Real Time Network Traffic Monitoring

Gerald Tripp

Technical Report: 5–99
Computing Laboratory, University of Kent

21 May, 1999

Abstract

This paper looks at the problems of real time network traffic monitoring. Some of the existing approaches are reviewed, looking at both simple filtering systems and also systems based on the use of finite state machines that can report specific events or capture data only when in particular states. Finally, some existing implementation techniques are examined and an outline proposal made for the design of a network monitoring system that uses finite state machines implemented using associative processing.

1. Introduction

A problem with many network-monitoring systems is that they generate data at a faster rate than it can be processed. This becomes more so with higher speed networks, particularly with networks operating at data rates of 100 Mbps and above. It is a particular problem if we wish to save the contents of packets, rather than just collect statistical information about data rates etc. One approach is to store data until either a certain period of time has elapsed or until we fill the available storage space. We then stop the data capture until the data has been processed. Another approach is that we start processing data as soon as it is available – thus freeing up storage. This can give significant increases in the capture period, but only if the processing rate approaches the data capture rate. Ultimately however, if the average data processing rate is less than the average capture rate, then we fill our storage and need to pause data capture. What is needed of course is a traffic monitoring system that can process data at the network rate. This can however be difficult with high-speed networks.

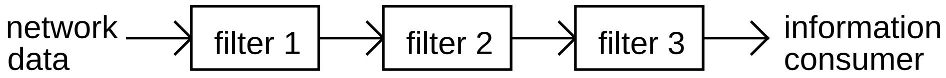
The next section looks at various techniques that processing systems can use to reduce the final amount of data captured, including state based monitoring systems. Section 3 looks at the use of associative processing as a method of performing a fast search for patterns that might match incoming data. Section 4 then gives an outline proposal for the design of a network monitoring system that uses finite state machines implemented using associative processing. The last section gives some conclusions and ideas for further work in this area.

2. Data Reduction Techniques

The difficulty for our monitoring system is that it may be dealing with a very large amount of data that – as far as the monitoring system is concerned – has a low information content. We may for example be monitoring a high speed backbone network but only be interested in traffic belonging to particular streams. In general, what is required is to use some method of filtering that removes network traffic that is not of interest.

2.1 Filtering systems

By using filtering systems, we aim to generate a new traffic stream which has a lower data rate, but effectively a higher information content – i.e. in the view of the next stage in the monitoring process the traffic now has a higher entropy. We may be able to implement our network traffic processing as a pipeline of processing – where each filtering stage gives this decrease in traffic and hence a corresponding increase in entropy.

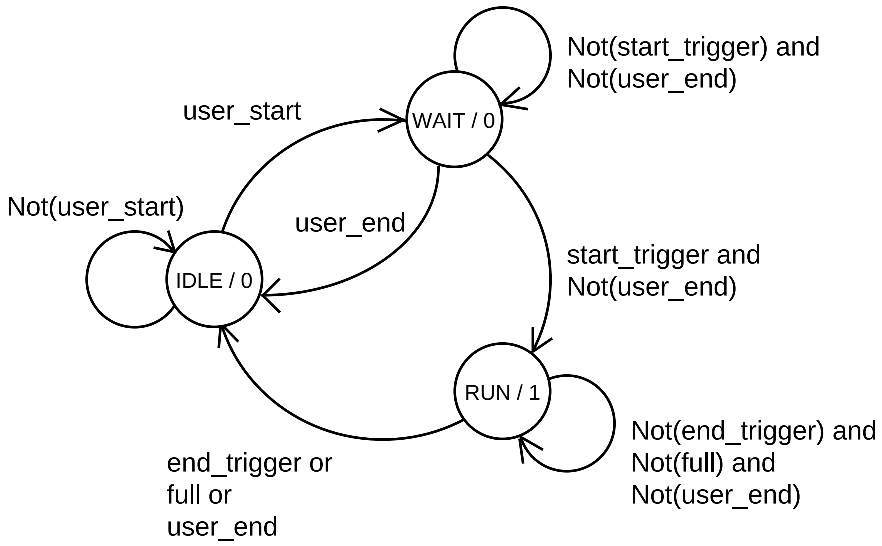


Initial stages in this pipeline may need to be implemented in hardware to be able to operate at the data rates involved – but only perform simple operations. The final stages in the pipeline may be implemented in software and perform quite complex operations – but only operate at a lower data rate. Intermediate stages could be based on embedded processing systems or programmable logic as appropriate.

Some filtering systems can be implemented quite easily: many network interface cards may allow filtering on the basis of network destination address, particularly those designed for shared media such as Ethernet. A number of traffic monitoring systems will provide filtering on the basis of network addressing, possibly using standard hardware or using custom network interfaces. This can be extended to allow patterns to be used to match against any part of the packet. As an example: Tripp [1] describes a system in which each bit in a small data packet is matched against 0, 1 or don't care. Only packets matching these patterns will then be captured for later analysis. This type of pattern matching can of course be extended in many ways – such as allowing the use of multiple patterns. We might therefore capture only those packets that are travelling between particular source and destination addresses and which also are for example "DATA" packets. The general function however is to use these filtering systems to determine if a packet – or part of a packet – should be saved for later processing.

2.2 State based systems

We can extend our filtering system by using the idea of state. As an example, logic analyzers – for debugging hardware – normally allow data capture to be started or stopped on detecting particular data patterns. Thus the monitoring system moves between different states and operates differently in each state. As an example, a finite state machine is shown below which might describe the behavior or a simple logic analyzer.



Here, a number of events may cause the logic analyzer to change state, such as actions by the user and also if a start or end trigger pattern is seen. A single output bit is generated that controls data capture, hence data will only be captured in state RUN. We can of course extend this type of system for our traffic monitoring to have many user-defined states.

Network protocols are often defined in terms of finite state machines (FSMs), so it is convenient to use a FSM in network monitoring. Rather than pass on all network data for an end system to decode, we can use one or more FSM to monitor the network data. The output from the FSMs will need to have some type to indicate what is being signaled and may also contain some attached data – such as data from one or more received packets.

2.3 Existing work

In his 'Information Collection Architecture' [2], Hershey uses finite state machines to recognize regular expressions that have been derived from strings of channel symbols. These FSMs generate event outputs, which indicate the occurrence of particular patterns on the input symbol stream. These event outputs then act as inputs to successor FSMs that control counters used to record the number of occurrences of a particular pattern in a measurement time interval (MTI). This system is soft, in that the state machine definitions can be loaded at run time. It also gives high data reduction, as it is used primarily to gather statistical information – which for any significant MTI will be a small fraction of the total data received from the network.

Richards [3] describes the use of state machines for monitoring broadband networks. His work looks at network data at a high level, taking the input to a state machine being an event, which contains fields for: length, type, timestamp, sequence number and a limited amount of attached data. At the lowest level this can carry a packet arriving from a network such as an ATM [4] cell. The 'Statechart' representation [5] is used for his state machines, which allows the use of hierarchy in the state structure. As the network probe he uses only allows standard FSMs, Richards' implementation allows flat state machines to be created from the statecharts. One of the examples he gives is measuring the duration of PDUs sent over an ATM network. This is implemented as two FSMs: one to run in a network probe and one to run in a monitoring PC. The FSM in the network probe generate events on start and end (and combined start & end for 1 cell PDUs) of AAL-5 frames – each of these events carries a timestamp giving the arrival time. The FSM in the PC receives the frame-start and frame-end events and generates its own frame-duration events.

Implementation techniques

The system described by Hershey [2] operates at a network symbol level and implements the FSMs using a table look up system. In this, the current state and the input symbol are used as indices to select the new state and any outputs. This is implemented in hardware – combining the indices to form an address to select the required data from RAM. This enables large finite state machines to be built that will operate at a rate primarily dependent on the speed of the RAM. Hershey also describes how it is possible to build state machines that operate on an input of more than one (consecutive) symbol per clock tick. He shows how using these techniques it is possible to build automata to monitor 1 Gbps networks.

The system referred to by Richards [3] operates at a high level with the bottom level input events consisting of cells arriving from the network. For the FSMs implemented on the network probe, a low-level format is generated consisting of a table of code words. This is interpreted using a software implementation written in the programming language occam [6] and is described by Linington in [7]. Each line in the table consists of four 32-bit words. The first word specifies the location in the cell to be tested, the second and third words give a match pattern and the final word gives an address to branch to if the match test fails. If the match succeeds, then execution proceeds to the next line. In addition, negative values for the first word indicate that one of a number of actions has to be performed, such as: generating output events or moving on to the next cell. As this system operates in software on a stored packet, it is able to have random access to the cell.

2.4 Implementation Problems

When implementing any finite state machine system, the input data size will have a great influence on the implementation and of course its performance. The more input data bits we can process in a single clock cycle, the faster we can process data from the network. However if we have a large number of input bits we may make the implementation of the FSM very complex. On the other hand, a small number of input bits may mean that our FSM may need a number of intermediate states as parts of a large input word are tested against a given pattern. In either case, this complexity might be hidden from the 'programmer'.

For the table lookup system described by Hershey [2], the amount of memory is very sensitive to the size of the input data. The amount of memory used for generating the next state being $s.2^{i+s}$ bits – where i is the number of bits in the input symbol and s is the number of bits used for the state variable (given that the maximum number of states is 2^s). This system does however have the advantage that we have a lookup table giving the next state for all possible combinations of the current state and the input bits. It also has the benefit that in using RAM for the tables, the FSMs implemented can be changed dynamically.

The code table system used by Linington [7] can operate on large numbers of input bits from the network. It is currently implemented in software, however it would be straightforward to construct a hardware version. However it has the disadvantage that a complex FSM could compile into a system that makes multiple tests on individual words of data. This means that it is not possible to place an upper bound on the processing time for a packet – which is undesirable in a real time system.

The standard method of implementing FSMs in hardware is to compile these into a network of logic gates, with the current state held in a number of flip-flops. This is the standard method used when building FSMs in field programmable logic arrays (FPGAs). Generally however, building logic for FPGAs consists of using tools to allocate logic cells and decide how signals are routed between the cells – this can be time consuming and in some cases may require user interaction. It is possible to vary the function of logic at run time – for example changing the contents of small internal look up tables – but we still have the same cell inter-connectivity unless we rebuild the design.

3. Associative processing

What we need for our FSMs is to be able to implement a logic function that has as its inputs the current state and a reasonably large word of data from the network. In terms of the specification, in each state we may have a number of different patterns that we wish to compare in turn against the network data and a default case when no match succeeds. Fortunately, a solution to this problem already exists: this is content addressable memory – which is described below.

3.1 Content addressable memory

The associative memory that is commonly used in production hardware is sometimes referred to as binary content addressable memory. With random access memory we obtain items from the memory by using an index as the address of the data to be read. With content addressable memory, things are the other way around; we provide a piece of data and the memory searches for its location. This typically operates by the user providing a 'key' to use for the search and details of which bits of the memory word should be used in the search. The memory chip compares the key with all locations in memory and gives a match/fail result. For a match, the memory chip will give typically give either the word's address or direct access to the word in memory. If multiple locations give a match, then the memory chip will often select the one with the highest `priority' – such as the one with the lowest (or highest) memory address.

Another form of content addressable memory is Ternary content addressable memory – which is sometimes referred to as functional memory [8]. This type of memory contains a mask word for every word of data that is stored in the memory. This allows us to store match patterns within the content addressable memory – with each bit having values of 0, 1 or don't care. When the memory chip is searching for a piece of information, it will search for a pattern that gives a match for the data key provided.

3.2 Existing work

The general area of using forms of content addressable memory to build computing systems is referred to as associative processing. Grosspietsch gives a survey of work in this area in [8], which is also a useful introduction. In general, associative processing systems consist of some type of processing logic that is closely coupled with content addressable memory. A lot of research work has involved the design of associative parallel processing systems – some of these have processing modules attached to small blocks of associative memory, such as a word. I am specifically referencing below some papers that describe the implementation of mono-processor systems as it appears at this point that these would be the most relevant when considering the application proposed in this paper.

A research group at J.W.Goethe-Universität Frankfurt has developed a processor called AM^3 (Associative Micro-programmable Multipurpose Monoprocessor) [9], [10]. This is based on the AMD 2900 series bit-slice components [11] and has closely coupled associative memory. A basic instruction set provides a standard von Neumann style of operation using the 2900 series components, plus an associative instruction set provides access to the operations performed using the associative memory.

A network specific processor is described in [12], which operates on an ATM cell stream. This is a single chip that performs a number of operations on a pipeline of ATM cells. A processor core is included that has a closely coupled area of content addressable memory. This enables searching and table look up operations to be defined by the application program that is stored in a separate on-chip instruction memory

One of the major uses of content addressable memory today is probably for implementing address tables in network routers and switches. This is particularly appropriate when we have a large number of bits in the address field and a sparse use of the address space. We can take the channel or destination address of a packet, search for this value in content addressable memory and use the result to determine the output queue and possibly a new value for the packet channel number. Performing this operation using a hardware content addressable memory is typically a lot faster than alternative software mechanisms, particularly when operating on large address fields.

4. Proposal

The system I am proposing here uses ternary content addressable memory as a kind of program store to implement FSMs for network monitoring. The aim here is to build a relatively simple processing system, and to that end, no conventional von Neumann style of operation is provided.

The program consists of a table of code words as shown below:

State	Data match pattern	Next state	Actions

When the system starts, an initial state is defined. This might be the same for all tests or it might be a state that refers to a particular data channel. The program executes by presenting the content addressable memory with the value of the current state and the word of data from the network to be checked. These are checked against the state and data match pattern fields respectively. If a match succeeds, then the result returned from the CAM is the next state and details of any actions to be performed. This process is repeated for each word of data contained in the packet. One of the final actions this might perform is to update the initial state either for all tests or for a particular data channel.

As the program executes, the current state will change. As the current state is used as part of the input to the content addressable memory, this will mean that different tests may be performed in each state. A large number of prioritised matches may therefore be performed in each state – limited only by the amount of memory that is available.

The system defined above requires that at least one test succeeds or no new next state will be defined. This will often require an else clause for each state which tests against don't care for all bits in the data match pattern.

4.1 Example

To show how this could operate in practice, here is an example showing how we might attempt to extract the source address from an IP datagram carried over ATM. We assume here that we have already determined that this is the first cell of an AAL5 [13], [14] frame and that this probably contains an IP datagram. We start by looking at the first word of the cell body. The IP datagram may have been carried directly in the AAL5 frame, or it may have used LLC/SNAP encapsulation for routed protocols [15]: this has a prefix of AA-AA-03, 00-00-00, 08-00. There may well be better ways to do this, but it gives a simple example.

The FSM here is described in a language that is based loosely upon VHDL. The word size used is 32-bits. The default number base is hex, with hex don't care shown as '?'. The DATA signal is updated to contain the next data word from the network on each clock tick. The current state of the FSM is defined by the state variable CS. This is updated with the value of next state (NS) on each clock tick – although to save space the code for this is not shown.

```
Architecture assoc of IPtest is

type state is: START, LLC2, IP1, IP2, IP3, IP4, IP5, ENDIP;
signal CS : state;    -- Current state
signal NS : state;    -- Next state
signal OK : boolean;
signal SA : std_logic_vector (31 downto 0);
signal DATA : std_logic_vector (31 downto 0);

Begin
    -- Note: for a VHDL specification, there also needs to be a process that
    -- copies NS into CS on the active edge of the system clock. (not shown)

    process(DATA, CS)
        case CS is
            START =>
                OK <= false;
                if DATA = "AAAA0300" then
                    NS <= LLC2;                -- looks like a LLC header
                elsif data = "4???????" then
                    NS <= IP2;                 -- may be an IPv4 frame
                else
                    NS <= ENDIP;               -- no, not IPv4
                end if;
            LLC2 =>
                if data = "00000800" then
                    NS <= IP1;                 -- yes it is an LLC/SNAP header
                else
                    NS <= ENDIP;               -- no, not IPv4
                end if;
            IP1 =>
                if data = "4???????" then
                    NS <= IP2;                 -- may be an IPv4 frame
                else
                    NS <= ENDIP;               -- no, not IPv4
                end if;
            IP2 =>
                NS <= IP3;                     -- ignore
            IP3 =>
                NS <= IP4;                     -- ignore
            IP4 =>
                SA <= DATA;                    -- save source address
                NS <= IP5;
            IP5 =>
                OK <= true;                    -- we probably have an IP frame
                NS <= ENDIP;
            ENDIP =>
                NS <= ENDIP;                   -- finished
        end case;
    end process;
end;
```

Taking the source code above, we should be able to compile this into a series of entries for the associative program store such as shown below. I have not attempted here to show how any actions are performed. It is likely that the memory word will need to be extended to contain a number of fields to allow operations to be performed on a set of global variables or variables related to a particular data channel.

State	Data match	Next state	Action
START	AAAA0300	LLC2	OK <= false
START	4???????	IP2	OK <= false
START	????????	ENDIP	OK <= false
LLC2	00000800	IP1	
LLC2	????????	ENDIP	
IP1	4???????	IP2	
IP1	????????	ENDIP	
IP2	????????	IP3	
IP3	????????	IP4	
IP4	????????	IP5	SA <= DATA
IP5	????????	ENDIP	OK <= true
ENDIP	????????	ENDIP	

The words of the program are shown with the highest priority at the top of the table.

5. Conclusions

The proposed system described above appears to be a feasible method of implementing FSMs for use in network monitoring. The use of associative memory to hold match patterns is a method in which we can performing quite complex matching of packet contents in real time. Unlike the table lookup system described by Hershey [2], this system cannot however guarantee to be able to match any possible input function, however it is likely that this should not be too much of a problem in practice. As network protocols are generally designed for ease of decoding by the receiving computer system, we might hope that in practice we may only be checking for a limited number of data word patterns or values in each state. An obvious departure from this will be data items such as network addresses. However, as we will often see sparse use of a large address field, this should not be too much of a problem – indeed associative memory is a standard method of implementing fast table look up in devices such as network switches and routers.

As this paper only constitutes a proposal for direction of research, there are evidently a large number of areas that need to be addressed. The following sub-section addresses some of these and gives ideas for further work.

5.1 Further work

Other specific areas that can be identified at this point for investigation include methods for implementing multiple FSMs in the monitoring system and also how we deal with multiple independent channels of data from the network without generating an explosion in the size of the FSMs used.

To aid investigation into this research area, the design of a reference implementation would be very useful. The process of designing such a system should highlight areas of weakness that require further work. This reference implementation will then need to be modelled and simulated to check the assumptions made in the design and also to measure the performance of the proposed implementation. Any proposed implementation will need to be based on existing or proposed hardware components so as to obtain an idea as to whether the design is practical and to gain realistic figures for performance.

To enable this system to be tested with a rich set of monitoring specifications, it is important that a set of software tools be created that will support the generation of code-tables from specifications of the FSMs.

Lastly, it would be beneficial to construct a prototype hardware implementation of such a system and connect this to a real network. This would help to determine the effectiveness of the system with real network traffic and could be a useful network-monitoring tool.

Acknowledgements

I would like to give acknowledgement to Robert Cole and his colleagues at Hewlett Packard Research Laboratories Bristol, as our source for the ideas of using state machine based filtering in network-monitoring systems.

References

[1] G.E.W.Tripp. A ring traffic monitor. Journal of Microcomputer Applications. 11, 87-93. Academic Press 1988.

[2] P.C.Hershey. Information collection architecture for performance measurement of computer networks. Ph.D. Dissertation. University of Maryland College Park, 1994.

[3] S.G.Richards. The Use of State Machine Technology in Broadband Network Monitoring. Dissertation submitted for the degree of M.Sc. in Distributed Systems. University of Kent. 1996.

[4] M. De Prycker. Asynchronous Transfer Mode: Solution for Broadband ISDN, 2nd edition. Ellis Horwood 1993.

[5] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming, Vol. 8, 1987, pp. 231-274.

[6] A.Burns, Programming in Occam 2. Addison Wesley 1988.

[7] P.F.Linington. Notes on the UKC Event Filter System, 95/8/31. Internal working paper.

[8] K.E.Grosspietsch, Associative Processors and Memories: A Survey. IEEE Micro Vol. 12, No. 3, June 1992, pp. 12-19.

[9] M. Schulz, Ch. Martini, D. Tavangarian, M.Darianian, K.Waldschmidt. An Associative Microprogrammable Bit-Slice-Processor for Sensor Control pp. C87-C95. 3rd Annual European Computer Conference (COMPEURO 89): VLSI and Computer Peripherals. IEEE 1989.

[10] B.Klauer, A.Bleck, K.Waldschmidt, The AM^3 Associative Processor. IEEE Micro 1995, Vol. 15. No. 2, pp. 70-78.

[11] J.Mick and J.Brick. Bit sliced Microprocessor design, McGraw Hill, 1980.

[12] A.Harasawa, T.Kaganoi, T.Kanoh, H.Nishizaki, M.Suzuki, H.Tomizawa, T.Shindou. NEC Corporation. An ATM Application Specific Integrated Processor. IEEE 1997 Custom Integrated Circuits Conference. Ch.129, pp.445-448.

[13] ITU-T Recommendation I.362, BISDN ATM Adaptation Layer (AAL) Functional Description, 1993.

[14] ITU-T Recommendation I.363, BISDN ATM Adaptation Layer (AAL) Specification, 1993

[15] Request for Comments: 1483. Multiprotocol Encapsulation over ATM Adaptation Layer 5. Juha Heinanen, Telecom Finland, July 1993.