Kenneth J. Turner and Iain A. Robin
(pre-publication version, copyright Elsevier Science).
An Interactive Visual Protocol Simulator, Computer Standards and Interfaces,
23:279-310, October 2001.

An Interactive Visual Protocol Simulator

Kenneth J. Turner and Iain A. Robin
Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, UK
Email kjt@cs.stir.ac.uk and iain_robin@hotmail.com

20th July 2001

Abstract

A flexible protocol simulator is presented that supports interactive visual experimentation with protocols. The simulator is modular and allows ready addition of new protocols. Use of Java permits protocol simulations to be delivered via the web, and also to be developed in an object-oriented manner. A range of simulations is presented, covering link-level protocols to application-oriented protocols. The Trivial File Transfer Protocol is used to illustrate in detail how a protocol simulation is implemented.

Keywords: ABP (Alternating Bit Protocol), BOOTP (Boot Protocol), Communications, Computer Science Education, HTTP (HyperText Transfer Protocol), IP (Internet Protocol), Java, Protocol, Simulation, SMTP (Simple Mail Transfer Protocol), SWP (Sliding Window Protocol), TCP (Transmission Control Protocol), TFTP (Trivial File Transfer Protocol), UDP (User Datagram Protocol), Visualisation

1 Introduction

1.1 Simulator Philosophy

Protocols play a key role in all communications, but it is surprisingly difficult to investigate them effectively. Static aspects such as message types and formats are easily understood. But protocols are inherently dynamic, so a presentation of their behaviour is often limited to general principles or to specific scenarios. The authors believe that protocol operation is best grasped through a combination of specification and hands-on investigation. Conventional protocol analysers offer a limited means to study protocols dynamically, so the authors have created the protocol simulator JASPER: Java Simulation of Protocols for Education and Research.

The focus on education and research has several implications for the simulator design. Since the aim is to study how protocols work rather than provide a complete implementation, it is possible to abstract from many details. For example, only abstract message formats need be presented; the actual formats are easily understood from a specification. It is also not necessary to simulate the fine details of a protocol. Efficient implementation does not need to be considered in a simulator. These simplifications allow straightforward protocol simulations to be written.

The choice of Java as the implementation language is very beneficial. With proper design, the same Java code can be run as an application as well as an applet. As an application, the simulator has additional facilities such as hard-copy print-out of a simulation run. Java offers portable graphics that can be used to illustrate protocols effectively. In particular, sequence diagrams are helpful in visualising the exchange of messages among several communicating entities. The object-oriented nature of Java makes it easy to write a simulator with 'plug-in' protocols. Protocol simulations are written in a modular manner, so adding a new protocol is easy. A protocol may specialise another one, as well as building on the protocol base classes. In fact, the use of object-oriented concepts in the simulator acts as a useful example for teaching Java.

Another important philosophy is that the simulation user should be in control. Protocol simulations are expected to manage basic activities automatically, such as acknowledging or reordering messages. However, major decisions are presented to the user through a menu of choices. For example, the simulation user should decide when to send a message or to close a connection. Since transmission problems often cause the greatest complications in protocol behaviour, the user is also allowed to control the behaviour of the medium. Thus the user can decide whether a message should be lost or fragmented during transmission.

It was envisaged that JASPER would be used in a number of ways:

- It can be used as a fixed collection of well-known protocols as an aid to teaching networking. To this end, the simulator is available as a set of pre-compiled classes that just need a Java-enabled web browser.
- It can be treated as an extensible simulator to which students can readily add new protocols. The idea here is to allow students to learn about protocols by implementing them in a simplified setting.
- It can be used to illustrate object-oriented design principles at work. The examples commonly used to explain object orientation can be somewhat hackneyed. JASPER is an object-oriented case study rooted in a realistic application. It can thus serve as an adjunct to a Java programming course.
- It can be regarded as a core simulator for researching new protocols. The extensibility and visualisation permitted by the simulator are important factors in achieving this.

To encourage its use in other institutions, the simulations described in this article have been made freely available [25]. The simulator can be tried out online. The downloadable code is precompiled so that anyone with just a web browser can use it immediately. The source code is also provided to permit modification and further development for new protocols.

1.2 Simulator Advantages

In an educational setting, the simulator has several advantages:

- The portable nature of Java means that simulation applets can easily be executed on a variety of platforms. This includes the typical mix of equipment found on a University campus as well as in students' own homes. Web-based delivery simplifies software maintenance as the simulator is distributed from a single server. The approach also lends itself to distance learning.
- The simulator is interactive and entirely under user control. This allows to students to experiment directly with a protocol. Instead of being limited to scenarios prepared by a lecturer, a student can explore aspects where greater understanding needs to be gained. Learning then becomes a student-centred activity. It is also possible to investigate unusual circumstances that do not arise in 'main path' behaviour.
- The visual nature of the simulation makes it easy for students to grasp what is happening and to control the simulation.
- The object-oriented nature of the simulator makes it a useful example for teaching Java. Java is commonly taught in computing degrees. Students should therefore be able to understand and to extend the simulator with new protocols or variations on existing protocols. In fact the development of a protocol simulation using JASPER is a good student exercise, both in learning a protocol and in learning Java.

In a research setting, the simulator has similar advantages:

- It is easy to develop throw-away simulations of a protocol as it is being designed. The functionality and the integrity of a protocol can be studied before its design is frozen.
- Interactive simulation means the user can control the exploration of a protocol in detail. Although automated exploration is possible with the simulator, this kind of investigation tends to be blind. Instead, user-directed simulation can be used to study unusual protocol situations and awkward 'corners'. Protocols often follow an '80:20 rule' that means much of their complexity is not normally exercised.
- The visual nature of the simulation helps to clarify the protocol behaviour. This is particularly valuable when complex protocol exchanges have to be considered. Diagrammatic representations are common in protocol research; they are an automatic by-product of JASPER.
- Although protocols have to be described in Java according to certain simulator interfaces, the bulk of a protocol description can be written quickly from a state machine. Since state machines or labelled transition systems are commonly used in formal specifications of protocols, a simulation can be obtained fairly readily from a (constructive) formalisation.

1.3 Related Work

Protocol simulation as considered here has received surprisingly little attention. A large number of network simulators[1] support modelling and performance analysis of networks and protocols. The goal is usually to study performance issues rather than to discover how protocols work. A major example is *ns/nam* (Network Simulator/Network Animator [16, 26]) that supports the analysis of (queueing) network models. A number of ready-made simulations are available for Internet and wireless protocols. As another example, the *cnet* (computer network) simulator [14, 15] can be used to model a network and then to view the statistical performance of the network as a whole. A number of network simulations have been developed under the auspices of the VINT (Virtual Internet) project. As a whole, these network simulators differ from JASPER in tackling performance rather than functionality issues.

Work on formal modelling of protocols often leads to protocol simulation. Such simulators are usually aimed at discovering faults in protocols rather than explaining how they work. Obviously there is the possibility of combining both goals. JASPER can in fact be used in protocol research, assisted by the visualisation of behaviour. However it has no notion of verification as with formal approaches. Formal methods are often weak on visualisation of behaviour. SDL (Specification and Description Language [9]) is unusual in that it can display behaviour graphically using MSCs (Message Sequence Charts [8]).

Another class of simulators deals with distributed algorithms, but these emphasise algorithm rather than protocol design. An example is [12] that includes an animation of the Alternating Bit Protocol using sequence diagrams. However simulation oriented towards algorithm design is quite different in scope from JASPER.

Other protocol simulators seem to be one-off developments. JavaScript has been used in conjunction with SDL to illustrate the operation of a simple data transfer protocol [1]. A Java applet [5] has been developed to simulate the ATM UNI (Asynchronous Transfer Mode User-Network Interface). This provides a graphical simulation of how signals are exchanged between the user and the network, together with textual output of protocol behaviour.

There do not seem to be many protocol simulators in the style of JASPER. DLPSIM (Data Link Protocol Simulator [10, 11]) specifically focuses on the data-link layer. Protocols may be coded in C and then exercised using a scripting language with simulated traffic to check for correct operation. The Pascal-based tool in [13] uses a scripting language to animate protocol interaction diagrams. In conclusion, JASPER seems to be almost alone in its goals and capabilities.

1.4 Structure of the Paper

The paper reflects the multiple uses described in section 1.1. It presents the simulations of a number of well-known protocols, in order to give a feel for the range and capability of the simulator. A detailed case study shows how to implement a 'new' protocol. The structure of the simulator is explained, as an example of object-oriented principles applied to protocol simulation.

Section 2 explains the organisation of the simulator and its major components. To show the capabilities of the simulator, sections 3 to 6 present a number of simulations ranging from link protocols to application-oriented protocols. The aim is to illustrate the range of protocol simulations supported by JASPER, without explaining the underlying protocols in too much detail. Appendix A illustrates the approach by showing how the Trivial File Transfer Protocol simulation has been implemented.

2 Simulator Organisation

This section explains the organisation of the simulator.

2.1 Overall Structure

The simulator package is designed according to the familiar MVC paradigm:

Model: The model of a protocol is its behaviour defined by a set of Java classes. This is concerned purely with the operation of the protocol. Although any style could be adopted for coding protocols, a state-based style is particularly appropriate. This is commonly used in protocol standards, and forms the basis of specification

[1] So-called network simulators are often discrete event simulations of queueing network models. They are often, but not necessarily, applied to the study of communications networks.

languages like ESTELLE [7] and SDL. State machines are easily represented in imperative programming languages. The major state is an enumerated type (a list of **final int** values in Java). Minor state variables are simply other instance variables. The state machine is encoded as a table, or as a collection of **if**/**switch** statements (as done in JASPER). The advantage of this approach is that implementing a protocol from a typical standard is straightforward. Several of the JASPER simulations were coded using ESTELLE or SDL specifications. It is also easy for a student to see the connection between the state machine representation of a protocol and its Java equivalent.

View: The view of a protocol is the graphical display of its behaviour. Several kinds of view were investigated for JASPER including MSCs. Currently the only supported view is the TSD (Time Sequence Diagram). Such diagrams are commonly used to illustrate protocols, although the notation varies among authors. The variant supported by JASPER is one developed by the principal author to support his teaching. It resembles the Service Conventions standard [6] used for OSI (Open Systems Interconnection). Time Sequence Diagrams have columns corresponding to the communicating parties, such as two protocol entities and the communications medium. More complex simulations may involve several protocol entities and also the protocol users. Time runs down the diagram, and sloping arrows show the exchange of messages. A diagram may be annotated to show conditions such as message loss, timeout and flow control blockage.

Controller: The base simulator class acts as the overall controller. It coordinates the activities of the communicating elements (service users, protocol entities, communications medium). The protocol rules determine which actions are offered to the user. The user's choice decides the next step in a simulation. The user may also undo or redo a previous choice so as to explore a tree of behaviours. As a protocol's behaviour unfolds, its actions are displayed using a Time Sequence Diagram.

2.2 Protocol Simulation Interfaces

Implementing a protocol simulation requires a number of classes to be realised for a new protocol. The abstract *Protocol* class provides basic functionality to manage the protocol entity classes, and must be subclassed for each protocol. The individual protocol entities are modelled by classes that implement the *ProtocolEntity* interface. Note that service users and the communications medium are also treated as implementations of *ProtocolEntity*. This allows all communicating objects to be treated in the same way. The *ProtocolEntity* interface requires the following methods to be realised:

initialise is called when simulation starts or is restarted; its main task is to initialise state variables

getName returns the name of an entity for display at the top of a Time Sequence Diagram column

getServices returns the services (actions) that an entity can currently perform; these are presented as menu choices to the tool user

performService notifies an entity of the user's choice; the entity performs the action and evolves to the next decision point in its behaviour

receivePDU notifies an entity that it has received a PDU (Protocol Data Unit)

setPeer notifies an entity of its peer entity

transmitPDU is called by an entity to transmit a PDU to another entity.

Timeouts defines an interface for handling message timers. Since the aim of simulation is mainly educational, the simulation user is allowed to control timeouts through menu choices. By deciding to time out a message, the user can explore protocol error recovery. Indeed this often uncovers the most interesting aspects of a protocol's design. The code for a protocol declares whether it uses timeouts. These are normal in lower-level protocols but not in higher-level ones. A protocol with timeouts (re)sets timers at appropriate points in its behaviour. Since the simulation is not meant to run in real-time, the actual value of a timeout is not modelled. A protocol entity that uses timers implements the following interface methods:

hasTimer is called with a PDU type to determine whether the protocol supports timeouts for this particular kind of PDU

setTimer informs an entity of the timer status for a particular PDU; the entity is expected to note and use this in deciding whether to retransmit on timeout, and to cancel a timeout on acknowledgement.

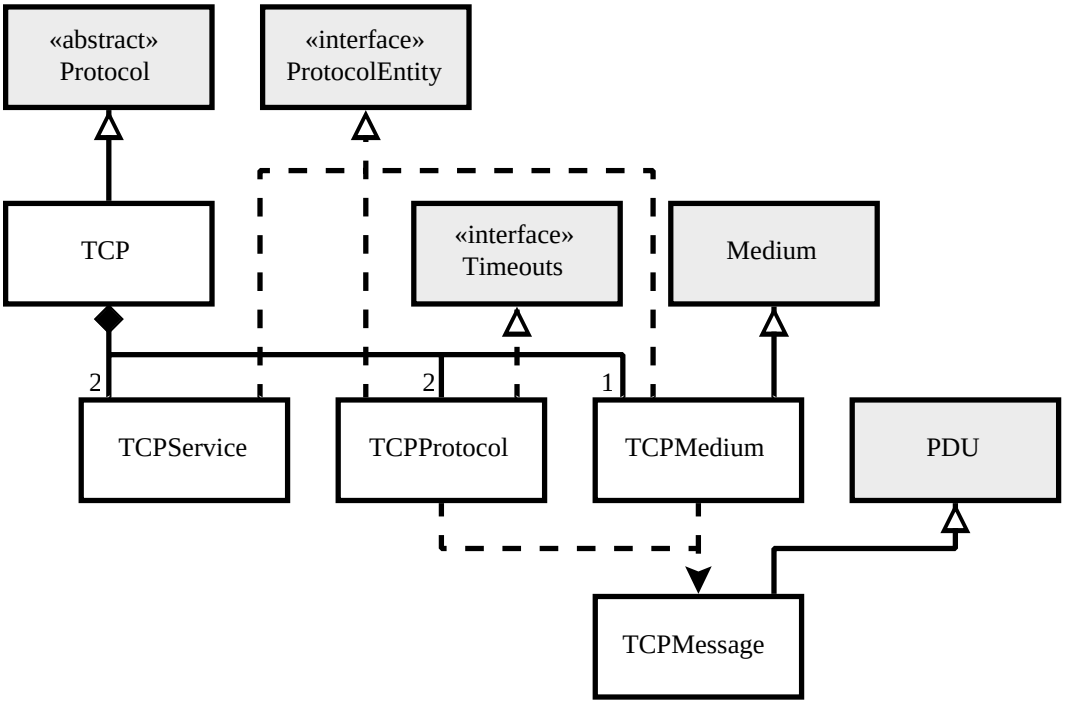

Figure 1: Outline of Classes for simulating TCP

The *Medium* base class describes the generic properties of a communications medium. Although it suffices for many protocols, it may be subclassed for particular cases. For example, message reordering in the medium must be added for IP (Internet Protocol) and TCP (Transmission Control Protocol). In general, inheritance allows for a hierarchy of medium classes.

Protocol entities and the medium are coordinated by exchange of *ProtocolEvent* values:

send a service user sends a message to a protocol entity
deliver a protocol entity delivers a message to a service user
transmit a protocol entity sends a message to the communications medium
receive a protocol entity receives a message from the communications medium
timeout causes a protocol entity to retransmit an unacknowledged message
lose loses a message within the communications medium
fragment fragments a message within the communications medium.

In addition there is a special *comment* event. This is used to explain why some activity is happening. For example, an entity may mention that its window is now unblocked or that it has retransmitted on timeout. This information is purely for the information of the simulation user, but is helpful in understanding what a protocol is doing. Comments are generated by the code defining a protocol.

2.3 Simulator Classes

The *SequenceDiagram* base class describes a generic view of protocol behaviour. In the present implementation of JASPER, only the *TimeSequenceDiagram* subclass exists. However the design of the simulator allows for other graphical views. These can be used as alternatives or together.

The *Simulator* class controls the overall behaviour of the protocol simulations. Because of the clean interfaces among the components, the core simulator does not need to know the actual protocol or medium behaviour. It simply drives these in a standard way.

The *PDU* base class represents protocol messages. PDU formats are important in practice, but for simulation can be abstracted to the key message fields. The base class allows for a message type, the source and destination protocol entities, a sequence number and an SDU (Service Data Unit or user message). Since the real data is

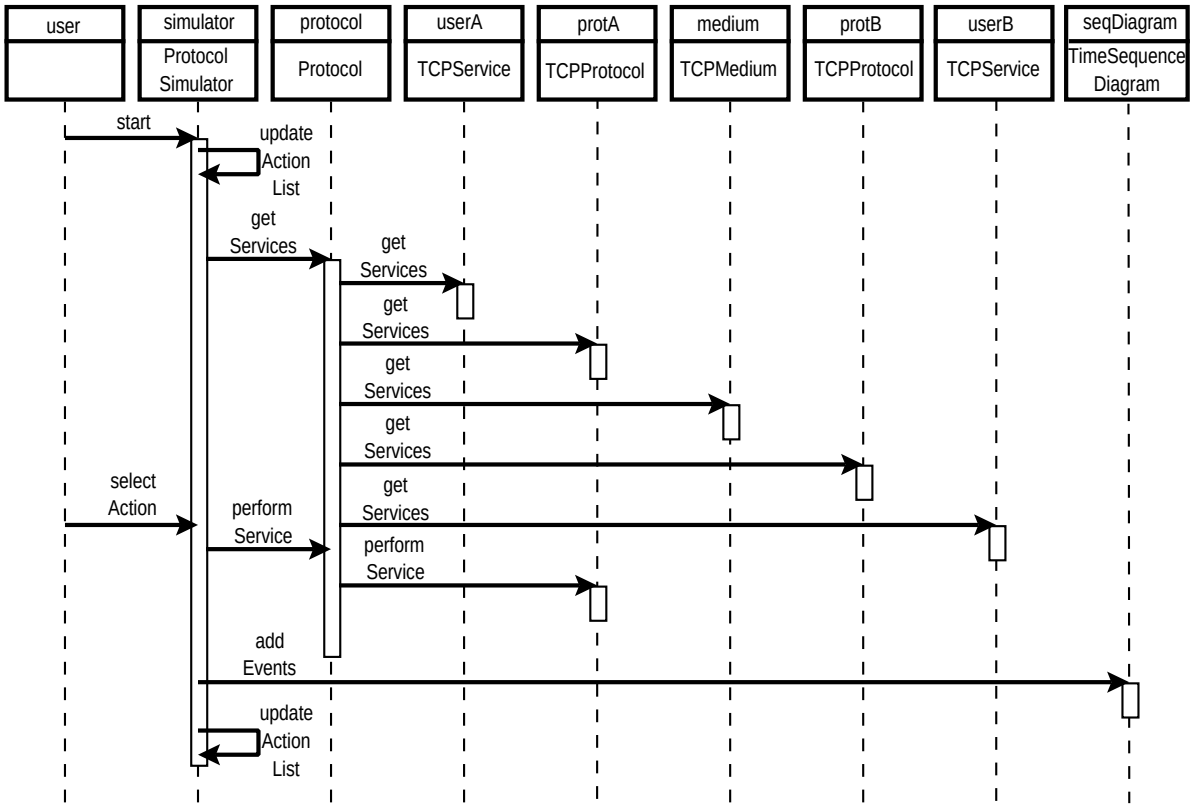

Figure 2: Interactions among Objects simulating TCP

largely unimportant for illustrating a protocol, the detailed content can be ignored in a simulation. For protocols that handle misordering or fragmentation of messages, it is necessary to keep track of data in an abstract way. Data content can be identified by generic labels like D0 or D1, or by the position of data in the stream. Protocols with specialised PDUs (like TCP) subclass *PDU*.

2.4 Simulator Example

As an example, figure 1 gives an outline of the main classes used to simulate TCP. The shaded classes are provided by the simulator; the others are specific to TCP. At the top level, an instance of *Protocol* such as *TCP* acts as a protocol manager. It is responsible for creating and coordinating its subsidiary protocol entities. The entities supporting a protocol conform to the *ProtocolEntity* interface. For TCP these are the service users (*TCPService*), the protocol entities proper (*TCPProtocol*), and the underlying medium (*TCPMedium*). An entity may also implement the *Timeouts* interface if it needs to run timers on protocol messages. *TCPMedium* specialises the generic *Medium* class. Both *TCPProtocol* and *TCPMedium* make use of *TCPMessage* – a sub-class of the generic *PDU*.

Exchanges among simulation objects are shown in the interaction diagram of figure 2. At each end of a connection there is an instance of *TCPService* (representing the user interface) and *TCPProtocol* (representing the protocol). *TCPMedium* represents the underlying communications medium. The *ProtocolSimulator* and *Protocol* objects use *getServices* to ask these entities which services (user choices) are currently possible. The simulator user selects one of these, causing *performService* to realise this choice with the corresponding object. During execution, a protocol may cause events such as fragmentation or delivery of a message. All significant actions are sent by *addEvents* to the *TimeSequenceDiagram* instance for display on the screen.

Figure 3 shows a screen shot of the simulator running TCP. Simulator parameters such as user message size and protocol receive window size are set in the top area. When the *Change Values* button is clicked, the simulation parameters are read and checked by JavaScript code on the web page. The parameters are then passed to the applet that is running in the window.

The buttons at the bottom left control the simulation. *Undo* undoes the last menu selection, in turn reversed by *Redo* (greyed out in the figure since nothing has been undone). *Run* allows the simulator to run in random mode,

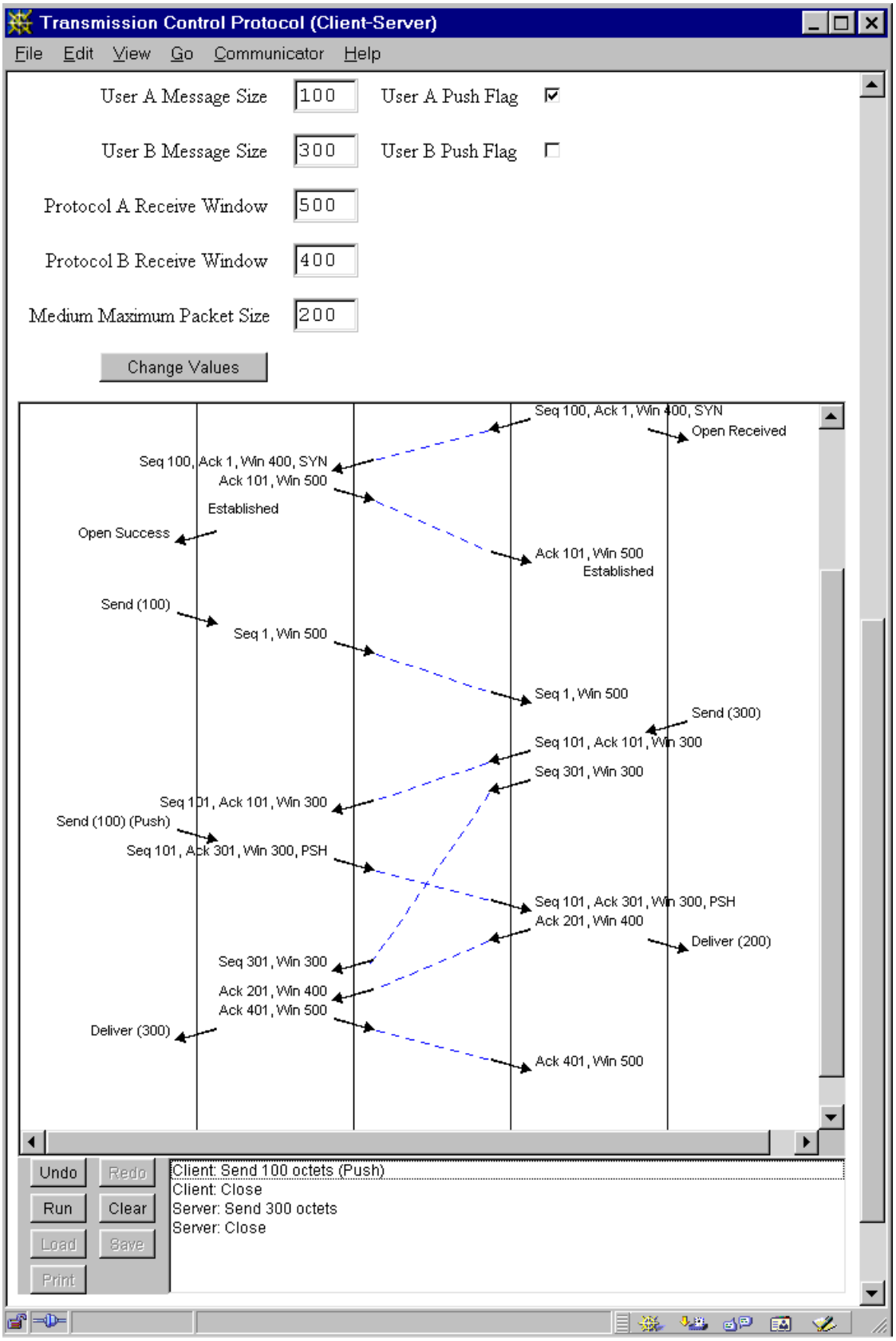

Figure 3: Sample Simulator Screen Shot

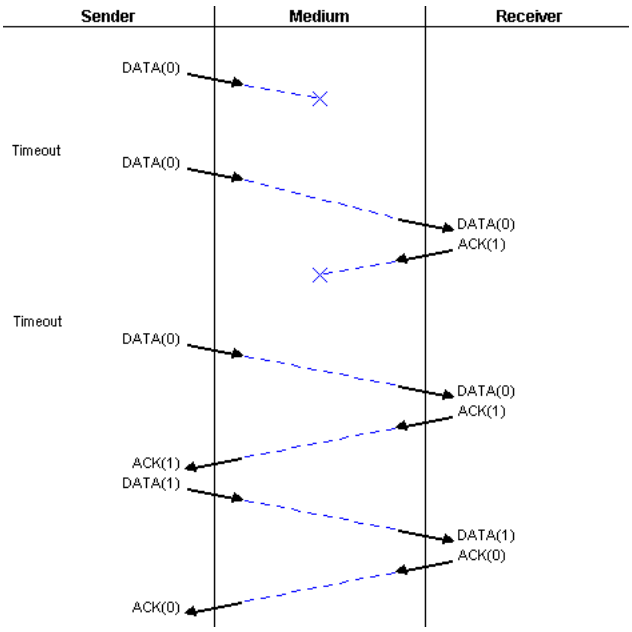

Figure 4: Alternating Bit Protocol transferring Data

making its own menu choices. *Clear* restarts the whole simulation. *Load* is used to enter a simulation scenario previously created with *Save*. *Print* is used for hard copy of the current simulation state. *Load*, *Save* and *Print* are greyed out since the simulation shown in the figure is running as an applet (and so has certain Java restrictions). When the simulator is run as an application, these actions are enabled.

To the right of the buttons are the menu choices. For example the simulation user may click on 'Client: Send 100 octets (Push)' to invoke this action. The sequence diagram will then be updated, if necessary scrolling down to the last protocol event.

The sections that now follow show the simulator in action on a range of protocols. The figures show screen-shots of a simulation window, omitting the simulation controls presented in figure 3.

3 Link Protocols

The simulations presented in this section are low-level protocols that are normally used over a single data link.

3.1 Alternating Bit Protocol

The ABP (Alternating Bit Protocol) is well-known although there are slightly different variants. The PAR protocol (Positive Acknowledgement with Retransmission) in [23, Chapter 3] was taken as the basis of this simulation. The Alternating Bit Protocol has been used as an example for many FDTs (Formal Description Techniques).

Message loss is under the control of the simulation user. After choosing to send a message, the user has the choice of delivering or losing the message in the communications medium. Due to loss of data or acknowledgement messages, the user is also presented with a timeout choice. Timeout comments are inserted as a reminder of where retransmission occurred. DATA and ACK (Acknowledgement) messages are numbered 0 or 1. The content of data messages is not important in illustrating protocol operation, so this is omitted.

The Alternating Bit Protocol is illustrated in figure 4. The sender made three attempts to transfer message 0 before succeeding. Message 1 was then sent successfully.

3.2 Sliding Window Protocol

The SWP (Sliding Window Protocol) is another well-known protocol. [22] is an early analysis of its formal properties. Many other Formal Description Techniques have been used to study this protocol, e.g. [24, Chapter 7].

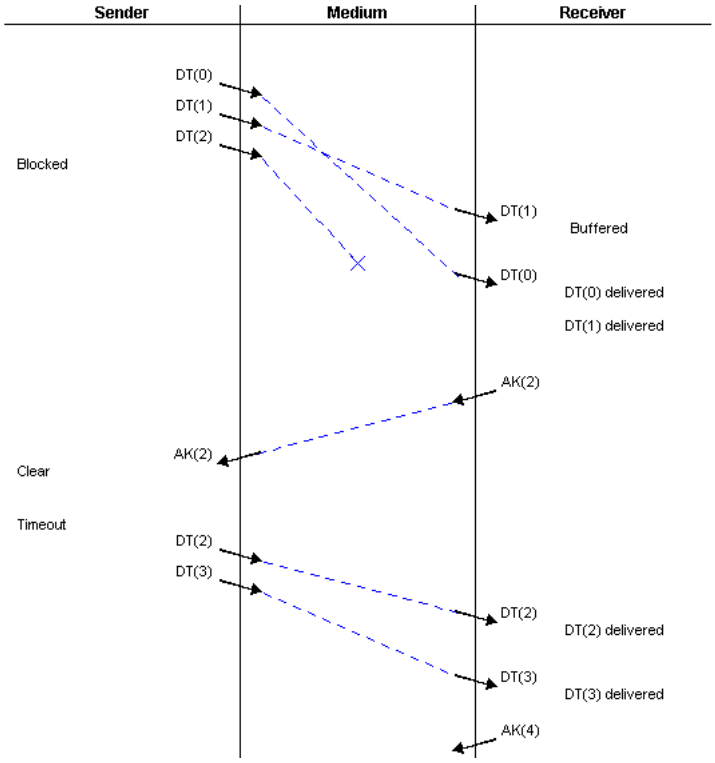

Figure 5: Sliding Window Protocol exercising Flow Control

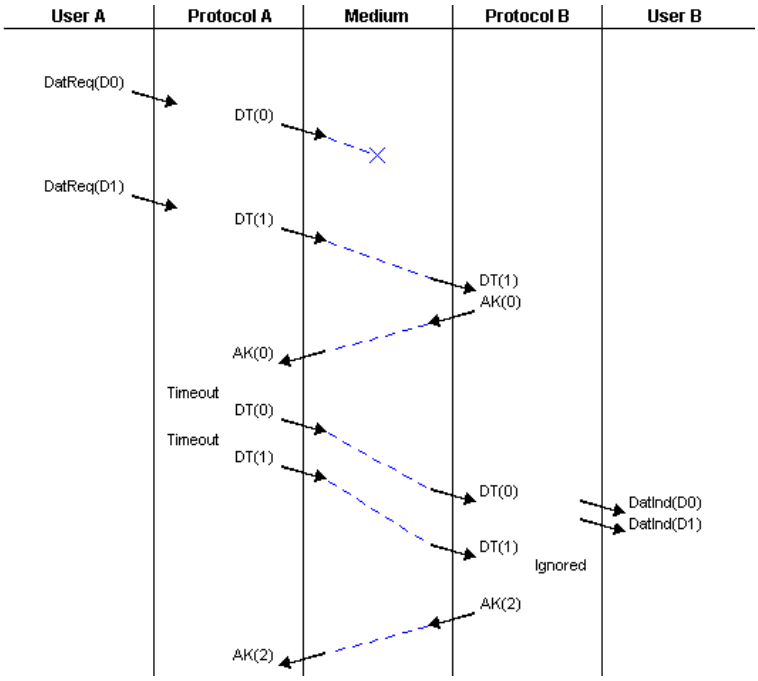

Figure 6: Sliding Window Protocol recovering from Message Loss

DT and AK (Data and Acknowledgement) messages carry sequence numbers. Although the protocol strictly uses unbounded sequence numbers, it has been shown that it can operate correctly if these are bounded. As in many related protocols such as HDLC (High-Level Data Link Control) and X.25 (packet-switching), sequence numbers are calculated modulo 8 in this simulation. A window size (3 by default) limits the number of outstanding messages waiting for acknowledgement. Setting the window to higher values (such as half the modulus or one less than the modulus) teaches the student important consequences of having a bounded sequence number.

The Sliding Window Protocol simulation is illustrated in figures 5 and 6. These show three-column and five-column diagrams, driven by the same protocol model. In figure 5, the transmitter sent three messages and then became blocked as the window was full. Messages 0 and 1 were misordered, so message 1 was buffered on receipt. When message 0 arrived, both it and message 1 could be delivered. However message 2 was lost. The acknowledgement therefore stated that message 2 was expected. Flow control restrictions were cleared at this point, but message 2 had to be resent on timeout. Message 3 was also sent. Both were delivered, so the acknowledgement cited message 4 as the next expected. Figure 6 is simpler in that it just shows several retransmissions on timeout.

4 Network Protocols

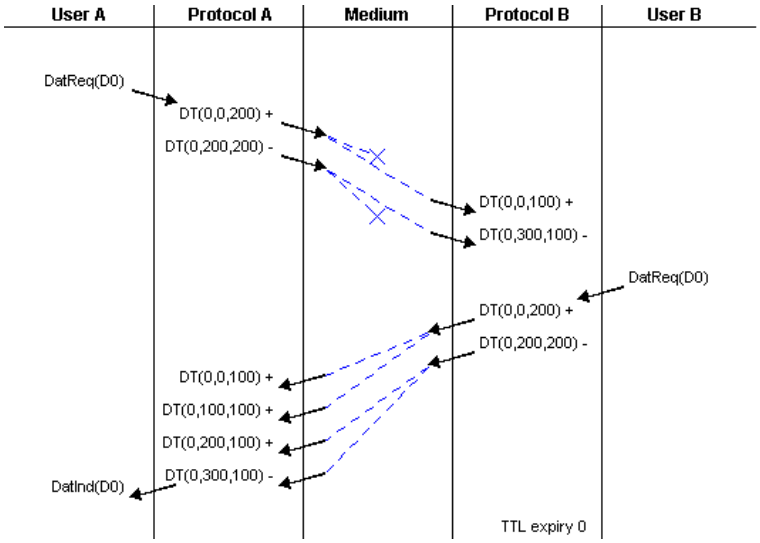

Figure 7: Internet Protocol showing Message Transmission Despite Loss

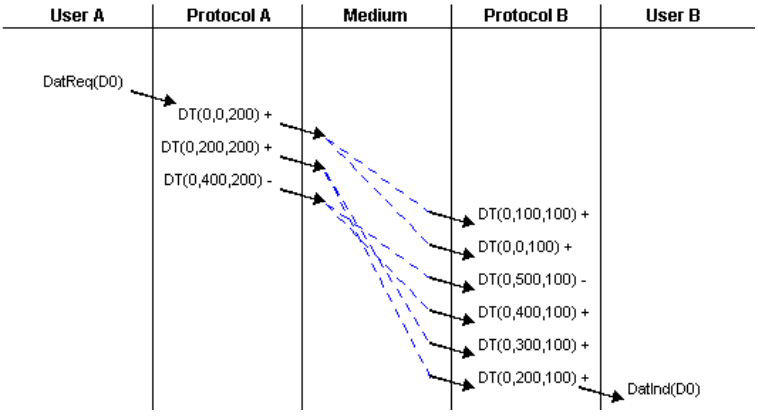

Figure 8: Internet Protocol showing Message Transmission despite Reordering

IP (Internet Protocol [3, 18]) is probably the most important network protocol in current use. The simulation is sufficiently high-level that it describes both version 4 and version 6. The IP header is complex as it allows

for many control fields and options. From an educational point of view, the key feature of IP is its handling of fragmentation and reassembly. The simulation therefore presents the transfer of data between a pair of service users whose addresses are unimportant for simulation purposes.

A DatReq or DatInd service primitive carries a quantity of data labelled symbolically as*n*. Within the protocol, DT messages are shown as carrying a message identifier *n*, the fragment offset and the fragment length. More fragments or last fragment is indicated by a + or − after the PDU. The protocol may fragment a user message if this exceeds the protocol message size. Fragmentation may also occur inside the communications medium. Fragment loss and reordering are also possible. Since possible transmission outcomes are much more complex with IP than with other protocols, the message loss and fragmentation are not controlled by the simulation user on a per-message basis. Instead, a probability is set for these characteristics along with other simulation parameters like message and fragment sizes. This allows the simulation to make choices about what happens to a message or its fragments. If necessary, the user can change the simulation parameters between simulation steps.

The simulation also supports the concept of TTL (Time To Live). Since simulation is not performed in real time, an actual value for this period is not meaningful. Instead the user may choose to let the TTL expire for a message. This gives insight into the complications of fragmentation and reassembly. Reassembly of message fragments is handled automatically by the simulation. If a message fragment is lost, the simulation user will find that delivery is not possible – only expiry of the message.

The Internet Protocol simulation is illustrated in figures 7 (user message size 400 octets) and 8 (user message size 600 octets). These simulations set the protocol message size to 200 octets, and the medium fragment size to 100 octets. The probability of message loss or reordering was set to 0.15. Fragmentation occurred both within the protocol and within the medium due to the differing maximum message sizes. In figure 7 the first and fourth fragments of the message from User A were lost, so the whole of message 0 eventually expired and was forgotten (*TTL Expiry 0*). The message sent by User B was fragmented but delivered without loss or reordering. Figure 8 shows the protocol reassembling misordered fragments. The user message was split into three protocol messages, and each of these into two medium fragments. These arrived in the order 1, 0, 5, 4, 3, 2 and were reassembled for delivery to the receiving user.

5 Transport Protocols

The simulations considered in this section deal with protocols that transport data end-to-end.

5.1 User Datagram Protocol

UDP (User Datagram Protocol [17]) is a widely used connectionless protocol. Datagrams are shown as carrying the source port, destination port and a symbolic label for data. The simulation user can set specific values for port numbers.

Figure 9 illustrates the User Datagram Protocol for transmission of datagrams between arbitrarily chosen port numbers (2430 to 1968, 1095 to 1624). The first message was lost, while the second two crossed independently in the network.

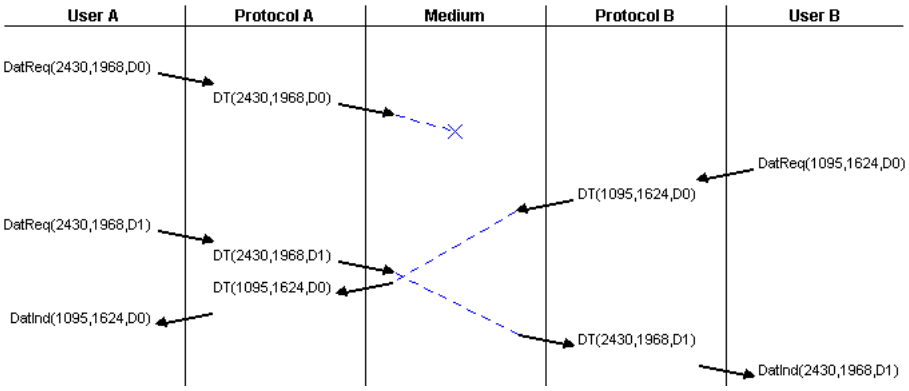

Figure 9: User Datagram Protocol showing Datagram Transmission

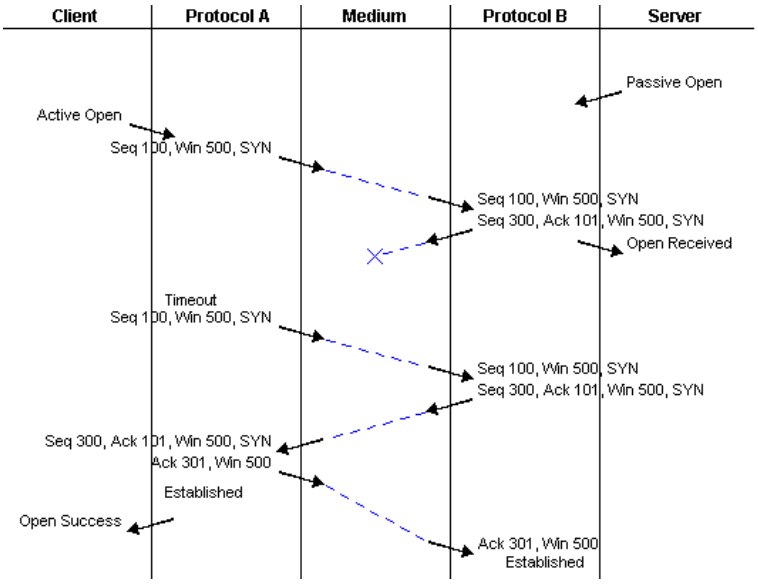

Figure 10: Transmission Control Protocol showing Client-Server Connection Setup

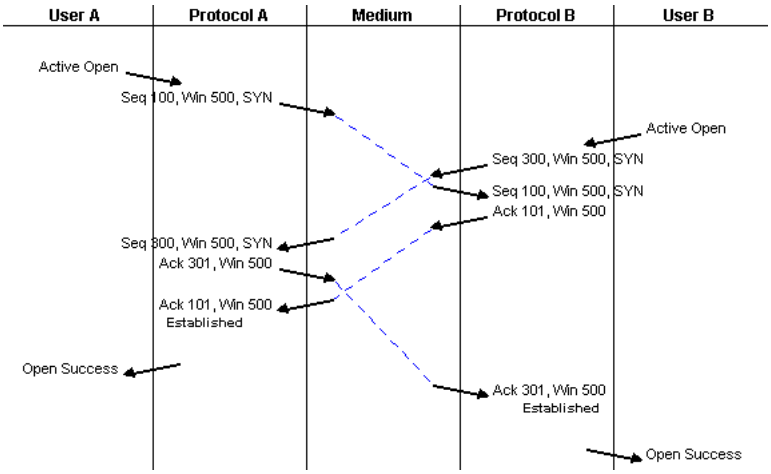

Figure 11: Transmission Control Protocol showing Peer-Peer Connection Setup

5.2 Transmission Control Protocol

TCP (Transmission Control Protocol [19]) is also well known, and is the most complex protocol simulation implemented so far in JASPER. Both client-server and peer-peer simulations are supported. These differ in that a server passively waits for a connection, while a client or peer actively opens a connection. TCP messages may contain a message sequence number, acknowledgement sequence number, window size and flags. The ACK (acknowledgement), FIN (finish – disconnect), PSH (push – deliver now), SYN (synchronisation – connect) and RST (reset) flags are supported by the simulation. For simplicity, URG (urgent) is not supported. Protocol messages are constructed in response to user requests such as Active Open, Send (with optional Push) and Close. Data is not explicitly identified. Instead just the offset and length of data are indicated. The simulation user may set various parameters such as the size of user messages and protocol fragments.

TCP is illustrated in figure 3 and in figures 10 to14. These diagrams should be read separately as they represent different connections with different parameters. Figure 10 shows client-server connection setup, with the client recovering from loss of the acknowledgement from the server. Figure 11 shows peer-peer connection setup, with the complication that both peers tried to set up a connection at the same time. Fragmentation and error recovery appear in figure 12. The user message of 500 octets was fragmented into 250 octet packets. The first fragment

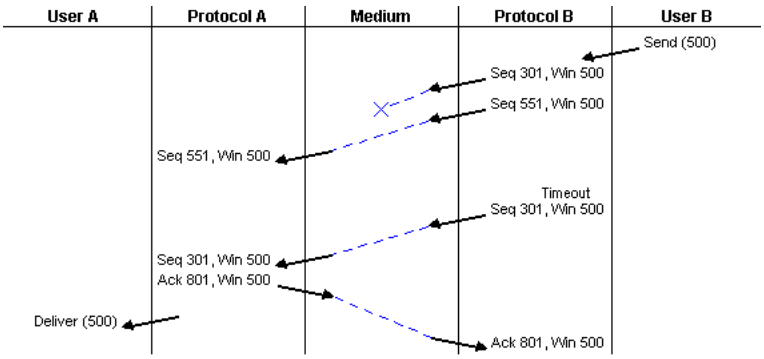

Figure 12: Transmission Control Protocol showing Data Transfer with Fragmentation and Error Recovery

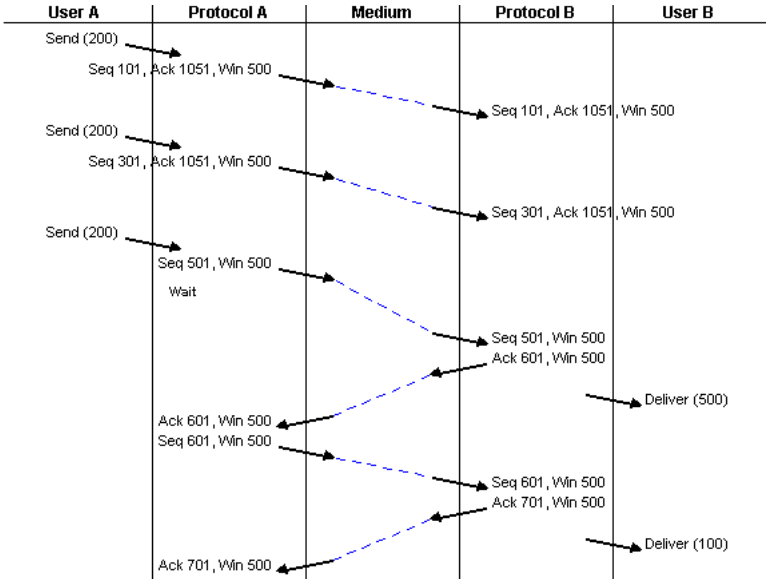

Figure 13: Transmission Control Protocol showing Data Transfer with Flow Control

was lost, so the second had to be buffered. Only when the first fragment was received after retransmission could the 500 octets be delivered to User A.

Figure 3 appeared earlier as an example of the simulator appearance. After establishing a connection, the figure shows piggybacking: the acknowledgement for the first 100 octets from Protocol A was combined with an outgoing message from Protocol B carrying 200 octets. Later in the simulation, the push flag for User A was set to send 100 octets with the PSH flag. Flow control is illustrated in figure 13. User A requested transmission of 200+200+200 octets. Because Protocol B has a window size of only 500 octets, the third packet was sent with just 100 octets. Only when an acknowledgement was received could the last 100 octets be sent. Finally, figure 14 shows graceful close in action. User B requested a close, but even though User A received a Closing message it was still allowed to send a further message (of 100 octets). Due to misordering, this arrived before the acknowledgement of the FIN message. Only when User A agreed to close the connection was a FIN response sent and (after retransmission) acknowledged. Both users were then informed of termination.

As will be seen from these diagrams, the protocol simulation is quite faithful and illustrates the complications that can arise with TCP.

5.3 Boot Protocol

BOOTP (Boot Protocol [2]) is typically used by a discless workstation to discover its Internet address and/or the name of its bootstrap file. Since BOOTP operates over UDP, it is included here with transport protocols

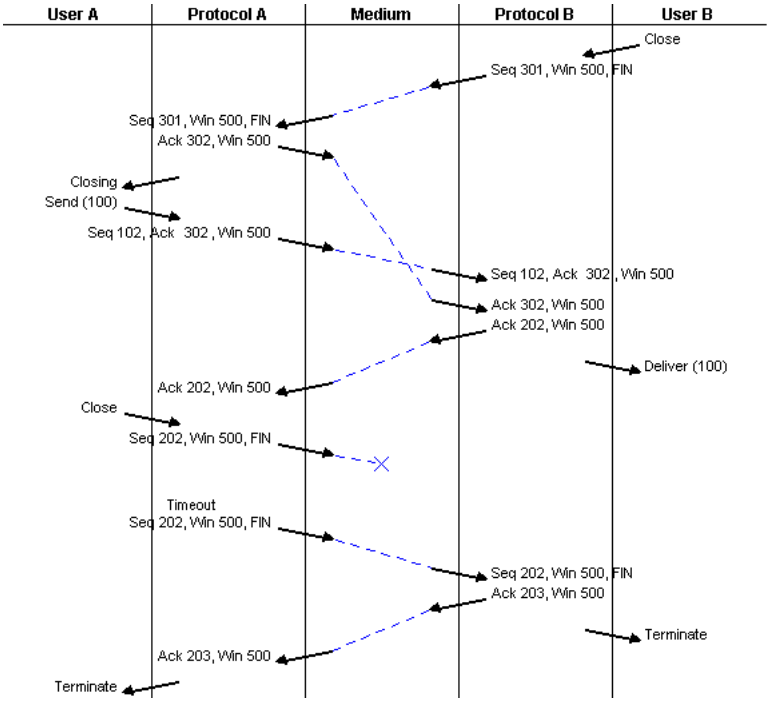

Figure 14: Transmission Control Protocol showing Disconnection

although it has more the character of a link protocol. BOOTP simply discovers the parameters needed for the bootstrap procedure. Typically, TFTP (Trivial File Transfer Protocol) is used to download the bootstrap file itself as described in section 5.4.

A boot client supplies an (arbitrary) transaction identifier and its hardware address. The client may optionally supply its network address; the server will allocate a unique network address if required. The client may optionally supply the name of the boot file; the server will supply the full path name if this file is given, or will determine the boot file the client needs. Addresses and the boot file are given symbolic names in the simulation.

The Boot Protocol is illustrated in figure 15. The client made a boot request with transaction identifier 24 and its hardware address (*hw*). Since the request was lost, the client timed out and retried. The server allocated the client address 86 within some subnetwork (*net.86*), and also determined the boot file path name (*path**boot*). The server replied with the original request parameters plus the address and boot file information.

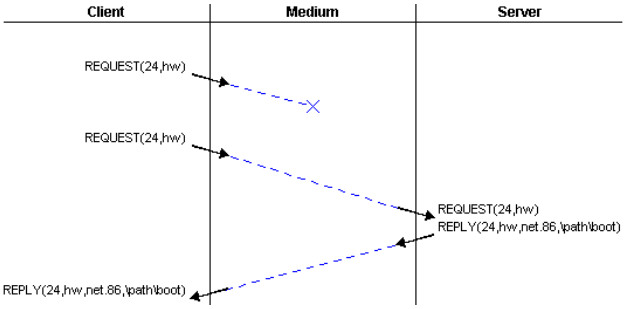

Figure 15: Boot Protocol providing Internet Address and Boot File Path

5.4 Trivial File Transfer Protocol

TFTP (Trivial File Transfer Protocol [21]) is an elementary file transfer protocol that is typically used by a discless workstation to download its bootstrap file. Since TFTP operates over UDP, it is included here with transport

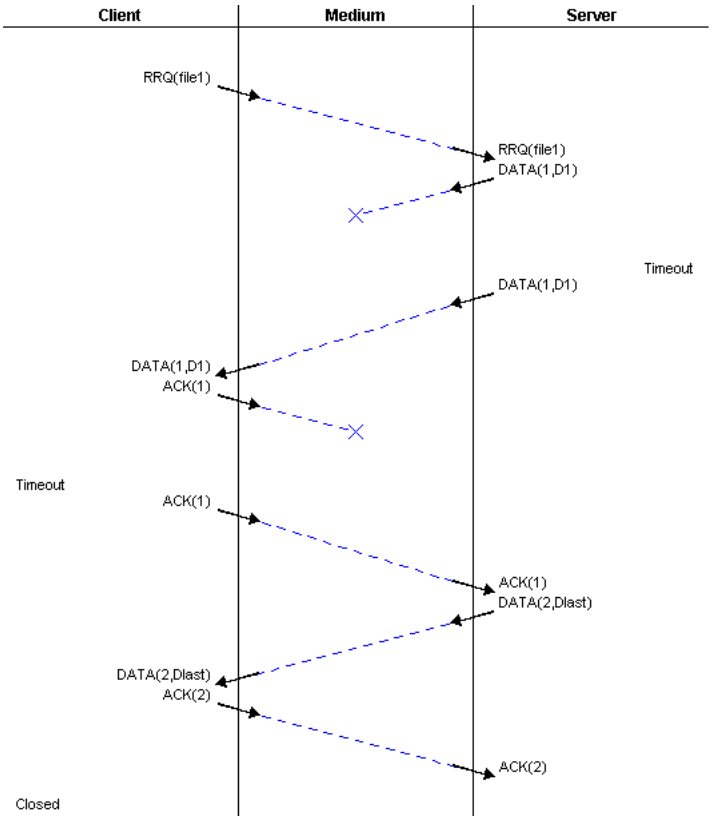

Figure 16: Trivial File Transfer Protocol reading a File

protocols although it has more the character of a link protocol. The implementation of TFTP is discussed in Appendix A as an example of how protocol simulations are written.

A client may issue a Read Request (RRQ) in order to retrieve a named file from the server. The server sends DATA blocks one at a time, numbered from 1 onwards; the client sends an ACK with the same number to acknowledge each block. All blocks except the last one are full; the final block is less than the normal size. The simulation shows this symbolically as D*n* for normal data or Dlast for the last data. A client may also issue a Write Request (WRQ) in order to send a named file to the server. In this case, it is the client that sends DATA blocks and the server that sends ACKs. The response to WRQ is an ACK with number 0.

The protocol is unusual in that both sender and receiver may time out. If the client does not receive a fresh DATA block after sending an ACK, it may retransmit the ACK in case it was lost. A complication is that the ACK for the last DATA may be lost. For this reason, the simulation (as recommended) waits until the receiver is satisfied that data transfer is complete. After completing transfer of a file, the client may start a new transfer. ERROR messages may be sent in situations such as incorrect sequence numbers or local disc errors. The full protocol allows for different modes of transfer, but the simulation supports only octet mode (the norm).

The Trivial File Transfer Protocol is illustrated in figure 16. The client issued an RRQ to read some file (*file1*). The server then sent two DATA blocks to which the client replied with ACKs. The first DATA block was lost, so the server retransmitted it on timeout. The ACK to this was lost, so the client timed out and retransmitted it. A further retransmission of the ACK was forestalled by the server sending the next DATA. As this was the final block, the client did not close the connection immediately in case its ACK had been lost. (Since timeouts are under the control of the simulation user, the decision to close is selected from a menu choice.)

6 Application-Oriented Protocols

This section describes the simulation of protocols that support applications and end-users.

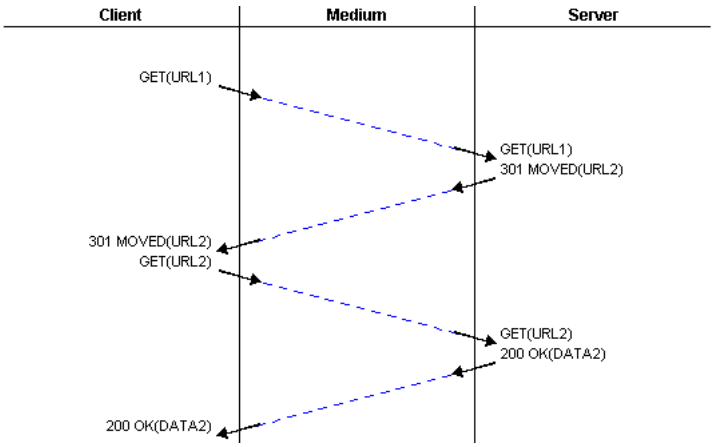

Figure 17: Hypertext Transfer Protocol showing Redirection

6.1 Hypertext Transfer Protocol

HTTP (Hypertext Transfer Protocol [4]) is familiar as the mechanism for delivering web pages. URLs (Uniform Resource Locators) and information contents are represented symbolically in the simulation. The protocol simulation deals with the main commands GET (get data for URL), HEAD (get header for URL), POST (append data to URL) and PUT (send data to URL). The simulation supports a limited range of HTTP response codes (200 OK, 301 MOVED, 400 ERROR) in the interests of simplicity.

Figure 17 shows the protocol in action. The client requested URL1, but the server indicated that the information had moved to URL2. The client then requested this URL and received the corresponding data.

6.2 Simple Mail Transfer Protocol

SMTP (Simple Mail Transfer Protocol [20]) is used to transfer email messages. The client connects to the server using TCP (Transmission Control Protocol [19]). The protocol simulation supports the main commands DATA (ask to send message data), HELO (name the client), MAIL FROM (name the sender), QUIT (finish the mail session) and RCPT TO (name the recipient). The simulation supports a limited but typical range of SMTP response codes (220 Server ready, 221 Server closing, various 250 OK messages, 354 Send mail, various 550 Invalid messages).

After the client connects to the server using TCP, the server reports its readiness with a 220 response. The client names itself in HELO, to which the server normally gives a 250 response. To send mail, the client issues MAIL FROM and normally gets a 250 response. Recipients are named in RCPT TO, normally obtaining 250 responses. However the server can reject a sender or recipient with a 550 response. Once all parties have been named, the client sends DATA to begin message transmission; a 354 response is expected. At this point, the protocol would send the lines of the message followed by a full stop. In the simulation, a single 'Mail Message' transfer stands for this. The server will normally give a 250 response and further messages can be sent. Finally, the client sends QUIT and the server responds with a 221 code. At this point the TCP connection is broken.

The Simple Mail Transfer Protocol is illustrated in figure 18. The client connected to the server, which responded it was ready. The client introduced itself with HELO, and was accepted by the server. The client named *sender1* as the source of an email message and *recipient2* as its destination. Since both were accepted by the server, the client initiated transfer with DATA and then sent the mail message. Following acceptance by the server, the client could have gone on to exchange other messages before finally quitting and closing the connection.

7 Conclusion

It has been seen that Java lends itself to visual simulation of protocols. Within a common framework it has been possible to build a variety of protocol simulators, from the simple Alternating Bit Protocol to the complex Transmission Control Protocol. The main goal was training in protocol design, though the simulators could

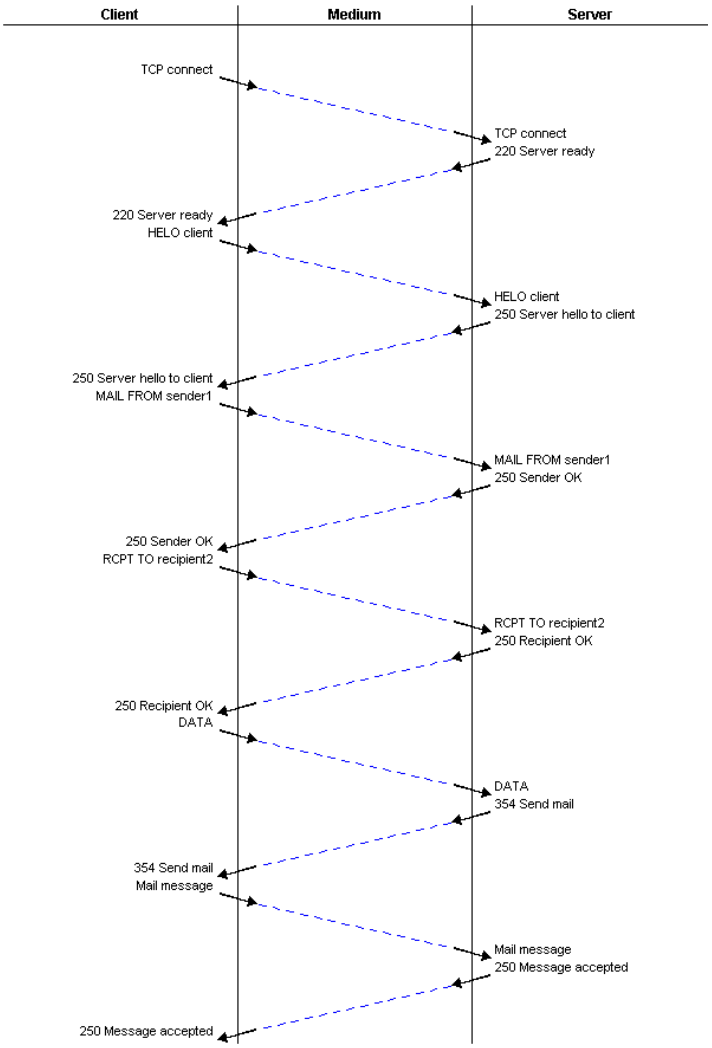

Figure 18: Simple Mail Transfer Protocol sending an Email Message

also play a role in protocol research. The educational orientation has influenced the design of the simulator, concentrating on broad functional behaviour and placing control with the user.

The simulations can be downloaded over the web as applets, making them ideal for use in teaching. The simulations can also be run as applications, when they have other capabilities such as local printing. The simulation package is portable and will run wherever Java will run. Students taking communications courses at the authors' University have reacted very positively to learning through simulation.

The object-oriented nature of Java makes it easy to develop simulations from a framework that can be extended to new protocols. Protocols can even be implemented as specialisations of other protocols. New simulations can be developed in a few days or weeks, depending on the complexity of the protocol. The open architecture of the simulator makes the addition of further protocols straightforward. To encourage its use in other institutions, the simulations described in this article have been made freely available.

Acknowledgements

I. A. Robin performed most of the simulator development work under the supervision of K. J. Turner while at the University of Stirling. P. K. Johnson and K. J. A. Whyte (University of Stirling) contributed to the simulations of BOOTP, HTTP, SMTP and TFTP. Dr. P. J. B. King (Heriot-Watt University) and Prof. E. H. Magill (University of Stirling) kindly reviewed a draft of the article.

A Simulation Implementation – Trivial File Transfer Protocol

This appendix aims to give a more exact feel for implementing a protocol simulation. It also acts as a 'how to' guide for those who wish to write new simulations. TFTP (Trivial File Transfer Protocol) is used as a concrete example; it was introduced in section 5.4. In fact TFTP is not as trivial as its name might suggest, and its simulation provides insight into some key aspects of the simulator.

A.1 Simulator Implementation

To give an idea of the size of the simulator development, the suite consists of about 70 Java and HTML files containing about 5500 non-comment source lines that took about six person-months to develop. The Java files can be compiled using an IDE (Integrated Development Environment, e.g. CodeWarrior, JBuilder, Kawa or RealJ). However, for simplicity the collection is supplied with a *make* file that automates the process of maintaining and running the simulations. GNU *make* is available for most platforms. As a convenience for Microsoft Windows users, DOS batch files are also provided to build the simulator. The suite is available precompiled, with HTML files and a common JAR (Java Archive) file that allow immediate use of the simulations via a web browser or from the command line. This permits use without the need for a development environment.

The latest Java innovations (such as Swing and the new printing model) have been deliberately avoided in order to maximise portability across platforms. The implementation has been tested with Javasoft's JDK (Java Development Kit, versions 1.1.17 and 3.0). Microsoft Internet Explorer (versions 4.0 and 5.0) and Netscape Navigator (version 4.6) have been successfully used to display simulations.

Once the framework for developing protocol simulators is understood, it is relatively straightforward to develop new simulators. Typical development efforts are a person-day (Alternating Bit Protocol), three person-days (Trivial File Transfer protocol) and three person-weeks (Transmission Control Protocol).

A.2 Protocol Structure in General

A three-column simulation for some example protocol *Example* normally requires the following files:

Example.java creates the underlying medium, the sending protocol entity and the receiving protocol entity

ExampleSender.java describes the behaviour of the protocol entity primarily responsible for sending messages (typically a client), though of course it receives messages too

ExampleReceiver.java describes the behaviour of the protocol entity primarily responsible for receiving messages (typically a server), though of course it sends messages too

Protocol behaviour is represented in Java. The most important manifestations of behaviour are the list of services (actions) the protocol can perform in a given state, the action of the protocol when one of these services is chosen, and the reaction of the protocol to an incoming message.

A five-column simulation has service and protocol entities. In place of a protocol sender and receiver it is conventional to write two files:

ExampleProtocol.java describes the rules for a protocol entity (usually a combined sender and receiver)

ExampleService.java describes the rules for a service entity (usually a combined sender and receiver).

The simulator framework allows for a basic protocol message format. If the protocol requires more than this, it is necessary to write two additional files:

ExampleMessage.java defines the format of a protocol message

ExampleMedium.java defines how to handle protocol messages in the medium.

Although the explanation above describes the current suite of protocol simulators, the developer is free to adopt other structures and naming conventions within the simulation framework.

The remainder of this appendix illustrates a typical simulator – TFTP (Trivial File Transfer Protocol) that is briefly described in section 5.4. Despite its supposedly trivial nature, TFTP contains a number of surprising complications. In terms of complexity, TFTP is intermediate between the Alternating Bit Protocol and the Transmission Control Protocol. A basic familiarity with Java is expected to understand the code that follows, though it has been extensively commented to help. The code is not reproduced in complete detail, but much of it is given so that a real understanding can be obtained of the approach.

A.3 General TFTP Support

Although TFTP can be invoked as an application on the command line, it is convenient to distribute the simulation as an applet via the web. An HTML file *TFTP.html* is therefore provided. This defines the main simulator class (*simulator.ProtocolSimulator.class*) and the JAR file that contains all the bundled classes (*ProtocolSimulator.jar*). The *protocol* parameter is given the value 'TFTP' to indicate which protocol is to be simulated.

```
<!doctype html public "-//ietf//dtd html 4.0//en">
<html>
  <head>
    <title>Trivial File Transfer Protocol Simulator</title>
  </head>
  <body>
    <h1>Trivial File Transfer Protocol Simulator</h1>
    <applet code="simulator.ProtocolSimulator.class"
        archive="ProtocolSimulator.jar" width="650" height="700">
      <param name="protocol" value="TFTP">
    </applet>
  </body>
</html>
```

The simulator does not need to be pre-configured with the protocols that have been implemented. Instead it uses Java reflection to instantiate the required protocol class. Here, the *TFTP* class will be found and instantiated automatically. This in turn will instantiate the other classes required.

A.4 TFTP

TFTP.java is the main file for the protocol. All the TFTP files start with a rubric like the following. Protocols are compiled as part of the *protocol* package. Protocol entity support files are compiled as part of the *entity* package. Many protocol actions involve Java vectors (lists of elements).

```
package protocol;                                   // protocol package

import java.util.Vector;                            // vector (list)
import entity.*;                                    // protocol entity support
```

A protocol simulation extends the *Protocol* class that defines the framework for any protocol. The TFTP class contains a constructor that creates an instance of the underlying medium, sending protocol entity and receiving protocol entity. The protocol entities are given their underlying medium and name. Each protocol entity is set as the peer of the other. Finally, the list of entities involved in the simulation (*entities*) is set up.

```
public class TFTP extends Protocol {                // TFTP protocol

  private TFTPSender sender;                        // protocol sender (client)
  private TFTPReceiver receiver;                    // protocol receiver (server)

  public TFTP () {                                  // construct protocol instance
    medium = new TFTPMedium ();                     // construct comms medium
    sender = new TFTPSender (medium, "Client");     // construct sender (client)
    receiver = new TFTPReceiver (medium, "Server"); // construct receiver (server)
    sender.setPeer (receiver);                      // sender is receiver's peer
    receiver.setPeer (sender);                      // receiver is sender's peer
    entities = new Vector ();                       // initialise protocol entities
    entities.addElement (sender);                   // add sender protocol entity
    entities.addElement (medium);                   // add comms medium entity
    entities.addElement (receiver);                 // add receive protocol entity
  }
}                                                   // end of TFTP class
```

A.5 TFTPSender

TFTPSender.java defines the sending protocol entity that is the TFTP client. It implements the *ProtocolEntity* and *Timeouts* interfaces that expect standard methods for all protocols. These interface methods were described in section 2.2. A protocol entity without timeouts (such as a passive receiver) implements only *ProtocolEntity*. A protocol entity typically stores its peer, its underlying medium and its own name.

```
public class TFTPSender implements ProtocolEntity, Timeouts {     // protocol sender (client)

    private ProtocolEntity peer;                                  // peer entity (server)
    private Medium medium;                                        // communications medium
    private String name;                                          // entity name
```

A protocol entity typically has a state and a number of state variables that are used to dictate protocol behaviour. Protocol message types are stored in constant strings for consistent reference.

```
    int state;                                                    // current protocol state
    final static int idle = 0;                                    // no connection
    final static int readRequest = 1;                             // file to be read
    final static int reading = 2;                                 // reading data
    final static int writeRequest = 3;                            // file to be written
    final static int writing = 4;                                 // writing data
    final static int waitLast = 5;                                // wait for last re-send

    private int pduNo;                                            // PDU sequence number sent/expected
    private PDU pduReceived;                                      // PDU received
    private PDU pduSent;                                          // PDU sent
    private boolean timerEnabled;                                 // whether timer is enabled
    private float errorProb = 0.2f;                               // file I/O error probability
    private int fileNo;                                           // file number to transfer

    final static String ack = "ACK";                              // acknowledgement message type
    final static String data = "DATA";                            // data message type
    final static String error = "ERROR";                          // error message type
    final static String rrq = "RRQ";                              // read request message type
    final static String wrq = "WRQ";                              // write request message type
```

The main constructor notes the underlying medium and the protocol entity name. It then calls the *initialise* method, that may later be called by the simulator kernel to clear a simulation or when undoing/redoing a simulation step. When starting from scratch, the file number to be requested by TFTP ('file*n*') is set to zero. Since several files may be transferred sequentially, protocol variables are set up in a separate *reinitialise* method that is called internally by the protocol.

```
    public TFTPSender (Medium m, String name) {                   // construct sender instance
        this.name = name;                                         // set protocol entity name
        medium = m;                                               // set underlying medium
        initialise ();                                            // initialise protocol
    }

    public void initialise () {                                   // initialise protocol
        fileNo = 0;                                               // initialise file number
        reinitialise ();                                          // re-initialise protocol
    }

    public void reinitialise () {                                 // re-initialise protocol
        state = idle;                                             // initialise state
        pduNo = 0;                                                // initialise sequence number
        pduReceived = null;                                       // initialise no PDU received
        pduSent = null;                                           // initialise no PDU sent
        timerEnabled = false;                                     // initialise no timeout
```

```
    fileNo++;                                   // to next file number
  }
```

As part of its interface, a protocol entity must be able to return its own name in response to a *getName* call. It must also accept a *setPeer* call to set its peer.

```
  public String getName () {                    // get protocol entity name
    return (name);                              // return protocol entity name
  }

  public void setPeer (ProtocolEntity peer) {   // set protocol peer
    this.peer = peer;                           // set this entity's peer
  }
```

The TFTP client needs to implement timeouts on all messages it sends. It therefore reports that all PDUs need timeouts in response to a *hasTimer* call. When *setTimer* informs it that a timer has been set for a PDU, it stores this in a local variable. Since TFTP has just one message outstanding at a time, it does not need to note which PDU has a timer running.

```
  public boolean hasTimer (String type) {       // PDU needs timer?
    return (true);                              // report it does
  }

  public void setTimer (PDU pdu, boolean b) {   // set timer status
    timerEnabled = b;                           // store timer status
  }
```

A protocol entity sends a message by calling the *transmitPDU* method with the PDU and the destination entity. Most protocols have a standard action on transmission. The PDU details are completed and it is stored locally (in case it has to be retransmitted). The medium is then told to receive the PDU for onward transmission. Simple protocols such as TFTP will then annul any PDU previously received in response, and will cancel any outstanding timeout.

```
  public void transmitPDU (                     // transmit PDU
   PDU pdu, ProtocolEntity dest) {              // for given PDU, destination
    pdu.setSource (this);                       // source is this entity
    pdu.setDestination (dest);                  // destination is as given
    pduSent = pdu;                              // copy PDU sent
    medium.receivePDU (pdu);                    // medium receives PDU
    pduReceived = null;                         // note no PDU in response
    timerEnabled = false;                       // note no timeout
  }
```

A protocol entity is notified that a PDU has arrived by *receivePDU* being called. The entity may choose to do minimal processing at this point and simply store the PDU locally. Alternatively the entity may perform extensive processing such as checking the validity of the PDU and making automatic responses such as acknowledgements. For this reason, the method may return a list of protocol actions (though this will be empty for a protocol that decides to respond later). TFTP is simple enough that an incoming PDU can simply be stored in *pduReceived* for later processing. However if this is an ERROR message, indicating a serious fault, the entity immediately reinitialises itself.

```
  public Vector receivePDU (PDU pdu) {          // handle received PDU
    pduReceived = pdu;                          // store PDU
    if (pduReceived.type.equals (error))        // error message?
      reinitialise ();                          // re-initialise protocol
    return (new Vector ());                     // return no events
  }
```

The most important parts of a protocol definition are the *getServices* and *performService* methods that define the real protocol behaviour. Most protocol simulations are implemented as a state machine, although this is not essential. These methods for TFTP are presented here in full detail so that the interested reader can get a real idea of what is involved in writing a protocol simulation of intermediate complexity. To understand the code fully will

need a good knowledge of the protocol and its various states. A simpler example like the Alternating Bit Protocol needs about 20% of the code for TFTP.

A call is made to *getServices* to determine the list of actions currently possible for a protocol. These may include user actions the protocol permits, new messages to be sent, and random events. It must be ensured that *getServices* merely determines possible actions and does not change the state of the protocol. Protocol changes must happen only in *performService*, depending on the user choice.

A typical *getServices* implementation evaluates all inputs (such as an incoming PDU *pduReceived*) in the current situation (*state*), and creates a list of possible actions (*events*). The fields of an incoming PDU may be inspected such as its type (*type*), sequence number (*seq*) and data content (*sdu*).

Possible actions are conventionally accompanied by an explanatory comment to help the user. For example when the TFTP client receives DATA block 3 it creates an action entitled 'ACK(3) – send acknowledgement'. If a Read Request is not followed by DATA block 1, a possible action is 'ERROR(wrong data sequence number) – report error'. These are the strings presented to the user in the simulation menu.

The actions may also include 'random' events such loss of data and timeout. As will be seen later, random numbers are generated by a method in the *TFTPMedium* class.

```
public Vector getServices () {                          // get protocol entity services
  Vector events = new Vector ();                        // list of events
  String pduType;                                       // received PDU type
  int pduSeq;                                           // received PDU sequence number

  if (state == idle) {                                  // not connected?
    events.addElement (                                 // read file
      rrq + ″(file″ + fileNo + ″) - ask to read file″);
    events.addElement (                                 // write file
      wrq + ″(file″ + fileNo +″) - ask to write file″);
  }
  else if (pduReceived != null) {                       // PDU received?
    pduType = pduReceived.type;                         // get received PDU type
    pduSeq = pduReceived.seq;                           // get received PDU sequence number
    if ((state == readRequest ||                        // read request or ...
         state == reading ||                            // reading or ...
         state == waitLast) &&                          // awaiting re-send, and ...
        pduType.equals (data)) {                        // data message?
      if (pduSeq == pduNo) {                            // expected sequence number?
        timerEnabled = false;                           // cancel timeout
        events.addElement (                             // send acknowledgement
          ack + ″(″ + pduNo + ″) - send acknowledgement″);
      }
      else if (state == readRequest)                    // wrong sequence, read request?
        events.addElement (                             // report sequence number error
          error +″(wrong data sequence number) - report error″);
    }
    else if ((state == writeRequest ||                  // write request or
              state == writing) &&                      // writing, and ...
             pduType.equals (ack)) {                    // write and acknowledgement?
      if (pduSeq == pduNo) {                            // expected sequence number?
        timerEnabled = false;                           // cancel timeout
        if (pduSent != null &&                          // valid previous PDU and ...
            pduSent.sdu.equals(″Dlast″))                // last data sent?
          reinitialise ();                              // re-initialise protocol
        else {                                          // more data to send
          events.addElement (                           // send data
            data + ″(″ + (pduNo + 1) + ″,D″ + pduNo + ″) - send data″);
          events.addElement (                           // send last data
            data + ″(″ + (pduNo + 1) + ″,Dlast) - send last data″);
```

```
            }
          }
          else if (state == writing &&                          // writing and ...
                   pduSeq == pduNo - 1)                         // repeated acknowledgement?
            events.addElement (                                 // send data
              data + "(" + pduSent.seq + "," + pduSent.sdu + ") - re-send data");
          else if (state == writeRequest)                       // wrong sequence, write request?
            events.addElement (                                 // report sequence number error
              error +"(wrong acknowledgement sequence number) - report error");
        }
        else if (pduType.equals (error)) {                      // error message?
          state = idle;                                         // back to not connected
        }
      }
      if (state == waitLast)                                    // waiting for last re-send?
        events.addElement (                                     // add close event
          "Close - presume all retransmissions over");
      if (state == writing &&                                   // writing and ...
          pduSent != null &&                                    // PDU was sent and ...
          !pduSent.sdu.equals ("Dlast") &&                      // PDU was not last and ...
          TFTPMedium.random () < errorProb)                     // file I/O error occurs?
          events.addElement (                                   // report I/O error
            error +"(data for write unavailable) - report error");
      if (timerEnabled)                                         // timer is enabled?
        events.addElement (                                     // add timeout event
          "Timeout - presume loss of message and resend");
      return (events);                                          // return list of events
    }
```

The *performService* method is called with the string corresponding to the action selected by the user from the simulation menu. Since this normally contains the PDU to be sent along with an explanatory comment, the string must be parsed to extract the information to be placed in the outgoing PDU. This is sent by calling *transmitPDU*, which also stores the transmitted PDU in the variable *pduSent*. If a PDU is transmitted, a TRANSMIT protocol event is notified to the simulator kernel. A protocol may also generate COMMENT events for the user's information, such as done here to indicate that the connection is considered to be finally closed. Understanding the following method will require a good knowledge of the protocol and the menu items generated by *getServices*.

```
    public Vector performService (String s) {                   // perform protocol service
      Vector events = new Vector ();                            // initialise events list
      int start, middle, end;                                   // start/middle/end subscripts
      int pduSeq;                                               // PDU sequence number
      String pduData;                                           // PDU data

      if (s.startsWith (rrq)) {                                 // send read request?
        start = s.indexOf ('(') + 1;                            // get filename start
        end = s.indexOf (')');                                  // get filename finish
        pduData = s.substring (start, end);                     // get filename
        transmitPDU (new TFTPMessage (rrq , pduData), peer);    // send read request
        pduNo = 1;                                              // expect sequence number 1
        state = readRequest;                                    // now requested write
      }
      else if (s.startsWith (ack)) {                            // send acknowledgement?
        start = s.indexOf ('(') + 1;                            // get sequence number start
        end = s.indexOf (')');                                  // get sequence number end
        pduSeq = Integer.parseInt (s.substring (start, end));   // get sequence number
        if (pduReceived != null &&                              // valid PDU received and ...
            pduReceived.sdu.equals("Dlast"))                    // last data?
```

```
            state = waitLast;                                   // wait for last re-send
          else {                                                // still reading
            pduNo++;                                            // to next sequence number
            state = reading;                                    // (now) reading
          }
          transmitPDU (new TFTPMessage (ack, pduSeq), peer);    // send acknowledgement
        }
        else if (s.startsWith (wrq)) {                          // send write request?
          start = s.indexOf ('(') + 1;                          // get filename start
          end = s.indexOf (')');                                // get filename finish
          pduData = s.substring (start, end);                   // get filename
          transmitPDU (new TFTPMessage (wrq , pduData), peer);  // send write request
          state = writeRequest;                                 // now requested write
        }
        else if (s.startsWith (data)) {                         // send data?
          start = s.indexOf ('(') + 1;                          // get sequence number start
          middle = s.indexOf (',');                             // get comma position
          end = s.indexOf (')');                                // get sequence number end
          pduSeq = Integer.parseInt (s.substring (start, middle));  // get sequence number
          pduData = s.substring (middle + 1, end);              // get data content
          transmitPDU (                                         // send data
            new TFTPMessage(data, pduSeq, pduData), peer);
          pduNo++;                                              // to next sequence number
          state = writing;                                      // (now) writing
        }
        else if (s.startsWith (error)) {                        // send error?
          start = s.indexOf ('(') + 1;                          // get error message start
          end = s.indexOf (')');                                // get error message end
          pduData = s.substring (start, end);                   // get error message
          transmitPDU (new TFTPMessage (error, pduData), peer); // send error
          state = waitLast;                                     // wait for last re-send
        }
        else if (s.startsWith (''Close'')) {                    // close?
          events.addElement (                                   // add closed comment
            new ProtocolEvent (ProtocolEvent.COMMENT, this, ''Closed''));
          reinitialise ();                                      // re-initialise protocol
        }
        if (s.startsWith (''Timeout'')) {                       // timeout?
          transmitPDU (pduSent, peer);                          // re-send PDU
          events.addElement (                                   // add timeout event and PDU
            new ProtocolEvent (ProtocolEvent.TIMEOUT, pduSent));
        }
        else if (pduSent != null)                               // PDU to send?
          events.addElement (                                   // transmit PDU
            new ProtocolEvent (ProtocolEvent.TRANSMIT, pduSent));
        return (events);                                        // return list of events
      }
    }                                                           // end of TFTPSender class
```

A.6 TFTPReceiver

TFTPReceiver.java defines the receiving protocol entity that is the TFTP server. It implements the *ProtocolEntity* interface. Slightly unusually for a receiver, it also implements the *Timeouts* interface. The receiver is essentially the mirror image of the sender and so is presented only in outline to show its structure. Variable declarations have been omitted as they are very similar to those of the sender. Only the methods headers are given, again because of the similarity to the sender.

```
public class TFTPReceiver implements ProtocolEntity, Timeouts {   // protocol receiver (server)

   ...                                                             // simulator variables
   ...                                                             // protocol variables
   ...                                                             // protocol state
   ...                                                             // protocol messages

   public TFTPReceiver (Medium m, String name)                     // construct receiver instance
   public void initialise ()                                       // initialise protocol
   public String getName ()                                        // get protocol entity name
   public void setPeer (ProtocolEntity peer)                       // set protocol peer
   public boolean hasTimer (String type)                           // protocol uses timer?
   public void setTimer (PDU pdu, boolean b)                       // set timer status
   public void transmitPDU (PDU pdu, ProtocolEntity dest)          // transmit PDU
   public Vector receivePDU (PDU pdu)                              // handle received PDU
   public Vector getServices ()                                    // get protocol entity services
   public Vector performService (String s)                         // perform protocol service
}                                                                  // end of TFTPReceiver class
```

A.7 TFTPMessage

TFTPMessage.java defines the protocol message format. Many protocols have a simple message format that is catered for by the basic simulator framework: the data content, or a sequence number and the data content. More complex formats require a definition of the PDU fields. Although TFTP could be accommodated within the generic framework, its message structure is irregular. A TFTP message class is therefore defined as an extension of the *PDU* class. It defines constructors for message values that invoke the basic *PDU* constructors:

```
public class TFTPMessage extends PDU {                             // protocol data unit format

   public TFTPMessage (String type, int seq) {                     // construct PDU type/sequence
      super (type, seq);                                           // use generic PDU constructor
   }

   public TFTPMessage (String type, String sdu) {                  // construct PDU type/SDU
      super (type, sdu);                                           // use generic PDU constructor
   }

   public TFTPMessage (String type, int seq, String sdu) {         // construct PDU type/sequence/SDU
      super (type, seq, sdu);                                      // use generic PDU constructor
   }
```

The real reason for having a separate TFTP message class is to override the standard *getLabel* method for a PDU. This is used by the simulator kernel to get the string to be printed for a PDU in a time sequence diagram. The default *getLabel* method shows either the sequence number or the data content but not both.

```
   public String getLabel () {                                     // get PDU "type(contents)"
      String label = type;                                         // get PDU type
      if (seq >= 0) {                                              // sequence number present?
         label += "(" + seq;                                       // append sequence number
         if (sdu.equals (""))                                      // SDU absent?
            label += ")";                                          // finish contents
         else                                                      // SDU present
            label += "," + sdu + ")";                              // append SDU, finish contents
      }
      else if (!sdu.equals (""))                                   // no sequence number, SDU present?
         label += "(" + sdu + ")";                                 // append SDU
      return (label);                                              // return PDU label
   }
```

```
}                                                   // end of TFTPMessage class
```

A.8 TFTPMedium

TFTPMedium.java defines specific medium methods to support the protocol. If, as for TFTP, it is necessary to introduce an extended class for PDUs, the common *Medium* class must also be extended. The standard medium can be used by a protocol that needs only the basic *PDU* class. For reasons to be explained, the class maintains a list of random numbers. The constructor method initialises this list after calling the standard medium constructor. Like a protocol entity, a medium also has an *initialise* method.

```
public class TFTPMedium extends Medium {           // protocol medium

   private static Vector randoms;                   // random number list
   private static int randomIndex;                  // random number index

   public TFTPMedium() {                            // construct medium instance
      super ();                                     // construct as generic medium
      randoms = new Vector();                       // initialise list of randoms
   }

   public void initialise () {                      // initialise medium
      super.initialise ();                          // initialise generic medium
      randomIndex = 0;                              // initialise randoms index
   }
```

A medium relies on the method *getMatchingPDU* to find a PDU in transit with a matching description. This is because in general there may be several PDUs in transit through the medium. The simulator must be able to find the PDU corresponding to its label. The method is **protected** because it is part of the *entity* package. For TFTP, the method parses the PDU label into its components fields (*type*, *seq*, *sdu*). It then searches the global list of PDUs in transit (*pdus*) for one with matching fields.

```
   protected PDU getMatchingPDU (String s) {        // get matching PDU on channel
      PDU pdu;                                      // PDU
      int seq = -1;                                 // sequence number
      String sdu = "";                              // data
      int sourceStart = s.indexOf ('[') + 1;        // get start of entity name
      int sourceEnd = s.indexOf (']');              // get end of entity name
      String sourceName = s.substring (sourceStart, sourceEnd);  // get PDU source
      int typeStart = s.indexOf (' ') + 1;          // get start of PDU type
      int typeEnd = s.indexOf ('(');                // get end of PDU type
      String type = s.substring (typeStart, typeEnd);  // get PDU type
      String[] params = getParams (s);              // get PDU parameters

      if (type.equals ("RRQ") ||                    // read request or ..
          type.equals ("WRQ") ||                    // write request or ...
          type.equals ("ERROR"))                    // error?
         sdu = params[0];                           // get filename/error message
      else if (type.equals ("ACK"))                 // acknowledgement?
         seq = Integer.parseInt (params[0]);        // get sequence number
      else {                                        // data
         seq = Integer.parseInt (params[0]);        // get sequence number
         sdu = params[1];                           // get data
      }
      for (Enumeration e = pdus.elements();         // get next PDU on channel ...
           e.hasMoreElements(); ) {                 // as long as more to check
         pdu = (PDU) e.nextElement ();              // get next PDU on channel
         if (pdu != null &&                         // valid PDU and ...
             pdu.type.equals(type) &&               // type matches and ...
```

```
            pdu.getSource().getName().                    // source ...
               equals(sourceName) &&                      // matches and ...
            seq == pdu.seq &&                             // sequence number matches and ...
            sdu.equals (pdu.sdu))                         // SDU matches
          return (pdu);                                   // return with this PDU
      }
      return (null);                                      // return no PDU as no match
   }
```

TFTP uses a random number generator in both the client and the server to decide whether a local I/O error has occurred. The medium class is a common place to define the *random* method. At heart, this uses the standard mathematics random method. However if this were the basis of random numbers, it would cause unrepeatability in the simulator if a step were undone and then redone. For this reason, all random numbers generated so far are stored in the *randoms* list. If a new random number is required (indicated by *randomIndex* being beyond the end of the list), this is generated as usual and stored in the list. If a previously chosen random number is required (because the user has undone one or more steps), this is taken from the list.

```
   protected static float random () {                     // random number (from list)
      Float rand;                                         // random number

      if (randomIndex < randoms.size ())                  // number is in list?
         rand = (Float) randoms.elementAt (randomIndex++); // get number from list
      else {                                              // make new random number
         rand = new Float (Math.random ());               // get random number
         randoms.addElement (rand);                       // add to list
         randomIndex++;                                   // increment list index
      }
      return (rand.floatValue ());                        // return random number
   }
}                                                         // end of TFTPMedium class
```

References

[1] Bayfront Technologies Inc. DataXfer protocol simulation in JavaScript. http://www.bayfronttechnologies.com/101fun.htm, Aug. 1997.

[2] B. Croft and J. Gilmore, editors. *Bootstrap Protocol*. RFC 951. The Internet Society, New York, USA, Sept. 1985.

[3] S. Deering and R. Hinden, editors. *Internet Protocol Version 6*. RFC 2460. The Internet Society, New York, USA, Dec. 1998.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, editors. *Hypertext Transfer Protocol Version 1.1*. RFC 2068. The Internet Society, New York, USA, Jan. 1997.

[5] D. Hudek. UNI 3.1 signalling package simulator. http://www.ultranet.com/∼dhudek/junidemo1.shtml, Feb. 1996.

[6] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conventions for the Definition of OSI Services*. ISO/IEC TR 10731. International Organization for Standardization, Geneva, Switzerland, 1992.

[7] ISO/IEC. *Information Technology – Open Systems Interconnection – ESTELLE: A Formal Description Technique based on an Extended State Transition Model*. ISO/IEC 9074. International Organization for Standardization, Geneva, Switzerland, 1997.

[8] ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 1996.

[9] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1996.

[10] P. J. B. King. Data link simulation. *IEEE Transactions on Education*, 40(3):172–178, 1992.

[11] P. J. B. King. Data link protocol simulator. http://www.cee.hw.ac.uk/∼pjbk/dlpsim/index.html, May 1998.

[12] H. Krumm. Verteilte Algorithmen. http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/OrdnerVertAlgo/VertAlgo.html, Dec. 1997.

[13] D. Lindsay. Visualising computer communications. In D. C. Bateman and T. Hopkins, editors, *Developments in the Teaching of Computer Science*, pages 72–79. University of Kent, Canterbury, UK, Apr. 1992.

[14] C. S. McDonald. A network specification language and execution environment for undergraduate teaching. In *Proc. ACM Computer Science Education Symposium*, pages 25–34. ACM Press, New York, USA, Mar. 1991.

[15] C. S. McDonald. The *cnet* network simulator. http://www.cs.uwa.edu.au/pls/cnet/, Dec. 2000.

[16] Network Simulator Team. Network Simulator version 2. http://www.isi.edu/nsnam/ns/, June 2000.

[17] J. B. Postel, editor. *User Datagram Protocol*. RFC 768. The Internet Society, New York, USA, Aug. 1980.

[18] J. B. Postel, editor. *Internet Protocol*. RFC 791. The Internet Society, New York, USA, Sept. 1981.

[19] J. B. Postel, editor. *Transmission Control Protocol*. RFC 793. The Internet Society, New York, USA, Sept. 1981.

[20] J. B. Postel, editor. *Simple Mail Transfer Protocol*. RFC 821. The Internet Society, New York, USA, Aug. 1982.

[21] K. Sollins, editor. *The TFTP Protocol*. RFC 1350. The Internet Society, New York, USA, July 1992.

[22] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1, 1976.

[23] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1997. Third edition.

[24] K. J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, Jan. 1993.

[25] K. J. Turner and I. A. Robin. JASPER (Java Simulation of Protocols for Education and Research) web page. http://www.cs.stir.ac.uk/∼kjt/software/comms/jasper.html, May 2001.

[26] H. Yu and N. Salehi. The network simulator *ns-2*. In *Proc. Network Simulator Workshop*. Cooperative Association for Internet Data Analysis, University of California, USA, June 2000.

Biography

K. J. Turner holds a B.Sc. in Electrical Engineering, and a Ph.D. in Artificial Intelligence. After 12 years working in the communications industry, he became Professor of Computing Science in 1987 at the University of Stirling, Scotland. His research interests focus on formal methods and systems architecture. He undertakes specification and analysis using the LOTOS and SDL formal languages, applying these in areas such as communications architecture, distributed systems, telecommunications services, hardware design and medical devices. His teaching interests include communications, compiler design, programming and software engineering.

I. A. Robin holds a B.Sc. in Physics and Astronomy, and a Ph.D. in Celestial Mechanics. He is presently a software engineer with Cadence Design Systems Ltd. in Livingston, Scotland. He is a member of the team developing the company's VCC (Virtual Component Co-Design) Links to Implementation tool. Much of the work reported in this paper is based on his M.Sc. in Software Engineering at the University of Stirling.