LANSF: A Protocol Modelling Environment and its Implementation

Paweł Gburzyński and Piotr Rudnicki

Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada T6J 2H1

December 23, 1994

email: pawel@cs.ualberta.ca, piotr@cs.ualberta.ca
phone: (403) 492 2347, (403) 492 2983
FAX: (403) 492 1071

SUMMARY

LANSF is a software package that was originally designed as a tool to investigate the behaviour of medium access control (MAC) level protocols. These protocols form an interesting class of distributed computations: timing of events is the key factor in them. The protocol definition language of LANSF is based on C, and protocols are specified (programmed) as collections of communicating, interrupt-driven processes. These specifications are executable: an event-driven emulator of MAC-level communication phenomena forms the foundation of the implementation. Some tools for debugging, testing, and validation of protocol specifications are provided. We present key features of LANSF at the syntactic level, comment informally on the semantics of these features, and highlight some implementation issues. A complete example of a LANSF application is discussed in Appendix.

KEY WORDS Communication protocols Modelling Simulation Distributed computations Executable specifications

Introduction

LANSF is a software environment for modelling discrete events in communication protocols. It was originally developed for investigating MAC-level (*M*edium *A*ccess *C*ontrol) protocols for Local Area Networks (LANs). In time, the system has evolved to permit the modelling of other physical systems (e.g. Long Haul Networks and distributed computer architectures). So far LANSF has been applied to study the correctness and performance of a number of LAN protocols (variations of Ethernet[7, 8, 9, 10, 24], ENET II[15], Token protocols[12, 13, 17], HYMAP[21, 22], and many others described in the LANSF manual[14]). It is also used as a teaching tool in a graduate course on networks. The possibility of playing with realistic network models and protocols seems to be very attractive to the students.

LANSF offers a language for specifying protocols and network configurations. This language constitutes the only user interface with the package and hides completely the underlying simulator. Basic objects built into the language include: data structures representing typical configuration elements of a distributed communication system (i.e. stations and communication channels), data structures representing information passed among stations (messages and packets), functions reflecting operations of a physical communication system (e.g. starting and terminating packet transmissions), and tools for handling processes that define the communication protocols (process creation and termination, process synchronisation).

A protocol description in LANSF closely resembles an implementation on a hypothetical hardware capable of recognising the language offered by the package. This description is directly executable in a virtual environment provided by the LANSF kernel—an event-driven, discrete-time simulator—which captures all the relevant phenomena of a realistic implementation (e.g. transmission delays, collisions of two or more transmissions in the channel, the limited accuracy of independent clocks).

The work reported in this paper was undertaken for the following reasons:

- *We needed a software environment to simulate a wide class of MAC-level protocols.* Before the creation of LANSF, these protocols were simulated by a number of dedicated programs. Writing a new program to investigate every new protocol had gradually become a nuisance. On the other hand, it was clear that all those programs had common features. The first

objective of the LANSF project was to identify these features.

- *We wanted to create a tool that would facilitate assessment of simulation results by independent researchers.* Communication networks at the Medium Access Control level are usually analysed by investigating formal simplified models. Then, an individually designed simulator is used to confirm the approximate analytical results. However, it is rather uncommon for a theoretically inclined researcher to give any details regarding their simulation model. In consequence, there is no easy way for an independent investigator to assess the validity of the experimental part.

- *In our research domain, no general-purpose simulation system offered more help than a regular programming language.* We couldn't find much use for general modelling packages potentially capable of expressing any simulation problem. Instead, we needed a collection of tools modelling accurately typical phenomena occurring in a communication system. The cost of implementing such tools in languages like SIMULA, SIMSCRIPT, or CSIM[23] seemed not much lower than the cost of programming them directly in C.

- *We wanted our system to be efficient.* Simulations are CPU-intensive computations. We postulated that the run-time performance of our modelling package should be totally under our control. This postulate was difficult or impossible to fulfill in a general purpose simulation system.

- *A simulation system for communication protocols can be turned into a protocol specification tool.* In recent years, two protocol specification systems Lotos[4] and Estelle[5] have gathered some audience. These systems are intended for specifying high level protocols and they are not expected to make these specifications directly executable. We had experimented with an implementation of Estelle and found it unsuitable for our purposes: many physical phenomena of the MAC protocol level (e.g. signal propagation in a medium, race conditions) were impossible to express in a reasonable way. The LANSF project was started with the assumption that our software would eventually evolve into an executable protocol specification system—an approach opposite to that of Estelle and Lotos.

LANSF is programmed in C and has been tested under versions 4.2 and 4.3 of BSD UNIX[1] (on VAXen and Sun workstations), and under UMIPS-BSD on a MIPS M/1000. The system source code is freely available from the authors upon request.

An overview of basic data structures

The number of tools offered by LANSF is small and they are almost orthogonal. The package is *configurable* in the sense that a separate stand-alone modelling program is created for each protocol to be investigated. A protocol is described by a collection of functions and data structures that must be provided by the user. These protocol-specific part is combined with the standard functions and data structures offered by LANSF. Syntactically, the protocol is programmed in C. The standard functions and data structures, together with a certain programming methodology imposed by the system, create an impression that LANSF offers a specialised language oriented towards specifying low-level communication protocols. In the subsequent sections, we explain how protocols are programmed in this language. In this section, we present a condensed overview of the LANSF environment. For a complete discussion of the package, the reader is referred to the LANSF manual[14].

Time

Time in LANSF is discrete which means that there is an *indivisible time unit* (*ITU*), and two events occurring during the same *ITU* are randomly ordered with respect to their chronological succession. The indivisible time unit may correspond to an arbitrary time interval in the modelled physical system. Time intervals are represented as objects of type `TIME`. This type and a number of standard functions implement multiple precision arithmetic on unsigned integers with practically unbounded range. Thus, the *ITU* can correspond to a very small unit of real time[2].

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] According to quantum mechanics, real time is also discrete. Namely, the notion of chronological succession breaks down when two or more events are separated by less than the so-called Planck time (about 10^{-43}s).

The topology: stations, links, and ports

By a distributed system (in particular a communication network) we understand a number of *stations* interconnected via *communication channels*. The stations communicate by sending and receiving information along the channels and their behaviour is defined by the protocol program. A channel is modelled in LANSF by an abstract data type called *link*. Models of bidirectional and unidirectional broadcast channels are provided. Models of complex channels can be built of these standard models.

A station is attached to a link through a *port*. One port connects one station to one link. A station can be connected to a number of links and it may have more than one port to a single link. For each pair of ports connecting some station(s) to a link, the *distance* between the ports is expressed as the time of propagating a signal from one port to the other. Thus the indivisible time unit (*ITU*) is also a unit of distance. The distance between two ports connected to two different links is infinite. Distances among ports are specified in the input data, separately for each link, in the form of a *distance matrix*.

Information sent along links is organised into sequences of *bits* called *packets*. Stations insert packets into links via ports. One attribute of a port is its *transfer rate* which says how fast information can be "pumped" into the port. The port transfer rate is expressed as the number of *ITU*s required to insert a single bit into the port.

Stations are assigned identifiers—integer values from 0 to `n_stations`-1, inclusively, where `n_stations` is a global variable of type `int`. Stations are represented as structural objects of type `STATION`. A station object consists of a number of attributes. Some of these attributes are protocol-specific and they are defined by the user. Other attributes are standard; among them are: the station identifier, the array of ports connecting the station to some links, the message queues holding messages awaiting transmission, and a collection of packet buffers. Depending on the protocol and the role of a station in the network, the configuration of some standard attributes may vary from station to station. For example, different stations may have different numbers of ports and/or packet buffers.

Ports are owned by stations. Within a given station, its ports are identified by consecutive integer numbers starting from 0. Thus, ports belonging to one station form an array whose elements are structures of type `PORT`. Among the attributes of such a structure are the already mentioned transmission rate and the *distance vector* specifying propagation distances (in *ITU*s) from the port to all ports connected to the same link. The port distance vector is the port's row of the link distance matrix.

The traffic: messages and packets

The purpose of a communication network is to transmit messages among stations. Messages are supplied by external "users" of the network which are modelled by the traffic generator—the so-called *client*. Each station has access to a number of queues storing messages to be transmitted. The client is responsible for filling these queues with messages, according to traffic pattern described in the input data. The message queues are the only perception of the client from the viewpoint of a station. The client is programmable, i.e. its standard functions (built-in traffic patterns) can be extended by a user program.

A message (an object of type `MESSAGE`) generated and queued by the client at some station represents a sequence of bits to be transmitted to another station. The protocol processes the messages by removing them from the queues, splitting them into packets, and transmitting them over the network to their destinations. Among the attributes of a message are: the identifiers of the sender and receiver, the length (expressed in bits), and the time when the message was queued. The last attribute is used for calculating message delay measures.

A packet represents a message fragment to be directly transmitted on one of the station's ports. Some of the attributes of a packet (e.g. its recipient) are inherited from the message the packet has been acquired from. There are two length attributes associated with a packet: the length of the information part (representing the proper contents of the packet) and the total length of the packet, including the frame (i.e. the header and trailer). A protocol may impose restrictions on the maximum and/or minimum length of the packet's information part. If a packet is acquired from a message (this is conveniently done by standard functions), it is guaranteed that its length neither exceeds the maximum required by the protocol nor is it shorter than the demanded minimum.

Besides the queuing time attribute copied from the message, the packet is equipped with one additional attribute of type `TIME`. This second attributed tells the time when the packet became ready for transmission and is used for calculating packet delay measures.

Performance measures

The global performance of a network is usually expressed as the correlation of *throughput* and *delay*. The performance statistics collected by LANSF during simulation include three different measures of delay; they are: the packet delay (excluding the message queuing time), the absolute message delay which includes the message queuing time, and the weighted message delay which includes the message queuing time and reflects the fact that long messages may be split into multiple packets reaching their destination at different times. Each measure is presented in the form of a random variable with a number of parameters including the minimum, maximum, mean, and the standard deviation. The throughput is measured separately for each link and globally for the entire network.

LANSF offers tools for collecting non-standard statistics. An abstract data type `STATISTICS` implements typical operations on random variables, e.g. adding a new sample, combining two random variables into one, and printing out the parameters of a random variable.

User interface

The LANSF modelling program configured for a specific protocol is parametrised by the input data set. This input data set defines: time units, the network topology (the configuration of stations, ports, and links), the collection of traffic patterns, protocol-specific data (e.g. the minimum and maximum packet length), and exit conditions (e.g. the maximum number of messages to be received, CPU time limit).

The output file produced when the program terminates contains a standard part and a user-defined part. The standard parts lists the standard performance measures. Typically, the output file is processed by tools like `sed`, `grep`, or `awk` to extract the interesting items.

LANSF is equipped with a dynamic display facility that allows the user to peek at the behaviour of the simulated network "on-line". The display facility works on regular non-graphic terminals with the cursor addressing capability and offers a collection of windows that can be selected and configured dynamically when the program is running.

Programming Protocols

The behaviour of stations is defined by a set of LANSF processes (which should not be confused with UNIX processes). In the simplest case, the protocol specification may be contained in the code of one process and each station runs a private copy of this process. The essential part of a process code defines a finite state machine and has the following structure:

```
process_code () {
    switch (the_action) {
        case INITIAL_STATE:

            . . . .
            . . . .

        case STATE_1:

            /* start and/or terminate activities */
            /* specify waking conditions         */
            /* go to sleep                       */

        case STATE_2:

            . . . .
            . . . .

        case STATE_n:

            . . . .
            . . . .
    }
}
```

Each case label corresponds to a process state. Whenever a process is to be run, its code function is called and the value of `the_action` identifies its current state. Then the process typically performs certain protocol-related operations (e.g. starts or terminates a packet transmission), specifies conditions that will restart it in the future, and puts itself to sleep. Thus, the process operation is event-driven: the process code can be viewed as a collection of interrupt service routines.

The behaviour of a station may be described by more than one process. Processes may communicate directly (not necessarily via links) by sharing data and by exchanging *signals*.

The protocol module

The user-programmed protocol is put into two C files: `protocol.c` and `protocol.h`. Standard global variables visible by a protocol module are declared as `extern` in a system file `user.h` which is usually `#include`d by `protocol.c`. Non-standard global variables and data structures used by the protocol processes are usually declared at the beginning of `protocol.c`.

The `protocol.c` file must contain a definition of two functions `in_protocol` and `out_protocol`. The former is called by the system when the simulator is initialised, its purpose is to read protocol-specific data and start the initial configuration of processes. The latter is invoked when the simulator reaches one of the termination conditions, it can be used to write non-standard results to the output file. The essential part of `protocol.c` consists of the functions that contain the code of protocol processes.

The `protocol.h` file contains declarations of non-standard station attributes (variables) and, possibly, symbolic constants and macros. The contents of `protocol.h` are `#include`d into the declaration of structure `STATION`; therefore, all variables defined in `protocol.h` are in fact declared as attributes of `STATION`. The file is visible to `protocol.c`—it is `#include`d as part of `user.h`—and any symbolic constants and macros put into `protocol.h` are available to the protocol code.

Protocol processes

Logically, a protocol process consists of its code and data. Some stations may obey the same protocol and share the same process functions; other stations may execute strictly private processes. In any case, two different processes can be told apart by their *environment*—the context in which they operate.

The environment of a process is formed by a number of global variables. When a process is awakened, the environment variables provide a bridge connecting the process with the rest of the modelled system. In particular, the variable `current_time` contains the virtual time of the simulated network (in *ITU*s). Another environment variable, `the_station`, points to the data structure representing the station owning the process. The process state is identified by the variable `the_action` which is typically used as the selector of the single switch statement constituting the process code.

Some additional environment variables are used to pass context-dependent information to the restarted process. They are only set up for some waking events. For example, if a process is awakened due to a packet heard at the station, the environment variable `the_packet` points to the object describing the packet (type `PACKET`) and `the_port` contains the port number at which the packet is heard.

A process is created by the following function:

```
new_process (code, version)
int  (*code)(), version;
```

which takes the pointer to the C function representing the process code as the first argument. The second argument is the so-called process *version number* which is used to assign distinct identifiers to processes that use the same code and run at the same station. No two processes with the same `code` and `version` can exist at one station simultaneously. The value of `version` is presented to the process in the environment variable `the_process_version` whenever the process function (pointed to by `code`) is called. The created process is bound to the station identified by the value of the environment variable `the_station` at the time of the process creation.

Immediately after a new process is created, a special event for this process is triggered with the value of `the_action` equal to `INITIALIZE` and the process code function is called. From now on, the process controls its own future. By executing `return`, the process puts itself to sleep. Normally, before doing so, the process specifies waking conditions—the list of events awaited by the process. If the process suspends itself without specifying any waking conditions, it blocks itself forever.

A process that wants to terminate itself gracefully should execute `terminate`, which removes the process description from the system. A process can also terminate another process belonging to the same station by calling `kill_process (code, version)`.

Wait requests specify events that will restart the process in the future. Multiple wait requests issued by a process before it suspends itself, are interpreted as alternative waking conditions. As soon as the **earliest** of the indicated events occurs, the process is restarted.

To issue a wait request, the process calls the following function:

```
wait_event (ai, event, act);
int   ai, act;
```

The first parameter of `wait_event` identifies the so-called *activity interpreter*—the agent expected to generate the event. The interpretation of `event` depends on the *activity interpreter*—in the manner described in the next section. The third argument, `act`, is a user-defined action number. Its value is presented to the process in the environment variable `the_action` when the `event` occurs and causes the process to be restarted.

If two or more different events specified as alternative waking conditions of the same process occur simultaneously (within the same *ITU*), one of them is selected at random and assumed to be the waking event. Thus, irrespective of how many events are awaited by a process, the process is always awakened by exactly one event. The information about all events awaited by the process is then erased and new waking conditions must be specified from scratch.

The function `continue_at (act)` provides a method of performing a structural branch to a specific state (`case` label) within the process. With this type of branching, as opposed to the regular `goto`, possible events scheduled for the same current time instant are given a chance to preempt the running process. By calling `skip_and_continue_at (act)`, instead of `continue_at (act)`, the process delays the branching by a single *ITU* (see the example in Appendix).

Activity Interpreters

Time flow

The flow of time in LANSF is modelled by a collection of *daemons* called *Activity Interpreters* or *AI*s, for short[3]. The goal of an *AI* is to emulate the behaviour of a certain part of the protocol's physical environment. There is a two way communication between the protocol and *AI*s. An *AI* accepts *activities* from processes and interprets them, i.e. transforms them into events that can be sensed and awaited by processes. In this way the *AI* predicts future events and their timing.

For example, a process starting a packet transmission on a port generates an *activity* in the corresponding link *AI*. The link *AI* determines when the packet will be heard at other ports connected to the link and schedules events to notify the processes interested in perceiving packet arrivals on those ports. As another example, consider a process willing to suspend itself for a specific period of time. Such a process initiates an activity interpreted by the timer *AI* and goes to

[3]Despite the misleading acronym, *AI*s have nothing to do with *Artificial Intelligence*.

sleep awaiting an event from the timer. When the requested period of time elapses the *AI* generates an event to wake up the process.

The major part of the semantics of *AI*s (which can be also viewed as the semantics of time in LANSF) is described in our paper[16]. Each *AI* offers a number of functions available to protocol processes that start/terminate activities or sense current events generated by the *AI*. By issuing wait requests (calling the generic function `wait_event`), processes ask the *AI* to be notified about future events.

The timer *AI*

The timer *AI* is an alarm clock that a process can set for some specific delay. By calling `wait_event (TIMER, delay, after_alarm)` a process sets up an alarm clock for `delay` time units. The call to `wait_event` initiates a timer activity which, if uninterrupted, will result in a timer event, i.e. the alarm clock going off. When it happens the process will be restarted in state `after_alarm` (this value will be passed in the environment variable `the_action`). If another event restarts the process before the alarm clock goes off, the timer event will not occur. In such case the `TIMER` activity is cancelled. By creating separate processes and using signals to communicate with them, a process can simulate an arbitrary number of physical timers running concurrently.

The link *AI*

The heart of LANSF is the set of data structures and functions modelling links, i.e. communication channels. There is a separate link *AI* for each link defined in the modelled network.

The protocol processes interact with the link *AI*s by starting and terminating (or aborting) activities on ports, by calling functions that *inquire* about the past or present status of ports, and by issuing wait requests addressed to ports. Since each port is connected to only one link, each port uniquely identifies the underlying link and thus the link *AI*.

Starting and terminating activities. There are two types of activities that can be inserted into ports: *transfers* and *jamming signals*. Jamming signals are special activities used in protocols based on collision detection. A packet transfer is started by a call to:

```
transmit_packet (port_id, packet, eot_action)
```

```
int     port_id, eot_action;
PACKET  *packet;
```

The station starts transmitting the packet (described by the contents of the structure pointed to by `packet`) on port number `port_id`. The transfer lasts as long as it is not interrupted by either `stop_transfer` or `abort_transfer` (see below). The time needed to transmit the packet is equal to the total length of the packet multiplied by the port transmission rate. When this time expires uninterrupted, the process is awakened in the state `eot_action` where it is supposed to stop the transmission explicitly.

At first sight, the philosophy that requires each packet transmission to be performed by two explicit operations, i.e. starting the transfer and then terminating it after the pertinent time delay, may seem primitive. One might prefer a single operation for transmitting a packet that would start the transfer and terminate it automatically after the entire packet has been sent. The rationale behind the assumed solution is that in a real MAC-level implementation of a LAN protocol, packet transmissions are actually started and terminated explicitly. Moreover, we must be able to differentiate between a transfer of a complete packet ended by `stop_transfer` and a partially sent packet whose transmission was aborted by `abort_transfer`. A packet terminated by `stop_transfer` is *complete*—its end can be recognised by the interested stations (e.g. the packet recipient). An aborted packet is *incomplete* which fact can also be perceived.

A successfully completed transmission of a packet acquired from the standard client is typically followed by a call to `release_packet (packet)` where `packet` is a pointer to the packet buffer supplied to the preceding `transmit_packet` call. The purpose of `release_packet` is to empty the contents of the packet buffer and indicate that the buffer becomes ready to accommodate the next packet.

The emission of jamming signals is controlled by functions analogous to those described above. With `transmit_jam` a process starts sending a jamming signal which is emitted until the process calls `end_jam`. Although jamming signals are essentially redundant (they can be simulated by "special" packets), their presence simplifies the programming of many protocols based on collision detection.

Link wait requests have the following general format:

```
wait_event (port_id, event, action)
```

where `port_id` identifies the port to which the request is addressed and `event` is a number representing the type of the awaited event. Typically, this number is specified as a symbolic constant. The following are the most often used event identifiers: `SILENCE`—the beginning of the nearest silence period heard on the port, `ACTIVITY`—the beginning of the nearest activity heard on the port (opposite to `SILENCE`), `EOJ`—the nearest end of a jamming signal, `BOT`—the nearest beginning of a packet transmission, `END_MY_PACKET`—the end of a complete packet addressed to the current station.

For all events generated by packets, the environment variable `the_packet` returns the pointer to the data structure representing the packet (an object of type `PACKET`). Variable `the_port` returns the port number on which the event has occurred.

Link inquiries are questions about the present or the past, in contrast to link wait requests which ask about the future status of the ports. The history of past link activities is archived for a limited time, determined by a user-definable link parameter. The value of this parameter is generally protocol-dependent; it must be large enough to assure correct protocol operation, yet it should be as small as possible, so that the simulator is not needlessly overloaded with a huge number of archived activities. The link inquiry service provided by link *AI*s frees the protocol process from the burden of remembering past history of activities and events.

A function performing a port inquiry accepts a port number as the only argument and returns the time of the last occurrence of a specific port event. A special value is returned when the information kept in the archive is insufficient to answer the inquiry. The following are the most often used inquiring functions: `last_boa_sensed` —returns the time when the last beginning of any activity (the end of the last silence period) was heard at the port, `last_eoa_sensed`—inquires about the beginning of the current silence period on the port, `last_eot_sensed`—the last end of a complete packet heard on the port.

There exist some low level inquiring functions used to examine multiple activities heard at a port at the same time. These functions are seldom used: their purpose is to provide ultimate tools for implementing tricky protocols for which simple inquiries may prove insufficient.

The propagation of activities in a link is simulated by link *AI*s. An activity inserted into a port, e.g. a packet being transmitted, propagates to other ports connected to the same link and

triggers events on these ports. For a given link, the timing of such events is determined by the link distance matrix. Let p_1 and p_2 be two ports connected to the same link l and $D(p_1, p_2)$ be the propagation distance between p_1 and p_2. If a packet transmission is started at port p_1 at time t_1, then the beginning of this packet will arrive at p_2 at time $t_2 = t_1 + D(p_1, p_2)$. Thus, if there is a process awaiting a packet arrival at p_2 (e.g. the `BOT` event), this process will be awakened at t_2, provided that no earlier event will interrupt its waiting.

Two types of communication channels are modelled by LANSF: an ether-type bidirectional medium and a unidirectional carrier. The distance matrix of an ether-type link is symmetric, i.e. for each pair of ports p_1 and p_2 connected to the link $D(p_1, p_2) = D(p_2, p_1)$. For a unidirectional link, signals propagate only in one direction, i.e. exactly one of the values $D(p_1, p_2)$, $D(p_2, p_1)$ is defined; the other distance is assumed to be infinite.

A collision is a situation when two or more transfer attempts in the same link overlap and the information carried by each of these transfers is, at least partially, destroyed. We say that a port perceives a collision if the packet being received is garbled by interference with another activity, or there is a jamming signal currently heard on the port. A packet is garbled if another activity is heard while the packet is being received. The detailed semantics of collisions is given in our paper[16].

The purpose of jamming signals is to represent special activities in the link which are always different from correctly transmitted packets. In LANSF, the explicit occurrence of a jamming signal is semantically equivalent to a collision. This semantics of jamming signals facilitates modelling of CSMA/CD protocols which use them to enforce the so-called collision consensus[24, 25].

The client *AI*

The purpose of the *client* is to supply stations with messages to be transmitted. The client behaviour is defined in the input data by specifying the distribution of messages to be generated. A number of independent traffic patterns can be defined, each such a pattern is called a *message type*. Messages types are numbered and can be referenced by the functions performing client inquiries. A process can inquire (poll) the client for a new packet. If the polling fails (e.g. the message queue is empty), the process may choose to wait for a message arrival.

Client inquiries are issued by a process ready to acquire a packet from the client. The following function serves this end:

```
int     get_packet (pkt, min, max, frm)
PACKET  *pkt;
int     min, max, frm;
```

The first argument of `get_packet` is a pointer to the packet buffer where the acquired packet is to be stored. The remaining three parameters denote the minimum packet length, the maximum packet length, and the length of the frame information (header and trailer), respectively. All these lengths are specified in bits. If the message is shorter than `min`, the packet is stuffed (*inflated*) with dummy bits—to make its proper part (i.e. excluding the frame information) exactly `min` bits long. The dummy bits are ignored when the network's performance measures are calculated. If the value returned by the function is `NO`, it means that no packet has been acquired. Otherwise, the function returns `YES` and the first message from the message queue (the one with the earliest arrival time) is used to create the packet. The environment variable `the_message_type` returns the message type identifier (number) representing the traffic pattern used to generate the message form which the packet has been acquired. There exist other functions for acquiring packets; in particular, it is possible to ask for a packet belonging to a specific message type or extract the packet from an explicitly indicated message, not necessarily the one with the earliest arrival time.

Client events can be awaited by a process when an inquiry for a new packet fails. For a client wait request, the first parameter of `wait_event` must be `CLIENT`. There are three types of client events that a process may await. `MESSAGE_ARRIVAL` occurs whenever a message of any type is queued at the station. Another event type is a non-negative integer number identifying a message type. A process waiting for such an event will be awakened only when a message of the specified type arrives at the station. The third event type is `MESSAGE_INTERCEPT`. This event always occurs together with `MESSAGE_ARRIVAL`, but has a higher priority and is serviced first. The purpose of `MESSAGE_INTERCEPT` is to allow a station process to intercept all messages arriving at the station (e.g. to preprocess them) before they are split into packets. At most one process at a station can await this event at a given moment.

Non-standard clients or non-standard extensions of the standard client can be programmed as LANSF processes that generate messages and queue them at stations. The package offers numerous high-level functions that are useful for this purpose as well as a collection of random number generators for implementing various probabilistic distributions.

The signal *AI*

The signal *AI* provides a tool for synchronising processes. A process can send an identifiable signal addressed to a specific station. In most cases, signals are passed between processes of the same station. There are two types of signals, namely *regular signals* and *priority signals*. A process starts awaiting a signal by issuing `wait_event` with `SIGNAL` as the first argument and the number of the awaited signal as the second argument. The following function is used to send a *regular* signal:

```
int   generate_signal (n, s, i1, i2)
int   n, s;
char  *i1, *i2;
```

This function sends a regular signal number `n` to station number `s`. Parameters `i1` and `i2` specify the values to be placed in environment variables `the_signal` and `the_sender`, respectively, when a process awaiting this signal is restarted. If a process belonging to station `s` is already waiting for the signal number `n`, the function returns `ACCEPTED` and the waiting process is restarted at the current *ITU*. Otherwise, the signal is queued at station `s` and the first process that issues a wait request for signal `n` will be restarted immediately. Only one signal with a given number can be pending at a station. When the signal is queued, `generate_signal` returns `PENDING`. Subsequent calls to `generate_signal` with the same signal number addressed to the same station and issued before the pending signal is accepted, return `REJECTED` and are ineffective.

One signal can restart at most one process. When the process is restarted, the signal is cleared. A signal `n`, pending at the station, can be removed (and thus ignored) by calling `clear_signal (n)`.

Due to the fact that most inter-process communication scenarios take place in the context of the same station, there is an abbreviation (a macro) for `generate_signal` which is used much more often than the original function. Thus `internal_signal (n)` sends a signal number `n` to the current station.

A priority signal is analogous to a regular signal, but has a higher priority. Priority signals are not queued and are serviced before other events scheduled for the same *ITU*.

The queue *AI*

The queue *AI* can be viewed as a generalisation of the signal *AI*. It provides an implementation of a common ADT *queue*. Each station in the network is equipped with a (practically unlimited) number of queues identified by integer values. Initially, all these queues are empty. A process may append an item (a pointer to an arbitrary object) to any queue of any station and can acquire an item from any queue at the station at which the process is running.

A process may wish to suspend itself and be awakened as soon as an item is added to a queue. Such a wait request is issued by calling `wait_event` with `QUEUE` as the first argument and the queue number as the second argument. The awaited event is triggered whenever an item is appended to the indicated queue. If the second argument of `wait_event` is `ANY` the event occurs when an item is put into any queue owned by the station. When the process is awakened by a `QUEUE` event, the environment variable `the_item` contains the newly-queued item and `the_queue` tells the number of the queue at which the item has been appended.

The following function is used to put an item into a queue:

```
int  queue_item (s, item, qn)
int  s, qn;
void *item;
```

where `s` is the identifier of the station owning the queue, `qn` is the queue number, and `item` is the pointer to be put into the queue. If some process at station `s` is waiting for an event on the `qn`-th queue, the function returns `ACCEPTED` and the waiting process is immediately restarted. Otherwise, the function returns `PENDING`.

The functions `get_item (qn)` and `get_any_item ()` are used to acquire an item and remove it from its queue. The first function examines the queue number `qn`; the second examines all queues at the station and selects the earliest queued item from them. `NULL` is returned when no item can be acquired.

Tools for protocol testing and validation

Rudimentary debugging tools in LANSF include tracing of events and dumping snapshot-states of links. The built-in tracing facility lists standard information about each station event to

be processed. This information is printed *before* the process being restarted is given control. The tracing can be switched on and off for specified time intervals. One can also print out snapshot states of links. By the state of a link we mean here the configuration of all activities present in the link at a given moment. The dynamic display feature (not discussed in this paper) facilitates on-line debugging by monitoring the protocol through a collection of dynamic windows.

Local testing is concerned with protocol properties that are expressed locally, within a single process. The known technique of assertions can be employed to terminate the program, if an abnormal condition arises. The function `assert (boolean_expression, string)` can be used for this purpose. If `boolean_expression` evaluates to false, the execution is aborted, `string` is printed and followed by standard information about the state of the aborted protocol.

Global assertions: observers. Certain properties of protocols are not locally expressible and simple assertions are insufficient to formulate them. For example, imagine that we want to test the validity of the following statement: "After a certain station transmits a packet at time t, the packet is received at another station not later than at $t + \Delta$". Such a statement describes a compound dynamic property which involves operations of multiple processes. Of course, it would be possible to modify the code of the protocol processes (and perhaps augment packets with additional information), so that the above property would be implicitly tested. However, one has to admit that this is not the right way of solving the problem: we end up with an unwarranted protocol extension, whose only purpose is to test a certain property of the original system.

Observers are tools for expressing global assertions that involve combined behaviour of more than one process. An observer has certain properties of a process; however, this special process does not interfere with the protocol execution. Observers never respond directly to events perceived by regular processes nor they generate events that could be perceived by regular processes. Instead, they react to *meta-events*. By a meta-event we understand the operation of awakening a regular process. An observer may specify that it is to be awakened whenever a specific regular process is restarted at a specific state. Thus, the observer is able to monitor the protocol behaviour viewed as a collection of finite-state machines. A similar approach to implementing self-checking distributed programs was proposed earlier[1] and recently refined[18]. The difference between these solutions and our approach consists in two aspects. Firstly, the "observed" processes are completely unaware of

the fact that their behaviour is monitored. Secondly, our observers have unlimited access to the data structures of the protocol processes: they can freely peek at them, or even modify them, e.g. to force an extreme or abnormal condition and see how the protocol copes with it.

Observers can be seen as a feature facilitating a non-executable specification of protocols. Indeed, an observer may specify the order in which certain events should occur, but it does not force them to occur in that order. However, if during the protocol execution some events occur in a way not foreseen by the observer, the event sequencing violation is detected and reported. It is an interesting and quite educational experiment[2] to create two different protocol specifications, preferably prepared by two different people. One specification (the executable one) is a collection of LANSF processes; the other comes as a number of observers that can be viewed as dynamic "formulas" describing the chains of legitimate state transitions.

An observer is programmed similarly to a process and is created similarly to a process by calling `new_observer (code)`, where `code` is the pointer to the C function describing the observer's behaviour. The structure of the observer code is analogous to the structure of a regular process code. An observer is not associated with any specific station: it runs on its own, paralleling the simulated system.

Observers are not permitted to call `wait_event`, to start activities, or to send signals. On the other hand, there are standard functions specific to observers which cannot be called from a regular process. The following function allows an observer to await a moment in the emulated time when a regular protocol process is restarted (this function is analogous to a wait request):

```
inspect (s, p, v, a, ma)
STATION *s;
int (*p)(), v, a, ma;
```

where

- `s` is a pointer to the station to which the process belongs,
- `p` is a pointer to the process code,
- `v` is the process version number,
- `a` is the action of process `p`,
- `ma` is the observer's entry point at which the observer wants to be restarted.

For example, let us assume that an observer has issued exactly one inspect request and executed `return`. As in the case of a regular process, by executing `return` the observer puts itself to sleep. The inspect request means that the observer wants to remain suspended until the simulator awakes a regular process in an environment matching the parameters of the `inspect`. Then, immediately after the regular process completes its action, the observer will be awakened and the global variable `the_observer_action` will contain the value passed to `inspect` through `ma`.

Any of the first four arguments of `inspect` can be specified as `ANY` which means that the actual value of the corresponding environment variable of the awakened process is irrelevant. For example, with the request:

```
inspect (ANY, transmitter, ANY, ANY, WAKE_UP);
```

the observer will be restarted after any process with the code `transmitter` is awakened. The observers will be restarted in state `WAKE_UP`. A restarted observer has access to the variables of the awakened process and to the attributes of the current station. These variables and attributes reflect the state of the process' environment after the process **has completed** the execution of its action.

When an observer is created (by `new_observer`), it is initially started with the value of `the_observer_action` set to `INITIALIZE`—similarly to a regular process. An observer can issue a number of inspect requests before it puts itself to sleep (by executing `return`). However, in contrast to a sequence of wait requests for a regular process, the order in which the multiple inspect requests are issued is significant. Whenever a regular process completes its current action and executes `return`, LANSF attempts to match the attributes of the process' environment with the arguments of the pending inspect requests. This is done in the order in which the inspect requests were issued.

As in the case of wait requests issued by regular processes, an observer has its inspect list cleared when it is restarted; therefore, new waiting conditions must be specified from scratch. This applies also to the function `timeout (delay, obs_action)` which implements alarm clocks for observers.

Applications

The first applications of LANSF were in re-examining the preformance characteristics of Ethernet-type[24] LAN protocols. These experiments confirmed the numerous published results on Ethernet behaviour (which is still not completely understood). They also pointed out that some results obtained from theoretical analysis of simplified slotted models[26] should not be taken at face value. A number of variations of the Ethernet protocol have been proposed[7, 8] and LANSF proved invaluable in studying their properties.

Some attempts to repeat simulation results obtained by other researchers failed. The case of ENET II[15] proves that even innocent-looking simplifications in a simulation model can result in very distorted results. The empirical study carried out with LANSF agreed with the later published theoretical analysis of the protocol. A similar lack of success was experienced for HYMAP[21] in which case we were also unable to repeat the simulation experiment quoted in the paper. Recently, the proponents of HYMAP[20] admitted that their original experiment had been inaccurate. What is the advantage of using LANSF for simulation experiments with LAN protocols? In our opinion, the readability of LANSF code which makes other people willing to read it is the most important factor. Moreover, once a protocol has been coded in LANSF it is easy to play with by modifying it and quickly trying its different versions.

The power and usefulness of observers have been proven on a number of protocols based on passing virtual tokens, e.g. protocols proposed by the present authors[12, 13, 17] and a unslotted version of a Tree Collision Resolution protocol[6, 2]. Experiments with the last protocol convinced us that every protocol implementation in LANSF should start with creating the observers (i.e. the non-executable part of the specification should come first). Only after preparing a number of observers for the TCR protocol we have fully understood all the invariants that guaranteed the correct operation of the executable part. This experience proved invaluable later, in implementing protocols for handling integrated services (data, voice, and video).

We do not keep track of how and where LANSF was and is being used. So far the package has been distributed to more than 50 universities and research laboratories. We are aware, however, of certain applications of LANSF for which it was not intended by its authors, e.g. to modelling circuit switching networks and distributed data bases.

LANSF under BSD UNIX

At this time LANSF has been installed and tested on Sun-2 and Sun-3 workstations, Sun-4, VAX-11/780, and MIPS/1000, all machines running various clones of 4.3 BSD UNIX.

The LANSF package comes as a collection of files containing the following items:

- protocol-independent declarations and functions (source files) constituting the proper part of the simulator;
- the source files of the library for multiple-precision integer arithmetic;
- the source code of the auxiliary (support) programs;
- sample protocols programmed in LANSF together with sample data files;
- the LaTeX version of the LANSF manual.

All these files are organised into a hierarchy of directories. The root directory of LANSF can be put anywhere in the user subdirectory tree.

***** Figure 1 should be placed around here.

Figure 1 depicts the standard structure of LANSF directories. Subdirectory `EXPER` is initially empty. It is used as the reference directory for running distributed simulation experiments. The source texts of the proper (protocol-independent) part of the system are in `SOURCES`. The total length of the (commented) source code is 0.5MB.

The protocol-specific parts of the simulator are kept in subdirectory `PROTOCOLS`. This subdirectory contains other subdirectories, each of them describing one protocol. Within a protocol subdirectory, LANSF expects to find three files: `protocol.h` and `protocol.c` that provide the sources of the protocol implementation, and `options.h` containing definitions of configuration parameters and options. Other protocol-related files, e.g. sample data files and results, are typically kept there, too.

Directory `SOURCES` contains all protocol-independent source files with the exception of functions for multiple-precision integer arithmetic, which are kept in directory `BIGLIB`.

Creating executable versions of the simulator. The manner in which standard source files are combined with specific versions of `protocol.c`, `protocol.h`, and `options.h` is described by the contents of file `makefile_pattern`. This file is preprocessed by a special program (`mk`) and then supplied as input to `make` to create a stand-alone executable copy of the simulator. The easiest and most natural way of creating a simulator copy is to call `mk` from the protocol directory (a subdirectory of `PROTOCOLS`). The simulator will be compiled with the protocol-specific files and linked into an executable module called `lansf`. The typical size of this module is about 180KB (on a Sun-3/80).

Conclusions and future plans

In the two years since LANSF was created, it has proven to be more flexible than ever planned. It has found applications in investigating a much wider class of physical systems than MAC-level protocols for local area networks.

The success of LANSF is due to its ability to model physical events in communication media at an arbitrarily low level. The growing run-time library provides enough tools to program a new MAC-level protocol within hours. The modularity of the design, and the open interfaces allow the users to plug LANSF into other existing software packages. The configurability of the system seems to be a serious advantage over other simulation packages.

Further development of LANSF is planned. It will be aimed at the development of: (a) a graphical output from simulation runs (a plotting package), (b) graphical tools for preparing input data, (c) graphically oriented language for specifying protocols, possibly with Statecharts[19] as a front-end, (d) a programming language built on top of LANSF, at the level of Esterel[3].

Although programmed in plain C, LANSF was developed with the "object-oriented" paradigm in mind. Currently we are working on moving the package into the C++ environment.

Acknowledgements

The authors would like to acknowledge the contribution of their friends, colleagues, and students whose comments and suggestions were essential for the successful completion of the project. In particular, we would like to mention: Wlodek Dobosiewicz, Piotr Findeisen, Ahmed Kamal, Luigi

Logrippo, Jacek Maitan, Jay Majithia, Mike McGregor, and John Wong. We would like to thank Marcel Berard for comments on the manuscript of this paper.

This work was supported in part by NSERC Grants No. OGP9183 and OGP9207.

References

1. J. M. Ayache, P. Azéma, and M. Diaz. Observer: a concept for on-line detection of control errors in concurrent systems. In *Proceedings of the 9th Symposium on Fault-Tolerant Computing*, pages 1–8, Madison, WI, June 1979.

2. M. Berard, P. Gburzyński, and P. Rudnicki. Developing MAC protocols with global observers. In *Proceedings of Computer Networks'91*, pages 261–270, June 1991.

3. G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to esterel. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 35–55. Elsevier, 1988.

4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

5. S. Budkowski and P. Dembinski. An introduction to ESTELLE a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

6. J. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, 25:505–515, 1979.

7. W. Dobosiewicz and P. Gburzyński. Ethernet with segmented carrier. In *Proceedings of IEEE Computer Networking Symposium*, pages 72–78, Washington, DC, Apr. 1988.

8. W. Dobosiewicz and P. Gburzyński. Improving fairness in CSMA/CD networks. In *Proceedings of IEEE SICON'89*, Singapore, July 1989.

9. W. Dobosiewicz, P. Gburzyński, and P. Rudnicki. An Ethernet-like CSMA/CD protocol for high speed bus LANs. In *Proceedings of IEEE INFOCOM'90*, pages 238–245, 1990.

10. W. Dobosiewicz, P. Gburzyński, and P. Rudnicki. On two collision protocols for high speed bus LANs. *Computer Networks and ISDN Systems*, 25(11):1205–1225, June 1993.

11. L. Fratta, F. Borgonovo, and F. Tobagi. The Express-net: A local area communication network integrating voice and data. In *Proceedings of International Conference on Performance of Data Communication Systems and Applications*, Paris, France, Sept. 1981.

12. P. Gburzyński and P. Rudnicki. A better-than-Token protocol with bounded packet delay time for Ethernet-type LAN's. In *Proc. of Symposium on the Simulation of Computer Networks*, pages 110–117, Colorado Springs, Co., Aug. 1987. IEEE.

13. P. Gburzyński and P. Rudnicki. Using time to synchronize a Token Ethernet. In *Proceedings of CIPS Edmonton '87*, pages 280–288. Canadian Information Processing Society, Nov. 1987.

14. P. Gburzyński and P. Rudnicki. *The LANSF Protocol Modeling Environment, version 2.0.* University of Alberta, Department of Computing Science, TR 89-19, Edmonton, 1989.

15. P. Gburzyński and P. Rudnicki. A note on the performance of ENET II. *IEEE Journal on Selected Areas in Communications*, 7:424–427, Apr. 1989.

16. P. Gburzyński and P. Rudnicki. On formal modelling of communication channels. In *Proceedings of IEEE INFOCOM'89*, pages 143–151, 1989.

17. P. Gburzyński and P. Rudnicki. A virtual token protocol for bus networks: correctness and performance. *INFOR*, 27:183–205, 1989.

18. R. Groz. Unrestricted verification of protocol properties in a simulation using an observer approach. In *Proceedings of the IFIP WG 6.1 6th Workshop on Protocol Specification, Testing, and Verification*, pages 255–266. North-Holland, June 1986.

19. D. Harel. On visiual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

20. A. Kamal. Comments on "Analysis of a Hybrid Multiple Access Protocol ..." and author's reply. *IEEE Transactions on Communications*, 38(2):142–143, Feb. 1990.

21. P. Nain, N. Georganas, and W. J. Stewart. Analysis of a hybrid multiple access protocol with free access of new arrivals during conflict resolution. *IEEE Transactions on Communications*, 36(7):806–815, July 1988.

22. M. Ríos and N. Georganas. A hybrid multiple-access protocol for data and voice-packet over local area networks. *IEEE Trans. on Computers*, 34(1):90–94, Jan. 1985.

23. H. Schwetman. Using CSIM to model complex systems. Aca-st-154-88, MCC, Austin, TX, 1988.

24. J. Shoch et al. Evolution of the Ethernet local computer network. *IEEE Computer*, 15(8):10–26, Aug. 1982.

25. W. Stallings. A tutorial on the IEEE 802 Local Network Standard. In R. Pickholtz, editor, *Local Area & Multiple Access Networks*, pages 1–30. Computer Science Press, 1986.

26. A. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

27. F. Tobagi, F. Borgonovo, and L. Fratta. Express-net: A high-performance integrated-services local area network. *IEEE Journal on Selected Areas in Communication*, 1(5):898–913, Nov. 1983.

APPENDIX: A Complete Example—Expressnet

The protocol

In *Expressnet*[11, 27], all stations are connected to a single unidirectional bus. Each station has two ports to the bus (see figure 2).

***** Figure 2 should be put around here.

The link is divided into two virtual channels: the *outbound channel* on which stations transmit their packets, and the *inbound channel* on which the packets are received.

The protocol of *Expressnet* operates in rounds. A station is allowed to transmit at most one packet per round. The packet is transmitted on the outbound port immediately after the station senses the end of an activity (the beginning of a silence period following a transmission) on the port. This event will be denoted by EAC. During some initial period of the transfer (Δ_p time units) the station transmits a preamble which contains no vital information and can be partly destroyed without affecting the packet consistency. The purpose of the preamble is to manifest the station's "presence" on the bus. If during the preamble transmission the station detects another activity in the "upstream" (left) part of the outbound channel, it immediately aborts the preamble and defers its transfer attempt until the next EAC on the outbound port. The leftmost backlogged station cannot be preempted.

Assume that one station, say S_1, detects an EAC event in the outbound channel and transmits its packet without being preempted by an upstream station. The EAC event generated by the end of this packet propagates to the right (downstream) and the first backlogged station located to the right of S_1 will append its packet immediately after the packet transmitted by S_1. This scenario repeats until all backlogged stations situated downstream with respect to the first successful transmitter insert their packets into the outbound channel. All these packets form the so-called *train* which looks like a single (almost) contiguous activity. Eventually, the train will arrive at the inbound channel and pass through it. Every station will hear all the packets of the train and possibly receive some of them. Moreover, each station is able to detect the end of the train—a period of silence following the last packet. If the period of silence is longer than the possible (tiny) gap separating two packets in the train, the station may assume that no more packets will follow.

Technically, the protocol operation consists of interpreting two event types: EAC on the outbound channel and ETR (for *end of train*) on the inbound channel. Upon detection of EAC, a backlogged station is allowed to start inserting its packet preamble into the outbound port. The preamble length (Δ_p) should be sufficient to compensate for the worst-case delay of a station in responding to the EAC event and for the time required to detect an interfering upstream activity. Thus, when the preamble has been transmitted without upstream interference, the station may

assume that it has acquired the bus access and its packet will be sent successfully.

After a successful transmission, the station waits for the *ETR* event on the inbound port. This event is assumed whenever an activity sensed on the port is followed by an appropriately long period of silence (we will denote it by Δ_t). Then the next round can be started.

To start the next round all stations detecting the *ETR* event, i.e. all operational stations in the network, send a short burst of activity (e.g. a packet preamble), even if they have no packets to transmit. The ends of these bursts, which all arrive on each outbound port at approximately the same time, provide the starting *EAC* event for the new train. The protocol must be initialised, i.e. upon network startup, the most upstream station is obliged to generate the first activity. Afterwards, the protocol operates without any designated "leader".

The `protocol.h` file

This file contains only macro definitions as there are no non-standard station attributes. We assign symbolic names for all *actions* of the protocol processes.

```
#define         WAIT_SYNC               1 /* Action constants for */
#define         CHECK_BUFFER            2 /* the transmitter */
#define         TRANSMIT_PACKET         3
#define         PREEMPTED               4
#define         START_TRANSFER          5
#define         END_PACKET              6

#define         WAIT_FOR_EAC            1 /* Action constants for the */
#define         CHECK_SYNC              2 /* inbound port synchroniser */

#define         WAIT_DT                 2 /* Action constants for the */
#define         END_TRAIN               3 /* outbound port synchroniser */
#define         NOT_END_TRAIN           4

#define         WAIT_FOR_PACKET         1 /* Action constants for */
#define         PACKET_RECEIVED         2 /* the receiver    */

#define         OP                      0 /* Outbound port identifier */
#define         IP                      1 /* Inbound port identifier */

#define         BUFFER                  packet_buffer (0)
#define         PREAMBLE                packet_buffer (1)

#define         SYNC_EAC                1 /* End of carrier signal */
#define         SYNC_ETR                2 /* End of train signal */

#define packet_pending (full (BUFFER) || \
```

```
        get_packet (BUFFER, min_packet_length, max_packet_length, \
                                                frame_info_length))
```

The last macro is used to check whether a station has a packet to transmit, and if so, to place the packet in the buffer (call to `get_packet`).

The `protocol.c` file

The LANSF implementation of *Expressnet* consists of four processes run by each station. The basic two are the transmitter and receiver. The transmitter is accompanied by two synchronising processes that monitor the two station ports (`OP` and `IP`) and detect *EAC* and *ETR* events. Below we list the contents of `protocol.c` and then briefly explain what happens there.

```
#include        "user.h"
char            *protocol_id = "Expressnet";
long    min_packet_length,      /* Minimum packet length  */
        max_packet_length,      /* Maximum packet length  */
        frame_info_length,      /* Header + trailer       */
        preamble_length;        /* The length of preamble */
TIME    delta_t;                /* ETR detection interval */
int     receiver (), transmitter (), op_synchroniser (), ip_synchroniser ();
                                /* Four processes run by each station */
in_protocol () {
  int  i;
  /* Read protocol-specific parameters from input */
  min_packet_length = read_integer ();
  max_packet_length = read_integer ();
  frame_info_length = read_integer ();
  preamble_length   = read_integer ();
  delta_t = read_time ();
  for (i = 0; i < n_stations; i++) {
    /* Create the station's processes */
    the_station = id_to_station (i);
    new_process (transmitter, NONE);
    new_process (op_synchroniser, NONE);
    new_process (ip_synchroniser, NONE);
    new_process (receiver, NONE);
  }
}

out_protocol () {
  out_string  ("Protocol specific parameters:\n\n");
  out_integer (min_packet_length, "MinPcktLength");
  out_integer (max_packet_length, "MaxPcktLength");
  out_integer (frame_info_length, "PcktHdrLength");
  out_integer (preamble_length,   "PreambleLength");
```

```
  out_time    (delta_t,           "ETRDelay");
}

transmitter             () {
  switch (the_action) {
    case INITIALIZE:
      *PREAMBLE = make_packet (the_station, NONE, preamble_length);
      if (the_station->id == 0) continue_at (TRANSMIT_PACKET);
    case WAIT_SYNC:
      wait_event (SIGNAL, SYNC_ETR, TRANSMIT_PACKET);
      wait_event (SIGNAL, SYNC_EAC, CHECK_BUFFER);
      return;
    case CHECK_BUFFER:
      if (! packet_pending) continue_at (WAIT_SYNC);
    case TRANSMIT_PACKET:
      transmit_packet (OP, PREAMBLE, START_TRANSFER);
      wait_event (OP, COLLISION, PREEMPTED);
      return;
    case PREEMPTED:
      abort_transfer (OP);
      continue_at (WAIT_SYNC);
    case START_TRANSFER:
      if (packet_pending) {
        abort_transfer (OP);
        transmit_packet (OP, BUFFER, END_PACKET);
        return;
      }
      stop_transfer (OP);
      continue_at (WAIT_SYNC);
    case END_PACKET:
      stop_transfer (OP);
      release_packet (BUFFER);
      continue_at (WAIT_SYNC);
  }
}

op_synchroniser () { /* Detects synchronising events on the outbound port */
  switch (the_action) {
    case INITIALIZE:
    case WAIT_FOR_EAC:
      wait_event (OP, EOT, CHECK_SYNC);
      return;
    case CHECK_SYNC:
      if (the_packet->sender != the_station)
        if (internal_signal (SYNC_EAC) != ACCEPTED)
          excptn ("Op_synchroniser: SYNC signal not expected");
      skip_and_continue_at (WAIT_FOR_EAC);
  }
}

ip_synchroniser () { /* Detects synchronising events on the inbound port */
```

```
  switch (the_action) {
    case INITIALIZE:
    case WAIT_FOR_EAC:
      wait_event (IP, EOT, WAIT_DT);
      return;
    case WAIT_DT:
      wait_event (TIMER, delta_t, END_TRAIN);
      wait_event (IP, ACTIVITY, NOT_END_TRAIN);
      return;
    case END_TRAIN:
      if (internal_signal (SYNC_ETR) != ACCEPTED)
        excptn ("Ip_synchroniser: SYNC signal not expected");
    case NOT_END_TRAIN:
      skip_and_continue_at (WAIT_FOR_EAC);
  }
}

receiver () {
  switch (the_action) {
    case INITIALIZE:
    case WAIT_FOR_PACKET:
      wait_event (IP, END_MY_PACKET, PACKET_RECEIVED);
      return;
    case PACKET_RECEIVED:
      assert (my_packet (the_packet), "Receiver: not my packet");
      accept_packet (the_packet, the_port);
      skip_and_continue_at (WAIT_FOR_PACKET);

  }
}
```

The file `user.h` is a standard file from directory `SOURCES` and contains a number of `extern` object declarations that protocol processes can access. These objects have been discussed throughout the paper.

Initialisation

The first function in `protocol.c` called by LANSF is `in_protocol`. It reads some protocol-dependent parameters and creates all protocol processes. Note that each collection of four processes is created in the environment of a different station (`the_station` is set before the processes are created).

Non-standard output

The non-standard output is very simple in this case. The function `out_protocol` will be called when the simulation run terminates and will print the protocol-dependent parameters. In general,

this function can can be programmed to output more data, e.g. non-standard statistics collected by the protocol.

The transmitter

The preamble is represented as a special packet. The length of this packet (`preamble_length`) corresponds to Δ_p discussed earlier. In our implementation of *Expressnet*, all activities inserted into the bus are packets. The *EAC* event is equivalent to the end of a (unaborted) packet (the `EOT` event in LANSF).

Upon initialisation of the transmitter, the preamble packet is created and stored in one of the standard packet buffers (cf. the definition in `protocol.h`). From now on, the contents of this buffer do not change. Station 0 is responsible for inserting the initial activity—to begin the first round. For now, let us assume that the transmitter runs at another station. The process sleeps until it receives one of the two synchronising signals: `SYNC_EAC` or `SYNC_ETR`. These signals are sent by the two auxiliary, synchronising processes (see below). Having received the `SYNC_EAC` signal, which represents the *EAC* event, the transmitter checks if the station has a packet to transmit. If there is a client packet awaiting transmission at the station, the process proceeds to `TRANSMIT_PACKET` where it transmits the preamble on the outbound port (`OP`) and, at the same time, monitors the port for a collision. If a collision occurs, which means that an upstream station is also inserting a preamble, the process aborts its transmission and returns to `WAIT_SYNC` where it continues awaiting one of the two synchronising events. If the preamble has been sent successfully, the transmitter wakes up at `START_TRANSFER`.

Action `TRANSMIT_PACKET` has a second purpose. This action is also used for transmitting the preamble without following it by a data packet—after the occurrence of *ETR*. In such a case, the preamble plays the role of a dummy activity starting the new round. Thus, when the transmitter gets to `START_TRANSFER`, it checks again whether there is a data packet awaiting transmission. If the answer is affirmative, the preamble is aborted (so that it will not trigger *EAC*) and it is immediately followed by the data packet. In the opposite case, the preamble is terminated (its end will now trigger the *EAC* event) and the transmitter branches to `WAIT_SYNC`. After the (necessarily successful) transmission of the data packet, the process releases the packet buffer and goes to sleep until a synchronising event occurs.

When the transmitter is awakened by the SYNC_ETR signal, which means that the *ETR* event has occurred, it goes directly to TRANSMIT_PACKET—to transmit the preamble. Then, if the preamble has been inserted with no interference from upstream, the station is free to transmit a data packet (if there is a data packet queued at the station). Note that upon initialisation, the transmitter of station 0 starts from TRANSMIT_PACKET, thus providing the first activity that "launches" the protocol.

The synchronising processes

The two signals controlling the transmitter operation come from two auxiliary processes: op_synchroniser which monitors the outbound port and detects *EAC* events, and ip_synchroniser responsible for sensing *ETR* events on the inbound port.

The op_synchroniser process intercepts all end of packet events on the outbound port. Every such event, with the exception of the end of a packet transmitted by the current station, is interpreted as an *EAC* event and results in the SYNC_EOT signal being sent to the transmitter. Note that the end of a packet inserted by the current station must be ignored, as otherwise the station would be able to transmit arbitrarily many packets in one round and potentially starve all downstream stations.

The ip_synchroniser process monitors all end of packet events on the inbound port (IP). If an end of packet is followed by undisturbed silence lasting for at least delta_t (Δ_t) time units, it is assumed that an *ETR* event has occurred and the SYNC_ETR signal is sent to the transmitter.

The receiver

This is the simplest of all four processes. Its role is to call the standard function accept_packet which updates performance measures. Most of the time the process waits for END_MY_PACKET. Upon the occurrence of this event the receiver calls accept_packet, and then continues waiting. Note the usage of skip_and_continue_at. With this statement, the process waits for 1 *ITU* before moving to WAIT_FOR_PACKET, so that it is not awakened twice by the end of the same packet.

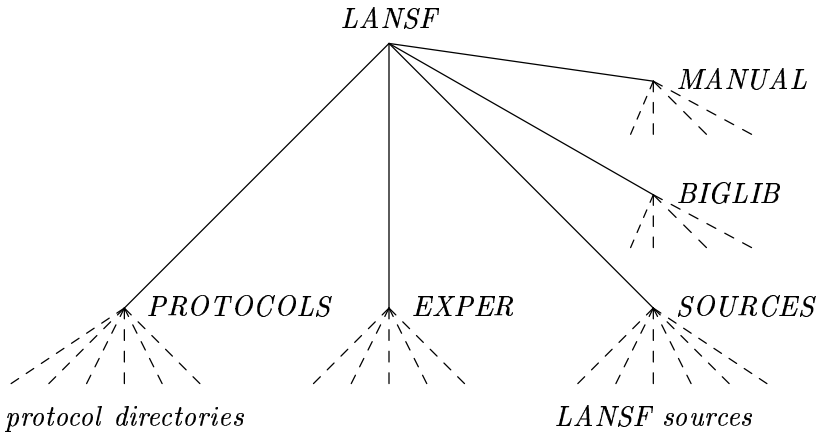

Figure 1: Structure of LANSF directories.

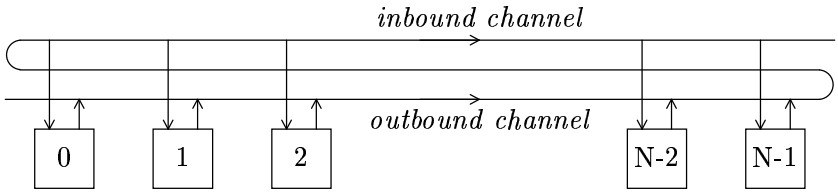

Figure 2: The topology of *Expressnet*.