15-453 Formal Languages, Automata, and Computation

Applications of Finite State Machines

Mihai Budiu

Lecture 3
January 24, 2000

Formal languages and automata are probably the biggest success story of theoretical computer science: such a big success story that all computer systems in use today apply this theory in several ways.

Lectures 3 and 4 will be dedicated to the description of real world applications of the Finite Automata. We will see applications in the following areas:

- formal specification — FA describe how a system should be built in an unambiguous way,
- classification and machine-learning — FA represent knowledge inside a computer,
- boolean function representation — used by virtually all CAD tools for manipulation of digital circuits,
- model checking — the main tool used to verify that hardware and software designs are correct.

All these topics could have a whole course of their own, so our treatment will necessarily be informal.

1 System and Protocol Specification

The first example we will consider is one of the fundamental components of the Internet: the TCP (transmission control protocol). A protocol specifies a set of rules that can be used by both parties to communicate. There are tens of implementations of TCP, but they implement the same protocol. Thus, a protocol is not a program.

1.1 TCP's Functions

TCP is a protocol that must ensure reliable, in-order transmission of data between two end-points. It uses the services of the IP (internet protocol) protocol, which can sometimes transmit data packets from one end to the other. IP may however loose, duplicate or reorder packets.

Virtually all network protocols which deal with this issue use the same scheme: *positive acknowledgements* and *retransmissions*, as in Figure 1.

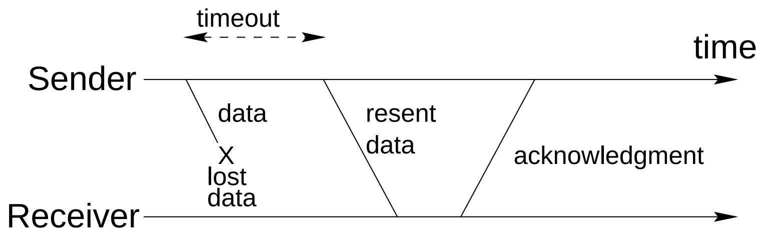

Figure 1: The positive acknowledgments are expected by the sender to certify that the data has reached the destination.

To prevent against interpreting old ACKs in the network as being ACKs for the current packet (the scenario in Figure 2), TCP needs to number both the data and the ACK packets, to be able to indicate the correspondence.

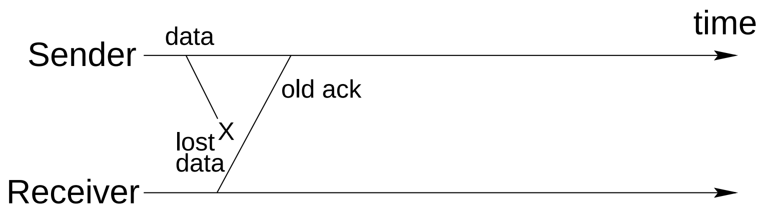

Figure 2: The data and ACK packets must be numbered, to ensure that the right ACK matches that data it is intended for.

1.2 Connection Set-Up

The numbering cannot always start at 0, because we would run the risk of confusing duplicate packets from earlier connections with packets from the current connection. Each connection must start with a random sequence number.

A protocol setup phase is necessary, where the two end-points that communicate agree on a common sequence numbering. Actually the two end-points agree on *two* sequences, one for each direction, because the TCP transmits both ways (i.e. it is a full-duplex protocol). They use packets labeled SYN (from synchronization) to achieve that.

The simple-minded approach in Figure 3 is not enough. (See if you can figure out why.)

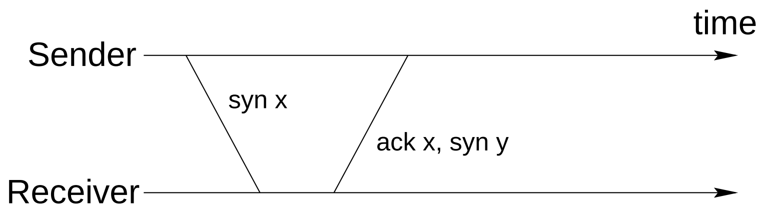

Figure 3: A flawed attempt to initiate the connection.

The scenario in Figure 4 would make the two hosts unable to communicate, because the receiver expects packets with a sequence number it will never see.

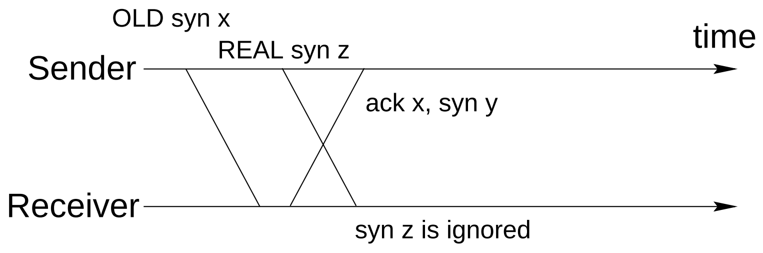

Figure 4: A failure case for the two-way handshake protocol.

The actual rule to establish (and also, in a modified form, to tear-down) connections is called "three-way handshake", and is displayed in Figure 5. The connection is considered established only after the three messages are received by their destinations.

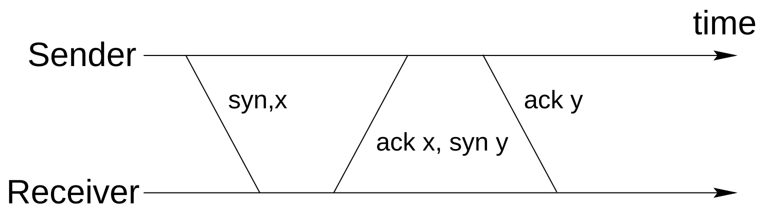

Figure 5: The actual three-way handshake connection set-up protocol used by TCP/IP.

Based on this kind of description, can you write one implementation of TCP? What would you do if, after you receive a SYN, you receive also an ACK? What happens if both parties send SYNs to each other, which cross paths in the network? What if you receive several SYNs in a row? What if. . . ?

There are many such questions to answer, and many possible scenarios for re-ordered, duplicated and lost packets. How do we describe succinctly the operation of TCP?

1.3 A Formal Specification of TCP

There has been too little material presented in the class so far, so the answer should be clear: using finite state machines (see attached copies of the protocol state-machine, from Douglas Comer's book, "Internetworking with TCP/IP, volume I, Principles, Protocols and Architecture", Prentice Hall 1995, pages 219–220). These FA are a little different from what has been presented, because they are not used to recognize languages. Here are some of their features:

- They have, like DFAs, a finite number of states and transitions.
- Each transition is triggered not by reading a character but by an external event, like the receipt of a packet of data or a timeout.
- Each transition also causes the automaton to *output* something, usually a data packet. The transitions are labeled with both the input and the output, separated by slash.

- There's no notion of an "accepting" state; such an automaton "computes" forever.
- If there's no transition from a state for a specific event (e.g. there's no arrow out of the LISTEN state labelled with the received packed FIN), it means that the automaton remains in the same state upon receipt of the event.

This kind of specification is not simply a pedagogical device, it is actually part of the "standard" which describes how TCP behaves. The Internet "standards" are called Request For Comments (RFC for short). TCP is described in RFC 793, and some corrections to the state-machine are made in RFC 1122. You can download these documents for instance from
`http://www.cis.ohio-state.edu/htbin/rfc/rfc793.html` and
`http://www.cis.ohio-state.edu/htbin/rfc/rfc1122.html`.

This automaton is a very concise and clear description of the functionality of the TCP protocol, which a programmer can use un-ambiguously to write a program, or which could even be used by an automated system to generate the skeletal code. We leave the explanation of the detailed states to a networking class: now you have the tools to fully understand it.

2 Machine Learning and Data Classification

First let us answer a question: are there finite languages (i.e. which only contain a finite number of words) which are also regular? Well, if there are finitely many words, we can definitely recognize them with finitely many states. So *all* finite languages are regular. Try to argue that a DFA which accepts a finite language cannot contain loops (cycles) in the state graph.

We can see the task of decision taking in the presence of a finite amount of data (e.g. answers to a finite number of questions) as a language acceptance problem; the following paragraphs will illustrate how.

2.1 Decision Trees

A *decision tree* is a kind of finite automaton in which the states "ask questions", and make transitions based on the answers. Decision trees are a very important tool of the Artificial Intelligence (see for instance the book "Machine Learning", by Tom Mitchell, published by McGraw-Hill in 1997, which has a whole chapter devoted to decision trees).

Figure 6 gives an example of a decision tree which helps us decide if we should play tennis on a given day according to weather. Again, this is not exactly the DFA we were taught about, but it has the main features:

- The states are labeled with questions.
- The arcs are labeled with answers.
- The final (accepting) states have no successors, and are labeled with the final decision.

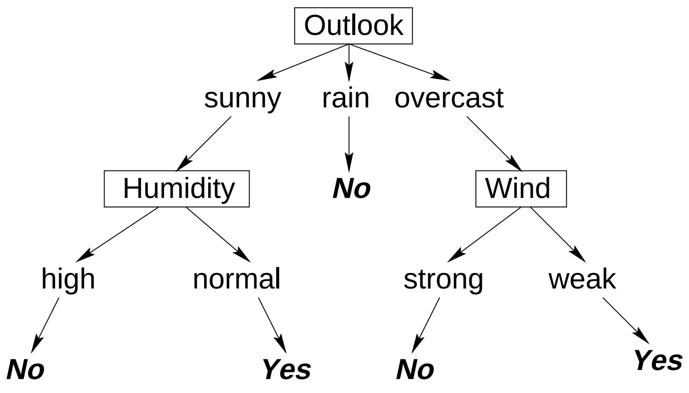

Figure 6: A decision tree for whether to play tennis.

There are lots of interesting questions about decision trees, like, how to build one given just some raw data, possibly from an expert, how to build a smallest tree, how to handle errors and omissions in the data, etc. We will leave these topics for a course specialized in Artificial Intelligence.

Decision trees have been successfully used to build expert systems for medical diagnosis, credit risk assessment for loan applicants, evaluating positions in the end-game of chess, etc.

Once a tree is built, you can use it to classify data. The reverse problem, of learning automatically trees from the data, is one of the subjects of machine learning.

2.2 Conclusion

What is interesting to us is the power of *representation* that FA have: very often they can successfully classify a huge amount of data in a very compact way: DFA can represent in a compact way information about large languages (the classical DFA taught in lecture 2 can represent even infinite languages, in a finite way!).

3 To Be Continued

In the next lecture we will see a very closely related relative of the decision trees, the Binary Decision Diagrams (BDDs), which are used to represent boolean functions of several variables $f(x_1, x_2, \ldots, x_n)$, (or, in the perspective we have now, to classify vectors of size n of boolean values: the vector $(v_1, v_2, \ldots, v_n)$ is classified according to the value of the function f when applied to that vector: if $f(v_1, v_2, \ldots, v_n)$ true or false? BDDs are are an ingredient found in most CAD tools for analyzing digital circuits, the main ingredient of computer systems.

We have also seen that we can *represent* systems and protocols as FAs; we will see a tool which can *analyze* such FA, to prove or disprove some of their properties: this is the area of *model checking.*