Four Logics and a Protocol

David Gray, Geoff Hamilton, David Sinclair
School of Computer Applications, Dublin City University
Glasnevin, Dublin 9, Ireland

Paul Gibson, James Power
Department of Computer Science, National University of Ireland, Maynooth
Maynooth, Co. Kildare, Ireland

Abstract

The Internet Protocol (IP) is the protocol used to provide connectionless communication between hosts connected to the Internet. It provides a basic internetworking service to transport protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These in turn provide both connection-oriented and connectionless services to applications such as file transfer (FTP) and WWW browsing. In this paper we present four separate specifications of the interface to the internetworking layer implemented by IP using four types of logic: classical, constructive, temporal and linear logic.

1 Introduction

The Internet Protocol (IP) [1] is used to implement a connectionless *internet* based on an arbitrary collection of interconnected physical networks. As such, IP provides a virtual internetworking service which allows *hosts* connected to different physical networks (possibly based on different network technologies) to communicate by exchanging packets of data.

The functionality provided by IP is deliberately limited to the connectionless exchange of individual packets of data so as not to require the underlying physical networks to support complex functionality. In particular, IP is defined to be *unreliable* so that underlying physical networks may (if need be) discard packets, duplicate packets or deliver packets in a different order to which they were sent[1].

As we show in this paper, connectionless transfer and unreliability makes the interface to IP very simple, but makes it difficult to prove *strong* properties about an IP internet.

The unreliability of IP also makes it impractical to implement applications that use IP directly and therefore, most Internet applications use Transmission Control Protocol (TCP) [2]. TCP provides a connection-oriented transport service between hosts, i.e., two hosts communicate by setting up a connection and exchanging data. Unlike IP, TCP is reliable; all data that is sent arrives exactly once in the same order it was sent. Since TCP uses IP to send and receive data over an internet, it needs to incorporate complex protocol mechanisms to overcome the inherent unreliability of IP[2].

In this paper we specify the service offered by IP using a number of different formalisms. In particular, we look at how unreliability can be captured and what properties can be established about the service offered by IP. Our longer term goal is to specify the services offered by TCP and prove that these TCP services can be realized by using TCP over IP.

[1]Of course, physical networks are not deliberately designed to be unreliable, but (for example) under heavy load conditions a physical network may be forced to discard data.

[2]Ultimately, since IP is unreliable, it is impossible to build a completely reliable TCP service. Thus, TCP reliability is only guaranteed while a connection can be maintained, i.e., if IP proves to be too unreliable, TCP can abort a connection.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

There is presently a vast range of formal specification techniques and methods, based on logic, algebraic and set-theoretic approaches, as well as various specialised formalisms for particular application areas. In this paper we deal with logic-based specification and, in particular, present specifications based on four main types of logic:

Classical Logic supported by set theory and logic-based modularisation constructs, and expressed here using the Z notation

Temporal Logic allowing for reasoning about a system in terms of states, and proving for quantification over transitions between these states; here we use the TLA formalism

Constructive Logic a variation of classical logic via an elimination of proof-by-contradiction and an enhanced role for induction; in this paper we use the Coq proof assistant

Linear Logic a resource-sensitive logic where the use of formulae as assumptions and conclusions of proofs mirrors the consumption and production of resources by a computational process

In presenting these specifications we hope to provide:

- a case study in the use of various logics
- a basis for comparison between the logics
- a formal foundation for further work with higher-level protocols

In what follows we informally review the fundamentals of IP, provide a description using each of the four logics, and compare the results. We assume a familiarity with first-order predicate logic; a brief overview of each formalism is given with the specification.

1.1 A Brief Description of IP

IP is a *protocol* that defines how two *hosts* connected by an internet can format and exchange packets of data known as *datagrams*. In particular, it defines:

- How hosts are addressed.
- The format of datagrams, i.e., it defines how a datagram is constructed from a *header* containing various protocol fields and a *body* containing the data being exchanged between hosts.
- The rules for how datagrams should be handled.

Each host will contain software that implements IP and this software will provide a *service* to other software within that host; typically this other software will implement TCP and UDP. The definition of IP [1] does not define an interface between an implementation of IP and other software, but gives a general description of what such an interface should look like.
In this paper, we are interested in specifying the services that are made available via such an interface, so in this section we give an informal description of these services. It should be noted that certain functionality of IP is not required to be visible across such an interface[3]. Therefore, the description of the services offered and the protocol fields in the header of an IP datagram, are limited to those that are visible across an interface.
The services offered by IP can be described as follows:

- Each host is identified by means of an *IP address*; this is a 32-bit integer value. IP addresses have an internal structure, but this is transparent to the user of the IP services.

[3]For example, IP supports the *fragmentation* of datagrams, i.e., if a datagram is too large to be transmitted across a particular physical network, it is fragmented into a number of smaller datagrams that are transmitted separately. These are then reassembled at the destination host. Fragmentation is supported via a number of protocol fields in the header of datagrams and is implemented by IP software. However, fragmentation and reassembly is transparent to the software that uses IP.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

- The header of a datagram has *source* and *destination* fi elds containing the IP addresses of the source and destination hosts of the datagram. These fi elds are used internally by IP to route a datagram from its source to its destination. How this routing is achieved is irrelevant.

- The header of a datagram has a *Time-To-Live* (TTL) fi eld; this is an 8-bit integer value. This value determines how long a datagram can remain within an internet before being discarded[4]. Here we consider the TTL fi eld to be the maximum time in seconds that a datagram can remain within an internet.

  It should be noted that (internally) IP decrements TTL values of datagrams, i.e., the TTL value in a datagram received will be less than or equal to the value when sent. In addition, it should be noted that IP does not require hosts (or internal nodes) to keep synchronised time. However, for simplicity, we assume that each host (and node) has an accurate internal clock that can record one-second intervals.

- The service offered by IP supports two operations; *send* and *receive*.

  - *send* allows a host to send a datagram to another host. The datagram being sent will contain the IP addresses of the source [5] and destination hosts and a suitable TTL value.
  - *receive* allows a host to receive a datagram that was sent by another host. The datagram received will contain the IP addresses of the source and destination hosts.

- IP implements a connectionless internet. Each datagram is sent and received independently and there are no facilities in the protocol for flow control, sequencing or acknowledgements.

- IP implements an unreliable internet. Datagrams may be lost, duplicated or delivered in a different order to which they were sent and no corrective action will be taken by the IP software.

- To protect against data errors, IP supports a *checksum* calculated over the header of a datagram, i.e., the header of a datagram contains a checksum fi eld that is calculated from the other header fi elds.

  IP software ensures that checksums are valid and discards datagrams that have invalid checksums. Therefore, we assume that if a host receives a datagram, then the header fi elds are valid and can never be corrupt[6]. However, since the body is not protected by the checksum, we may receive a datagram with corrupt user data.

  However little the IP specifi cation guarantees, it does say that:

  - a message which has been received with an uncorrupted header must have originated from the source indicated in that header
  - if you wait suffi ciently long enough between sending two message, and if both messages are received then an ordering of the received messages is guaranteed.

  In each of the following four sections we present a specifi cation of IP along with these two consistency results.

2 Specification of IP using Z

In the Z [3][4] specifi cation of IP, we present a schema (*Datagram*) describing a datagram and a schema (*IP*) describing the state of an entire IP system. We then present operations (*Send*, *Receive* and *Tick*) that transform the state of an IP system and prove properties about these transformations.

[4]The TTL fi eld ensures that datagrams that cannot be delivered do not remain indefi nitely within an IP internet, e.g., datagrams addressed to non-existent hosts are eventually discarded.

[5]For simplicity, we assume that the source address placed in an IP datagram is the actual IP address of the source host. In reality, a host can *masquerade* as another host by using its IP address in a datagram.

[6]Of course, there is a very small possibility that a datagram could have been corrupted in such a way as to still have a valid checksum. We will ignore this possibility.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

2.1 IP Datagrams

We can define an IP datagram in terms of some simple types.

$$
\begin{array}{l}
\mathit{IPAddress} \mathrel{\widehat=} 0..2^{32}-1 \\
\mathit{TTL} \mathrel{\widehat=} 0..255 \\
\mathit{Byte} \mathrel{\widehat=} 0..255
\end{array}
$$

$$
\begin{array}{|l}
\textit{Datagram} \\ \hline
\mathit{source} : \mathit{IPAddress} \\
\mathit{destination} : \mathit{IPAddress} \\
\mathit{ttl} : \mathit{TTL} \\
\mathit{data} : \mathbf{seq}\ \mathit{Byte} \\ \hline
\mathit{source} \neq \mathit{destination} \\ \hline
\end{array}
$$

2.2 IP

The state of an IP system can be captured as the *bag*[7] of datagrams that have been sent but not received and which still have a non-zero TTL value. An IP system is initialised by setting *transfer* to the empty bag.

$$
\begin{array}{|l}
\textit{IP} \\ \hline
\mathit{transfer} : \mathbf{bag}\ \mathit{Datagram} \\ \hline
\forall\, d \sqsubset \mathit{transfer} \bullet d.\mathit{ttl} > 0 \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{IPInit} \\ \hline
\mathit{IP}' \\ \hline
\mathit{transfer}' = [\![\,]\!] \\ \hline
\end{array}
$$

2.3 The *Send* Operation

If IP were reliable, we could send a datagram by simply adding it to the bag of datagrams in transit. However, since datagrams can be corrupted, discarded or duplicated, we need to define *Send* so that each datagram in the final state is *derived* from either the datagram being sent or from one of the datagrams in the original state[8]. We specify this by using *bag derivation* ($\preceq$) as follows:

$$
\begin{array}{|l}
\textit{Send} \\ \hline
\Delta \mathit{IP} \\
d? : \mathit{Datagram} \\ \hline
\mathit{transfer}' \preceq \mathit{transfer} \uplus [\![d?]\!] \\ \hline
\end{array}
$$

A formal definition of $\preceq$ is given below. Informally, $t_1 \preceq t_0$ if every member of t_1 is *derived* from some member of t_0, i.e., t_1 can be obtained from t_0 by corrupting, discarding or duplicating members of t_0[9]. Thus, in the definition of *Send*, *transfer*$'$ is derived from *transfer* and $d?$ by corrupting, discarding or duplicating datagrams.

[7] Since IP datagrams need not be delivered in the order in which they are transmitted, we use a bag rather than a sequence. In addition, since an IP datagram may be duplicated or indeed, two unrelated datagrams may be identical, we use a bag rather than a set.

[8] Strictly speaking we could preserve all datagrams from the original state unchanged, but allowing them to be corrupted, discarded or duplicated does not affect the overall specification.

[9] As we will see below, derivation also caters for TTL values being decremented.

2.4 The *Receive* Operation

Again, if IP were reliable, we could receive a datagram by simply removing it from the bag of datagrams in transit. However, to allow for unreliability, we permit a copy of a datagram to be received **and** the bag of datagrams in transit to be changed so that **any** datagram can be corrupted, discarded or duplicated. In particular, there is no requirement that the datagram received should be removed from the bag of datagrams in transit. Of course, one would expect a good implementation to do its best to remove the received datagram.

$$STATUS ::= yes \mid no$$

Receive

$$\begin{array}{|l}
\Delta IP \\
d! : Datagram \\
delivered! : STATUS \\
\hline
transfer' \preceq transfer \\
delivered! = no \lor (delivered! = yes \land d! \sqsubseteq transfer)
\end{array}$$

2.5 Datagram Derivation

To allow for the possibility of the body of a datagram being corrupted or its TTL value being reduced, we define a relation which specifies when one datagram is *derived* from another datagram, i.e., $d_1 \preceq d_0$ if d_1 is the same as d_0, except that some *small* amount of corruption has occurred to the data of d_0 or the TTL value of d_0 has been reduced. To do this, we assume that we can define when the data segments of two datagrams are approximately equal ($\approx$).

$$\begin{array}{|l}
\_ \approx \_ : \mathbf{seq}\, Byte \leftrightarrow \mathbf{seq}\, Byte \\
\hline
\forall s_1, s_2 : \mathbf{seq}\, Byte \bullet s_1 \approx s_2 \Leftrightarrow \cdots
\end{array}$$

$$\begin{array}{|l}
\_ \preceq \_ : Datagram \leftrightarrow Datagram \\
\hline
\forall d_0, d_1 : Datagram \bullet \\
\quad d_1 \preceq d_0 \Leftrightarrow \\
\qquad d_1.source = d_0.source \land d_1.destination = d_0.destination \land \\
\qquad d_1.ttl \leq d_0.ttl \land d_1.data \approx d_0.data
\end{array}$$

We can extend datagram derivation to define *bag derivation* as follows:

$$\begin{array}{|l}
\_ \preceq \_ : \mathbf{bag}\, Datagram \leftrightarrow \mathbf{bag}\, Datagram \\
\hline
\forall b_0, b_1 : \mathbf{bag}\, Datagram \bullet (b_1 \preceq b_0 \Leftrightarrow (\forall d_1 \sqsubseteq b_1 \bullet \exists d_0 \sqsubseteq b_0 \bullet d_1 \preceq d_0))
\end{array}$$

2.6 Properties

Given the operations defined above, we can use composition to derive new schemas to explore the behaviour of an IP system. For example, given datagrams d_0 and d_1, we can define the behaviour (A_0) of sending these datagrams, followed by a single *Receive*.

$$A_0 \mathrel{\widehat{=}} IPInit \fatsemi Send[d_0/d?] \fatsemi Send[d_1/d?] \fatsemi Receive$$

The schema A_0 can be rewritten as:

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

$$\begin{array}{|l}
\multicolumn{1}{l}{A_0} \\
\Delta IP \\
d_0, d_1 : Datagram \\
d! : Datagram \\
delivered! : STATUS \\
\hline
transfer' \preceq [\![d_0, d_1]\!] \\
delivered! = no \vee (delivered! = yes \wedge (d! \preceq d_0 \vee d! \preceq d_1)) \\
\hline
\end{array}$$

A_0 states that if we send two datagrams on an IP system with no datagrams in transit, then **if** we receive a datagram it will be derived from one of the two datagram that were sent. In addition, there may be any number of datagrams (derived from the two datagrams sent) waiting to be received.

2.7 TTL Handling and the *Tick* Operation

The defi nition of *datagram derivation* given above, requires the TTL value in the derived datagram to be less than or equal to the TTL value in the original datagram[10]. Therefore, with the operations *Send* and *Receive*, datagrams in transit can have their TTL values reduced; provided that they are always greater than zero[11].
For TTL handling, we have a stronger requirement, i.e., TTL values **must** be reduced every second. Therefore, we defi ne *strict derivation* ($\prec$) for both datagrams and bags.

$$\begin{array}{|l}
\_ \prec \_ : Datagram \leftrightarrow Datagram \\
\hline
\forall d_0, d_1 : Datagram \bullet (d_1 \prec d_0 \Leftrightarrow d_1 \preceq d_0 \wedge d_1.ttl < d_0.ttl)
\end{array}$$

$$\begin{array}{|l}
\_ \prec \_ : \mathbf{bag}\, Datagram \leftrightarrow \mathbf{bag}\, Datagram \\
\hline
\forall b_0, b_1 : \mathbf{bag}\, Datagram \bullet (b_1 \prec b_0 \Leftrightarrow (\forall d_1 \sqsubseteq b_1 \bullet \exists d_0 \sqsubseteq b_0 \bullet d_1 \prec d_0))
\end{array}$$

Using strict derivation we can defi ne an operation *Tick*, which is performed every second, to reduce the TTL values of datagrams in transit.

$$\begin{array}{|l}
\multicolumn{1}{l}{Tick} \\
\Delta IP \\
\hline
transfer' \prec transfer \\
\hline
\end{array}$$

2.8 Useful Relations and Functions

For convenience, we defi ne a useful relation (*in*) and a useful function ($\sharp$).
The relation *in* determines if a bag of datagrams contains a datagram that was (or could have been) derived from a given datagram.

$$\begin{array}{|l}
\_ \; in \; \_ : Datagram \leftrightarrow \mathbf{bag}\, Datagram \\
\hline
\forall d : Datagram;\ \forall b : \mathbf{bag}\, Datagram \bullet (d \; in \; b \Leftrightarrow (\exists d' \sqsubseteq b \bullet d' \preceq d))
\end{array}$$

The function $\sharp$ returns the maximum TTL value of the datagrams in a bag that have been derived from a given datagram, i.e., $d \sharp b$ is the maximum TTL of any datagram in b that was (or could have been) derived from d.

$$\begin{array}{|l}
\_ \sharp \_ : Datagram \times \mathbf{bag}\, Datagram \rightarrow \mathbb{N} \\
\hline
\forall d : Datagram;\ \forall b : \mathbf{bag}\, Datagram \bullet (d \sharp b = max(\{0\} \cup \{d' \sqsubseteq b \mid d' \preceq d \bullet d'.ttl\}))
\end{array}$$

Note that *in* returns 0 if there are no datagrams in the bag that were derived from the given datagram.

[10]Since the TTL component of a datagram is protected by the checksum, it can never be corrupted.
[11]Datagrams with zero TTL values are not permitted by the invariant of the schema *IP*.

2.9 Properties involving *Tick*

All operations on *IP* (*Send*, *Receive* and *Tick*) are non-deterministic[12]. The specifications of *Send* and *Receive* are very weak in that they only require the output bag of datagrams $transfer'$ to be derived from the input bag *transfer* and (in the case of *Send*) the datagram being sent. This leads to the possibility of states that would, in practice, not be produced by an implementation. For example, a *Send* of a datagram *d*, followed by an infinite number of *Receive* operations each returning *d*, is a possible behaviour.
However, the *Tick* operation is much more constrained. Like *Send* and *Receive*, *Tick* requires its output bag $transfer'$ to be derived from its input bag *transfer*, but the TTL value of each datagram in *transfer* is strictly less than the TTL value of the datagram from which it was derived. Thus, while each *Tick* operation can result in a datagram being corrupted and replicated an arbitrary number of times, after a suitably large number of *Tick* operations, all these replicates will have been received or discarded.
For example, consider the schema A_1 specified by the composition a *Send* operation followed by a number of *Tick* operations:

$$A_1 \mathrel{\widehat{=}} Send \mathbin{⨟} Tick^N$$

where N is the TTL value of the datagram $d?$ sent by the *Send* operation.
For each *Tick* operation, we can show that the maximum TTL value of datagrams derived from $d?$ is reduced, i.e.,

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Tick}} \\ \hline
\Delta IP \\ \hline
transfer' \prec transfer \\
\forall d : Datagram \bullet d \sharp transfer' \neq 0 \Rightarrow d \sharp transfer' < d \sharp transfer \\ \hline
\end{array}$$

Therefore, by induction over the maximum values in each intermediate state, we can show:

$$\begin{array}{|l}
\multicolumn{1}{l}{A_1} \\ \hline
\Delta IP \\
d? : Datagram \\ \hline
transfer' \prec transfer \\
d? \sharp transfer = 0 \Rightarrow d? \sharp transfer' = 0 \\ \hline
\end{array}$$

A_1 states that datagrams derived from the sent datagram $d?$ will only be present in $transfer'$ if they were already in *transfer*, i.e., all datagrams introduced by the *Send* operation will have been removed from IP after N ($d?.ttl$) *Ticks*. We can also express this as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{A_1} \\ \hline
\Delta IP \\
d? : Datagram \\ \hline
transfer' \prec transfer \\
\neg\ (d?\ in\ transfer) \Rightarrow \neg\ (d?\ in\ transfer') \\ \hline
\end{array}$$

2.10 Ordering of Received Messages

As shown by A_0 above, in general, if two datagrams are sent and one is received, then the received datagram will be derived from either of the two datagrams that were sent. This result can easily be extended to show that in general, if two datagrams are sent and received, the order of reception is not fixed.

[12]This non-determinism is not just the specification of a *don't care* property; it is assumed that implementations of IP will be non-deterministic.

However, if two datagrams are sent with a suffi cient number of intervening *Tick* operations, then if they are received, they must be received in the order in which they were sent. To express this property in Z requires us to consider **all** sequences of operations characterised by this requirement.
Let use assume that we send d_0 and d_1 in that order, i.e., we are interested in sequences of operations of the form:

$$A_2 \mathrel{\widehat{=}} \mathit{IPInit} \mathbin{⨟} \mathit{Send}[d_0/d?] \mathbin{⨟} \mathcal{A} \mathbin{⨟} \mathit{Send}[d_1/d?] \mathbin{⨟} \mathcal{B}$$

where:

- $\mathcal{A}$ is the composition of one or more receive operations with one of the form $\mathit{Receive}[d_0'/d!, \mathit{first}/\mathit{delivered}!]$, interleaved with at least $d_0.\mathit{ttl}$ *Tick* operations.
- $\mathcal{B}$ is the composition of one or more receive operations with one of the form $\mathit{Receive}[d_1'/d!, \mathit{second}/\mathit{delivered}!]$, interleaved with *Tick* operations.

For **any** such sequence A_2, we can show:

$$\begin{array}{|l}
\multicolumn{1}{l}{A_2} \\ \hline
\Delta \mathit{IP} \\
d_0, d_1 : \mathit{Datagram} \\
\mathit{first}, \mathit{second} : \mathit{STATUS} \\
d_0', d_1' : \mathit{Datagram} \\ \hline
\mathit{transfer}' \preceq \langle\!\langle d_1 \rangle\!\rangle \\
\mathit{first} = \mathit{no} \lor (\mathit{first} = \mathit{yes} \land d_0' \preceq d_0) \\
\mathit{second} = \mathit{no} \lor (\mathit{second} = \mathit{yes} \land d_1' \preceq d_1) \\ \hline
\end{array}$$

3 Specification of IP using the Temporal Logic of Actions

3.1 A Quick Overview of TLA

3.1.1 Introduction

Temporal logic extends classical logic by handling modalities such as fairness and eventuality. TLA [7] is a temporal logic of actions for specifying and reasoning about concurrent systems. Systems and their properties are specifi ed in the same logic, so the assertion that a system meets its specifi cation and the assertion that one system implements another are both expressed by logical implication. TLA has four levels:

1. Constants: this level is concerned with formulas which are state-independent (*constant*). The variables which are used in this level are called *rigid variables* and cannot change values between states.

2. States: this level allows reasoning about individual states. The formulas in this level can either be *state functions* (non-boolean functions) or *state predicates* (boolean expressions). The variables used in this level can be *flexible variables*, which can change values between states.

3. Pairs of States: this level concerns reasoning about pairs of states. The formulas in this level can either be *transition functions* or *transition predicates* (*actions*). The variables used in this level can be primed. Unprimed variables refer to the old state, primed variables to the new. For a state function or predicate f, f' is obtained by replacing each flexible variable v in f by v'.

   An action $\mathcal{A}$ is a relation between states which assigns a boolean to $s[\![\mathcal{A}]\!]t$ for states s and t, where s is the old state and t is the new state. This is called an "$\mathcal{A}$ step" if it is true. *enabled* $\mathcal{A}$ is true for a state if it is possible to take an $\mathcal{A}$ step starting in that state.

   Stuttering steps are steps in which specifi ed variables do not change. These are used to help show equivalences between behaviours. For an action $\mathcal{A}$ and a state function f, a stuttering step is written as follows:

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

$$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f)$$

A step corresponding to an action $\mathcal{A}$ in which the variables in a state function f do change is written as follows:

$$< \mathcal{A} >_f \triangleq \mathcal{A} \wedge (f' \neq f)$$

4. Sequences of States: the fourth level allows reasoning about behaviours, which are infinite sequences of states. A behaviour is denoted by $< s_0, s_1, s_2, ... >$, where s_0 is the first state, s_1 is the second, etc.

   An action $\mathcal{A}$ is true iff the first pair of states in the behaviour is an $\mathcal{A}$ step.

$$< s_0, s_1, s_2, ... > [\![\mathcal{A}]\!] \triangleq s_0 [\![\mathcal{A}]\!] s_1$$

3.1.2 TLA Formulas

A TLA formula can include the following forms:

P	Satisfied by a behaviour iff P is true for the initial state.
$\Box[\mathcal{A}]_f$	Satisfied by a behaviour iff every step satisfies $\mathcal{A}$ or leaves f unchanged.
$\Box F$	Satisfied by a behaviour iff F is true for all suffixes of the behaviour (F is always true). $< s_0, s_1, s_2, ... > [\![\Box F]\!] \triangleq \forall n \in Nat :< s_n, s_{n+1}, s_{n+2}, ... > [\![F]\!]$
$\Diamond F$	Defined to be $\neg\Box\neg F$ (F is eventually true). $< s_0, s_1, s_2, ... > [\![\Diamond F]\!] \triangleq \exists n \in Nat :< s_n, s_{n+1}, s_{n+2}, ... > [\![F]\!]$
$\mathrm{WF}_f(\mathcal{A})$	Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is infinitely often not enabled, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur (weak fairness). $\mathrm{WF}_f(\mathcal{A}) \triangleq \Box\Diamond < \mathcal{A} >_f \vee \Box\Diamond\neg Enabled < \mathcal{A} >_f$
$\mathrm{SF}_f(\mathcal{A})$	Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is only finitely often enabled, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur (strong fairness). $\mathrm{SF}_f(\mathcal{A}) \triangleq \Box\Diamond < \mathcal{A} >_f \vee \Diamond\Box\neg Enabled < \mathcal{A} >_f$

The specification of a system in TLA has the following form:

$$Initial \wedge \Box[N]_x \wedge F$$

where *Initial* is a state predicate describing the initial state, N is a transition predicate describing the possible steps, x is a state function indicating the variables which cannot change in any step other than a N step, and F is the conjunction of fairness conditions.

3.2 Specification of the IP Layer Interface Using TLA+

We now give a specification[13] of the IP layer interface using TLA+, a specification language which is an extension of TLA and includes additional data types, operators and modules.

constants

DATAGRAMS

DATAGRAMS is the set of valid datagrams

variables

sent, *received*, *transit*

These three variables represent the current state: *sent* is a set containing all the datagrams which have been sent so far, *transit* is a set containing all the valid datagrams which are still in transit, and *received* is a set containing all the

[13]The proofs have been banished to the appendices

datagrams which have been received so far. A datagram is represented as a four-tuple giving its source, destination, time to live and message. The actions which can be performed are as follows:

$$\begin{aligned} \mathit{Send}(\mathit{datagram}) \triangleq\ & \land\ \mathit{sent}' = \mathit{sent} \cup \{\mathit{datagram}\} \\ & \land\ \mathit{transit}' = \mathit{transit} \cup \{\mathit{datagram}\} \\ & \land\ \textsc{unchanged}\ \mathit{received} \end{aligned}$$

When a datagram is sent, it is added to the set of sent datagrams, and also to the set of valid datagrams which are currently in transit. The set of received datagrams is not affected.

$$\begin{aligned} \mathit{Receive}(\mathit{datagram}) \triangleq\ & \land\ (\mathit{datagram} \in \mathit{transit}) \\ & \land\ \mathbf{let}\ \mathit{dg} \triangleq \mathbf{choose}\ d : \land\ (d \in \mathit{transit}) \\ & \qquad\qquad\qquad\qquad\quad \land\ d = \mathit{datagram} \\ & \quad \mathbf{in}\ \ \mathit{received}' = \mathit{received} \cup \{\mathit{dg}\} \\ & \land\ \textsc{unchanged}\ < \mathit{sent}, \mathit{transit} > \end{aligned}$$

A datagram which is received must be one which is currently in transit. This is not removed from the set of datagrams in transit as it may have been duplicated. The datagram is added to the set of those which have been received.

$$\begin{aligned} \mathit{Timeout}(\mathit{datagram}) \triangleq\ & \land\ (\mathit{datagram} \in \mathit{transit}) \\ & \land\ \mathbf{let}\ \mathit{dg} \triangleq \mathbf{choose}\ d : \land\ (d \in \mathit{transit}) \\ & \qquad\qquad\qquad\qquad\quad \land\ d = \mathit{datagram} \\ & \quad \mathbf{in}\ \ \mathit{transit}' = \mathit{transit} - \{\mathit{dg}\} \\ & \land\ \textsc{unchanged}\ < \mathit{sent}, \mathit{received} > \end{aligned}$$

A datagram which is timed out must be one which is currently in transit. This is removed from the set of datagrams which are in transit, but the sets of datagrams which have been sent and received are not affected.

$$\begin{aligned} \mathit{Change}(\mathit{datagram}) \triangleq\ & \land\ (\mathit{datagram} \in \mathit{transit}) \\ & \land\ \mathbf{let}\ \mathit{dg}1 \triangleq \mathbf{choose}\ \mathit{dg} : \land\ (\mathit{dg} \in \mathit{transit}) \\ & \qquad\qquad\qquad\qquad\qquad \land\ \mathit{dg} = \mathit{datagram} \\ & \quad \mathbf{in}\ \ \mathbf{let}\ \mathit{dg}2 \triangleq \mathbf{choose}\ \mathit{dg} : \land\ (\mathit{dg} \in \mathit{DATAGRAMS}) \\ & \qquad\qquad\qquad\qquad\qquad\quad \land\ \exists\ s, d, t, m1, m2 : \land\ \mathit{dg}1 = \ < s, d, t, m1 > \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{dg} = \ < s, d, t, m2 > \\ & \qquad \mathbf{in}\ \ \mathit{transit}' = \mathit{transit} \cup \{\mathit{dg}2\} \\ & \land\ \textsc{unchanged}\ < \mathit{sent}, \mathit{received} > \end{aligned}$$

Here, we model the corruption of datagrams which are currently in transit. We only include the case where the header of the datagram remains unchanged, as these are the only ones which will be valid. Although the actual message will not be changed very much, we do not specify this. The original uncorrupted datagram is not removed from the set of datagrams in transit as it may have been duplicated. If the result of corruption produces invalid datagrams, the set of datagrams in transit will therefore not be affected.

$$\mathit{Transition} \triangleq\ \exists\ \mathit{dg} \in \mathit{DATAGRAMS} : \mathit{Send}(\mathit{dg}) \lor \mathit{Receive}(\mathit{dg}) \lor \mathit{Timeout}(\mathit{dg}) \lor \mathit{Change}(\mathit{dg})$$

Each transition within a behaviour must be a send, receive, timeout or change action.

$$\mathit{Fairness} \triangleq\ \forall\ d \in \mathit{DATAGRAMS} : \mathit{WF}_{\langle \mathit{sent}, \mathit{transit}, \mathit{received} \rangle}(\mathit{Timeout}(d))$$

Messages which remain in transit are eventually timed out.

$$Init \triangleq (sent = \{\}) \land (transit = \{\}) \land (received = \{\})$$

Initially, no messages have been sent, received or are in transit.

$$Specification \triangleq Init \land \Box\,[Transition]_{\langle sent, transit, received \rangle} \land Fairness$$

3.3 No Message Creation

In order to prove that no messages are created, we prove that if a message is received with a specified source and destination, then a message must also have been sent with the same source and destination. This property is stated as follows:

$$NoCreation \triangleq \forall s, d, t1, m1 : \Box(< s, d, t1, m1 > \in received \Rightarrow \exists t2, m2 : < s, d, t2, m2 > \in sent)$$

To prove this invariant, we prove the following two properties:

$$NoCreation1 \triangleq \forall s, d, t1, m1 : \Box(< s, d, t1, m1 > \in transit \Rightarrow \exists t2, m2 : < s, d, t2, m2 > \in sent)$$

$$NoCreation2 \triangleq \forall s, d, t1, m1 : \Box(< s, d, t1, m1 > \in received \Rightarrow \exists t2, m2 : < s, d, t2, m2 > \in transit)$$

$NoCreation1$ can be proved by induction on behaviours:

1. $Init \land \Box[Transition]_{\langle sent, transit, received \rangle} \Rightarrow NoCreation1$
   Proof Outline:
   1.1. $Init \Rightarrow NoCreation1$
   1.2. $NoCreation1 \land Send(dg) \Rightarrow NoCreation1'$
   1.3. $NoCreation1 \land Receive(dg) \Rightarrow NoCreation1'$
   1.4. $NoCreation1 \land Timeout(dg) \Rightarrow NoCreation1'$
   1.5. $NoCreation1 \land Change(< s, d, t1, m1 >) \Rightarrow NoCreation1'$
   1.6. $NoCreation1 \land < sent, transit, received > = < sent', transit', received' > \Rightarrow NoCreation1'$
   1.7. Q.E.D.

$NoCreation2$ can now also be proved by induction on behaviours:

2. $Init \land \Box[Transition]_{\langle sent, transit, received \rangle} \Rightarrow NoCreation2$
   Proof Outline:
   2.1. $Init \Rightarrow NoCreation2$
   2.2. $NoCreation2 \land Send(dg) \Rightarrow NoCreation2'$
   2.3. $NoCreation2 \land Receive(dg) \Rightarrow NoCreation2'$
   2.4. $NoCreation2 \land Timeout(dg) \Rightarrow NoCreation2'$
   2.5. $NoCreation2 \land Change(< s, d, t1, m1 >) \Rightarrow NoCreation2'$
   2.6. $NoCreation2 \land < sent, transit, received > = < sent', transit', received' > \Rightarrow NoCreation2'$
   2.7. Q.E.D.

3.4 Ordering of Received Messages

Here we prove that if two datagrams are received, then if they are sent far enough apart they will be received in the same order in which they were sent. This property can be stated as follows:

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

$(\Diamond\Box(dg_1 \in sent \wedge dg_1 \in received \wedge dg_2 \in received)) \Rightarrow$
$\Diamond\Box(dg_2 \notin sent \Rightarrow (dg_1 \in received \wedge dg_2 \notin received))$

Before we can prove this property, we need to make the following assumption:

$$(dg \notin sent \wedge \Diamond\Box(dg \in received)) \Rightarrow dg \notin received \tag{1}$$

This states that any message which is eventually received and has not yet been sent cannot have already been received. We can now prove the bounded reliability property as follows:

3. $(\Diamond\Box(dg_1 \in sent \wedge dg_1 \in received \wedge dg_2 \in received)) \Rightarrow$
   $\Diamond\Box(dg_2 \notin sent \Rightarrow (dg_1 \in received \wedge dg_2 \notin received))$
   Proof Outline:
   3.1. $\Diamond\Box(dg_1 \in received)$
   3.2. $\Diamond\Box(dg_2 \notin sent \Rightarrow (dg_2 \notin received))$
   3.3. Q.E.D.

4 Specification of IP in Linear Logic

4.1 A Crash Course in Linear Logic

One way to look at a logic is to divide its rules into three categories:

1. *Axioms*. These are the defi ned truths of the logical system. In intuitionistic logic, the sole axiom represents the tautology that from hypothesis *A* one can deduce hypothesis *A*.

2. *Structural rules*. These specify how hypotheses can be manipulated. In intuitionistic logic these are the Exchange, Contraction and Weakening rules.

3. *Logical rules*. These defi ne the logical connectives, which in the case of intuitionistic logic are conjunction($\times$), disjunction($+$) and implication($\supset$).

Linear logic[5, 6] belongs to the family of sub-structural logics. These logics remove, or weaken, the *structural rules* of Exchange, Contraction and Weakening. In order to remain expressive they typically introduce addition *logical rules*. In the case of linear logic, the Contraction rule, which allows hypothesis to be duplicated and the Weakening rule, which allows hypothesis to be discarded, are removed. The effect of this is to make the logic "resource conscious". You cannot duplicate or discard hypotheses. If you have one instance of a hypothesis *A* then you cannot create additional instances of *A* out of mid-air; nor can you simply sweep *A* under the carpet and pretend you never had *A*.
Though linear logic has lost two *structural rules* it has gained a series of *logical rules*. These are:

- Linear implication is written $A \multimap B$ and is pronounced "consume *A* yielding *B*. If you have two hypotheses *A* and $A \multimap B$ then you can derive *B* but the hypothesis *A* has been consumed and is no longer available.

- Multiplicative conjunction is written as $A \otimes B$ and is pronounced "both *A* and *B*". When used, both hypothesis are consumed and are no longer available.

- Additive conjunction is written $A \& B$ and is pronounced "choose from *A* and *B*". When used, you have the choice of which hypothesis is consumed and no longer available.

- Additive disjunction is written $A \oplus B$ and is pronounced "either *A* or *B*". When used, it represents an external choice as to which hypothesis is consumed and no longer available.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

- Exponentiation is written $!A$ and is pronounced "of course A". This is a new form of connective which represents a producer of the hypothesis A. It allows us to represent *intuitionistic truth* in linear logic. We can also use it to define intuitionistic implication $\supset$ as $(!A) \multimap B$.

Assumptions can be *linear*, written $<A>$, or *intuitionistic*, written $[A]$. Though assumptions of the form $<!A>$ are in a sense equivalent to assumptions of the form $[A]$ there is very important difference. Linear assumption, including those of the form $<!A>$, can only be used *once* in a proof whereas intuitionistic assumptions may be used any number of times.
Judgements are written $\Gamma \vdash A$. Only the assumptions in the judgement are labelled with square or angle brackets and these brackets may only appear on the left hand side of $\vdash$ and never on the right. Alternatively judgements can be written $\Gamma;\ \Delta \vdash A$ which states that with the unrestricted intuitionistic context Γ and the restricted linear context Δ we can judge A to be a true conclusion. With the combination of intuitionistic and linear hypotheses we need an additional rule called *Dereliction* to allow us to copy an intuitionistic hypothesis to a linear hypothesis.

4.2 Specification of IP Layer Interface

There are two operations available to the user of the IP layer:

1. $Send(x, y, ttl, m)$
   Sends a message m from node x to node y with a "time to live" value of ttl.
2. $Rcv(x, y, ttl, m)$
   Receives a message m from node x to node y with a "time to live" value of ttl.

The equations that define the user interface to the IP layer are:

$\forall x.\forall y.\forall ttl.\forall m.$

$$Send(x, y, ttl, m) \multimap Datagram(x, y, lower(ttl), m) \tag{2}$$

Sending a message adds a single datagram to the system. *lower* is a function that reduces its operand to some non-zero integer in the range $(0, t]$.

$\forall x.\forall y.\forall ttl.\forall m.$

$$Datagram(x, y, ttl, m) \multimap (Datagram(x, y, lower(ttl), m) \otimes Datagram(x, y, lower(ttl), m)) \oplus \mathbf{1} \tag{3}$$

A datagram can be duplicated, or be lost.

$\forall x.\forall y.\forall ttl.\forall m.$

$$\begin{aligned} Datagram(x, y, ttl, m) \otimes Listen(y) \multimap\ & Listen(y) \otimes \\ & (Rcv(x, y, lower(ttl), m) \oplus Rcv(x, y, lower(ttl), change(m))) \end{aligned} \tag{4}$$

If a datagram addressed to node y exists and node y is listening for it, then node y will receive the message m or some corrupted version $change(m)$ of the message m.

4.3 Verification

The IP specification guarantees very little about a message sent from one node to another; however, as mentioned earlier we can prove results relating to the origin of a valid datagram, and the ordering of receipts under certain conditions.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

4.3.1 No message appearing from mid-air

If a node receives a properly formatted message some node must have sent a message with the same header (but the message may be corrupted). In fact, if the initial datagram is not lost, a messages sent from node A to node B may result in one or more messages, with correct header, being received by node B.
The unrestricted context Γ contains the rules for the system, i.e. equations 2,3 and 4. For the remainder of this section we use the following shorthand:

- $D(m)$ for $Datagram(x,y,ttl,m)$.
- $S(m)$ for $Send(x,y,ttl,m)$.
- $R(m)$ for $Rcv(x,y,ttl,m)$.
- L for $Listen(y)$.

There are at least two styles of proofs in linear logic: one which is applicable when there is an overall goal (such as planning), and the other when there is no overall goal but there is an evolution of state. In the latter case the right-hand side of the judgement should be "empty", which is modelled by **0**, the impossible goal. Then the following partial derivation shows that, given a set of rules Γ, the state Δ_0 can evolve into state Δ_1.

$$\begin{array}{c} \Gamma;\ \Delta_1 \vdash \mathbf{0} \\ \vdots \\ \Gamma;\ \Delta_0 \vdash \mathbf{0} \end{array}$$

If we assume that $j-1$ copies of the sent message (or a similar message with the correct header) were received then there exits a previous state that can also evolve into the reception of j copies of the sent message. Thus, letting the multiplicative conjunction of N linear assumptions A be denoted by $\otimes_{i=1}^{N} A$, we can prove

$$\begin{array}{c} \Gamma;\ L, D(m), \otimes_{i=1}^{N}(R(m)\oplus R(change(m))) \vdash \mathbf{0} \\ \vdots \\ \Gamma;\ S(m), L \vdash \mathbf{0} \end{array} \tag{5}$$

4.3.2 Ordering of received messages

If two messages m_1 and m_2 are sent and successfully delivered then the precondition necessary to guarantee that m_1 arrives before m_2 is that the interval n between sending m_1 and m_2 is greater than m_1's *time to live* value.
Let $T_s(E)$ be a function that returns the time of an event E as seen by the *sender's* clock. Events can be either $Send()$ or $Rcv()$ operations.
A corollary of 5 is that before a message is received it must have been sent, and therefore the time a message is sent (as seen by the sender's clock) is less than or equal to the time the message was received (as seen by the sender's clock). Since equations 2 to 4 are only valid when $0 < ttl' \leq ttl$, the following axiom is true.

$$\begin{aligned} &\forall x. \forall y. \forall ttl. \forall ttl' \mid 0 < ttl' \leq ttl. \forall m. \\ &\quad T_s(Send(x,y,ttl,m)) \leq T_s(Rcv(x,y,ttl',m)) \leq T_s(Send(x,y,ttl,m)) + ttl \end{aligned} \tag{6}$$

Therefore, if $T_s(Rcv(x,y,ttl'_1,m_1)) \leq T_s(Rcv(x,y,ttl'_2,m_2))$ then:

$$\begin{aligned} &(T_s(Send(x,y,ttl_1,m_1)) \leq T_s(Rcv(x,y,ttl'_1,m_1)) \leq T_s(Send(x,y,ttl_1,m_1)) + ttl_1) \\ \leq\ &(T_s(Send(x,y,ttl_2,m_2)) \leq T_s(Rcv(x,y,ttl'_2,m_2)) \leq T_s(Send(x,y,ttl_2,m_2)) + ttl_2) \end{aligned}$$

and thus:

$$(T_s(Send(x,y,ttl_1,m_1)) + ttl_1) \leq T_s(Send(x,y,ttl_2,m_2))$$

Let message m_2 be sent n seconds after message m_1, then:

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

$$(T_s(Send(x,y,ttl_1,m_1))+ttl_1) \leq (T_s(Send(x,y,ttl_1,m_1))+n)$$
$$\Rightarrow ttl_1 \leq n$$

Hence, if message m_1 is received before message m_2 then the sender must have waited for more then ttl_1 between sending message m_1 and sending message m_2.

5 Specification of IP using the Calculus of Constructions

5.1 Constructive Logic and Coq

Constructive logic can be seen as a variation of classical logic where we remove the “law of the excluded middle” ($A \vee \neg\, A$) and related axioms, such as proof by contradiction. This results in a system where a proposition A is true precisely when there exists a proof of A, as opposed to classical logic which will also allow a refutation of $\neg\, A$ as evidence. Under the standard Curry-Howard isomorphism we can identify each proof of a proposition A with a program implementing the specifi cation A; logical conjunction, disjunction and implication translate into product, sum and function types respectively.
Our specifi cation below was developed using the Coq [8] proof assistant, based on the Calculus of Inductive Constructions [9]. As well as the normal benefi ts of a proof assistant such as uniformity of notation and verifi cation of type-correctness, Coq also provides three enhancements to ordinary constructive logic:

- Coq implements a higher-order constructive logic, facilitating in particular, the descriptions of object logics within the framework
- Coq has two type hierarchies: `Set` of constructive types, and `Prop` for classical logic. This allows specifi cations to be developed in ordinary classical logic, and then verifi ed in the same framework against programs written within `Set`.
- Coq supports inductive (and co-inductive) defi nitions, giving a natural logical extension of the defi nition-by-cases style of programming found in functional languages

5.2 Datagrams

The basic unit of communication between the IP layer and the network layer is the datagram, which encapsulates the data being sent, and various pieces of auxiliary information. Since we do not propose to treat the components of a datagram in any great detail, we may take the type of these components as given, declaring:

```
Parameter IPaddress : Set.
Parameter data : Set.
```

Our datagram then can be regarded as a record, the fi elds of which are described in section 3.1 of [1]. Abstracting from this, we specify:

```
Record datagram : Set :=
mkDG
  source : IPaddress;
  dest : IPaddress;
  ttl : nat;      (*  Time to live *)
  checksum : nat;
  contents : data
.
```

Next we assume that a function to re-calculate a checksum for a datagram is available, and state that this calculation is independent of the value in the checksum fi eld. Based on this, we can specify what is means for a datagram to be valid:

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

```
Parameter calcChecksum : datagram -> nat.

(*  Checksum is independent of the checksum field *)
Axiom howToCalc :
  (dg:datagram)
  let cs = (calcChecksum dg) in
  (calcChecksum (mkDG (source dg) (dest dg) (ttl dg) O (contents dg))) =
  (calcChecksum (mkDG (source dg) (dest dg) (ttl dg) cs (contents dg))).

Definition validDG : datagram -> Prop :=
  [dg:datagram]((checksum dg) = (calcChecksum dg)).
```

Here `validDG` is defined to be a function from `datagram`s to `Prop` - i.e. a predicate over datagrams; the square brackets are used to delimit the argument here.
We can also define a function `newDG` that constructs a new datagram from pieces of information, and construct a proof `ValidNewDG` to demonstrate that a datagram so constructed satisfies the validity property above.[14]

```
Definition newDG :=
  [ newSource:IPaddress; newDest:IPaddress; newTTL:nat; newContents:data ]
  let newCS = (calcChecksum (mkDG newSource newDest newTTL O newContents))
  in  (mkDG newSource newDest newTTL newCS newContents).

Lemma ValidNewDG :
  (src,dst:IPaddress)(dt:data)(tm:nat)
  (validDG (newDG src dst tm dt)).
(*  Proof omitted *)
```

5.3 Sending and Receiving

Since we have omitted many details of the internal operation of the IP layer, it is hardly any surprise that we choose a simple interface between the IP and network layers. The IP layer must be able to dispatch and collect datagrams to and from the network layer; we assume that this is achieved via two functions, which we call `depart` and `arrive` respectively.
Our specification of the IP interface naturally centres on the two main functions: SEND and RECV. In each case the main purpose of the definition is to explicitly link a successful SEND or RECV with a corresponding departure or arrival action.
Allowing for the possible error conditions specified in section 3.3 of [1], we given an inductive definition for the SEND function:

```
Variable departure : IPaddress -> datagram -> Set.

Inductive SEND : IPaddress -> IPaddress -> nat -> data -> Set :=
  sendOk :
    (src,dst:IPaddress)(tm:nat)(BufPTR:data)
    (departure src (newDG src dst tm BufPTR)) ->
    (SEND src dst tm BufPTR)
| sendError :
    (src,dst:IPaddress)(tm:nat)(BufPTR:data)
    (SEND src dst tm BufPTR).
```

[14]The variables introduced in a lemma or axiom between ordinary parenthesis are taken to be universally quantified.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

Again, variables between square brackets are arguments, those between ordinary brackets in each of the two cases are universally quantified; the "`->`" in the `sendOk` case denotes implication. Giving this as an inductive definition means that if a successful SEND has occurred we can be sure that a new datagram (which is valid by the previous lemma) has departed onto the network layer.
The definition of RECV is similar, except now we omit the error case assuming that we block until a successful receipt:

```
Variable arrival : IPaddress -> datagram -> Set.

Inductive RECV : IPaddress -> IPaddress -> data -> Set :=
  recvOk :
    (dg:datagram)(arrival (dest dg) dg) -> (validDG dg) ->
    (RECV (source dg) (dest dg) (contents dg)).
```

Note that the definition abstracts away from details of datagrams which have arrived but have not answered a RECV call, or RECV calls pending that have not been satisfied by a corresponding arrive. A more model-oriented specification might seek to implement this functionality with queues of datagrams, but this is not necessary for our purposes.

5.4 A Basic Consistency Proof

As a simple example of a consistency proof, we wish to show that any valid datagram received from the IP layer must have been sent from the host indicated in the header.
To do this, we must assume that any datagram "in transit" has indeed been sent by the indicated source:

```
(*  Any valid DG must have departed from its source *)
Axiom SendValid :
  (dg:datagram)(validDG dg) -> (departure (source dg) dg).
```

A similar assertion cannot, of course, be given for `arrive`, since the datagram may be lost at any stage. The proof is now straightforward, centred around the above assumption:

```
(*  Theorem : Any Received DG has been sent by its source *)
Lemma RecvMeansSend :
  (dg:datagram)
  (r:(RECV (source dg) (dest dg) (contents dg)))
  {tm:nat & (SEND (source dg) (dest dg) tm (contents dg))}.
(*  Proof Omitted *)
```

The statement of the lemma can be read as: "for any datagram `dg` and any answered RECV `r`, there exists some number `tm` (to act as the original time-to-live value) such that there was a corresponding SEND call". The last line in the statement beginning "`{ tm:nat ...`" is actually the specification of a set; in constructive logic we "prove" this specification by showing how an element of that set can be built.
While the lemma itself is quite simple, it did play a useful role in the construction of the specification, serving as a validation that the requisite elements were present. Indeed, the process of constructing a specification and then a consistency proof is often an iterative one, since an inability to build the proof indicates a gap in the specification. We suggest that this is one particular advantage of using a proof assistant, since such proofs (particularly relatively simple ones) are often just "assumed" in a specification constructed by hand, occasionally leaving omissions unexposed.

5.5 Time and Ordering of Received Messages

Our second proof has a temporal element - we wish to specify that the time-to-live field places a bound on the maximum amount of time between sending and receiving a datagram. Thus we choose at this point to introduce what is essentially a temporal variable in the form of a `timeOf` function for send and receive events.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

First of all, for convenience, we introduce the type `Event` as a common name for sends and receipts:

```
Inductive Event : Set :=
  sndEvent : (src,dst:IPaddress)(tm:nat)(BufPTR:data)
    (SEND src dst tm BufPTR) -> Event
| recEvent : (src:IPaddress)(dst:IPaddress)(BufPTR:data)
    (RECV src dst BufPTR) -> Event.

Coercion sndEvent : SEND >-> Event.
Coercion recEvent : RECV >-> Event.
```

Next, we assume a function to tell us when a particular event occurred:

```
Variable timeOf : Event -> nat.
```

Now we must state the relevant properties of our events; in particular, that a RECV always occurs *after* the corresponding SEND, but before the time-to-live fi eld has expired:

```
Axiom recvAfter :
  (dg:datagram)
  (s:(SEND (source dg) (dest dg) (ttl dg) (contents dg)))
  (r:(RECV (source dg) (dest dg) (contents dg)))
  (lt (timeOf s) (timeOf r)).

(*  Time-to-receipt (if any) is bounded above by TTL *)
Axiom timeBound :
  (dg:datagram)
  (s:(SEND (source dg) (dest dg) (ttl dg) (contents dg)))
  (r:(RECV (source dg) (dest dg) (contents dg)))
  (lt (timeOf r) (plus (timeOf s) (ttl dg))).
```

We note here the need to explicitly order the SENDs and RECVs in the axiom `recvAfter`. This arises from the strategy of modularising our specifi cation into two parts: a “logical” part based purely on cause-and-effect, culminating in the lemma `RecvMeansSend`, and a “temporal” part, based around the time at which events occur. Indeed, the axiom `recvAfter` could be seen as the temporal manifestation of the `RecvMeansSend` result.
To verify the existence of a time-bound on liveness, we assume that we have two datagrams, `dg1` and `dg2`, both of which have been successfully received:

```
Variable dg1,dg2:datagram.
(*  First/Second datagram sent and received *)
Variable s1:(SEND (source dg1) (dest dg1) (ttl dg1) (contents dg1)).
Variable r1:(RECV (source dg1) (dest dg1) (contents dg1)).
Variable s2:(SEND (source dg2) (dest dg2) (ttl dg2) (contents dg2)).
Variable r2:(RECV (source dg2) (dest dg2) (contents dg2)).
```

Next we assume that the time between sending the fi rst datagram exceeded the time-to-live of the second. The expected result - that the second datagram can only then arrive *after* the fi rst - follows by simple transitivity:

```
(*  Time between sends exceeds TTL for first datagram *)
Hypothesis SendGap : (lt (plus (timeOf s1) (ttl dg1)) (timeOf s2)).

Lemma recvOrdered :
  (lt (timeOf r1) (timeOf r2)).
(*  Proof Omitted *)
```

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

6 Conclusions

The case study chosen is relatively small and certainly does not cover all aspects of each of the chosen formalisms. However it *is* an example of a “real” problem that is independent of any of the natural domains of each of the given logics. In addition, the specifi cations themselves provide a flavour of each of the formalisms, and serve as a basis for some comparative remarks.
In each case the specifi cations were naturally modularised around the two main proofs: one relating solely to the send and receive functions, the other connecting the two via some kind of ordering. Within these natural sections, we can see that in each case the main units of organisation were the datagram (the main “data-structure”) and the two function calls send and receive.
There is a clear dichotomy here between the Z and TLA specifi cations on the one hand, and the linear and constructive logic based specifi cations on the other.

- The fi rst two are explicitly *state-based*, and in each case this style of specifi cation seemed to lead naturally to the same basic structures - for example, the Z specifi cation has a bag *transfer* of datagrams in transit, with a corresponding set *transit* in the TLA specifi cation. Indeed, we should point out that the choice between using a bag or set here was largely one of individual style, rather than any inherent bias in the formalism used.
- On the other hand, the other two specifi cations can be regarded as being more *declarative*, eschewing an explicit state and relying on their respective logical structures to provide the necessary links between send and receive actions. In particular, in linear logic the connection between valid receipts and corresponding sends was achieved via an explicit transfer of resources over linear implication, but in constructive logic this was achieved by a direct axiomatic link.

It was in the presentation of the second result that a clear difference emerged between Z and TLA. In particular, signifi cant diffi culty was encountered in arriving at a formal specifi cation of this property within the Z schema calculus as the statement of the result was essentially a quantifi cation over sequences of send and receive *schemas*. On the other hand, the built-in temporal operators in TLA provided a direct route towards reasoning about sequences of operations, and thus towards establishing properties regarding the ordering of receive events.
Since neither linear logic nor the constructive formalism used here support temporal operators primitively, the simplest approach in each case was to index events using a relative time-clock, expressed as T_s in the linear-logic specifi cation, and as the `timeOf` function in the Coq specifi cation. Indeed, once this had been introduced the proof of ordering devolved to relations over natural numbers, rather than any distinctive features of the logics used.

6.1 Future Work

The results presented above are part of an ongoing collaboration between the authors in the general area of formal methods and communication protocols. Specifi cally it is hoped that this work can now be used as a basis for the specifi cation of TCP, in particular

- specifying the TCP/user interface
- specifying the TCP protocol (based on the IP specifi cation) and
- verifying implementation consistency between the protocol internals and the TCP/user interface

In addition to providing an interesting case study in the use of the four formalisms, it is hoped that this can serve as a basis for an analysis of the suitability of each of the logics for protocol specifi cation, as well as a foundation for further work in the area.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

References

[1] Postel J (ed). RFC 791 - Internet Protocol. Defense Advanced Research Projects Agency, 1981.

[2] Postel J (ed). RFC 793 - Transmission Control Protocol. Defense Advanced Research Projects Agency, 1981.

[3] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall, 1992.

[4] Hayes I. Specifi cation Case Studies. Prentice Hall, 1993.

[5] Girard J-Y. Linear Logic. Theoretical Computer Science 1987; 50:1-102.

[6] Girard J-Y. On the unity of logic. Annals of Pure and Applied Logic 1993; 59:201-217.

[7] Lamport L. The temporal logic of actions. ACM Transactions on Programming Languages and Systems 1994; 16:872-923.

[8] Cornes C, Courant J, Filliâtre JC, et al. The Coq Proof Assistant Reference Manual, Version 5.10. Rapport Technique No. 177, INRIA, 1995.

[9] Coquand Th., Huet G. The Calculus of Constructions. Information and Computation 1988; 76:95-120.

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

Appendices

A TLA Proofs

1. $Init \wedge \Box[Transition]_{\langle sent, transit, received\rangle} \Rightarrow NoCreation1$
   PROOF:
   1.1. $Init \Rightarrow NoCreation1$
   PROOF: follows since $sent = \{\} \wedge transit = \{\}$
   1.2. $NoCreation1 \wedge Send(dg) \Rightarrow NoCreation1'$
   PROOF: follows since $sent' = sent \cup \{dg\} \wedge transit' = transit \cup \{dg\}$
   1.3. $NoCreation1 \wedge Receive(dg) \Rightarrow NoCreation1'$
   PROOF: follows since $sent' = sent \wedge transit' = transit \cup \{dg\}$
   1.4. $NoCreation1 \wedge Timeout(dg) \Rightarrow NoCreation1'$
   PROOF: follows since $sent' = sent \wedge transit' = transit - \{dg\}$
   1.5. $NoCreation1 \wedge Change(< s, d, t1, m1 >) \Rightarrow NoCreation1'$
   PROOF: follows since $sent' = sent \wedge \exists t2, m2 : transit' = transit \cup \{< s, d, t2, m2 >\}$
   1.6. $NoCreation1 \wedge < sent, transit, received > = < sent', transit', received' > \Rightarrow NoCreation1'$
   PROOF: follows since $sent' = sent \wedge transit' = transit$
   1.7. Q.E.D.
   PROOF: follows from 1.1-1.6 and INV1.

2. $Init \wedge \Box[Transition]_{\langle sent, transit, received\rangle} \Rightarrow NoCreation2$
   PROOF:
   2.1. $Init \Rightarrow NoCreation2$
   PROOF: follows since $transit = \{\} \wedge received = \{\}$
   2.2. $NoCreation2 \wedge Send(dg) \Rightarrow NoCreation2'$
   PROOF: follows since $transit' = transit \cup \{dg\} \wedge received' = received$
   2.3. $NoCreation2 \wedge Receive(dg) \Rightarrow NoCreation2'$
   PROOF: follows since $transit' = transit \wedge dg \in transit \wedge received' = received \cup \{dg\}$
   2.4. $NoCreation2 \wedge Timeout(dg) \Rightarrow NoCreation2'$
   PROOF: follows since $transit' = transit - \{dg\} \wedge received' = received$
   2.5. $NoCreation2 \wedge Change(< s, d, t1, m1 >) \Rightarrow NoCreation2'$
   PROOF: follows since $\exists t2, m2 : transit' = transit \cup \{< s, d, t2, m2 >\} \wedge received' = received$
   2.6. $NoCreation2 \wedge < sent, transit, received > = < sent', transit', received' > \Rightarrow NoCreation2'$
   PROOF: follows since $sent' = sent \wedge received' = received$
   2.7. Q.E.D.
   PROOF: follows from 2.1-2.6

3. $(\Diamond \Box (dg_1 \in sent \wedge dg_1 \in received \wedge dg_2 \in received)) \Rightarrow$
   $\Diamond \Box (dg_2 \notin sent \Rightarrow (dg_1 \in received \wedge dg_2 \notin received))$
   PROOF:
   3.1. $\Diamond \Box (dg_1 \in received)$
   PROOF:
   follows from initial assumption
   3.2. $\Diamond \Box (dg_2 \notin sent \Rightarrow (dg_2 \notin received))$
   PROOF:
   3.2.1. $\Diamond \Box (dg_2 \in received)$
   follows from initial assumption
   3.2.2. Q.E.D.
   follows from 3.2.1 and assumption 1.
   3.3. Q.E.D.
   follows from 3.1 and 3.2

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

B Linear Logic Proofs

B.1 Proof of $\otimes L^{-1}$

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma;\ X\vdash X}\qquad\overline{\Gamma;\ X\vdash X}}{\Gamma;\ X,X\vdash X\otimes X}\otimes R\qquad \Gamma;\ X\otimes X\vdash Y}{\Gamma;\ X,X,X\otimes X\multimap X\otimes X\vdash Y}\multimap L}{\Gamma;\ X,X\vdash Y}\,Der.$$

B.2 Proof that no message appears from mid-air

The proof is by induction; here we use C as a shorthand for $(R(m)\oplus R(change(m)))$ and the multiplicative conjunction of N linear assumptions A is denoted by $\otimes_{i=1}^{N}A$.

- **Base Case** The base case is that:

$$\dfrac{\dfrac{\dfrac{\Gamma;\ D(m),L,C(m)\vdash\mathbf{0}}{\vdots}\qquad\dfrac{\Gamma;\ L\vdash\mathbf{0}}{}}{\vdots\qquad\qquad\vdots}}{\Gamma;\ S(m),L\vdash\mathbf{0}}$$

The proof is as follows:

$$\dfrac{\dfrac{\overline{\Gamma;\ S(m)\vdash S(m)}\qquad \dfrac{\dfrac{\overline{\Gamma;\ D(m)\vdash D(m)}\qquad \dfrac{\dfrac{\dfrac{\Gamma;\ D(m)\otimes L\multimap L\otimes C(m),D(m),D(m),L\vdash\mathbf{0}}{\Gamma;\ D(m),D(m),L\vdash\mathbf{0}}\,Der.}{\Gamma;\ D(m)\otimes D(m),L\vdash\mathbf{0}}\otimes L\qquad \dfrac{\Gamma;\ L\vdash\mathbf{0}}{\Gamma;\ \mathbf{1},L\vdash\mathbf{0}}\mathbf{1}L}{\Gamma;\ (D(m)\otimes D(m))\oplus\mathbf{1},L\vdash\mathbf{0}}\oplus L}{\Gamma;\ D(m)\multimap(D(m)\otimes D(m))\oplus\mathbf{1},D(m),L\vdash\mathbf{0}}\multimap L}{\Gamma;\ D(m),L\vdash\mathbf{0}}\,Der.}{\Gamma;\ S(m)\multimap D(m),S(m),L\vdash\mathbf{0}}\multimap L}{\Gamma;\ S(m),L\vdash\mathbf{0}}\,Der.$$

$$\dfrac{\overline{\Gamma;\ D(m)\otimes L\vdash D(m)\otimes L}\qquad \dfrac{\Gamma;\ L,D(m),C(m)\vdash\mathbf{0}}{\Gamma;\ L\otimes C(m),D(m)\vdash\mathbf{0}}\otimes L}{\Gamma;\ D(m)\otimes L\multimap L\otimes C(m),D(m),D(m),L\vdash\mathbf{0}}\otimes L^{-1},\multimap L$$

- **Inductive Case** The inductive case is that:

$$\dfrac{\dfrac{\dfrac{\Gamma;\ D(m),L,\otimes_{i=1}^{j}C(m)\vdash\mathbf{0}}{\vdots}\qquad\dfrac{\Gamma;\ L,\otimes_{i=1}^{j-1}C(m)\vdash\mathbf{0}}{}}{\vdots\qquad\qquad\vdots}}{\Gamma;\ D(m),L,\otimes_{i=1}^{j-1}C(m)\vdash\mathbf{0}}$$

The proof of this is presented in figure 1

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999

$$
\dfrac{
\dfrac{}{\Gamma;\ D(m) \vdash D(m)}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Gamma;\ D(m)\otimes L \vdash D(m)\otimes L}
\quad
\dfrac{\Gamma;\ L, D(m), \otimes_{i=1}^{j} C(m) \vdash \mathbf{0}}{\Gamma;\ L\otimes C(m), D(m), \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ \otimes L, \otimes L^{-1}
}{\Gamma;\ D(m)\otimes L \multimap L\otimes C(m), D(m), D(m), L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ \otimes L^{-1}, \multimap L
}{\Gamma;\ D(m), D(m), L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ Der.
}{\Gamma;\ (D(m)\otimes D(m)), L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ \otimes L
\quad
\dfrac{\Gamma;\ L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}{\Gamma;\ \mathbf{1}, L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ \mathbf{1}L
}{\Gamma;\ (D(m)\otimes D(m))\oplus\mathbf{1}, L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ \oplus L
}{
\dfrac{\Gamma;\ D(m) \multimap (D(m)\otimes D(m))\oplus\mathbf{1}, D(m), L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}{\Gamma;\ D(m), L, \otimes_{i=1}^{j-1} C(m) \vdash \mathbf{0}}\ Der.
}\ \multimap L
$$

Figure 1: Proof of inductive case

C Coq Proofs

- Any newly created datagram is a valid one

```
Lemma ValidNewDG :
  (src,dst:IPaddress)(dt:data)(tm:nat)
  (validDG (newDG src dst tm dt)).
Proof.
  Intros; Hnf; Apply howToCalc with dg:=(mkDG src dst tm O dt).
Qed.
```

- Any Received DG has been sent by its source

```
Lemma RecvMeansSend :
    (dg:datagram)
    (r:(RECV (source dg) (dest dg) (contents dg)))
    {tm:nat & (SEND (source dg) (dest dg) tm (contents dg))}.
  Proof.
    Induction r.
    Clear r dg; Intros dg ARR VAL.
    Exists (ttl dg).
    Left.
    Apply SendValid
      with dg:=(newDG (source dg) (dest dg) (ttl dg) (contents dg)).
    Apply ValidNewDG.
  Qed.
```

- The first datagram is received before the second

```
Lemma recvOrdered :
  (lt (timeOf r1) (timeOf r2)).
Proof.
  Apply lt_trans with m:=(timeOf s2).
  Apply lt_trans with m:=(plus (timeOf s1) (ttl dg1));
    [ Apply timeBound | Apply SendGap ].
  Apply recvAfter.
Qed.
```

Accepted for the 3rd. Irish Workshop in Formal Methods
Galway, Ireland, July 1-2, 1999