Bolt Beranek and Newman Inc.

LEVEL II

AD A072423

(12)

Report No. 4149

AD A072424

Development of a Voice Funnel System

Quarterly Technical Report No. 3
1 February 1979 to 30 April 1979

June 1979

Prepared for:
Defense Advanced Research Projects Agency

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	(9)
4. TITLE (and Subtitle) (6) DEVELOPMENT OF A VOICE FUNNEL SYSTEM, QUARTERLY TECHNICAL REPORT NO. 3	5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical Rept. No. 3 1 Feb 1979 to 30 Apr 1979,
(14) BBN-4149	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) M. Hoffman R. D. Rettberg	8. CONTRACT OR GRANT NUMBER(s) (15) MDA903-78-C-0356, W/ARPA Order-3653
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street, Cambridge, MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3653
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE (11) 29 June 1979
	13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

(12) 41 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly switch, Multiprocessor, Pluribus, Wideband Satellite Network, PSATNET, Flow Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed during the period noted on two projects: (1) the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network, and (2) the analysis of Pluribus performance and end-to-end flow control in the wideband satellite network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

060100 JCB

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 3
1 February 1979 to 30 April 1979

29 June 1979

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract Expiration date: 30 November 1980
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Deputy Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either express or implied, of the Defense Advanced Research Projects Agency or the United States Government.

QUARTERLY TECHNICAL REPORT III

Contents

1. Introduction

This Quarterly Technical Report, Number 3, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 February 1979 to 30 April 1979. This report covers two independent projects encompassed by this contract: (1) system considerations for the wideband packet satellite channel, and (2) the design of the Voice Funnel system. These two projects are reported in the following two sections.

2. Wideband Packet Satellite Channel

During the previous months, we have concentrated our efforts on two areas which have significant impact on the operation of the wideband satellite network: (1) determining the ability of a Pluribus Satellite IMP to handle a multi-megabit channel data rate, and (2) designing an end-to-end flow control procedure for the network. Below, we summarize our principal results; complete details will be given in a final report. During the next quarter we plan to investigate channel congestion control and several related issues.

2.1 Pluribus Throughput

The Pluribus is configured as follows. There are four I/O interfaces: host in, host out, satellite in, satellite out. In order for data to be transferred between the I/O interfaces and memory they must be serviced by a software poller. The four I/O channels share a common I/O bus; each of three processor buses is shared by two CPUs. The three processor buses and the I/O bus contend for access to two common memory buses. The distribution of code and data in the memory is flexible.

We have analyzed two issues in particular. The first is the ability of the software poller to service each of the four I/O interfaces at the rate demanded by the megabit channel. On the basis of our results, the poller was redesigned to provide

improved performance. The second issue is the ability of the Pluribus to handle the processing burden placed upon it by the wideband application. We have developed an approximate analytic model that can be used to obtain order-of-magnitude estimates of Pluribus performance for a given configuration once more information is available on PSAT code. In dealing with both questions, we have not attempted to develop detailed analytic models but have made reliable, albeit rough, estimates in order to locate potential bottlenecks.

2.1.1 Software Poller

If one assumes that the software poller uses a round-robin discipline and that a processor requires 4 microseconds to execute an instruction, then the poller as originally implemented could service each interface once every 528 microseconds assuming that all interfaces had to be serviced on each round. An interface requires servicing whenever a message boundary arrives at the interface or whenever a 1024-bit buffer is filled.

Given the above, we seek to determine if, in the worst case, the poller can keep up with the data rate. The worst case means that data crosses all four interfaces at the full channel rate, an assumption which may be realistic if the host is the Voice Funnel. In this worst case, we calculate that each interface requires polling $k/1024 + 1$ times per message, where k is the message length. If one assumes that the messages are 6000 bits

long, then one needs to poll each interface an average of once every 600 microseconds at 1.5 Mbps and once every 300 microseconds at 3 Mbps. Thus, under its original implementation, the poller was inadequate at 3 Mbps.

Partly as a result of this analysis, the poller was redesigned so as to require 458 microseconds to service all four interfaces. This is still not adequate for the 3 Mbps case if only one processor is dedicated to polling. If two processors are dedicated to polling, then in the worst case, they can service each interface every 240 microseconds. While this should provide adequate performance at 3 Mbps, the impact of reducing the number of processors available to the rest of the system is not clear at present.

2.1.2 Pluribus Processing

All Pluribus configurations examined share the following features. There are four I/O channels, each of which operates at 1.5 (or 3) Mbps and attempts to access common memory one 16-bit word at a time. There are 3 pairs of CPUs similarly attempting to access common memory. The two critical areas of contention are the ability of the processor bus to service both CPUs and of the memory bus to service the processor and I/O buses.

It is assumed that all processor bus accesses to common memory are reads, while all I/O accesses consist of reads and

writes in equal proportions. It is further assumed that half of all accesses to common memory are to the extended portion of the bus. These assumptions, together with the Pluribus memory access parameters, imply that the total time to service an I/O access is 760 nanoseconds, while the average time to service a processor access is 910 nanoseconds. These numbers indicate the amount of time the memory bus is tied up by a given access. The accessing (processor or I/O) bus sees a longer service time, caused by a 1.2 microsecond bus coupler delay.

Additional assumptions are as follows. We assume that 25% of all memory references from the processors are to common memory. In addition, although one or two of the processors may be dedicated to polling the I/O interfaces and will therefore access common memory less frequently than the other processors, this is ignored. That is, it is assumed that all processors are behaving similarly and would like to access common memory with equal frequency. Finally, four I/O channels are treated as a single simplex line accessing common memory at four times the satellite channel bandwidth.

We seek to answer the following question: for a given satellite channel bandwidth, what is the maximum number of references that each processor can make to common and local memory. This problem is answered for two separate configurations of the Pluribus. In the first configuration, code and data are

accessed from the same memory bus. In the second configuration, they are accessed from different buses.

In the case where code and data are accessed from the same memory bus, contention for the common memory is modeled as an M/G/1 queue with four sources. This calculation yields the system response time for processor accesses to common memory as a function of memory bus utilization. Using this response time value, we compute processor bus utilization assuming that processor accesses to common memory see an additional 1.2 microsecond processor delay and that there are three times as many processor accesses to local memory as there are to common memory.

The following chart summarizes the results.

satellite channel bandwidth (Mbps)	utilization of m-bus	accesses/processor/second ($x10^2$)	p-bus utilization
1.5	.5	158	.41
1.5	.7	304	.84
1.5	.8	377	>1
3	.7	95	.26
3	.9	229	.7

The above case assumes that code and data are accessed from the same bus. We have also examined the configuration in which all of the code is accessed from one memory bus while all of the

data is accessed from the second bus, so that the I/O and processor buses no longer contend. The results (which are independent of data rate) are summarized below:

m-bus utilization	total processor accesses/sec ($x10^3$)	p-bus utilization
.6	440	>1
.5	366	.93

Finally, although this was ignored for the purposes of the above computations, a 3-Mbps channel bandwidth implies that a single I/O bus cannot handle all four interfaces if one assumes that the I/O bus must see the 1.2 microsecond bus coupler delay.

The above tables should be interpreted as follows. For a given memory bus utilization, we have computed the total number of processor accesses to local and common memory per second, and the corresponding processor bus utilization. The point at which the utilization of either the processor bus or the memory bus equals 1 indicates an upper limit to the total number of memory accesses for the given configuration.

We must emphasize again that the above calculations are intended only to indicate the methodology and to provide order-of-magnitude performance estimates. Because we have ignored the effects of contention for the local bus, the calculations may be optimistic. On the other hand, the above numerical results are based upon some specific assumptions about

the PSAT code. More realistic performance estimates can be obtained only after information about the actual code is available. (For example, we do not know how many memory accesses the processors will seek to perform for a given channel data rate, and we have assumed that the I/O channels and processors function more or less independently.) When more complete information is available, the model can be used for a rough determination of satisfactory Pluribus configurations. More accurate performance estimates will probably require simulations.

2.2 End-to-End Flow Control

The end-to-end flow control procedure we have developed is intended to control the rate at which traffic is input from all sources so as to prevent the occurrence of steady-state overloads at any particular destination. In this context, steady state means that the procedure is not designed to prevent short-term overloads resulting from temporary surges in traffic.

Without end-to-end flow control, there is nothing to stop some source (or some combination of sources) from sending data to a destination port at a rate which exceeds the rate at which the destination host can remove the data from the network. This situation does not benefit the sources in any way; they cannot get better delay or throughput than if they were sending at a slower rate which did not overload the destination. Although

allowing overload has no salutary effect, it does, however, have several negative effects which can cause serious degradations of network performance. One such effect has to do with the utilization of the satellite channel bandwidth. To say that some set of sources could transmit at a slower rate without increasing delay or reducing throughput is to say that they could leave more channel bandwidth for other traffic flows, which may be very important if the channel is heavily utilized. To put the point another way, if a set of sources is overloading a destination, they are simply wasting valuable bandwidth on the satellite channel.

Overloading a destination port can also interfere with the flow of traffic to other ports at the same PSAT node. Overloading a port will cause a large queue (of potentially unbounded length) to form. This will cause fewer buffers to be available to other ports, possibly reducing their throughput and increasing their delay, without any countervailing advantages to the destination port. Furthermore, it is possible that some of the packets in these large queues will have to be discarded before delivery to the host. This can happen if a buffer needs to be pre-empted (for a higher priority packet, say), or if the packet is queued so long that it times out (every packet in the network will have a "maximum holding time", after which it will be discarded.) If a packet must be discarded at the destination PSAT, then its use of resources at the destination PSAT, the

source PSAT, and the satellite channel have all been wasted. In summary, end-to-end flow control is essential for the efficient use of network resources, at all points in the network.

The requirements associated with the wideband network impose three restrictions upon end-to-end flow control. First, resources should not be allocated to traffic of a lower priority while traffic of a higher priority is being throttled. Second, destination resources should be fairly divided among all sources wishing to send data at a particular priority level. Also, use of network resources should be maximized. These criteria imply a scheme in which destination resources are dynamically allocated among sending ports and priority levels.

The flow control procedure that we propose is a distributed, rate-controlled procedure which exploits the broadcast nature of the network. The PSAT associated with a particular destination port informs all other PSATs (via piggybacking on data traffic or by a separate control packet) of the maximum rate that data can be sent to all hosts that are accessed by the network through that port. Each PSAT monitors all traffic in the downlink and therefore knows the rate at which data is actually being sent to that destination port at each priority level. With these quantities, each PSAT can then compute the unused bandwidth at each destination. From this information, each PSAT makes stream setup decisions and determines the rate at which datagrams at

each priority level will be accepted from each of its source ports for each destination port. This assigned rate is equal to the sum of the rate at which each source has been sending to the destination plus a fraction of the available bandwidth at the destination. The various aspects of the flow control procedure are presented in more detail in the following paragraphs.

Output Capacity: The output capacity of a destination port j, C_j, is a measure of the maximum rate at which data can be sent out that port. This rate will vary with the condition of the access line and the status of the destination host. The port's associated PSAT must therefore periodically measure the output capacity and communicate the results to all other PSATs. The measurement of output capacity can be performed as follows. Assume that PSAT A had announced at time t' that its destination port i could receive data at maximum rate C_j. PSAT A keeps counters which indicate the amount of data accepted since t' and the amount of time since t' during which all of the following conditions held: (a) the line which accessed A through port i was idle, (b) no data was queued, (c) the line was not blocked by some other protocol and (d) a buffer was available for sending data out that line. Thus, this counter indicates the amount of time the line was not utilized. If the utilization is low, but the sending rate is high relative to C_j, then we conclude that C_j is too low. If the utilization is high but the sending rate is low compared to the ostensible maximum C^j, then C_j is too high.

In either case, C_j is changed accordingly, the updated value is communicated to the network and all counters are cleared.

Measurement Periods: We define a unit of time (possibly different for each destination port) called a measurement period. Each PSAT monitors the traffic in the downlink and maintains tables indicating the total amount of data sent at each priority level to each destination port since the beginning of the current measurement period. These tables are kept for every priority level in such a way that the traffic volume computed for a particular destination and priority level is the sum of the traffic actually sent to the destination at that priority level and at all higher priority levels. Since these tables are used as the basis of throttling decisions, this procedure guarantees that lower priority traffic flows cannot result in the throttling of higher priority traffic because the lower priority traffic is invisible to all higher priority levels.

At the end of every measurement period, say at time t_1, each PSAT knows the following quantities:

$S_{jk}(t_1)$ is the bandwidth utilized by all streams to destination port j of priority class k or higher at time t_1.

$D_{jk}(t_1)$ is the average bandwidth utilized by datagrams to destination port j of priority

class k or higher over the last measurement period.

$d_{ijk}(t_1)$ is the average bandwidth utilized in the last measurement period by datagrams from i to j of priority class k or higher.

Using these tables, the flow of traffic is controlled in the manner described below.

Stream Setup: Suppose that a request is made for a stream of priority k from port i (associated with PSAT A) to port j (associated with PSAT B). The request is made in the form of a datagram specifying priority, stream interval, and other relevant information. All PSATs hear the request and all decide whether to accept the call as follows. Suppose the call is requested at time t_2 during a measurement period whose boundaries occur at t_1 and t_3. We calculate whether or not we have enough bandwidth to handle the stream by subtracting from the destination's capacity the bandwidth used by existing streams to j and the average bandwidth utilized by datagrams to j over the last measurement period. Thus, the available bandwidth for streams of priority k is given by:

$$A^s_{jk}(t_3) = C_j(t_3) - S_{jk}(t_3) - D_{jk}(t_3)$$

If this quantity is greater than the minimum required by the requested stream, then the setup is accepted.

Datagrams: At the beginning of a measurement period each PSAT must determine the maximum rate at which its source ports will be allowed to input traffic to each destination during the next measurement period. This is done in the following manner. First, the unused bandwidth at the destination is determined by subtracting from the capacity, C_j, the bandwidth used by existing streams, S_{jk}, and the average bandwidth used by datagrams over the previous measurement period, D_{jk}. A fraction, b_j, of this bandwidth is available for datagrams. The reason that we do not assign the entire unused capacity to datagrams is that this might result in unacceptably long queues. We therefore have for the available bandwidth for datagrams of priority k from source port i to destination port j:

$$A^d_{jk} = b_j(C_j(t_1) - S_{jk}(t_1)) - D_{jk}(t_1)$$

Each source PSAT can then determine the maximum rate at which it will accept datagrams of priority k from port i for port j. This rate is equal to the rate at which i sent datagrams of priority k to j during the last measurement period plus a fraction, f_j, of the unused bandwidth at j for datagrams of priority k. Thus the

maximum rate at which i will be allowed to send to j at priority k is given by:

$$M^{d}_{ijk}(t_1) = d_{ijk}(t_1) + f_j * A^{d}_{jk}(t_1)$$

It should be noted that this scheme utilizes a feedback mechanism to prevent overloads. Each PSAT is appropriating bandwidth for its sources without any knowledge of what the demand for destination resources will be for the next measurement period. As the total utilization of destination resources approaches the capacity, the total available bandwidth computed by each PSAT will decrease and therefore the additional bandwidth being appropriated by each PSAT for its source ports will decrease. Should overloads occur, this will be detected at the end of the next measurement period as a negative available bandwidth. In this case, the new maximum sending rate will be less than the previous maximum. The sending rates will be continually throttled until the overload disappears.

Fairness: Should all of the resources at the destination j be occupied by existing traffic, then there will be no available bandwidth for additional datagrams. Hence, a host wishing to initiate traffic to that destination will not be able to do so. Our fairness criterion is thereby violated. In order to remedy

this situation, we implement the following scheme. Every n measurement periods we examine the unused bandwidth at each priority level available for datagrams. If this number is less than some specified value, each PSAT recomputes the maximum sending rate for each of its source ports at the particular priority level according to:

$$M^d_{ijk}(t_1) = F_j*(d_{ijk}(t_1) + f_j*A^d_{jk}(t_1))$$

The effect of this computation is to throttle all sources by an amount which is proportional to their sending rate over the last measurement period and to evenly divide the unused bandwidth among all sources. The destination resources will therefore be more fairly allocated.

Note that the same sort of "unfair" situation might exist with respect to streams. That is, existing streams might be using so much bandwidth that no new streams can be initiated. However, unlike the case with datagram, it is not feasible to reduce the bandwidth used by an existing stream in order to make room for new streams. A stream user is supposed to have guaranteed bandwidth, once his stream is initiated. This means that it is inappropriate to reduce a stream's bandwidth for reasons of fairness. If fairness among contending stream users is a problem, the most sensible procedure would be to set a

maximum duration for each stream. Then, under periods of high utilization, the network would be permitted to delete streams of a certain age, if the bandwidth they are using is needed elsewhere. There are no technical problems with this procedure; whether to implement it or not is a policy decision.

Local Traffic: According to the above, each PSAT monitors the traffic going to all destination ports with whom their hosts are communicating in order to determine at what rate they may accept data for a particular destination. The accuracy of this information arises from the broadcast nature of the network, i.e., each PSAT hears all datagrams in the downlink. This scheme is undermined if there exists a substantial volume of traffic which is not sent via the satellite link. Such traffic is termed local and consists of data between two hosts connected to the same PSAT. So that all PSATs can accurately determine the available bandwidth at each destination, the destination PSAT must periodically communicate via the satellite link the volume of its local traffic.

Stream Preemption: The stream set-up procedure discussed above determines whether or not there is sufficient bandwidth for a datagram based upon the number of existing streams and the average datagram flow over some period in the past. There exists the possibility that a stream will be set up only to be followed by an increase in datagram traffic at a higher priority level.

Since streams will probably be associated with ongoing voice conversations, one does not want to preempt streams frivolously, i.e., in response to short-term surges in higher priority datagrams. To avoid this problem, the destination PSAT will have the option of preempting individual stream packets in order to service an increase in higher priority datagram traffic. If the rate of packet preemption exceeds some threshold the stream will be deleted.

Datagram Source Throttling: At the beginning of a measurement period a source port will have assigned to it a maximum sending rate for each priority level. These rates, or "quotas", will be strictly decreasing with priority. The PSAT makes its throttling decisions as follows. For each source-destination pair, the PSAT maintains four counters, one for each priority level. Each time a datagram arrives from the source for the destination, the counter corresponding to that priority level and all lower levels is incremented. The "quotas" specify data units per time unit. At the beginning of the time unit all counters are zeroed. When the counter at each level reaches the specified quota, no more packets of that level are accepted for the duration of that time unit.

3. Overview of Voice Funnel Design

This section presents our current thinking on the design of a Voice Funnel System. During this quarter we have reviewed and updated our earlier ideas concerning the design of the Butterfly Switch and we have carried the switch design to a finer level of detail. We have also begun the design of the processor node and have performed the top level design and analysis of the application software and the operating system. Naturally, at this stage not all areas of the design are equally detailed, and we expect to modify the design somewhat as our work progresses. Nevertheless, we believe that we have considered all major areas of the system at least at a high level and that we have a coherent overall design.

Anticipating that both the Voice Funnel and the Butterfly Multiprocessor will be successful, making eventual enhancement of both a likely possibility, we have been careful to include in our design the potential for expansion upon our early work. Thus in many cases we suggest software features which may not be included in the initial version. While we do not expect to be able to do all of the things mentioned here, we are confident that we can produce a very useful and interesting initial system. Looking beyond the initial system, we feel that it is important to describe more than we might implement initially, because this will provide others with an early indication of our thinking and

will increase our own assurance that our design is sufficiently robust to permit extension.

3.1 Voice Funnel Software

The task of the Voice Funnel will be to interface to a large number of speech terminals which generate INTERNET packets. It will multiplex their data streams into a wideband satellite channel and demultiplex these streams for delivery to speech terminals at other sites. The initial Voice Funnel will support 1.5 Mbps of total traffic, although it is expandable to permit operation at higher speeds later. Since the interface to the Voice Funnel is INTERNET packets, any host generating INTERNET packets may be connected to the Voice Funnel. The Voice Funnel is particularly well suited to hosts which generate traffic in the form of streams.

The functions of the Voice Funnel divide into two categories, main line tasks and background tasks. Main line tasks involve the actual handling of speech packets, and must be performed quickly enough so that the total delay of a speech packet in the Voice Funnel is kept sufficiently small (about 50 msecs). Background tasks are not directly involved with the transmission of individual packets; as a result, there is no firm latency requirement.

Speech data is received by the Voice Funnel as INTERNET packets, which include NVCP (or NVP-II) packets within them. Each such packet includes digitized speech in the form of a small number of parcels or a stream of bits. In order to interface simple encoders which transmit only parcels or bit streams, a Voice Funnel option would be able to convert between simple speech data streams and INTERNET packets.

The Voice Funnel can be viewed as a network, with a central network switch in the Voice Funnel and network hosts implemented both within the Voice Funnel software and by external devices. External hosts include speech terminals; internal hosts include a gateway to the PSATNET and the hosts which interface to simple encoders. The use of the network structure allows implementation of functional modules as hosts which may be attached to or removed from the network as desired.

The main line functions of the Voice Funnel will be performed by a group of modules which will accept packets from speech terminals, aggregate them for transmission over the PSATNET (a wideband satellite network), segregate them at the other end, and deliver them to their respective destination speech terminals. Eventually the Voice Funnel will support speech terminals which are connected directly, through a local net, or over a network of Voice Funnels or other machines, and which transmit packets, parcels, bytes, or a continuous bit

stream. Where needed, device-dependent modules will convert between the format used at the interface and INTERNET packets. Other modules will take packets destined for the satellite channel and merge them in stages into a large message buffer. Finally, a satellite network header will be attached as a prefix and the buffer will be transmitted.

Incoming messages will be broken up into individual INTERNET packets, and the packets will be passed to modules handling individual speech terminals or encoders. Where needed these modules will perform speech reconstitution, a procedure by which the speech in the incoming packets will be restored to its original order, missing speech will be replaced by silence or by extrapolated sounds, and the resulting speech transmitted at regular intervals to restore smooth speech. Other modules will operate in the background. Background tasks will include call and conference establishment and control, and accounting.

An external controller may be provided outside the Voice Funnel in order to provide control functions such as call placement, conference chairing, and accounting. In the absence of the external controller, an internal default controller will provide simple versions of these functions. Use of alternate controllers will be simple since the controllers are implemented as hosts on the network.

The goals of low delay and high bandwidth have influenced much of the design of the application software. The Voice Funnel will employ several techniques for achieving high bandwidth in the Voice Funnel and over the satellite channel, including: multiplexing messages on stream traffic; minimizing copying of data and using the block transfer mode of the switch interface when appropriate; and increasing parallelism by using many small processes rather than a few large processes.

3.2 Butterfly Switch

The Voice Funnel System will be implemented on a newly designed machine, the "Butterfly Multiprocessor". This machine will consist of from one to several hundred processing units called "processor nodes" which will be interconnected by a switch called the "Butterfly Switch".

The Butterfly Switch is a network of elements, called "switch nodes", which route messages between processors and remote memories. All of the switch nodes in a Butterfly Switch are identical. Each node routes messages which appear on its input ports to one of several output ports on the basis of address bits at the beginning of the message. The route a message takes through the switch is determined by the interconnection pattern of the switch nodes.

A preliminary description of the Butterfly Switch has been presented previously.* Efforts to further refine the design of the switch and understand its performance have led to several important improvements. The following points summarize these design refinements:

- The number of inputs and outputs on one switch node has been increased from two to four, significantly reducing the number of switch nodes in a Butterfly Switch.
- The data paths through the switch are four bits wide to increase the bandwidth of the switch and reduce the delay experienced by a message as it travels through the switch.
- The "retreat" strategy, rather than the "wait" strategy, will be applied when conflicts occur in the switch.
- Each message will carry a checksum which can detect errors in the data or routing of the message.
- An extra column will be added to large switches in order to minimize the effect of a failure in a switch node.

Increasing the number of input and output ports on a switch node from two to four reduces the number of switch nodes by a factor of four. We use the term "base" to denote the number of input or output ports on a switch node. The table below gives the number of nodes required to implement switches with base-4 and base-2 nodes.

* R. D. Rettberg and M. F. Kraley, "Introduction to the Voice Funnel System", Bolt Beranek and Newman Inc., Report No. 3990, December 1978

Ports	Number of Switch Nodes	
	Base-4	Base-2
4	1	4
16	8	32
64	48	192
256	256	1024

Base-4 nodes also reduce the number of columns in a switch and thus the delay through the switch and the likelihood of contention. The primary disadvantage of base-4 switches is that they grow less gracefully than base-2 switches. Each base-2 switch has twice as many ports as the next smaller size switch while each base-4 switch has four times as many ports as the next smaller switch. This makes it harder to match the desired number of ports to the size of the switch.

We have discovered that the simple "wait" strategy for resolving conflicts is prone to deadlocks. Substituting the "retreat" strategy eliminates the deadlock and permits a switch interface design which has no known deadlocks.

Two mechanisms will make the switch more robust in the event of switch node failures: checksums and extra columns. Each message will carry a 4-bit checksum to help detect errors in its data, and the destination processor node address will be included in the checksum to detect errors in message routing.

A Butterfly Switch is particularly vulnerable to the failure of a switch node which is on the interior of the switch since

such a failure may isolate many processor nodes. An extra column of switch nodes provides an extra path through the switch which may be used to route messages around the faulty switch node. Since a 16 input port switch has only two columns, there are no "interior" columns, and adding an extra column would not make sense. As a result, extra columns will be added only to larger switches.

Simulations of the Butterfly Switch have now been performed which show that the effect of references to global memory becomes important as the number of global memory references approaches 5% of the total. The expected global memory reference rate for the Voice Funnel is unknown at this time. However, by executing instructions from local memory and requiring that the stack and private variables also be local, we will be able to reduce the global reference rate significantly below what it would be if we ignored locality entirely.

In order to calibrate the effect of the Butterfly design upon the switch, we also simulated a crossbar switch having the same data path width. Interestingly, the simulations also show that the performance of a Butterfly Switch does not differ greatly from that of a crossbar switch of equal path width.

An MSI implementation of the switch node has been designed which requires between 40 and 60 MSI TTL packages. We expect that this design will yield a switch with a 48 Mbps potential

data rate between any pair of input and output ports. For small machines, such as the prototype Voice Funnels, a switch node of this complexity could be implemented on a moderately small printed circuit card. A large machine with, say, 256 processor nodes would have 256 switch nodes, making an LSI implementation of the switch node more attractive.

Up to now, we have assumed that request messages and reply messages would travel through the switch independently. As an alternative, we might design the switch so that the path from the source to the destination would be held after the request message has been sent so that a reply could be returned over the same path. This approach could have several advantages:

- the source processor node address need not be sent, shortening the request message;
- the reply message would not suffer contention either within the switch or at the source processor node;
- the "wait" strategy would be free from deadlocks and could be employed if desired.

The "bidirectional" switch is more complicated than a "unidirectional" switch and may run slower. Holding the path for the reply would cause some switch paths to be occupied while no data was being sent. This would increase the number of conflicts in the switch. The balance between these advantages and disadvantages is unknown at this time.

3.3 Processor Node

Processing in the Butterfly Multiprocessor will be performed by modules called "processor nodes". A processor node will consist of a processor, I/O devices, memory, and an interface to the Butterfly Switch. The current design permits Butterfly Multiprocessors which contain up to several hundred processor nodes.

The processor used in the processor node will be Zilog's Z8000. The Z8000 is a highly integrated microprocessor which represents the state of the art in commercial microprocessor design. Several characteristics make the Z8000 particularly attractive:

- the "segmented" version supports 23-bit virtual addresses;
- a compatible memory management unit has been announced; and
- the Z8000 is a reasonably fast 16-bit processor.

The ability of the Z8000 to support a virtual address space of greater than 16 bits is particularly important, since a multiprocessor is often a large machine and is expected to handle large problems. Large problems, and large programs to solve them, require a large memory space. In current machines, several techniques such as base registers and mapping registers have been used to expand a 16-bit address into a larger address. These solutions miss the point, however. In a large data structure,

pointers to objects must be atomic units in order to eliminate translations when using a pointer. The most straightforward way to provide this is to support longer virtual addresses at the outset. The Z8000 permits manipulation of 23-bit virtual addresses and 23-bit pointers embedded in 32-bit double word operands. Eventually even larger address spaces may be required, but by that time we expect microprocessors with larger virtual address spaces to be available.

The previous discussion has concentrated on the address space seen by the programmer (or compiler). The hardware supports another address space -- the physical address space. During each memory access, the virtual address is translated into a physical address by a device called a Memory Management Unit. The Memory Management Unit uses a translation table maintained by the operating system. Memory management facilitates protection by limiting a process to only those areas of physical memory which are specified in the translation table and by allowing each process to access memory only as specified by protection flags in the translation table.

A benchmark program has been written for the Z8000 and has been used to compare the Z8000 with the the LSI-11. These results are based on preliminary specifications of the Z8000 and should be regarded as preliminary themselves. Briefly, we have found that:

- The code density of the Z8000 is equivalent to that of a PDP-11;
- It is straightforward to code those things that programs do most often;
- A few nonuniformities in the Z8000's instruction set might make code generation slightly more difficult than for the PDP-11;
- The Z8000 in the segmented operating mode is about 2.6 times as fast as an LSI-11;
- The average instruction execution time is about 2 microseconds.

Commercial microprocessors are not designed for use in tightly coupled multiprocessor systems. As a result, one must expect some incompatibilities between the processor and the rest of the processor node design. For example, the Z8000 expects that all of its memory is "close" to the processor. As a result, it seizes the bus for the duration of a memory access. In a Butterfly Multiprocessor, some accesses will be made to memory which is in the "global" address space. Were we to permit the Z8000 to hold the bus while the access is in progress, important real time events might be blocked. Other examples include the method of refreshing main memory and timing considerations. In each of these cases, specialized circuitry is required in order to correct for these deficiencies.

Several I/O devices are common to every processor node:

- The memory management unit described above;
- A counter-timer which will be used to provide interrupts for real-time or scheduling events;

- A Read Only Memory which is used for system loading and debugging;
- A serial interface for loading, system control, and debugging.

In addition to these standard I/O devices, the Butterfly Multiprocessor will require interfaces to support the application. To the extent possible, these interfaces will be implemented using standard LSI interface circuits such as USARTs or advanced communications protocol chips. Where bandwidth considerations are important, these interfaces will be direct memory access devices.

Each processor node will contain an interface to the Butterfly Switch. This interface will offer several communications services. It will identify memory accesses which are to global memory and convert them into messages addressed to the correct processor node. It will also respond to request messages from other processor nodes for accesses to its own local memory. The programmer will be unaware when an access uses the switch.

The switch interface will also be able to transfer a block of words between the memory in its own processor node and that in another processor node. This type of transfer must be specified explicitly by the programmer, but it will make very efficient use of the bandwidth of the switch. To use this mode, the programmer specifies the direction of the transfer, the starting virtual

addresses of both the local and remote memory regions, and the number of words to be transferred. The switch interface will perform the transfer, making up request messages and dealing with replies. In order to reduce the impact of long messages on other users of the switch, long transfers will be broken into blocks of at most 8 or 16 words.

In addition to data transfers across the switch, the switch interface will support special messages which perform functions such as interrupting a remote processor, amputating the processor, or locking a resource.

3.4 System Software

Supporting the Voice Funnel application will be an operating system which will provide the following services:

- processor management (or scheduling),
- memory management,
- inter-processor communication,
- process synchronization,
- interrupt handling,
- system configuration, and
- reliability assurance.

The operating system will provide an environment in which the work of the Voice Funnel application can be partitioned into a number of individual tasks which can be performed in parallel

by several processes running simultaneously on different processors. In our model, tasks and processes are parallel concepts: a task is the unit of work performed by a process (although a process which cycles may handle a series of tasks).

Because the Butterfly Switch will permit all processors to access all available memory directly, cooperation between processes on different processors will be almost as easy as between processes on the same processor. Not only will the operating system take advantage of this feature itself, but it will also provide facilities to make it safe and easy for application processes to communicate freely regardless of which processor they are running on. While accesses to remote memory may be convenient, however, they will encounter both delay and contention, and excessive use of remote memory could seriously degrade overall system performance.

In order to efficiently use the available hardware while still achieving high parallelism and low delay, the application software and the operating system will observe certain conventions:

- a process may only run on the processor in whose local memory it resides; in consequence, processes which might run on several processors must be replicated;
- a task, however, may be performed by any copy of the process written to handle it, regardless of the processor the process runs on, except when a specific process is required for some reason;

- in order to minimize delay for individual tasks, processes which will perform frequently occurring tasks will be created in anticipation of their use (on one or more processors) and will remain resident;

- except under unusual circumstances, processes will not be moved from one processor to another; normally, when a process must be run on another processor, a copy of the process will be created there.

Scheduling in the Voice Funnel must provide a mechanism for efficiently using many processes and processors to perform a large number of tasks with low delay. Because tasks can often be performed by any copy of the process which handles them, it is possible to take advantage of multiple processors by having a global scheduling function which directs incoming tasks to suitable processes on processors which are not too busy. Because processes will be created in advance of their use and can only be run on the processors to which they are local, global scheduling of processes will not be worthwhile. However, local scheduling will be necessary in order to allocate time on a priority basis among the processes assigned to each processor.

Three potential global scheduling algorithms are under consideration:

- an implicit algorithm having no explicit global scheduling module, but instead relying upon natural tendencies in the system to cause tasks to flow toward processors which are less busy;

- a decentralized, but explicit, global scheduler which would pull tasks toward processors as capacity became available to them; and

- a centralized, but migrating, global scheduler which would push tasks towards the processors with the most excess capacity.

The implicit algorithm will be implemented first because of its simplicity and naturalness. Its performance characteristics will then be observed, and one of the other algorithms will be considered if the implicit algorithm does poorly in practice.

Each process will have access to a large virtual memory (up to 8 megabytes on the Z8000). Because the memory will be segmented (up to 127 segments on the Z8000), and because any process can directly access memory in any processor, it will be possible for processes to selectively share memory with one another in order to collaborate on a problem or to communicate rapidly, while concealing information such as instructions or private data from processes which should not be able to read or modify it. If not used carefully, these memory access and protection features could add significant overhead through memory fragmentation, context switching, or translation of shared references. Thus, virtual address spaces (with their size, protection, and sharing attributes) will be specified at link time, rather than at run time, retaining most of the flexibility offered by the memory management hardware without incurring significant overhead.

Reliability mechanisms similar to those found in the Pluribus Multiprocessor will be provided in order to exploit the

reliability potential inherent in a multiprocessor. Among the ideas which will be borrowed from the Pluribus are:

- discovery and verification of the hardware configuration at system initialization time;
- development and maintenance of a "consensus" regarding the set of correctly functioning common resources;
- periodic checking of critical resources;
- use of redundant information for verification of data structures;
- timeouts for locks on common resources; and
- blocking of higher level activities when lower level activities have detected possible errors or inconsistencies in resources upon which the higher level activities depend.

A UNIX time-sharing system will provide the basic environment for software development. A limited number of tools will be created (mostly by modifying existing tools) to permit development and maintenance of the application and the operating system to proceed rapidly. Both the application software and the operating system will be written primarily in the C Programming Language, although certain frequently executed or machine-dependent modules will be written in assembly language.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency

Dr. Robert E. Kahn (2)

Defense Supply Service -- Washington

Jane D. Hensley (1)

Defense Documentation Center (12)

Bolt Beranek and Newman Inc.

Library
Library, Canoga Park Office
R. Brooks
P. Carvey
P. Castleman
F. Heart
M. Hoffman
D. Hunt
M. Kraley
W. Mann
J. Mayersohn
J. McQuillan
R. Rettberg
I. Richer
E. Rosen
E. Starr
D. Walden
E. Wolf