A knowledge-based algorithm for the Internet protocol TCP

Freek Stulp, Rineke Verbrugge
Cognitive Science and Engineering, University of Groningen
Grote Kruisstraat 2/1, 9712 TS Groningen, The Netherlands
E-mail: {freeks,rineke}@tcw2.ppsw.rug.nl

Abstract

Using a knowledge-based approach, we derive a protocol for the sequence transmission problem, which provides a high-level model of the Internet protocol TCP. The knowledge-based protocol is correct for communication media where deletion and reordering errors may occur. Furthermore, it is shown that both sender and receiver eventually attain depth n knowledge about the values of the messages for any n, but that common knowledge about the messages is not attainable.

1 Introduction

In their classical paper [6], Halpern and Zuck showed that epistemic logic provides a transparent way to specify and verify a number of protocols (like the alternating-bit protocol) that have been introduced for error-free transmission of sequences of messages over a distributed network. In particular, they introduced two knowledge-based protocols, A and B, that could solve the following problem. Let two processors be given, called the sender S and the receiver R. The sender has an input tape with an infinite sequence X of data elements. S reads these elements and tries to send them to R, which writes the elements on an output tape. The protocols are required to guarantee that (a) at any moment the sequence of data elements received by R is a prefix of X (safety) and (b) if the communication medium satisfies certain so-called fairness conditions, every data element of X will eventually be written by R (liveness). Fairness here means that infinitely many message instantiations from S to R and from R to S are delivered, guaranteeing that every message arrives eventually.

It is easy to see that no protocol can guarantee these properties in an environment were deletion errors, mutation errors, and insertion errors may all occur. For, suppose that the symbols transmitted over the channel are 0, 1, and λ (where λ denotes that nothing is sent), and that the elements of the input sequence X are 0s and 1s. Now any sequence of messages in $\{0, 1, \lambda\}^*$ sent by S may be changed by the communication channel to any other sequence of the same length as the original.

Halpern and Zuck did however solve the sequence transmission problem for communication media where any two kinds of the above-mentioned errors occur together. In order to do this, they used for each combination of two errors a

special encoding of messages ensuring unique decodability and error detection. Thus, the knowledge-based protocols A and B were implemented in different ways to solve the sequence transmission problem in different kinds of communication media. (See [6] or for more background [9, 5]).

In this paper it is our goal to use epistemic methods to model and analyse some important aspects of a protocol that is actually used in today's technology: the Transmission Control Protocol (TCP). Because this protocol is hardwired into today's Internet it is at present probably the most frequently used protocol. The epistemic analysis of TCP will be done in much the same fashion as has been done with other protocols in the past. Before doing this we will have to abstract from technical aspects that are irrelevant for our analysis. We will eventually acquire a knowledge-based protocol, represented by a simple algorithm. As we shall see, this algorithm beautifully demonstrates the windowing principle used by TCP.

The analysis of this algorithm yields some interesting results. We will show that the depth of knowledge the sender and receiver can accumulate about messages that are sent is dependent upon the length of the tape and the position of information on the tape. If an infinite tape models the transmitted data, the following can be shown. For any n and any piece of data on the tape, at some point n-fold depth of knowledge arises about the message, although common knowledge can never be achieved. Another interesting aspect of TCP is that it may almost be viewed as a generalisation of protocol B - but not quite, as protocol B uses only a finite message alphabet and the knowledge-based protocol for TCP does not.

The rest of the paper is structured in the following way. Section 2 gives a short introduction to the Transmission Control Protocol and its role for the Internet. In Section 3, we present knowledge-based algorithms that model TCP. Section 4 contains an epistemic analysis of these algorithms, giving bounds on the state of knowledge achieved by sender and receiver. Finally, Section 5 gives some conclusions.

2 The Transmission Control Protocol

In this section we will discuss the history of the Internet, and the role which TCP plays in it. The birth of the Internet as we now know it goes back as far as 1969 [10]. It was then that the U.S. Defence Department sponsored the development of the Advanced Research Projects Agency Network (ARPANET). The ARPANET consists of four layers. The lowest one is called the Network Interface Layer and comprises the physical link between devices. The second is the Internet Layer, which insulates hosts from network-specific details. The Internet Protocol (IP) was developed for this purpose. The third layer, the Service Layer, is very important because it guarantees that packages are delivered. Two protocols were developed for the Service Layer. TCP was introduced in 1973 and is used when a very reliable delivery is requested. The User Datagram Protocol or UDP, the counterpart of TCP, is used when the reliability requirements cannot be met. The combination of the TCP and IP protocols, called TCP/IP, is so frequently used that they are almost always found together. In this article we will only discuss TCP. The highest layer is the Process/Application Layer, which supports user-to-host and host-to-host processing. This layer includes ap-

plications such as Telecommunications Network (TELNET) and File Transfer Protocol (FTP).

The Internet meets a lot of problems when it comes to the correct delivery of a package from one computer to the other. First of all it should facilitate communication between a wide variety of servers, operating systems, and Internet browsers. This part is taken care of by the IP. It also has to deal with the limited capacity a network might have, and with possible deletion and reordering problems that may arise from overloading such a network. TCP does not only deal with these problems, but it also provides efficient use of a network, adapting transmission speed to the network load.

Note that TCP is a communication protocol, not a piece of software. Thus, the protocol does specify the format of the data and acknowledgements that computers exchange as well as the way computers initiate and complete a TCP stream transfer. On the other hand, TCP does not dictate the details of the interface between an application program and TCP itself [4]. This gives a programmer flexibility when implementing TCP for a particular computer's operating system. At the same time, the high-level specification of TCP makes it amenable for an analysis using epistemic logic, as we shall see in the future sections.

3 Implementation of the TCP

Now that we have a better understanding of what the TCP's role is in the Internet, and the problems it encounters in playing this role, we proceed by explaining the techniques it uses to cope with these problems. We will see that data is segmented and sequenced to solve insertion errors. Furthermore, data is acknowledged, enabling the TCP to spot deletion errors. Probably the most striking feature of the TCP is the sliding window. This window is not essential for a correct delivery of data, but it enables the TCP to make use of the network in a very efficient way.

3.1 Segmentation of the Application data

The first problem the TCP encounters in correctly transmitting data across a network is the size of the data, called *Application Data*, to be transmitted. Often networks have a Maximum Transmission Unit (MTU), that determines the maximum size of a unit of data that is sent across the network. Its value can be set by the network designer, and a common value is 1500 bytes. Application Data is almost always larger than the MTU [1]. To enable the transmission of these large chunks of Application Data, they are divided over several smaller units.

A TCP-header is added to these smaller units, which will now be called segments. Their maximum size is determined by the Maximum Segment Size (MSS). The TCP-header (20 bytes) contains information about the source and destination port, sequence numbers, acknowledgement numbers, and a lot more. The specification of ports ensures that data arrives at the right place. Because the data does not always arrive in time, the sequence and acknowledgement

[1] For example, to download the Postscript version of this article, the TCP has to send 0.54Mb across the Internet. This is more than 350× an MTU of 1500 bytes.

number are included to specify which part of the Application Data this Datagram encapsulates. This is TCP's way to deal with insertion errors. Not all the information in the header is always used. Certain flags can be switched on or off to indicate if a certain field contains relevant information.

IP-headers (another 20 bytes) are also added to these segments, converting them into IP-Datagrams, which are sent over the network. Note that a Datagram may never be larger than the MTU [2].

Before the Datagrams are sent over the network, a connection between the computers must be established. During this *connection setup phase* some agreements are made between the sender and the receiver about certain parameters and offsets. One of the offsets is the sequence number. Both sender and receiver have an internal parameter that counts the number of bytes that have been sent or received. They do not have to start at zero, and the sequence number of the sender and the receiver do not have to coincide.

3.2 Acknowledgements and Retransmission Time-Out

Once Application Data is being transmitted, other problems arise. Datagrams might get lost on the network. In order for the sender to know whether a Datagram has arrived or not, the receiver sends acknowledgements of Datagrams. The way this is done is shown in Figure 1. Suppose the sender is sending the Roman alphabet, and every segment contains exactly one letter as Application Data. An example Datagram may contain sequence number '0' and Application Data '*A*'. Assume that sequence numbers for both sender and receiver are 0 at the start of the transmission. Once the receiver has received this Datagram, it will acknowledge it by sending a Datagram with the acknowledgement field set to 0, because all the Datagrams up to 0 have been received correctly[3].

If a Datagram gets lost on the network (by a deletion error), it never reaches the receiver, who will thus not acknowledge it. Note that the receiver *will* keep sending acknowledgements of a previous Datagram. If the sender has to wait too long for an acknowledgement of a certain Datagram it assumes that a deletion error has occurred, and re-sends it. The time the sender waits before re-sending a Datagram is called Retransmission Time-Out (RTO). The process is shown in figure 1. The RTO is implemented by starting a retransmission-timer as soon as the Datagram is sent, like winding up an egg-timer. The computer keeps track of these timers, incrementing them in real-time. If a timer reaches RTO (the egg-timer rings), the Datagram it belongs to is re-sent, and the timer is reset (the egg-timer is rewound). If an acknowledgement of a Datagram is received before the RTO is reached, it is assumed that all the messages up to this acknowledgement have been transmitted correctly, and the timer is no longer needed. In figure 1 the first Datagram arrives correctly and is acknowledged as

[2]Since a IP-Datagram may never be larger than the MTU, and the segment size (data plus TCP-header) is always the size of the IP-datagram minus 20 bytes (for the IP-header), the MSS may never be larger than the MTU minus 20 bytes. If the MTU is 1500, than the maximum MSS value can be 1480. Often the MSS is chosen to be 1024, which is well within this 1480 range.

[3]If all the packages up to n have been received, TCP will actually send a 'request' for package $n+1$ in stead of an acknowledgment of n. Since requesting package $n+1$ implies that all the packages up to n are known, they are functionally equivalent. We will use the terminology of acknowledgements to enhance our knowledge-based approach. In the TCP-literature, the $n+1$ form is always used.

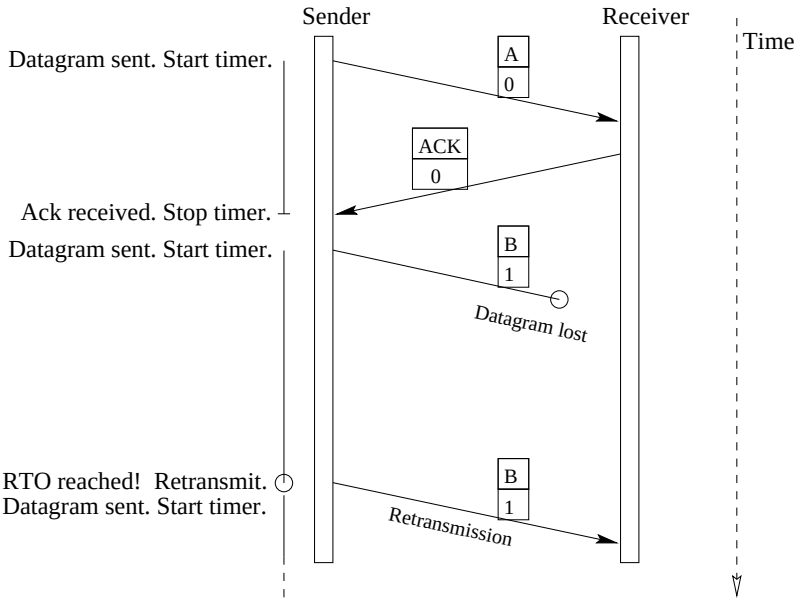

Figure 1: Acknowledging and retransmitting Datagrams

soon as the sender receives the acknowledgement, the second message is sent and a new timer is started.

Modern versions of the TCP have ingenious methods of estimating the RTO for optimal use of the network. This is done by measuring the Round Trip Time (RTT), the time between sending a Datagram and receiving its acknowledgement. See [4] for a good description of how the Round Trip Time is used to estimate an appropriate RTO. Other versions allow the network designer to specify the RTO. In these versions the RTO does not change during transmission.

3.3 Sliding Windows

The implementation of the TCP we have seen so far is much like a hardware version of Halpern and Zuck's protocols A and B, in which the sender sends one package only, and just resends it until the receiver has successfully acknowledged it (see [6, 9] for a more detailed account). These protocols are reliable, but do not take into account the restrictions that a real-life network imposes on data transmission.

A basic problem is that transmission of receiver acknowledgements in one direction might hold up the flow of data in the other direction [12]. The data being held up is the actual Application Data the receiver is requesting with its acknowledgements! Protocol A and B ignore this problem. For every message sent by the receiver, at least one acknowledgement has to be sent. This one-on-one transmission would be highly inefficient on a real network.

Another network issue is that, although the MTU is the maximum size of a transmission unit, a network is often capable of coping with a sequence of units

if they are separately sent with short intervals between them. Protocol A and B certainly do not make use of this feature, since they sequentually only send one message at a time.

The TCP counters these real-life networking problems with a beautiful solution: *sliding windows*. Sliding windows allow the TCP to send more than one Datagram at a time [3], without having to wait for acknowledgements. Which series of Datagrams are to be sent is determined by a window which is placed across the segments.

In figure 2, the window is represented by a box, enclosing the first four segments. In this example segments 0/1/2/3 (again containing letters of the Roman alphabet) may be sent without waiting for any acknowledgements. The efficiency in this method lies in the fact that the receiver can acknowledge multiple Datagrams with one acknowledgement. If it receives 0/1/2/3 for instance, as shown in figure 2, it will only acknowledge the Datagram with the highest consecutive sequence number, in this case 3.

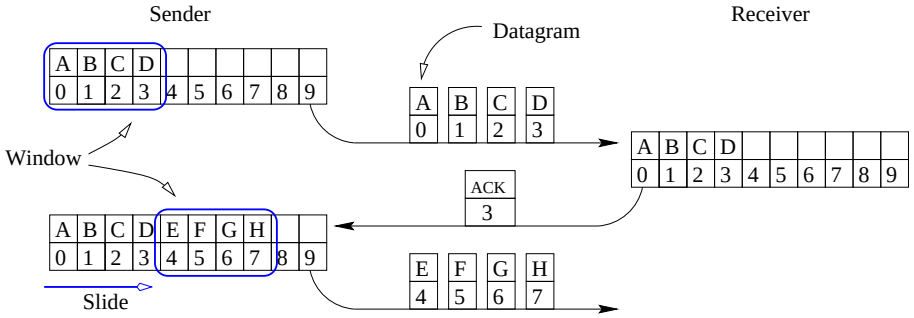

Figure 2: Acknowledging multiple Datagrams and sliding windows

Once the sender receives this acknowledgement, it knows that the receiver has received the first four Datagrams. This is where the sliding takes place. Because the first four Datagrams have arrived, re-sending them is useless. For this reason the sender's window *slides* across the segments considered to be done onto a fresh, unsent patch of data. Application Data is thus transmitted Datagram by Datagram, but also window by window.

Yet another problem arising in data transmission is the limited buffer space a receiver has for each connection. When running an Internet related application on the operating system, incoming Datagrams (segments of Application Data) are not directly sent to the application, but kept in a intermediate buffer. This relieves the application from dealing with all the networking details. It only has to access the internal buffer to get the Application Data it needs. The buffer is constantly changing in size, because incoming Datagrams are added, while the application is reading data from it. This means that at certain times it is able to cope with more incoming data than at others. Based on the size of the buffer, the receiver *advertises* a window-size. The more incoming data a buffer can process, the larger the advertisement. This advertisment is sent along with the acknowledgements. The sender can adapt its window-size accordingly. Usually, the window-size is an integer multiple of the Maximum Segment Size. If the MSS is 1024 for instance, common window-sizes vary between 0 and 4096 (0-4

times the MSS).

Two things should be mentioned in this context. First of all, if the receiver's buffer is incapable of handling any Datagrams, it will advise the sender to make its window size 0, effectively shutting down the transmission. Transmission can easily be opened by the receiver by transmitting a new advertisement with a window-size larger than 0. Another aspect that is important to our implementation of the knowledge-based algorithms is that a sender cannot change its window-size until the entire current window has been sent and acknowledged [12].

4 The Knowledge-based Algorithms

To give a knowledge-based interpretation of the TCP we will first have to convert the technical Internet language to knowledge-based terms. We will explain the language we use, and justify certain simplifications.

4.1 Data Format

First of all, in compliance with [6] we will consider our Application Data to be a *tape*. The tape we use as an example in this article contains the alphabet. Each letter on the tape is located at a certain *slot*, with an integer value (called the *position*) as its reference. The letter 'A' for instance can be referred to by position '0' on the tape. When it doesn't matter what the letter is, α or $-$ is used. We do not specify a MTU, but will set the MSS as being exactly one letter on the tape. In this article we will not consider the connection setup phase. We assume that the initial offsets and parameters are known to both sender and receiver at the start of the transmission. The sequence numbers are 0 for both at the beginning.

During transmission, TCP-headers are added to the Roman letters on the tape. The TCP-header will not contain all the information it usually does. For instance, since there is only one sender and one receiver, which we will often call S and R, we do not need to include the source *and* destination port. We will indicate who sent the message (the source port) by the subscripted knowledge operator K_R or K_S. As sending a message implies that one know what that message was, the knowledge operator is justified and enhances our knowledge-based approach.

Our header will also include the *window-size*, which is always sent both ways. Note the subtle difference between a window-size sent by the Sender and the Receiver. The Sender sends the size of the window from which the package came. The Receiver doesn't send the actual current window-size, but the window-size it would prefer at the moment.

No IP-header will be added, as this article only considers the TCP. We will call the knowledge-based version of a IP-Datagram a package. Packages have the format: $K_{R/S}(\mathit{position}, \mathit{data}, \mathit{window\_size})$ Sometimes we will use $K_{R/S}(\mathit{position}, \mathit{data})$ to specify a package when the window-size is not relevant. Although packages will never be sent this way, it is a useful abbreviation. Examples of correct packages are:

$K_S(3, D, 4)$**:** A message from Sender. Remember that K_S at the beginning of a message means that S is the source port, so the destination port must

be R. The message contains data-element 'D' at position 3 on the tape, and the current window-size is 4.

$K_R(6, -, 2)$: Receiver acknowledges having received all the packages up to and including the one with sequence number 6. Its buffer can now optimally process two packages at a time, so the window-size advertisement is set to 2. The '$-$' indicates that there is no need for Receiver to send Sender the Data, as Sender already knows this.

Another aspect that is worth mentioning is that if a tape is very long, the position marker will become larger and larger. The first package might be $(0, \alpha)$, whilst a latter package might be $(1000, \alpha)$. The first position requires one bit, the latter ten. Luckily, our algorithm has infinite computing capacity. But the Internet does not. The TCP-Header of the IP-Datagram always reserves 32 bits to indicate the sequence number, or position. This allows more than $4 * 10^9$ sequence numbers to be generated, which is sufficient in practice. On the total scale of an IP-Datagram, these 32 bits are not that substantial.

4.2 Modelling the TCP features

We have constructed a knowledge-based algorithm that models the TCP using the language used in the previous section.

In order to visualize how the knowledge-based algorithm of the TCP works, we have constructed an applet simulating the simplified case where the window-size is kept constant [11]. The applet link in the upper menu of this page will take you to an interactive applet that shows how the algorithms at work. A small manual is added to explain the applet.

The algorithms incorporate the Retransmission Time-Out, window sliding and varying window-size features that have been discussed in previous sections. Comment is added in the algorithms to explain how they work. Presently some more general issues concerning the algorithm will be discussed.

First of all we will only model versions of the TCP that have a fixed RTO. We feel that determining the RTO is really an issue of implementation, and should be left out of the knowledge-based algorithms.

The algorithms assume that both Sender and Receiver have an internal clock, though no global clock is necessary (or realistic). Resetting the local timer is represented as `timer` $= 0$; from this moment it is assumed that the timer increases in real time. For instance, if (`timer` $== 1000$) is true, this means that 1000 ms or 1 second has passed since the timer was reset. Receiver only needs one timer, because it only needs to send one acknowledgement at a time. Sender needs more timers because it can send multiple packages. Its timers are subscripted with a position, indicating which package is being timed.

Another aspect we feel is too hardware related to include is the ability of Receiver to make an estimation of an optimal window-size by checking the input-capabilities of its buffer. We assume that the system takes care of this and that the algorithm makes an outside call to the system to get this estimation. This call is called `estimateOptimalWindowSize`(). The choice of window-size will have no influence on the proofs given in future sections, if we assume that infinite window-sizes do not occur. The proofs can cope with any sequence of window-sizes, either constant, random or generated by `estimateOptimalWindowSize`().

To guarantee fairness, however, we do assume that the window-size is infinitely often greater than 0.

While protocol A and B do not have to function in real-time, we would like to model real-time in our algorithms. For this reason, processing incoming and outgoing packages is done separately, although the in- and output-sides of the algorithms can share global variables. We feel that this is more true to the nature of the TCP. It is probably best to see the algorithm as real code (but written in pseudo-code), in that every line of the algorithm can be processed very quickly. The timers of course keep track of the real time.

4.3 The Algorithms

Before we present Sender's and Receiver's algorithms we will explain briefly what the variables and functions refer to. All of the descriptions have been given more elaborately in previous sections.

`seq_number`	:	A sequence number.
`ack_number`	:	An acknowledgement number.
`high_cons_seq`	:	Used by R. Keeps track of the highest consecutive sequence number of the packages it has received.
`high_cons_ack`	:	Used by S. Keeps track of the highest consecutive acknowledgement number of the packages it has received.
`window_size`	:	The window-size.
`offset`	:	The offset of the current window. Only used by S.
`latest_advert`	:	Used by S. Keeps track of the most recent advertisement the sender received.
`estimate-Optimal-WindowSize()`	:	A function that calls to the operating system. It returns an integer that specifies what the best window-size would be for the current state of the buffer.
`time_out`	:	The RTO or Retransmission Time-Out.
`timer`	:	Timer used by R. Starts when an acknowledgment is sent.
$\texttt{timer}_{seq}$	:	Timers used by S. The index refers to the package being timed by this timer. Starts when that package is sent.

Sender's algorithm: Incoming packages

```
1  while true do
   {Get ready for receiving an infinite tape.}
2      when received Kr(ack_number,-,advertisement) do
       {You have received a package. Prepare for processing.}
3          latest_advert = advertisement
           {The last advertisement you have received is this one}
4          if (ack_number > high_cons_ack) do
           {If this acknowledgement is higher than the highest consecutive
            acknowledgement received so far....}
5              high_cons_ack = ack_number
               {This is your new highest consecutive acknowledgement}
6              forall ack with (ack ≤ high_cons_ack) do
               {For all the packages up to the highest acknowledgement }
7                  store KsKr(ack,-,window_size)
                   {Store the fact that you know the receiver knows it.}
8              end
```

```
9           end
            {Acknowledgements updated.}
10      end
        {Finished processing of incoming package.}
11  end
```

Sender's algorithm: Outgoing packages

```
1   window_size = 4
    {Set initial window-size.}
2   time_out = 20
    {Retransmission Time-Out (RTO). Common value is 20 ms}
3   offset = 0
    {Start reading the tape at position 0}
3   while true do
    {Start reading and sending tape}
4       forall seq with (offset =< seq < offset+window_size) do
        {For all the packages in the current window}
5           read(seq,alpha)
            {Read value from tape.}
6           store Ks(seq,alpha,window_size)
            {Store information in your knowledge base}
7       end
        {Tape within window has been read. Facts stored.}
8       while (high_cons_ack ≠ offset+window_size-1)
        {While not all the packages in the window have been acknowledged}
9           forall seq with (offset =< seq < offset+window_size) do
            {For all the packages in the current window}
10              if ¬KsKr(seq,-,-) do
                {If package 'seq' has not been acknowledged yet, }
11                  if (timer_seq >= time_out) do
                    {And its retransmission time has expired, }
12                      send Ks(seq,alpha,window_size)
                        {Resend the package to the Receiver.}
13                      timer_seq = 0
                        {Reset the timer.}
14                  end
15              end
16          end
17      end
        {All the packages in the window have been acknowledged... }
18      offset = offset + window_size
        {So slide the window!}
19      window_size = latest_advert
        {Set the window-size to the last advertisement the receiver made}
20  end
```

Receiver's algorithm: Incoming packages

```
1   while true do
    {Get ready for receiving an infinite tape.}
2       when received Ks(seq_number,alpha,window_size) do
        {You have received a package. Prepare for processing.}
3           store KrKs(seq_number,alpha,window_size)
            {Store the received package in your memory.}
4       end
```

```
        {Finished processing incoming package.}
5   end
```

Receiver's algorithm: Outgoing packages

```
1   when KrKs(0,-,-)
    {Wait until the first message is received.}
2   high_cons_seq = 0
    {Get ready for receiving an infinite tape.}
3   time_out = 20
    {Retransmission Time-Out (RTO). Common value is 20 ms}
4   timer = 0
    {Reset timer}
5   while true do
    {Get ready to acknowledge packages}
6       while ¬Kr(high_cons_seq+1,-,-) do
        {Still not received package with sequence number 'high_cons_seq+1'}
7           if (timer >= time_out) {
            {Time to retransmit acknowledgement?}
8               window_size = estimateOptimalWindowSize()
                {Estimate the best window-size for the state your buffer is in.}
9               send Kr(high_cons_seq,-,window_size);
                {Send acknowledgement }
10              timer = 0;
                {You've just sent a package. Reset the timer. }
11          end
12      end
        {You have received message high_cons_seq+1!}
13      high_cons_seq = high_cons_seq + 1
        {You now know the next message. Increment high_cons_seq.}
14  end
```

5 Epistemic analysis of TCP

We will first describe some choices we made in modeling the sequence transmission problem for the Internet.

An important aspect of the model of sequence transmission given by Aho and others and analysed in Halpern and Zuck's paper, is that they assume message transmission to proceed in synchronous clocked rounds. One may think of these rounds as consisting of three consecutive phases: a send phase, a receive phase, and a local computation phase. In their model, messages are received in the same round as they are sent, if they are received at all, so reordering problems do not appear [2]. This model is not adequate for studying TCP, where a global clock is an unlikely assumption and reordering is one of the most important problems to be solved. The model was relaxed by Halpern and Zuck to include asynchronous systems, where S and R perform an action only when they are scheduled. In order to assure liveness, it is assumed that S and R are scheduled infinitely often [6]. We adopt this extension to asynchronous systems.

Like Halpern and Zuck, we also extend the model of Aho and others by allowing messages to come from an alphabet larger than $\{0, 1, \lambda\}$. We even assume that the strings for $K_S(i, \alpha)$ and $K_R(i, -)$ are distinct for every value of i, so either we need an infinite alphabet or the strings grow longer as i becomes larger; we chooses the latter representation, as discussed in section 4.1. This assumption is necessary to solve the

sequence transmission problem in communication media where reordering is possible. For example, suppose that messages $K_S(10, \mathrm{a})$ and $K_S(1000, \mathrm{a})$ are represented by the same string and R receives $K_S(100, \mathrm{a})$ just after sending its acknowledgement about package 999. Then, due to possible reordering problems, R will not be able to decide whether it is a new message or an overly late version of $K_S(10, \mathrm{a})$.

When proving properties of knowledge-based protocols, it is usual to make use of a semantics of interpreted systems representing the behavior of the two processors over time (see [6, 9, 8]). We give a short review here. When modeling distributed systems, it is usual to make the assumption that at each point in time, each of the processors is in some state, which is referred to as its *local state*. All of these local states, together with the environment's state, form the system's *global state* at that point in time. These global states will be the possible worlds in a Kripke model. Thus, if one represents the global state as a vector of the local states, a system consisting of two processors R and S in environment e may be in global state $s = (s_e, s_R, s_S, s)$, where s_R and s_S are the local states of the two processors; in our asynchronous environment, such a local state may be represented as the sequence of distinct observations of the processor. As mentioned, the state of the environment is also included in the global state; it consists of those aspects of the distributed system that are relevant to an analysis of the problem at hand but is not part of the local states of the processors. The accessibility relation is defined according to the following informal description of "knowledge" of a processor. The processor R "knows" φ if in every other global state which has the same local state as processor R, the formula φ holds. In particular each processor knows its own local state.

A *run* is a (finite or infinite) sequence of global states, which may be viewed as running through time. Notice that time here is taken as isomorphic to the natural numbers, or a finite part of them. Because we do not want to demand a fixed time bound on the TCP process from the beginning, we allow time to run over the set of natural numbers. Note that there need not be any accessibility arrow between two global states for them to appear in succession in a run.

5.1 Correctness result

The algorithms implementing the knowledge-based protocol in section 4.3 can solve the sequence transmission problem in communication media where deletion errors and reordering errors, but no other kinds, occur. Formally, this can be proved using a semantics of interpreted systems I (i.e. set of runs) that are consistent with the knowledge-based protocol.

Theorem 1 *Let I be an interpreted system consistent with the knowledge-based protocol given in section 4.3. Then every run of I has the safety property and every fair run of I has the liveness property.*

See the introduction to this paper for definitions of the notions of safety, fairness, and liveness. Intuitively, safety for TCP is obvious since R writes a data element only if it knows its value. Assuming fairness, one can show the window eventually slides over every cell of the tape, thus that every message eventually arrives and is written by the receiver.

A formal proof of such a correctness result is still quite long and complicated, however, and we do not give it here (see the journal version of [6] for very similar proofs).

5.2 Accumulating knowledge

In the first knowledge-based algorithm studied by Halpern and Zuck, protocol A, the sender S needs explicit depth 4 knowledge of the form $K_S K_R K_S K_R(x_i)$ before

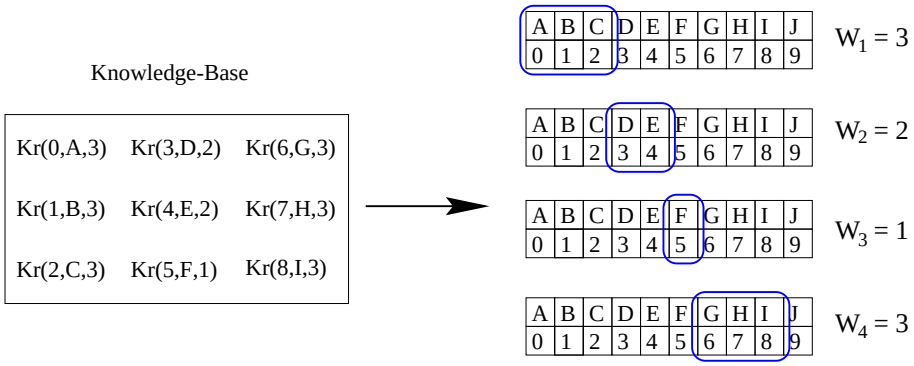

Figure 3: Reconstructing a sequence of windows

sending the next data element x_{i+1} to R. In addition to protocol A, Halpern and Zuck constructed protocol B in which S does not need to wait until it attains depth 4 knowledge before sending the next message. It was shown, however, that S eventually does attain such knowledge, in particular, S knows that always if R writes x_{i+1}, then R will know that S knows that R knows x_i (see the conference version of [6]).

In this subsection, we show a stronger result for the TCP algorithm. In fact, using TCP, the sender and receiver accumulate more and more knowledge about messages sent previously: when the window has moved n complete window sizes past the i-th package on the tape n knowledge of that package is attained by both participants (the receiver knowing even a bit more). For example, after the window has moved two complete window sizes, the receiver knows that the sender knows that the receiver knows that the sender knows that the receiver knows that the sender knows the first data element, or formally $(K_R K_S)^3(0, \alpha)$.

The result holds in communication media where there may be deletion and reordering errors, but no other kinds of error. In particular, there should be no undetected mutation and insertion errors.

???Hier moet Rineke nog meer over schrijven

Before we can present the theorem we will have to explain the concept of a *sequence of windows*. Since S and R have recorded packages in which the window-size was included, they can easily deduce which package has come from which window. Using this information they can construct a trace of all the windows that have been used. We will call this trace a sequence of windows. In figure 3 we show how the sequence of windows can be reconstructed from the knowledge-base.

Knowing the sequence of windows allows us to express every position as a certain number i plus a summation over previous window-sizes. We will formulate this as:

$p = i + \sum_{j=1}^{n} w_j \qquad$ with $i >= 0$ and $n >= 0$

In which p is the position of a package that has been received, n specifies the number of windows used, and w_j is the window-size of the j-th window in the sequence of windows. The values of w_j can be found at the right of figure 3. We will show the summation process for package 7. The position 7 of the package can be expressed in four ways.

$n = 0$ and $i = 7$ gives $p = 7 + \sum_{j=1}^{0} w_j = 7 + 0 = 7$;
$n = 1$ and $i = 4$ gives $p = 4 + \sum_{j=1}^{1} w_j = 4 + 3 = 7$;

$n = 2$ and $i = 2$ gives $p = 2 + \sum_{j=1}^{2} w_j = 2 + 5 = 7$;
$n = 3$ and $i = 1$ gives $p = 1 + \sum_{j=1}^{3} w_j = 1 + 6 = 7$;

We will use this way of expressing a position as a summation over the window-sizes plus a value i in the proof.

We will also need some definitions in order to formulate and prove the result formally.

Definition 1 *We use the following abbreviations and temporal notation:*

α *is a variable that can be any letter of the Roman alphabet.*

β *is a variable that can be any integer.*

$K_R(n, \alpha)$ *stands for* $K_R(n, \alpha, \beta)$ *(leaving out the window-size.)*
similarly for $K_S(n, \alpha)$.

$K_R(n, \alpha)$ *stands for "R knows that the n-th data element is* α*";*
similarly for $K_S(n, \alpha)$.

$K_R(n)$ *stands for "R knows the value of the n-th data element";*
similarly for $K_S(n)$.

The temporal operator $\Box$ *stands for the future operator on a run, and includes the present state; thus,* $\Box\varphi$ *stands for "*φ *holds now and at all moments in future on this run". Note that we do not really include temporal logic in the epistemic language with which we describe agents' knowledge at states. We simply use some facts we know from temporal semantics in our proof, in order to describe the processors' behavior on appropriate runs.*

Theorem 2 *Let R be any set of runs where:*

- *The safety property holds (so that at any moment the sequence Y of data elements received by R is a prefix of the infinite sequence X of data elements on S's input tape);*
- *S's state records all data elements that it has read and all acknowledgements that it has received;*
- *R's state records all the data elements it has written.*

Let w_j *be the window-size of the* j*-th window in the sequence. Then for all runs in R and all* $n \geq 0, i \geq 0$ *the following hold:*

[Forth]: *R stores* $K_R K_S(i + \sum_{j=1}^{n} w_j, \alpha) \rightarrow \Box(K_R K_S)^{n+1}(i, \alpha)$.

[Back]: *S stores* $K_S K_R(i + \sum_{j=1}^{n} w_j, -) \rightarrow \Box K_S(K_R K_S)^{n+1}(i)$.

Before we give a formal proof, we will try to give an intuitive feel of the knowledge the sender and the receiver can accumulate about each other with respect to the packages. First of all S will know a package (i, α) as soon as it has been read from the tape ($K_S(i)$). R knows a package (i, α) when it is received ($K_R(i)$). A package received by R must have been read by S, so R also knows that S knows the package ($K_R K_S(i)$). When S receives an acknowledgement of a package, S knows that R has received it. The sender can deduce $K_R K_S(i)$ from this (thus $K_S K_R K_S(i)$). This is actually the case when n is chosen to be zero.

Now we reach a more interesting case ($n = 1$). We will show what R can deduce once it receives a package $K_S(i + w_j, \alpha)$, in which w_j is the size of the j-the window in the window sequence. R will reason like this. If S has sent $K_S(i + w_j, \alpha)$ then its

window must contain $(i + w_j, \alpha)$. A window with size w_j that contains $(i + w_j, \alpha)$ cannot also contain (i, α). Apparently S has already shifted its window past (i, α). S would only have done this if it had received $K_R(i)$ from R. Since R now knows that S has received $K_R(i)$, and that (as we showed above) S would deduce $K_S K_R K_S(i)$ from this, R can deduce $K_R K_S K_R K_S(i)$. It also works the other way around. Once S receives $K_R(i + w_j)$ it knows that R has received $K_S(i + w_j, \alpha)$. S knows that R will deduce $K_R K_S K_R K_S(i)$ from this package (as above), thus $K_S K_R K_S K_R K_S(i)$. Thus, the further the tape-transmission progresses, the more information can be deduced about all parts of the currently read tape.

All these deductions rely on a simple reasoning mechanism. Suppose you are the Sender or Receiver and you receive something. This implies that the other must have sent it at some time in the past. If the other has sent it, the other must know it (then and now). I now know that the other knows it. In *psuedo-logic* (same holds for R):

$$R \text{ receives } \phi \rightarrow K_R(S \text{ has sent } \phi) \rightarrow K_R(S \text{ knew/knows } \phi) \rightarrow K_R K_S \phi$$

Proof We prove the theorem by induction on n. In the proof, we freely use three general principles.

First, a principle from tense logic: $P(\Box\varphi) \rightarrow \Box\varphi$, where P stands for "sometime in the past on this run".

Second, R and S are assumed to store all relevant information from their message history. Thus, if R knows a positive modality (like $K_S K_R K_S$) about data elements now, it will know it always in future, i.e. $K_R(\varphi) \rightarrow \Box K_R(\varphi)$ for appropriate φ, and similarly for S. This is implied in the algorithms, in which S and R actively 'store' their knowledge in memory.

Third, messages sent have the format $K_R\phi$ or $K_S\phi$. When S/R receives this, it kan be interpreted beyond the simple format of data. The knowledge-operator actually acknowledges that fact ϕ is known, now and in the future (previous principles), to the one who sent it. This is why $K_R K_S \phi$ or $K_S K_R \phi$ is stored directly after receiving a message. This receiving/storing combination is epistemically justified, and made quite explicit in the algorithms. Conclusive (similar for S):

$$R \text{ receives } K_S\phi \rightarrow R \text{ stores } K_R K_S \phi \rightarrow K_R K_S \phi \rightarrow \Box K_R K_S \phi$$

n=0 The third principle basically provides us with the **Forth**-part of the theorem for n=0.

$$R \text{ stores } K_R K_S(i, \alpha) \rightarrow \Box K_R K_S(i, \alpha)$$

Furthermore, R will only send an acknowledgement of a message if it has been received, and thus, is known. Using the first and third principle we derive:

$$R \text{ sends } K_R(i) \rightarrow P(R \text{ receives } K_S(i, \alpha)) \rightarrow P\Box K_R K_S(i, \alpha) \rightarrow \Box K_R K_S(i, \alpha)$$

S only stores acknowledgements if they have been received. If S receives an acknowledgement, it knows that R has sent it in the past.

$$S \text{ stores } K_S K_R(i) \rightarrow K_S P(R \text{ sends } K_R(i)) \rightarrow \ldots$$

We can now use one of the previously proven facts, as well as the second principle to derive:

$$\ldots \rightarrow K_S P(R \text{ sends } K_R(i)) \rightarrow K_S P(\Box K_R K_S(i)) \rightarrow K_S \Box K_R K_S(i) \rightarrow \Box K_S K_R K_S(i)$$

The first and last literals of this long formula are exactely the **Back**-part of the theorem for n=0.

induction step Suppose as induction hypothesis that **Forth** and **Back** hold for $k-1$, where $k \geq 1$. We will prove that **Forth** and **Back** hold for k itself.

Because S only moves its window forward after it has received acknowledgements about all data elements in the window, we have the following:

$$S \text{ sends } (i + \sum_{j=1}^{k} w_j, \alpha) \rightarrow P(S \text{ receives } (i + \sum_{j=1}^{k-1} w_j)).$$

We may combine this fact with the **Back**-part of the induction hypothesis and the first general principle to derive:

$$S \text{ sends } (i + \sum_{j=1}^{k} w_j, \alpha) \rightarrow \Box K_S (K_R K_S)^k (i).$$

R knows the above fact. Now if R receives a data element with position marker $\sum_{j=1}^{k} w_j$ from S, it knows that S has sent it sometime in the past which implies by the above fact and our two general principles that $\Box K_R K_S (K_R K_S)^k (i)$. Thus, we have

$$R \text{ writes } (\alpha, i + \sum_{j=1}^{k} w_j) \rightarrow \Box (K_R K_S)^{k+1} (i),$$

which is exactly the **Forth**-part of the theorem for n=k.

As in the base case, R sends an acknowledgement about the $i + \sum_{j=1}^{k} w_j$-th data element only if it received that element in the past, so we derive by our first general principle:

$$R \text{ sends } (i + \sum_{j=1}^{k} w_j, -) \rightarrow \Box (K_R K_S)^{k+1} (i).$$

S knows the above fact, so if it receives an acknowledgement about the $i + \sum_{j=1}^{k} w_j$-th data element, it knows that R has sent this in the past, so by the two general principles we conclude:

$$S \text{ receives } K_R (i + \sum_{j=1}^{k} w_j, -) \rightarrow \Box K_S (K_R K_S)^{k+1} (i),$$

which is exactly the **Back**-part of the theorem for n=k.

We assumed from the start that the input tape is infinite and that infinitely many messages from R to S and from S to R are delivered. Thus, the above theorem shows that for any n and any message, depth n knowledge of that message will eventually be reached. Subsection 5.4 shows that common knowledge of the message remains nevertheless out of reach.

5.3 Comparison with protocol B

As mentioned in the introduction, TCP could inexactly be viewed as a generalisation of Halpern and Zuck's protocol B, by allowing other windows than of size 1. As a reminder, the knowledge-based protocol B is given here in a presentation similar to the one in [9]:

Protocol B Sender's algorithm:

```
1  i := 0;
2  while true do
3        begin read(x_i);
             {Read the package with position 'i' from the tape.}
4            send(x_i; "K_S K_R(x_{i-1})") until K_S K_R(x_i);
             {Send a combined message of the data element you have just read
              and an acknowledgement of R's last acknowledgement (none if i=0).}
5            i := i + 1;
             {Set pointer 'i' to the next position on the tape.}
6        end
```

Protocol B Receiver's algorithm:

```
1  when K_R(x_0) set i:=0;
   {You can start the algorithm when the first data element has been received.}
2  while true do
   {After the initialisation has taken place, get ready to receive the tape.}
3        begin write(x_i);
             {Write the received data element with position i to your output tape.}
4            send "K_R(x_i)" until K_R(x_{i+1});
             {Send an acknowledgement of the last received package.}
5            i := i + 1;
6        end
```

In the special case where TCP operates with constant window-size 1, it will send one package at a time and the window may only be shifted if the acknowledgement for this package has been received, exactly as in protocol B.

However, there is an important difference between the two algorithms as well. When implementing protocol B, a finite message alphabet is used so that e.g. the "$K_S K_R(x_{i-1})$" messages are not distinct for all values of i. For protocol B, which is meant to work in environments where reordering problems do not occur, this does not present a problem. Afek and others have shown, however, that protocols using only a finite message alphabet can never solve the sequence transmission problem in environments where both reordering and deletion errors may occur, see [1]. Thus it is essential that our knowledge-based protocol for TCP uses an infinite message alphabet or messages growing in size.

In the previous section we discussed Halpern and Zuck's result about the eventual attainment of depth 4 knowledge when using protocol B. It is surprising to note that their result can be strengthened to eventual attainment of depth n knowledge, similarly as for TCP. Because of the problems mentioned above, it is an essential condition that no reordering problems occur. We also make the usual assumption that the message that encodes (x_{i+1}; "$K_S K_R(x_i)$") sent by S is different from the code of its predecessor (x_i; "$K_S K_R(x_{i-1})$"), and that R's message encoding "$K_R(x_{i+1})$" differs from the code of the previous "$K_R(x_i)$".

Theorem 3 *Let R be any set of runs consistent with knowledge-based protocol B, where:*

- *Messages are never reordered;*
- *The safety property holds (so that at any moment the sequence Y of data elements received by R is a prefix of the infinite sequence X of data elements on S's input tape);*
- *S's state records all data elements it has read and acknowledgements it received;*
- *R's state records all the data elements it has written.*

Then for all runs in R and all $n \geq 0, i \geq 0$ the following hold:

[Forth]: *R writes $x_{i+n} \rightarrow \Box(K_R K_S)^{n+1}(x_i)$.*

[Back]: *S receives "$K_R(x_{i+n})$"$\rightarrow \Box K_S(K_R K_S)^{n+1}(i)$.*

Proof By induction on n. The proof is completely analogous to the proof of theorem 2 with a fixed window-size of 1, as is the case in protocol B. In other words, for all j, we assume $w_j = 1$.
Using this, we rewrite the summation over the window sequence as:
$\sum_{j=1}^{n} w_j = \sum_{j=1}^{n} 1 = n$. Replacing $\sum_{j=1}^{n} w_j$ with n in the proof of theorem 2 and rewriting the messages (where sequence number and window-size are left out) gives us the proof of theorem 3.

The extra condition of this theorem, namely the absence of reordering problems, enables S and R to correctly interpret the position of new incoming messages. For example, suppose that R has already written data elements in the positions $0, \ldots, i$ on its output tape and it receives a message from S that differs from S's previous message. Then R will interpret the new message as (x_{i+1}; "$K_S K_R(x_i)$"), even though i is not explicitly encoded in the message, in contrast to the case for TCP.

Note that for n=1, theorem 3 gives the depth 4 knowledge that was shown to be eventually attained in the conference version of [6].

5.4 Negative result: bounds on the depth of knowledge

In subsection 5.2, we proved that for any n, an n-fold depth of knowledge about messages among R and S is eventually realized when using TCP. This might lead one to hope that also infinite depth knowledge, that is, common knowledge, might be attainable when using TCP. However, even when using TCP, the two processors will at any moment only have a finite depth of knowledge about each message passed between them. The exact bound is given by the following theorem.

Theorem 4 *Let R be any set of runs consistent with the knowledge-based TCP protocol.*

Let w be the window-size. Then for all runs in R and all $n \geq 0, i \geq 0$ the following hold:

[Bound-R]: *S reads $(i + \sum_{j=1}^{n} w_j, \alpha) \rightarrow \neg(K_R K_S)^{n+1}(i)$.*

[Bound-S]: *R stores $K_S K_R(i + \sum_{j=1}^{n} w_j, \alpha) \rightarrow \neg K_S(K_R K_S)^{n+1}(i)$.*

Proof We prove the theorem by induction on n. In the proof, we freely use two general principles. The first is that both R and S obey positive introspection, i.e. $K_R\varphi \rightarrow K_R K_R\varphi$ and $K_S\varphi \rightarrow K_S K_S\varphi$ are valid. We use the contrapositions of both principles below. The second principle, well-known from modal logic, is that if $\varphi \rightarrow \psi$ is valid, then so are $\neg K_R\neg\varphi \rightarrow \neg K_S\neg\psi$ and $\neg K_S\neg\varphi \rightarrow \neg K_S\neg\psi$.

n=0 It is clear that at the moment S reads (α, i), then, according to the algorithm, S has not yet sent any message about the i-th data element to R who cannot know about it in any other way, so indeed $\neg(K_R K_S)(i, \alpha)$, which is the **Bound-R**-part of the theorem for n=0. On the other hand, at the moment that R writes (α, i), it has not yet sent S any acknowledgement about the i-th data element, so S does not know that R knows that S knows the i-th data element: $\neg K_S(K_R K_S)(i, \alpha)$, which is the **Bound-S**-part of the theorem for n=0.

induction step Suppose as induction hypothesis that **Bound-R** and **Bound-S** hold for $k-1$, where $k \geq 1$. We will prove that **Bound-R** and **Bound-S** hold for k itself.

Suppose that S reads $(i + \sum_{j=1}^{k} w_j, \alpha)$, then its window has just moved to position $i + \sum_{j=1}^{k} w_j$. At this moment, S has not yet sent any message about the data element on position $i + \sum_{j=1}^{k} w_j$ to R, so R doesn't know that S's window does not contain $i + \sum_{j=1}^{k-1} w_j$ anymore. Therefore, R does not know whether S received R's message $(\text{ack}, i + \sum_{j=1}^{k-1} w_j)$ yet. Supposing that the $i+\sum_{j=1}^{k-1} w_j$-th data-element is β, this implies that R does not know that S knows that R is not at this very moment writing $(\beta, i + \sum_{j=1}^{k-1} w_j)$, or more formally: $\neg K_R K_S \neg(R \text{ writes } (\beta, i + \sum_{j=1}^{k-1} w_j))$. By the **Bound-S** part of the induction hypothesis and our second principle, this implies $\neg K_R K_S K_S (K_R K_S)^k(i)$. Using positive introspection axiom for S, this implies $\neg (K_R K_S)^{k+1}(i)$, which is exactly the **Bound-R**-part of the theorem for n=k.

Now suppose that R writes $(i + \sum_{j=1}^{k} w_j, \alpha)$, then S has not received any acknowledgement about the $i + \sum_{j=1}^{k} w_j$-th data-element yet, so S does not know that R knows that S is not at this very moment reading $(\alpha, i + \sum_{j=1}^{k} w_j)$, or formally $\neg K_S K_R \neg(S \text{ reads } (\alpha, i + \sum_{j=1}^{k} w_j))$. So, using our second principle and the **Bound-R**-part for n=k that has just been proved, this implies that $\neg K_S K_R (K_R K_S)^{k+1}(i)$. Now, using the positive introspection for R, this gives $\neg K_S (K_R K_S)^{k+1}(i)$, which is the **Bound-S**-part of the theorem for n=k.

6 Conclusions

In this article we have shown that an essential body of a real-life protocol such as TCP can be modeled by knowledge-based algorithms. These algorithms can be analysed to determine the robustness of the protocol. The following questions can then be studied. Can mutation, deletion or insertion errors be handled by this protocol? If not, what are the practical consequences? How much knowledge can the protocol attain about delivered data?

One of the results of this analysis has been the proof that the sender and the receiver using TCP can acquire depth n knowledge about the values of the messages for any n. The proof technique can also be applied to protocol B, one of the first protocols that were analysed using epistemic logic; thus far, it was assumed that protocol B would give rise to knowledge up to a depth of only four. On the negative side, it had been proven that protocol B could never ensure common knowledge between the sender and the receiver. We have shown that this result also holds for TCP, by giving specific bounds on the depth of knowledge at each moment of data transmission.

As to further research, it would be interesting to investigate whether the correctness results of section 4 may be extended. We conjecture, for example, that an environment with any two kinds of deletion, mutation, and insertion errors can be handled by implementing the knowledge-based algorithm for TCP using encodings similar to [6].

Applying knowledge-based techniques to other protocols could also yield interesting results. The User Datagram Protocol mentioned briefly in section 2, for instance, might have an entirely different *epistemic character*, as it has not been built to guarantee a perfect delivery.

The Internet has been the subject of our research, but also the means by which some of the results have been presented. The Internet has proven to be a useful tool in the clarification and visualisation of certain concepts. The Internet and its techniques are readily available, so we hope it will inspire others to do likewise.

Acknowledgements

We would like to thank Egon Baars and Jeroen Meijer, who (together with author Freek Stulp) were contributors to the student project which inspired the writing of this article. Also thanks to Bill Bitner who answered some of our questions about TCP as it works in practice. Finally, the remarks of the anonymous referee and the editors of this special issue were very helpful and even pointed us in the right direction to find some new results.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. J. Fischer, N. Lynch, Y. Mansour, D. Wang and L. D. Zuck, Reliable communication over unreliable channels, *Journal of the ACM*, Vol. 41, No. 6 (1994), pp. 1267–1297.

[2] A.V. Aho, J.D. Ullman, A.D. Wyner, and M. Yannakakis, Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, vol. 8, nr. 3 (1982), pp. 205-214.

[3] B. Bitner and R. White, *Care and Feeding for Better VM TCP/IP Performance*, IBM Endicott, 1992.
http://vmdev.gpl.ibm.com/dEVPAGES/Bitner/presentations/tcpip/ipcare.html

[4] D.E. Comer, *Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture*, Upper Saddle River, Prentice Hall, 2000.

[5] R. Fagin, J.Y. Halpern, Y. Moses and M.Y. Vardi, *Reasoning about Knowledge*, Cambridge (MA), MIT Press, 1995.

[6] J.Y. Halpern and L.D. Zuck, A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM* 39, nr. 3 (1992), pp. 449-478. Earlier version appeared in: *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987, pp. 269-280.

[7] J.Y. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment, *Journal of the ACM* 37, nr. 3 (1990), pp. 549-587.

[8] R. van der Meyden, Common knowledge and update in finite environments, *Information and Computation* 140, No. 2 (1998), pp. 115-157.

[9] J.-J. Ch. Meyer and W. van der Hoek, *Epistemic Logic for AI and Computer Science*, Cambridge, Cambridge University Press, 1995.

[10] M. Miller, *Troubleshooting TCP/IP: Analyzing the Protocols of the Internet*, San Mateo (CA), Prentice-Hall, 1992.

[11] F. Stulp, Visualisation of the Transmission Control Protocol, http://tcw2.ppsw.rug.nl/~freeks/tcp/

[12] K.Washburn and J.T. Evans, *TCP/IP: Running a Succesful Network*, Addison-Wesley, 1993.