THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 6, July-August 1982
Printed in U.S.A.

Automated Repair Service Bureau:

System Architecture

By R. L. MARTIN

(Manuscript received June 29, 1981)

The four main functions being served by the Automated Repair Service Bureau are (i) Customer Line Card Maintenance—a large, complex data base problem; (ii) Trouble-Taking and Tracking—a simple transaction problem; (iii) Loop Testing—a process-control problem; and (iv) Trouble-History Review—a large, filter, sort, and count report-generation problem. All of these functions were implemented among different computers, which were then networked together to form the full ARSB architecture. This decomposition resulted in an architecture that has been adaptable to Bell Operating Companies' needs over a full decade.

I. INTRODUCTION

The architecture of a system is the product of the history of the organization which builds it, the present and near-present technology, and the intended application. The architecture of the Automated Repair Service Bureau (ARSB) is the result of several iterations, and many decisions. In retrospect, many of the decisions which we tortured over seem, given that the system works, clear and obvious. Hopefully, our experiences will be useful lessons for other similar developments.

After describing the characteristics of the present ARSB architecture and the history which led to it, the major design decisions in its formation will be discussed. This will be followed by a slightly more detailed discussion of the architectures of the subcomponents of the system.

1.1 History

The Loop Maintenance Operations System (LMOS), as an operations system for the Repair Service Bureau, was first conceived as a system

in August, 1970, as part of a broad-reaching systems engineering study. Specific development work on what ended up being its prototype, the Mechanized Line Record System (MLR) started in April, 1971. The first MLR repair service bureau (RSB) was placed on-line in Manhattan, New York, on December 8, 1972, on an IBM 370/155 and DEC PDP* 11/20 computer. The 370/155 provided all the major user functions. The 11/20 initially provided simple backup when the 370/155 was down by recording trouble entry and status transactions for later submission to the 370 when it recovered; also, it provided loop-testing facilities via the Line Status Verifier (LSV), an early automated testing system. After going on-line, the next year and a half of MLR application development had two major activities:

(*i*) The transactions were restructured for ease of use and to utilize the IBM 3270 synchronous display terminals.

(*ii*) The data bases were restructured to improve computer performance and simplify program development by changing a combined line record/trouble data base to separate line record and trouble data bases.

As other Bell operating companies (BOCs) (Southwestern Bell Telephone Company and South Central Bell Telephone Company, in particular) expressed interest in MLR, we decided to totally rewrite the system because:

(*i*) The existing data base structure could not easily accommodate the new Universal Service Order (USO) requirements. (USO is the language used for the entry of customer service requests for new services or changes to existing services.)

(*ii*) An IBM 370-based system would not have been economical for the "small" towns (small, at that time, was anything less than 750,000 people.)

(*iii*) The IBM 370-based system's one-half hour or so mean-time-to-repair or daily availability of 16 hours was not satisfactory for a back-bone customer support system. It is important to note that given a fixed monthly availability, e.g., 98.5 percent, the user is more concerned with mean-time-to-repair than with mean-time-to-failure.

We initially planned to build two systems, an IBM 370 maxi-based system for the large cities and a separate DEC 11/45 mini-based system for the suburban/rural areas. They were to have identical user views to minimize development cost and time, and to facilitate moving RSBs and personnel from one system to the other. J. Cloutier of Bell Laboratories suggested the distributed architecture which we subsequently implemented as LMOS and which this paper addresses.

Though no MLR code was included in LMOS, the MLR system provided two major benefits to the LMOS project. First, it was a working model

* Registered trademark of Digital Equipment Corporation.

or pilot plant from which we could gather data and experience for the LMOS design decisions. Its role in this capacity cannot be over emphasized. Second, it acted as an existence proof that such a system could be used as part of an on-line customer interaction and could save the Bell System money.

II. ARCHITECTURE OVERVIEW

The hardware/software distribution of function and data is summarized in Fig. 1, while Table I summarizes the operational characteristics of the network.

The function/data distribution was such that the system could be partitioned into large complex-data "maxi" pieces and high-transaction-volume/small simple-data "mini" pieces. (See Fig. 1.) They were as follows:

(*i*) *Line card maintenance (a host maxifunction).* Copies of the customer's service and service history are maintained—a low-volume but large-complex data base function.

(*ii*) *Trouble taking and tracking (a front-end minifunction).* The customer's trouble is entered and subsequent repair tracked (via on-line status and reports)—a high-volume but simple data base function.

(*iii*) *Loop testing (a front-end minifunction).* The loop is tested for fault presence and location—a low-volume but complex algorithm function.

(*iv*) *Trouble history review (a host maxifunction).* A 40-day history of troubles is reviewed via "batch" reports for analysis of equipment and personnel performance trends—a large sort-and-count function.

The LMOS transaction load follows the classic 80/20 rule. That is, 80 percent of the total transaction load is generated by a few high-use

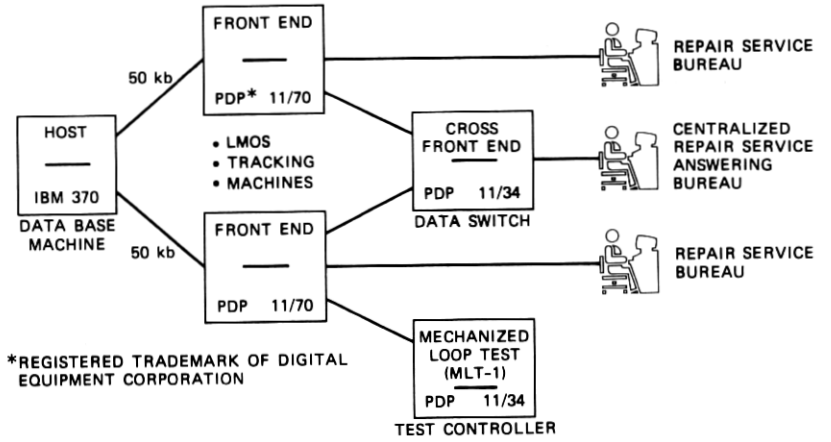

Fig. 1—Automated repair service bureau architecture.

Table I—ARSB data sheet: typical configuration serving five million customer lines

	Host	Front End	Cross Front End	MLT Controller
COMPUTERS				
Number in system	1	7	2	40
Processor type	IBM-370-303x	PDP* 11/70	PDP 11/34	PDP 11/34
Main memory (per machine)	4 megabytes	1 megabyte	1/4 megabyte	1/4 megabyte
Secondary storage (per machine)	6,000 to 8,000 megabytes	300 megabytes	0	0
TOTAL SYSTEM TRAFFIC				
Busy hour transactions	17,000	28,000	12,000	12,000
Daily transactions	145,000	175,000	80,000	60,000
OPERATIONAL PERFORMANCE				
Scheduled availability	24 hr (1)	22 hr	24 hr	24 hr
Operational	98.5%	99.5%	99.5%	99.5%
Mean time to recover	40 min	7 min	4 min	8 min
SIZE OF SOFTWARE (LINES)				
Batch	140,000	23,000	0	0
On-line	35,000	225,000	22,000	30,000
Operating system	(2)	34,000	12,000	12,000
Utilities	(2)	45,000	0	0
Total	175,000	327,000	34,000	42,000

* Registered trademark of Digital Equipment Corporation.
(1) On-Line 12 to 18 hours (depending on BOC). Available for batch processing all other times.
(2) IBM Multiple Virtual System and Information Management System.

transaction types (about 20 percent of all transaction types). These characteristics were first verified during the MLR pilot operation and justifies the distributed architecture of LMOS which places the high-use, simple data transactions on several front-end computers. The projected cost of the distributed architecture was 30 percent less than either an all-maxi or all-mini approach and allowed us to build one standard system rather than two.

The best way to understand the distributed architecture is to trace how a trouble is reported and repaired. This is as follows:

1. *Enter trouble description*—A customer calls a repair service attendant at the Centralized Repair Service Answering Bureau (CRSAB).[1] The attendant enters the telephone number of the line in trouble. The entered telephone number is switched by the cross front end[2] to the front end[3] having the copy of the customer's miniline card. (The miniline card is roughly 10 percent the size of the full line card which is kept in the host[4] IBM system; however, it contains enough "critical" data to support trouble entry, as well as limited operations when the host is "down.") In the case of numeric telephone numbers, a cross front end table is used to determine the proper front end. In the case of nonnumeric circuit identifiers, the cross front end interrogates all front ends to determine which has the miniline card. After

receiving the telephone number, the front end returns a description of the customer's line (and any existing trouble information) to the attendant via the cross front end.

2. *Test loop*—Simultaneous with trouble entry, the front end initiates a test of the line in trouble via the Mechanized Loop Testing (MLT)[5] controller computer. When the MLT controller returns the test results to the front end, roughly 25 seconds later, it routes them, again via the cross front end, to the attendant. The attendant, now armed with customer service information and test results, enters the trouble description and an agreed-upon repair commitment time.

3. *Route trouble description to* RSB—This final trouble description is sent via the cross front end back to the front end. The front end then ships this trouble description to the host computer for combination with the full customer line record and associated trouble history. This combined record is shipped via the front end, and perhaps cross front end, to the RSB having repair responsibility. (If the host computer is unavailable, the trouble description is combined with the miniline record for the RSB.)

4. *Repair trouble*—While being repaired, the status of the trouble, e.g., tested, is entered into the front end. In this way, trouble repair can be tracked by the bureau management using on-line reports, can be relayed to the customer, and can be analyzed after trouble clearance.

5. *Post repair analysis*—After repair, a description of the closed trouble is entered and sent to the host computer for a 40-day running historical file used for analysis of various components of the repair process. The resultant reports that are generated by the trouble report evaluation analysis tool (TREAT)[6] are routed from the host computer to the RSB (optionally via the front ends or cross front end).

While the repair process is taking place, the customer line records are being updated either manually or automatically from data received over the BOC service order distribution network on the host computer. Complete copies of the miniline record for all changed line records are sent to the front ends the night after the change is entered into the host computer. A night-time versus day-time update was chosen to off-load the front ends and the 50-kb lines connecting the host to the front ends (see Fig. 1).

III. MAJOR DESIGN DECISIONS

There were, in effect, two levels of ARSB architecture-design decisions: (*i*) those affecting the overall system (discussed in this section), and (*ii*) those affecting an individual component of the system, e.g., the front end (discussed in the later sections). The decisions that affected the overall system had one of two motivations—either trying

to satisfy the end users operational objectives or trying to satisfy our development objectives. The interplay between what *should* be done and how it *could* be done is the most exciting and yet exasperating experience for the designer. If either perspective—the user or the developer—gets out of hand, the end system will surely fail.

The major user and developer issues and lessons will be discussed in the following sections.

3.1 User issues—don't forget the end user will rely on and supplement your product

The major user issues which impacted the architecture evolved around the entry level skills of the end user, the desire to install the system quickly, and the high efficiency of the existing manual repair process. These factors all interrelated to lead to the following decisions:

(i) The System would not be based on a pure data base—don't make the machine enforce what the field won't.

After much interaction with the operating companies who wanted, initially, to have the software enforce a pure data base, i.e., reject a line record with machine detectable inconsistencies (e.g., a cable assigned to two telephone numbers), we decided that it was better to put the line record in the data base with its known errors than not to put it in until the errors were resolved. Known errors are, of course, flagged for future resolution. This decision was contrary to that being made by the facility assignment systems, e.g., COSMOS and BISCUS/FACS which assign loop facilities to provide service. We could not afford the difference in the cost of conversion—more than 6 to 1 for each customer record. Further, the end user of the system, repair people, historically had been able to use impure data for repair operations and would not tolerate the expense of trying to keep it purer than it had been in the past, though there would be tools to do this. (We have since found that repair does not in fact, keep the data pure, nor is it necessary.)

(ii) High-volume operations—do what you can, don't do everything.

The RSB operation is essentially a high-volume repetitive job wherein a 100,000-line RSB will process 500 troubles a day. The majority of these troubles is in residence and simple business services. This gave us the opportunity to save money by leaving the hard cases out of the system, both in the data base and in the automated testing. A human being handles these relatively low-frequency, very complex cases. Thus, the system could and can be viewed as the high-volume mechanized adjunct to the human being. The decision was: do not try to do everything; let the human augment the machine (and conversely!).

(iii) Part of the on-line customer flow—sit in the end user's job and view availability from that perspective, for example, 98.5 percent availability is 2½ hours down time a week!

The system is an adjunct to the daily customer interaction process. Thus, we had to ensure that high availability was provided for parts of the system that were used for daily interactions between BOC personnel and their customers. Further, these operations had to have a relatively predictable response time. Thus, we allowed no algorithm which had high run time variance, otherwise customer contact would be unpredictable. Also, we learned that the BOC user can manually augment missing features and, thus, will tolerate, though not happily, their absence; however, they will not, and should not, tolerate bad performance or availability.

As mentioned before, in the operations support world, mean-time-to-repair is much more crucial than average uptime. (Two hours down Monday morning is intolerable.) A rapid mean-time-to-repair allows less stable software to be introduced to the field. The result of this issue was a front end and cross front-end hardware configuration which had at least one passive backup unit for each two on-line units. This backup hardware could be switched on-line via a network switch and disk switches. Also, a very fast data base recovery system was built, (less than two minutes recovery time for all but the head crash).

3.2 Developer issues—manage complexity

The development decisions were all motivated toward reducing development complexity. All the major design decisions were oriented towards decoupling the development of the subcomponents. (An extension of this theme was not to make any changes in the IBM software for the front-end system to work.) We partitioned the software tasks into pieces which could be done by teams of three to five people for development efficiency and risk containment. Further, by uncoupling the development activities from each other, the companies could decide on the order of installation for the various pieces of the system; however, there was a natural and commonly followed installation sequence of TREAT,[6] host, front end, cross front end, and MLT. It is important to note, however, that the overall architecture and development approach was for an integrated system which could be developed in phases as compared to stand-alone pieces which could, somehow, be forced to fit together.

The resultant major decisions to allow this decoupling were as follows:

(i) Duplicate data to simplify distributed processing—reduce risk by using known technology.

We decided that no single computing activity would be dependent

on the on-line interaction between computers. That is, we did not want a single user transaction to have to get data from more than one machine. We did not know how to solve the data base locking, consistency, and data base recovery problems that accompanied that form of operation and so we avoided it. (For example, we did not want to solve the clean-up problem resulting from transaction A on machine 1 locking data base B on machine 2, updating it, going down, and simultaneously having the line go down.) Instead, we duplicated a small amount of data across machines in a master (host data base)/slave (front end) relationship to support the transaction activities.

(ii) Communications design—do the riskiest first.

The most complex design problem—though we did not know it at the time—was the distributed communications network. For MLR, the PDP 11/20 communications software we wrote handled solely the synchronous terminals (not the printers!) in a stand-alone fashion (i.e., we did not have a full pilot plant).

Our initial design was quite simple. We decided that to each sending computer, the receiving computer would look like a TELETYPE* teleprinter 40/4 controller. The IBM host computer thought that the DEC front-end computer was a pair of 3270 controllers, one with 32 printers and one with 32 CRTs. The CRTs were used as "virtual" terminals by the PDP computer and were assigned or coupled to a real terminal when it used an IBM resident transaction. In this way, the IBM transactions did not have to be tested for use with the front end and no special work had to be done in the front end when an IBM transaction changed or was added.

Two of the "virtual" CRTs were used to effect the batch data transfers between machines. That is, the miniline cards, for example, were blocked into CRT-like messages and were sent to one of these two CRTs.

The handling of printers and front-end-to-host messages almost killed the project. The early strategy had four parts:

(a) By mapping the printers on the front end to one of the "32 on the 50-kb line," the host system would do all the spooling for Host generated messages.

(b) To reduce spooling logic, the front end would separately spool its output for the printers. The front end would control actual printing by either "unspooling" its messages or control passing through of the host messages.

(c) To simplify the software, no messages from the front end to the host would be spooled to disk. Rather, as these messages were created, they were put in a core buffer for transmission to the host.

(d) When the host was "attached" to a printer or CRT, all terminal

* Registered trademark of Teletype Corporation.

and printer status messages would be passed directly from the front end to the host.

The first two decisions were good ones—the last two were somewhat less than optimal. By not spooling to disk, we implicitly counted on predictable bandwidth to the host, i.e., the core buffers could fill up on the front end and stop the system. Further, by sending the status directly to the host, sometimes it would take the whole 50-kb line down because, for example, it thought a bad printer was on the line, i.e., we confused network and terminal control. These two design problems took 3 to 6 months to fix, generated user dissatisfaction, and "work-arounds" that complicated the communications manager design. The major lesson was that we tried to schedule the development of something that neither we (nor anybody else) had done before.

(iii) Operational decoupling—reduce the need for organizations to communicate in the field.

We decided that the basic operations of the host and front end had to be viewed by the computer operations personnel as though, for all practical purposes, the other system did not exist. For example, if a data base recovery took place on one system, then an operator (or software) would not have to initiate one on the other. This decision actually simplified the overall final design but was hard to implement as we built a variety of ad hoc message synchronization schemes and (over the years!) found holes in them.

We shall next discuss the architectures of the individual subsystems or components.

IV. HOST DESIGN

The design of the host software was driven by two major issues—the use of the IBM support software and the largeness of the data bases (several billion bytes on line). The decisions were as follows:

(i) Use PL/I and only simple features of IMS—avoid local optimization and use of complex features.

PL/I was chosen for its relative development efficiency. The other feasible alternatives of COBOL and Assembly were rejected because of their awkwardness or resulting code complexity. Further, as far as we could tell, we suffered little to no performance impact by choosing PL/I over Assembly.

The Information Management System (IMS) was used to create simple hierarchical data bases. We avoided using any new IMS programming feature for approximately two years after it was announced. By placing these restrictions on the use of IMS, we made sure that the feature had been debugged and we substantially improved the overall system field reliability and performance, as well as easing our internal training problems.

(ii) Data base size—big data bases have lots of inertia.

The projected size of the data bases (many billion bytes) resulted in two major design decisions. First, a series of inverted files were created from the line card file so that data could be indexed and retrieved other than by telephone number. Without these indices, the query of telephone numbers by cable pair, for example, would take hours. These files included a cable file, office equipment file, as well as three other small files. Each of the programs which updated the line card file also updated these inverted files. (A common, but poor practice of the computer science community at that time was allowing individual programmer access to the indices. Data base structure and indices should be hidden from the programmer by access routines.)

Given the need to reorganize data bases, take image copies, and recover them, care had to be taken to split a potentially large single data base into several logically smaller parts. While we recognized this for the line card data bases, unfortunately we did not for the cable data base. The cable data base became exceedingly large in one of the "rural" BOCs, which had extensive cable networks, and several 3350 disk drives were required. Reorganizing, recovering, and loading this one data base soon became the pacing item of the host computer operations.

(iii) Performance prediction—"off-line" not "on-line" will get you.

The performance prediction for the host computer was simple for the real-time day and surprisingly complex for the night time, off-line batch operations. The transaction processing in an IMS system inevitably is CPU limited. (The majority of the CPU cost is due to I/O processing; however, the CPU, not the channels or disks, saturates first.) Thus, all we had to do was to estimate the CPU utilization of the on-line transactions. Since we had the MLR system in New York for a model, we estimated the performance of the distributed system by removing the load of those transactions which would migrate to the front end. The only problem that we ran into was that as the system matured at a BOC, terminals accessing the data base for its review spread like wild fire. As experience with the system grew at BOCs, this extra use added an additional 20 percent to 30 percent load to the repair-only predicted load.

The nighttime load prediction had two problems. First, our MLR model experience was in Manhattan which was, at the time, having very little, if not negative, growth. As the system spread to the expanding areas, the need to do massive rearrangements of the data base for area transfers, major cable throws, etc., added what was to us an unexpected load. Most importantly however, we did not leave enough time for operator or machine error. In effect, we had planned

the operational evening too tightly. To meet the original load projections, we had to redo some of the programs so that they could run during the real-time day and in effect cut down the need for "off-line" time. The only real lesson here is that one really needs to understand all the environments of the end system, including the variants of the offered load and the realities of how big machines operate and are operated.

V. FRONT-END DESIGN

5.1 The operating system—there is no such thing as a simple operating system

The front-end system was to be, in effect, a transaction processor. The first design decision was related to whether we should use what at that time was a rather new operating system from Murray Hill—UNIX* program operating system—or build our own. Because of the need for a robust file system, high-performance interprocess communications, and the need to handle forty-eight 4.8-kb synchronous communications lines, we decided to build our own. It is still not clear whether this was our best or worst decision. It was best from the viewpoint that subsequently evolving the UNIX program operating system to handle the high-performance, high-availability, synchronous-terminal-driven, transaction-processing application has been most complex. It was the worst in that we might have been able to substantially modify the UNIX program operating system to satisfy our needs and in the process would have had earlier access to C language and the UNIX software development environment. (Two years into the development, we moved our development environment to the UNIX program operating system and started to use C as the programming language of choice. Later, (see Ref. 3), we decided to modify UNIX software, given our experience with the special-purpose operating system.) The other impact was that, as all operating system developers are, we were plagued with the never-ending minor utilities for operational and maintenance ease, e.g., the utility to do system file transfer.

The other major design features of the front end were all oriented towards high predictable performance and low mean-time-to-repair.

5.2 Design for availability—(It's easy if you include it in the design from the beginning.)

The low mean-time-to-repair was achieved by providing a standby

* UNIX is a trademark of Bell Laboratories.

processor, extra peripherals, and a rapid software recovery system. The recovery system was based on

(a) Using the communications terminals for message recovery—a message was processed in its entirety, one in and one out. The terminal held the input message until completion. If the system went down, the operator would re-enter the message.

(b) Keeping preupdate copies of a transaction's intended updates on disk to roll back the data base in case of error.

(c) Keeping a log tape of all before and after data base copies which were used to reconstruct the data base in case of a head crash.

This rather simple design resulted in a mean-time-to-repair of approximately two minutes for all but the head crash which would take approximately an hour. During the hour recovery, a simple back-up system was given to the user so that troubles and statuses could be entered for journaling on tape. This journaled information would then be read into the system when it was brought on-line so that the bureaus would not have to do any special manual catch-up work.

5.3 *Performance—(Keep the model simple and worry about it from the start.)*

The performance design was based on assuming the system would be designed to be single-threaded, i.e., once a transaction started, no other would run. This solved the data set locking problem, simplified recovery, and seemed like a reasonable approach given that at that time we only had two disks. Given this and using an application load model from the New York Telephone Company (which remained relatively invariant across other companies), we projected what the single thread load would be at peak busy hour. *Before announcing system capacity*, we designed each transaction and counted their disk accesses. Since the transactions were all simple, their elapsed time was directly related to the number of disk accesses. We then sized every other component of the system, the 4.8-kb lines, the 50-kb line, and the core buffers for holding transaction input, so that they would not be a limiting resource. Simple single server and multiserver queuing equations were used for the sizing.

The system algorithms were then designed to drive this single-server queue, i.e., the transaction stream. For example, the priorities for polling the terminals were such that once work came into the system, the polling priority would be lowered so that the transaction would run to completion. The major performance tuning was in handling the 50-kb line to the host and the communications buffers in the front end. These tuning needs were driven by all the idiosyncrasies of handling the bisynchronous protocol in an environment with noisy and failing lines.

VI. 11/34 MLT CONTROLLER DESIGN—Reexamine your architectural decisions under change.

The 11/34 MLT controller originally supported a special-purpose terminal which is no longer used in the system. This terminal, the Status Entry Device, was specially designed and built to be used to enter numeric status information into the system. Originally, it was used in the MLR system and was viewed by the computer as a TELETYPE CRT. On changing to the IBM 3270, we decided to use an 11/10 to simulate an IBM 3270 controller and to co-locate it with the Status Entry Devices. Once the 11/10 was at the bureau, it seemed natural to use it for controlling the loop tests. (The 11/10 was replaced later with the 11/34.) The building of the special terminals was a well-motivated detour. They were built because normal terminals were too big to use with the test desk and were too expensive. However, their production volume was never enough to allow cost reductions; thus, they did not track the cost reductions in the terminal field, but more importantly, the tester ultimately needed a full-feature terminal as the ARSB system grew in feature.

Once we decided to co-locate the 11/34 in the bureau, its design was actually very simple. For reliability, it had to have no moving peripherals. However, while it was appropriate to have a separate computer for driving the loop testing system, the 11/34 was probably not a good design choice. A better choice, both in cost and later development and deployment flexibility might have been to have a separate PDP 11/70 computer for testing as compared to several 11/34's. We recognized this too late in the development/deployment cycle.

VII. CROSS FRONT END—Geography is a bad division.

The cross front end was the last major addition to the system. Though we had recognized the potential need for such a system in the large cities, it represented a nicety and not a necessity. The design guidelines for the cross front end were identical to that for the MLT controller, with one addition—all cross front end tables had to be automatically derivable from the front ends. We could not stand the thought of the operator difficulty, then error, and then our repair, of having to keep the systems in constant synchronization. The performance approaches for the cross front end were almost identical to the front ends. The only exception was in assuring that one heavily loaded, or failing, front end did not use up all the buffers in the cross front end and, thus, jeopardize access to the other front ends.

Though the cross front end did the job, it was the early warning signal for the need to redo the ARSB architecture. Until the cross front end arrived, the system was completely hierarchical serving a smaller, self-contained geographic entity. The cross front end, in effect, re-

sponded to the desire of the BOCs to avoid the need to organize their operations according to this rather rigid computer-imposed geographic view. In several installations, the cross front end use grew beyond its original design intent of serving just the CRSABs to serving coin bureaus, and special business bureaus. This growing need led to the second architecture for the ARSB which is described in Ref. 3.

VIII. MAJOR LESSONS

Some of the lessons already described and one or two others deserve special mention as follows:

8.1 Time scales and tolerance to change

The LMOS system, from its inception to its full Bell System penetration, will take 13 years to complete. There is no way that any operations systems designer is 13 years smart. Thus, the early architecture must be very carefully examined to determine impact of BOC organizational change, changing data needs, etc. The issue of performance was very significant to the first generation of the ARSB. This issue of flexibility versus performance will have to be balanced exceedingly carefully in succeeding generations of the system.

8.2 Data view

The host system was designed from a functional view point, i.e., a conversion system, an automated line record update system, an on-line system etc. This view did not recognize that the most complex and unyielding issue was the large and unwieldy data base. Future systems should take a more data-centered view and build for example, a cable system, a customer service record system, etc. This should be done because it is hard to evolve mechanisms which allow easy minor changes of the basic view of the data base.

8.3 Data synchronization

The growing use of the LMOS data base resulted in our having to synchronize it with many other systems. We found this to be an almost impossible task unless the synchronization was done using self-contained groups of information, e.g., a full miniline record. Even then, minor differences in item definition (it took 6 to 9 months to resolve cable/pair status definition between two systems) would lead to terrible confusion. The only lesson is that if at all possible, do not duplicate data; however, if you have to, synchronize it in very big blocks.

8.4 Modularity—planned and achieved

Perhaps the major advantage of the distributed architecture was that it enforced modularity on the system. It seems that only physical

boundaries, either different machines or 16-bit address space or time slots, enforce that modularity after the early designers leave.

8.5 Models and pilot plants

The major lesson towards building a predictable architecture and system is to build a pilot plant first. Make that pilot plant exercise the technical risk and use it to gather data for the performance versus flexibility equation.

IX. ACKNOWLEDGMENTS

A system can have an architecture only after someone conceives the need for it. M. W. Bowker of Bell Laboratories deserves credit for that vision. Major contributors to the architecture were L. S. Dickert, J. M. Hunt, III, D. S. Watson, D. L. Kessell, S. G. Glover, D. Lloyd, S. Hensdale, and E. Mays.

REFERENCES

1. M. W. Bowker et al., "Automated Repair Service Bureau: Evolution," B.S.T.J., this issue.
2. J. P. Holtman, "Automated Repair Service Bureau: The Cross Front End: The Context-Sensitive Switch," B.S.T.J., this issue.
3. S. G. Chappell, F. H. Henig, and D. S. Watson, "Automated Repair Service Bureau: The Front-End System," B.S.T.J., this issue.
4. C. M. Franklin and J. F. Vogler, "Automated Repair Service Bureau: Data Base System," B.S.T.J., this issue.
5. O. B. Dale, T. W. Robinson, and E. J. Theriot, "Automated Repair Service Bureau: Mechanized Loop Testing Design," B.S.T.J., this issue.
6. S. P. Rhodes and L. S. Dickert, "Automated Repair Service Bureau: The Trouble Report Evaluation and Analysis Tool," B.S.T.J., this issue.