parameter values as output queries, and sends them to the mapper. The mapper translates the abstract values into concrete ones and sends them to the device. Any concrete output from the device is translated back to abstract values, and is returned to the learner. The mapper may use state variables to do the translations, and these state variables can be updated when it receives input symbols from the learner or responses from the device. A reset-input should set all state variables to an initial value. A graphical overview of the learner and mapper is given in Figure 4

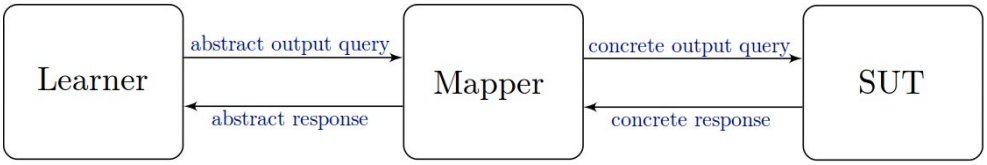


Figure 4: An overview of the learner, the mapper and the SUT

3.4 The TCP-mapper

The mapper is a module which receives abstract output queries from the learner, and translates them to concrete output queries which are then sent to the SUT, and vice versa. This mapper has been based on the one used by Aarts et al. [2]. In this case, all messages were TCP segments with three parameters: The sequence number *seq*, the acknowledgement number *ack*, and the *flags*-parameter. The numbers *seq* and *ack* have concrete domains of 32 bit unsigned integers, and the corresponding abstract parameters seq_{abs} and ack_{abs} can have the values *valid* or *invalid*. The abstract flags-parameter can have the values ACK, SYN, FIN, RST, or any combination of these flags (such as SYN+ACK). The concrete flags-parameter is the bitfields of flags in the TCP-frame, in which flags are set or unset with 1 or 0. Each element of *flags* defines which flags have been set: all flags mentioned are set, all other flags are not.

With these three parameters, an abstract input symbol is of the form $input(flags, seq_{abs}, ack_{abs})$. However, due to time constraints and practical constraints, only valid sequence and acknowledgement numbers have been

used as parameters of input symbols. The input alphabet used to generate the models in section 5 therefore consists of several combinations of flags, with valid sequence and acknowledgement numbers. As sequence and acknowledgement numbers do not vary in the abstract input alphabet, an input symbol can be named only by its flags. The input alphabet can therefore be described by $I = \{\text{ACK}, \text{SYN}, \text{SYN+ACK}, \text{FIN}, \text{FIN+ACK}\}$. Attempts have been made to include invalid numbers as well, but these attempts have not yielded complete models.

The abstract output alphabet contains all combinations of abstract sequence and acknowledgement numbers, and flags. All possible combinations of flags are in this flags-parameter, including flags such as RST which are not used in the input alphabet. In addition, an extra symbol *timeout* is used to denote when the SUT does not give any response. Therefore, a message from the SUT is of the form *timeout* or $response(flags, seq, ack)$. The latter will be abbreviated to $flags(seq, ack)$.

To make the proper translation between abstract and concrete, the mapper needs state variables to determine which concrete values of *seq* and *ack* to use. The state variables *lastSeqSent* and *lastAckSent* are set to the last values of the *seq* and *ack* parameters that were sent to the SUT in a valid request. The variables *lastSeqReceived* and *lastAckReceived* are set to the last parameters that were received in a valid response from the SUT. The parameters remain unchanged when sending an invalid request or when receiving an invalid response. The validity of requests or responses corresponds to the specifications of TCP [7]. The mapping from abstract to concrete requests is as follows:

- if $seq_{abs} = valid$:
  $seq = lastAckReceived$,
  $lastSeqSent = lastAckReceived$
- if $seq_{abs} = invalid$:
  $seq = 0$
- if $ack_{abs} = valid$:
  $ack = lastSeqReceived + 1$,
  $lastAckSent = lastSeqReceived + 1$
- if $ack_{abs} = invalid$:
  $ack = 0$

Invalid sequence and acknowledgement numbers are not used to generate the models in section 5, but they are mentioned for clarity.

The mapping from concrete to abstract responses is as follows:

- if $seq = lastAckSent \mid lastSeqReceived = 0$:
  $seq_{abs} = valid$,
  $lastSeqReceived = seq$

- otherwise:
  $seq_{abs} = invalid$

- if $ack = lastSeqSent + 1$:
  $ack_{abs} = valid$,
  $lastAckReceived = ack$

- otherwise:
  $ack_{abs} = invalid$

Furthermore, when the response is a *timeout*, all state variables remain unchanged.

4 Experimental setup

To be able to let the learner learn a model of a SUT, it should be able to send output queries to and receive responses from it. The pipeline from learner to SUT needed multiple components to achieve this.

LearnLib is the library which contains the learning algorithm, written in Java. The learner was connected to the mapper. The mapper was also written in Java, so that the learner could directly call methods of the mapper. A Java-program called Tomte [11] is used to call LearnLib-functionalities and to connect the learner and the mapper.

The translated messages from the mapper then needed to be sent to the SUT. This was done by a separate python-module, which was connected to the mapper through a local socket connection. The concrete values of the symbol parameters were sent through this socket connection. The python-module made use of the scapy library [9], which allows network packets to be customized and sent on the network. For building the fragments, the parameters were directly used in their appropriate fields. For other fields, default values were used if possible. Only the source port and destination port needed to be manually set to ensure that the fragments reached the right destination. This python module also sniffed the network for returning packets, and extracted the concrete values of the symbol parameters from them. Those where then sent back to the mapper. When no network packet returned for a certain time, a timeout-message was returned.

The SUT was another computer on the local network. A TCP server-port was opened by a java-application on this computer, so that connections from the python-module could be accepted. This application does nothing besides opening a server port. Computers with different operating systems (OSes) have been used as SUT: Windows 7, Windows Vista, Ubuntu 12.10 and Ubuntu 10.4.

To run the experiment, several issues had to be addressed. One of those was that the learner issues resets, to let the SUT return to its start state.

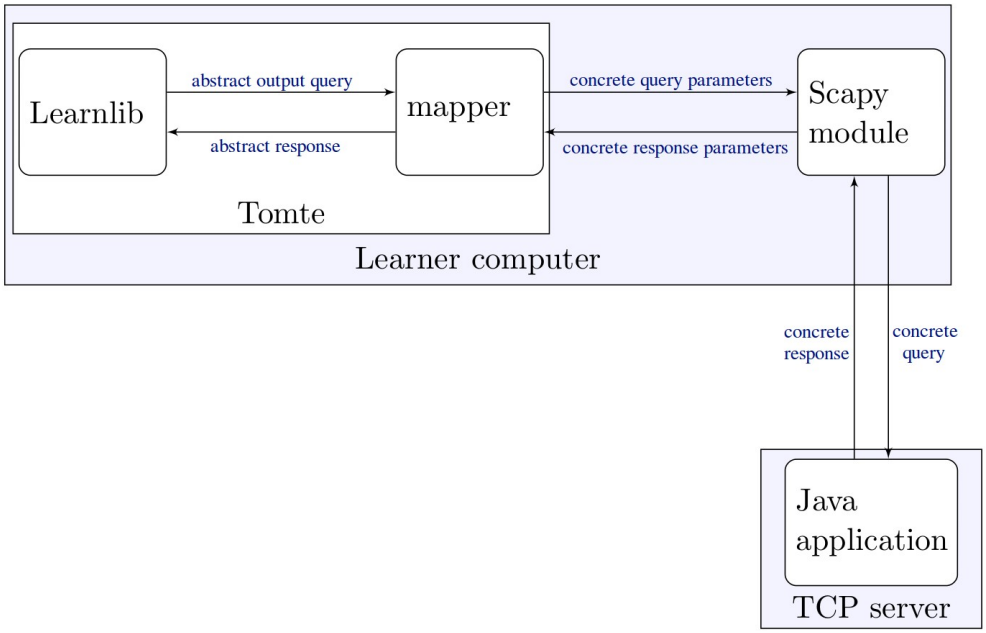


Figure 5: An overview of the experimental setup

This can be done by sending a fragment with a RST-flag. When a connection is closed with a RST-message, the behaviour for a new connection may be different than an entirely new connection. If that happens, a RST-message is not a valid reset as assumed by the learner. As explained in section 2, a connection is defined by the four-tuple (source-IP, destination-IP, source-port, destination-port), so this problem could be solved by using a different four-tuple after every reset. This was achieved by using a different port in the python-module. A large domain of integers was chosen to choose from, and the used port was incremented by one after every reset. This domain could be chosen large enough to give each connection its own four-tuple. By doing this, a totally new connection is used after each reset, and the experiments showed that the server behaved consistently after resets.

Another issue that had to be solved, was that the operating system on which the python module was running, responded automatically to TCP fragments. The operating system is unaware of what network packets are sent with scapy, and it will therefore receive responses over the network that it does not recognise. As a TCP-connection is set up by the learner, the operating system will notice a connection that it has not set up itself, and it will respond with a RST-fragment to shut down the connection. This problem could be solved by blocking outgoing packets with a firewall, so that the RST-fragments are not sent. In Ubuntu, scapy allows sending packets regardless of the firewall, so any packets sent for the sake of the experiment will not be blocked. Obviously, the computer used as SUT should allow all

relevant TCP fragments.

Other issues mostly concerned getting the network packets to the destination: in addition to the IP-address of the remote device, a mac-address is also needed when sending on a local network. When using the scapy library to send packets, the mac-address is not automatically retrieved. This could be solved by manually entering the right address in the python-module.

5 Results

On all OSes, a model was generated with only valid sequence numbers and acknowledgement numbers used for output queries. In this section these models will be compared to the theoretical model from section 2. The names of the states in the generated models are based on the names used in the theoretical model. A comparison between the theoretical model and the generated models is made based on the transitions from every state, mainly on the transitions following the normal path of opening and closing a connection.

The resulting abstract Mealy machine found for the four different OSes are presented in Figures 6, 7, 8 and 9. Some transitions have more than one input symbol, seperated by vertical lines; this means that this transition can be made for all of those input symbols. Additionaly, for each state, any input which is not shown yields a transition to that same state, with a timeout as output. These transitions have been left out for readability. V and INV represent *valid* and *invalid*, respectively. The states and transitions that are normally used when opening and closing a connection are green. The states LISTEN2 and UNRESPONSIVE (as described in the next section) are red.

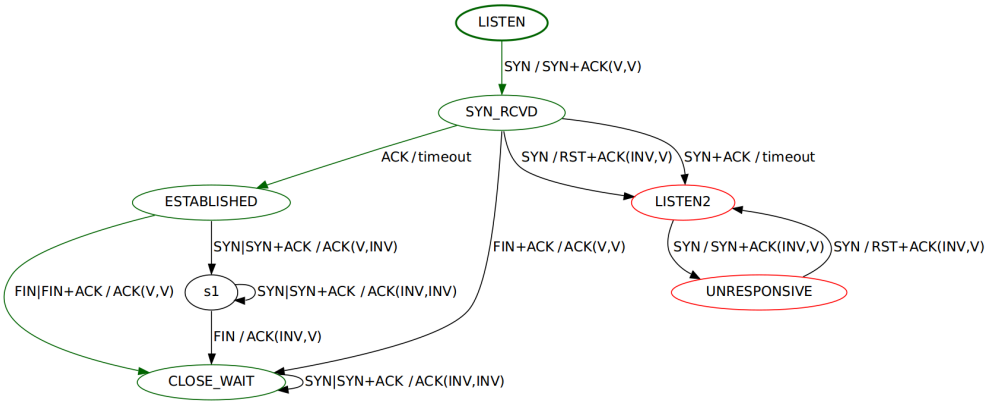


Figure 6: A Mealy machine describing the behaviour of Ubuntu 12.10.

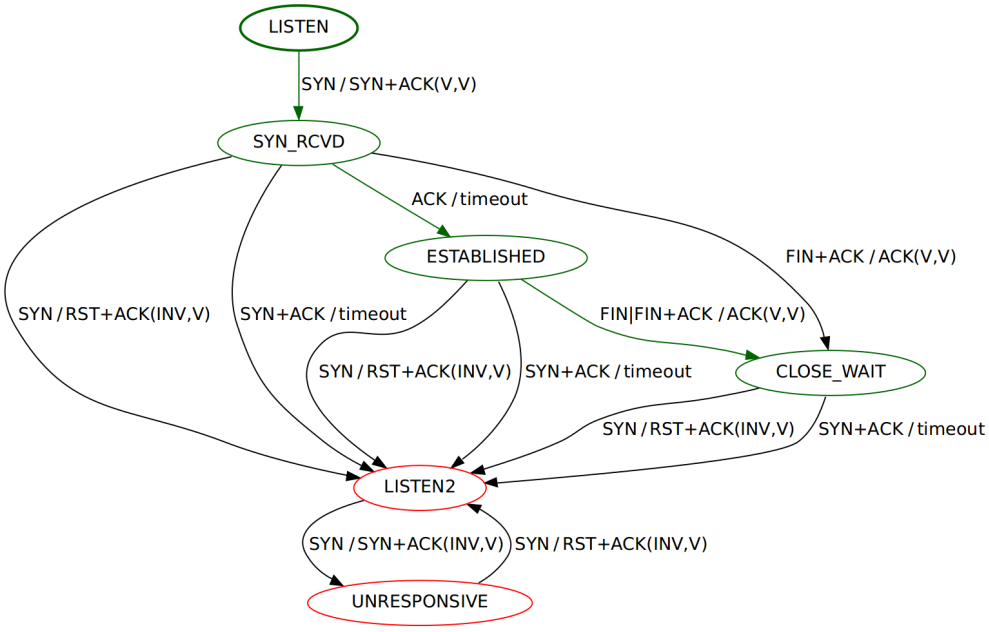


Figure 7: A Mealy machine describing the behaviour of Ubuntu 10.4.

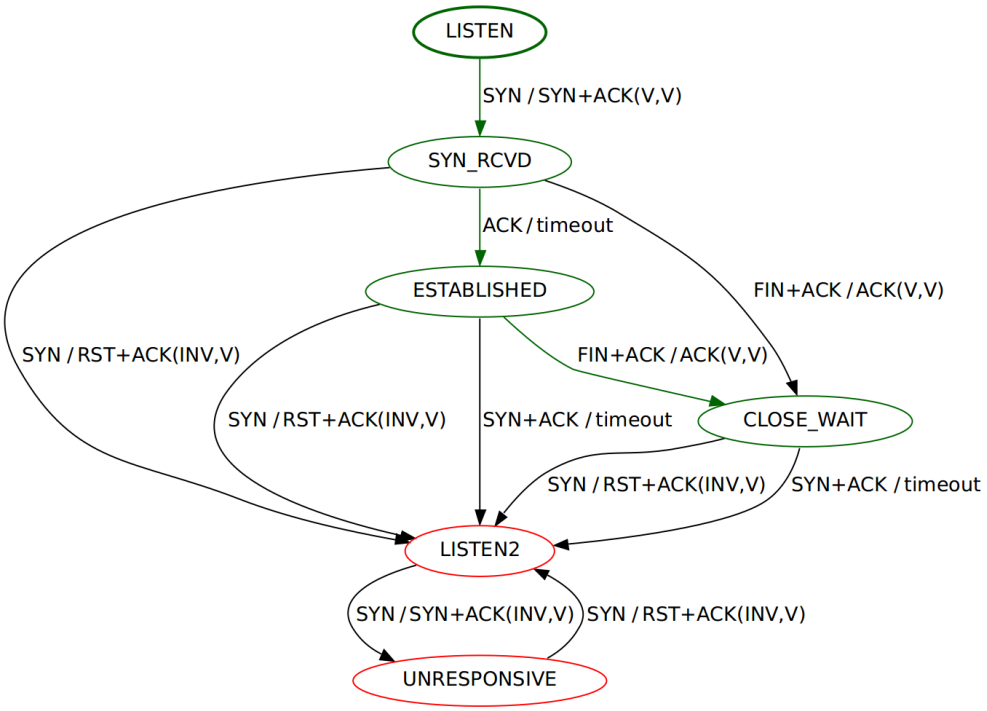


Figure 8: A Mealy machine describing the behaviour of Windows 7.

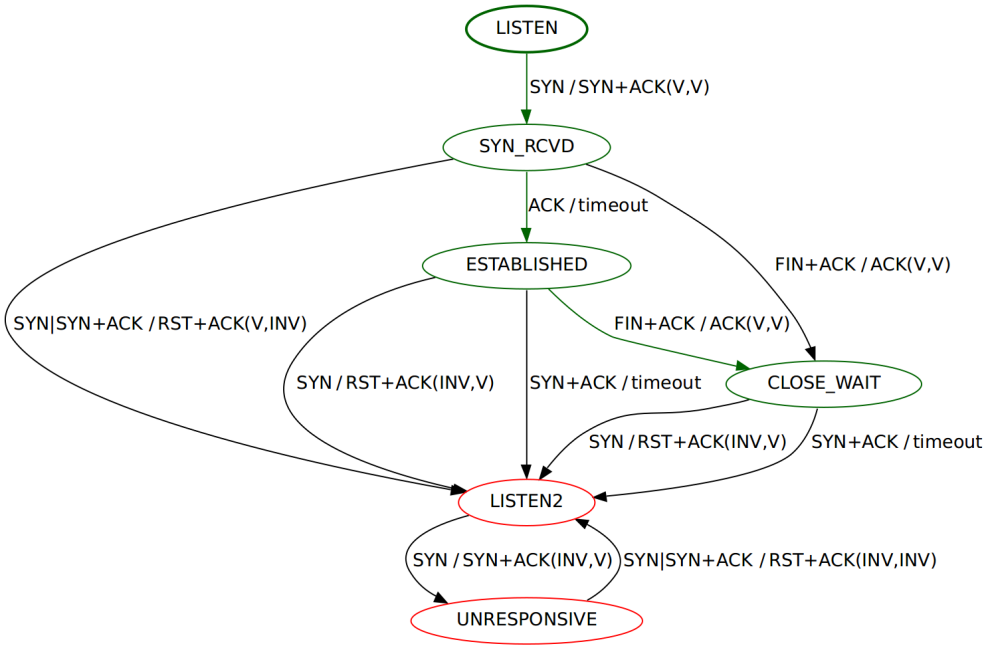


Figure 9: A Mealy machine describing the behaviour of Windows Vista.

5.1 The normal path

The four generated models share some similarities. Obviously, the normal way of opening and closing a connection works in the same way for all four OSes. LISTEN is certainly the starting state, as this is the state after the server-application on the SUT has only opened the connection. In this state, the SUT does not respond on any input except for a SYN-fragment. The response is always a SYN+ACK message, and the SYN_RCVD state is entered. An ACK-fragment can now be sent to enter the ESTABLISHED state. The connection can now be closed using a FIN or FIN+ACK, and the CLOSE_WAIT state is entered.

5.2 Sending multiple SYN-fragments

There is another similarity between the four models which is not in this normal path. After a connection has been set up, a state can be entered which will be referred to as LISTEN2. The outputs of this state are the same as the outputs of the first LISTEN state. This state can be entered by sending another SYN after the first SYN has been sent, which will be responded to with a RST+ACK. On all OSes except Windows 7, this second LISTEN2 state can also be reached with a SYN+ACK, which will be responded to with a RST+ACK on Windows Vista. Ubuntu does not send back a response, but the transition to LISTEN2 is made nonetheless.

LISTEN2 gives the same output as LISTEN; it only responds to a SYN, to which it responds with a SYN+ACK. Thus it seems to accept a new connection. The previous connection that has been accepted with a SYN from LISTEN, responded to with a SYN+ACK, seems to be aborted because of the second SYN (or SYN+ACK). In the case when a RST has been sent by the SUT this seems logical, as a RST can imply the abortion of a connection. However, Ubuntu can also reach this state without responding with a RST.

The normal path of setting up and aborting a connection cannot be made from this LISTEN2 state. When in the LISTEN2 state, a SYN will be responded to with an ACK, making a transition to another state. This state is not in the standard model, and will be named UNRESPONSIVE. In this state, a SYN can be sent again to abort the connection and return to LISTEN2. This will be responded to with a RST+ACK. On Windows Vista, this can also be done by sending a SYN+ACK. All other queries will not be responded to, and as such this connection does not function properly: a FIN or FIN+ACK should be responded to with an ACK.

The behaviour of the LISTEN2 and UNRESPONSIVE states could in rare cases cause problems; a full three-way handshake can be done, after which the client can conclude that a connection has been established properly. However, if the client tries to close the connection with a FIN, this fragment is never responded to. This situation will not occur often, as the client

must first send two fragments with a SYN-flag, which is not according to the specifications.

5.3 Sequence and acknowledgement numbers

The sequence and acknowledgement number of the responses are always valid when following the normal path of opening and closing a connection. When multiple SYN- or SYN+ACK-requests are made, the response on those SYN-requests never has both a valid sequence number and a valid acknowledgement number. In the models of Windows 7 and Ubuntu 10.4, each response is either a timeout or a fragment of the form *flags*(*invalid*, *valid*). In the models of Windows Vista and Ubuntu 12.10, either the sequence number, the acknowledgement number, or both are invalid.

A response with an invalid sequence number only occurs upon requests with a SYN-flag set. This can be explained by the normal use of a SYN-flag; if the server is in the LISTEN-state, it will choose its own sequence number upon receiving a SYN-fragment. This number is randomly chosen for each connection. When multiple SYN-fragments are received, the server might consider these as seperate connections, and choose a new sequence number for each new connection.

A response with an invalid acknowledgement number, and with the RST-flag set, can also be explained. For a RST-fragment, the acknowledgement number does not necessarily need to be correct.

5.4 Ubuntu 12.10

The model of Ubuntu 12.10 has some major differences with the other models. One difference is that in the three other models, the LISTEN2-state can be reached by sending a SYN or SYN+ACK when in ESTABLISHED or CLOSE_WAIT. In the model of Ubuntu 12.10, LISTEN2 cannot be reached from these states. Instead, a SYN or SYN+ACK leads to CLOSE_WAIT, or to another intermediate state (s1 in Figure 6) which is not in the theoretical model. The response to those requests is an ACK-fragment with an invalid acknowledgement number. Further testing showed that this number was not incremented by one properly after each request, and therefore the acknowledgement number stayed constant during all further responses while in CLOSE_WAIT.

5.5 Comparision with ns-2

As mentioned, Aarts et al. [2] have done a similar experiment with the ns-2 network simulator, and generated a model for it. There are several differences between this model and the models found for this thesis. First of all, the ns-2 model did not seem to respond to a FIN, only to a FIN+ACK. This is not the case with the real TCP-implementations; both FIN and FIN+ACK

cause transitions and outputs. Furthermore, the general shape of the model is very different. The model of ns-2 shows many states which are not in the models of real implementations. It also lacks the LISTEN2 and UNRESPONSIVE states that all four models of real implementations have.

5.6 Performance

For each input query, the python-module needed to wait for a certain time before concluding a timeout. Choosing a too small time causes a risk of missing a slow response. A small delay in the SUT would thus result in an incorrect timeout. This incorrect timeout might contradict other responses, leading to non-determinism, which let LearnLib crash. Over ethernet, a waiting time of 12ms turned out to be long enough to generate the models presented in this thesis, although the setup still crashed sometimes. This waiting time is the bottleneck of the learning algorithm. Some statistics about the learning process are found in Table 1.

OS	time(h:m:s)	learning queries	testing queries	hyphotheses
Ubuntu 12.10	7:07:44	216	10010	1
Ubuntu 10.4	2:11:08	223	5027	2
Windows 7	3:10:15	258	5027	3
Windows Vista	2:07:50	186	5000	1

Table 1: The amount of time, the number of queries and the number of hyphotheses needed to generate the models.

5.7 Invalid parameters as inputs

Attempts of using invalid numbers in the input alphabet have also been made. However, they led to abstract state machines that were too big to let the learner terminate. The learning time was much longer than for valid inputs only. A side-effect of this is that the chance of the learner missing a slow response of the SUT increases as the learning time increases, making it more difficult to correctly complete the learning and testing. Terminating the learner before a complete hyphothesis was found showed that the model contained at least 200 states. This premature model was untested and unfinished; it included many states which had no outgoing transitions, so a correct hyphothesis might very well have many more states. It is not yet clear whether a correct state machine would actually be finite and deterministic.

6 Conclusions

The experiments carried out for this thesis have shown that the framework proposed by Aarts et al. [2] can be used to generate models of a real TCP-server. It also shows that this is not a trivial task: for large input alphabets, such as an input alphabet with invalid sequence and acknowledgement numbers as well as different flags, the model can become quite large, and possibly infinite. The resulting models show that the TCP-implementations contain states which are not in the standard model. Also, for the inputs which were tested, implementations of different OSes differ in some details but show large similarities in general shape. Not only the behaviour as described in the standard model is equal among all OSes, but also the states LISTEN2 and UNRESPONSIVE are found in all implementations. They also show that real implementations differ greatly from the implementation of the ns-2 network simulator.

7 Future work

There is much room for improvement of the generated models. This can mainly be done by extending the input alphabet. Many useful additions to the input alphabet can be done, such as

- invalid sequence and acknowledgement numbers,
- triggers from the application layer, as many transitions described in the theoretical model have such triggers as inputs, and
- additional flags. The RST-flag and the PSH-flag can easily be added, although the latter would only make sense if the TCP-fragment also contains a data-payload.

Furthermore, the validity of parameters is still quite simple. A finer abstraction may also be chosen. Numbers which are one lower than the expected value might give different results than values which are higher than the expected value. A finer abstraction is also needed when data-transmission is added to the experiments, as the acknowledgement number can then be incremented by more than one.

Also, connecting to a TCP-implementation over localhost can be attempted. This has not been done for this thesis as the influence on the results was not known, but it could significantly speed up the experiments.

Acknowledgement

I would like to thank my supervisors, Frits Vaandrager and Fides Aarts, for all help, hints and discussions. Also I would like to thank Harco Kuppens, for the technical support in using Tomte.

References

[1] F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *4th International Workshop on Security Testing, Luxembourg, March 22, Proceedings*.

[2] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[5] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 426–439, New York, NY, USA, 2010. ACM.

[6] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02, 5^{th} Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[7] J. Postel (editor). Transmission Control Protocol - DARPA Internet Program Protocol Specification (RFC 3261), September 1981. Available via `http://www.ietf.org/rfc/rfc793.txt`.

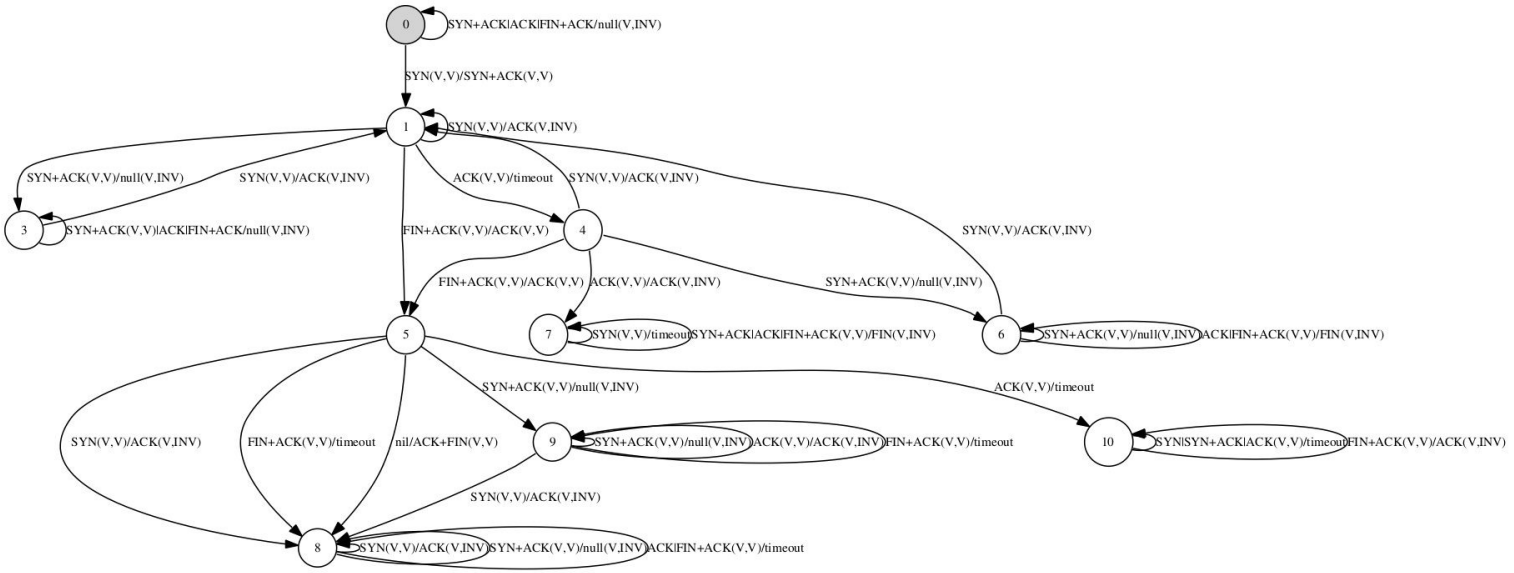
[8] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

[9] Scapy. `http://www.secdev.org/projects/scapy/`.

[10] W. Smeenk, F. Vaandrager, and D. Janssen. Applying automata learning to embedded control software, 2013.

[11] Tomte. `http://www.italia.cs.ru.nl/tomte/`.

A Model of TCP in ns-2



A model of a TCP-server, implemented in the ns-2 network simulator, as found by Aarts et al. [2]