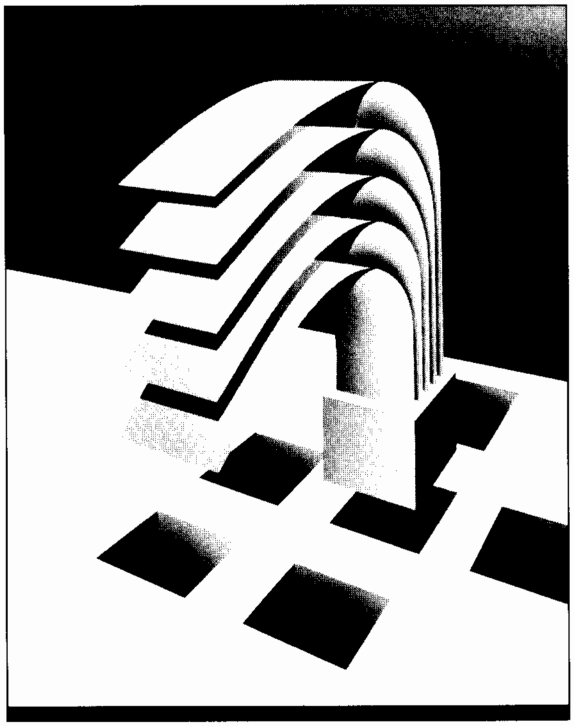
TANDEM SYSTEMS REVIEW

VOLUME 4, NUMBER 1

FEBRUARY 1988



TMF Autorollback:
A New Recovery Feature

The ENABLE Program Generator
for Multiple Applications

The PATHWAY TCP:
Performance and Tuning

The GUARDIAN Message System
and How to Design for It

Using FOX to Move
a Fault-tolerant Application

INTRODUCTION

This is the first issue of the *Tandem Systems Review*. The purpose of this publication is to provide programmer-analysts who use Tandem computer systems with useful technical information about Tandem's software releases and products. This information includes descriptions of product design, implementation, and performance; practical information to help users install, use, and tune products; and support information, such as software release plans, software manual information, and course offerings. From time to time, the *Review* may also contain technical customer profiles and articles on industry topics of relevance to users of Tandem systems.

Subscriptions to the *Tandem Systems Review* are free. Its publication schedule will be coordinated with Tandem product releases rather than following a regular quarterly schedule. Two issues are planned to support the B00 software release and new product releases in the first half of 1985. Volume 1, Number 2 is planned for April, and Volume 1, Number 3 for July.

I hope you'll find the *Tandem Systems Review* useful and interesting. Please send me your comments and suggestions.

Carolyn Turnbull White
Editor

TANDEM SYSTEMS REVIEW

Volume 1, Number 1, February 1985

Editor
Carolyn Turnbull White

Technical Advisor
Geary Arceneaux

Copy Editor
Sarah Rood

Art Director
Terri Hill

Designer
Joanne Danforth

Cover Art
Stephen Stavast
John Tomasini

The *Tandem Systems Review* is published by Tandem Computers Incorporated.

Purpose: The *Tandem Systems Review* publishes technical information about Tandem software releases and products. Its purpose is to help programmer-analysts who use our computer systems to plan for, install, use, and tune Tandem products.

Subscription additions and changes: Subscriptions are free. To add names or make corrections to the distribution data base, requests within the U.S. should be sent to Tandem Computers Incorporated, Sales Administration, 19191 Vallco Parkway, Cupertino, CA 95014. *Requests outside the U.S. should be sent to the local Tandem sales office.*

Comments: The editor welcomes suggestions for content and format. Please send them to the *Tandem Systems Review,* 1309 South Mary Avenue, Sunnyvale, CA 94087.

TMF Autorollback: A New Recovery Feature

Autorollback is a new feature of Tandem's Transaction Monitoring Facility (TMF™). It is used to restore logically inconsistent data files to their most recent consistent states after a TMF crash. It recovers the inconsistent files much more quickly than TMF rollforward. Autorollback has been available to a limited number of users since July 1984 and is available to all users of TMF as of the B00 software release. This article describes the concept, benefits, and limitations of autorollback.

Autorollback Features

Autorollback is available only on Tandem™ NonStop II™ and NonStop TXP™ systems. Its main features are listed below:

- Autorollback is initiated automatically and requires no operator intervention.
- It is significantly faster than rollforward.
- It improves TMF on-line performance.

Some users find the interaction between TMF and the operator too complex, and thus prone to error. Autorollback improves the user-friendliness of TMF by eliminating the need for operator intervention to start the recovery processes. Autorollback initiates recovery automatically when a volume containing logically inconsistent audited files is being enabled for TMF transaction processing (with the START TMF or ENABLE VOLUMES commands).

The time it takes autorollback to restore logically inconsistent data files to their most recent consistent states depends on several factors, such as the number of files to be recovered and the amount of system resources available. The time required by Autorollback to complete recovery, however, is much less than that required by rollforward.

The Rollback Concept

Before autorollback, TMF provided two forms of recovery: backout and rollforward. Backout reverses the effect of a single transaction in response to the ABORTTRANSACTION request. Rollforward restores from tape an old (and fuzzy) image of a data base and reapplies the after-images of all committed transactions.

Since the most recent on-line dump could be several days old, rollforward can be very time consuming. Ideally, it should be required only in the event of a media failure, something that is very rare in a Tandem system with mirrored discs. As the size of application data bases increased, a fast recovery mechanism became more and more important to the success of TMF. (Other vendors' products with functions similar to that of TMF, e.g., IBM's data-base management systems, IMS, SQL/DS, and DB2, already provided rollback.)

Rollback basically involves "undoing" (or backing out) all incomplete transactions at the time of a crash. In a 16-processor system, there are at most 2304 active (16 * 128 local + 256 network) transactions at any time. Therefore, rollback has to undo at most 2304 transactions to return a data base to its last consistent state. Rollforward, however, has to restore a copy of the data base from tape and then "redo" hundreds of thousands of transactions. It is because of this that autorollback is much faster than rollforward.

Autorollback could not have been implemented without a change to the disc process. Before autorollback, the disc process implemented what is known as the *write-through-cache algorithm*, in which an update to a file is immediately reflected in the file on disc.

Figure 1 illustrates the write-through-cache concept. The deletion of record R from the file on disc is part of the WRITE-UPDATE processing. After the file on disc has been updated, the before and after images for record R are still in the audit buffer and have not been written to the audit-trail file on disc. The audit for the update is only written to the audit-trail file on disc as part of ENDTRANSACTION processing or when the audit buffer becomes full. (The example in Figure 1 assumes that the audit buffer does not become full.)

Suppose a TMF crash occurred after record R had been deleted from the file on disc but before the audit buffer were written to disc. The file would then be logically inconsistent because the transaction would not have been committed and record R would have been deleted from the file. Worse yet, there would be no audit record indicating that record R had been deleted. The only way to restore the file to its most recent consistent state would be to perform a rollforward.

Even if the audit buffer had become full and had been written to disc before the TMF crash, rollforward would still be the only way to recover the file because there would be no guarantee that all audit buffers had been written to disc before the TMF crash.

To support autorollback, the disc process has been modified to implement a *write-in-cache* algorithm. In this algorithm, updates to a file are not written to disc immediately. Instead, the updated page of a file is kept in cache until the memory occupied by the page is required by another process. The write-in-cache concept is especially suitable for batch-oriented transactions, as several logical updates to the same data page result in only one physical update.

Figure 1

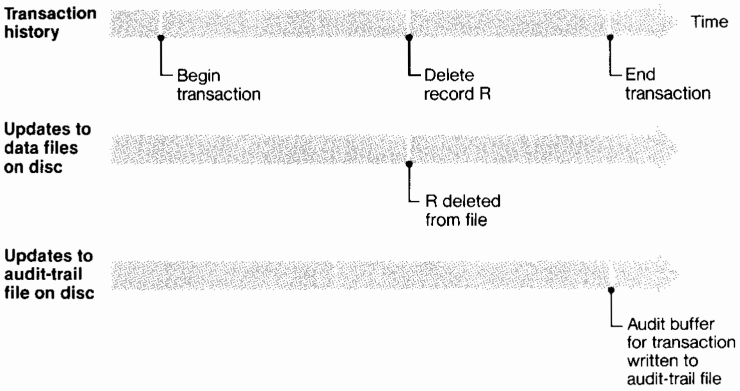

Transaction history
Time
Begin transaction
Delete record R
End transaction
Updates to data files on disc
R deleted from file
Updates to audit-trail file on disc
Audit buffer for transaction written to audit-trail file

Figure 1.
With the write-through-cache algorithm, an update to an audited file is reflected on disc immediately after the WRITE or WRITE-UPDATE request. The log for the update is added to the audit-trail file on disc at a later time.

Figure 2

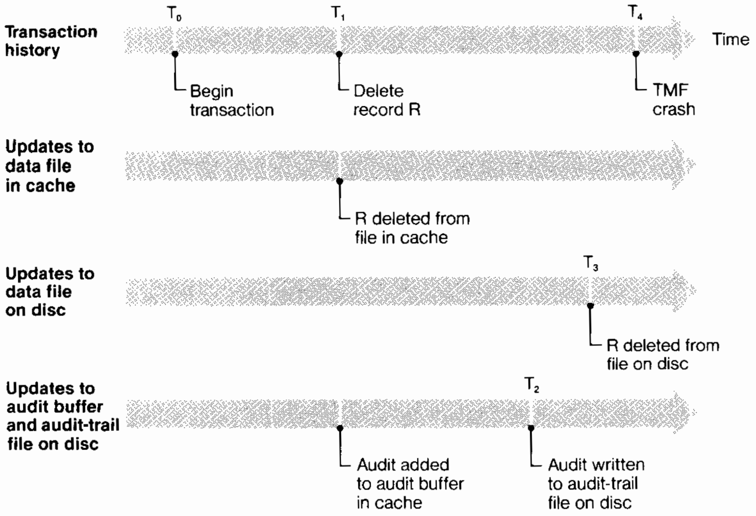

Transaction history
T_0
T_1
T_4
Time
Begin transaction
Delete record R
TMF crash
Updates to data file in cache
R deleted from file in cache
Updates to data file on disc
T_3
R deleted from file on disc
Updates to audit buffer and audit-trail file on disc
T_2
Audit added to audit buffer in cache
Audit written to audit-trail file on disc

Figure 3

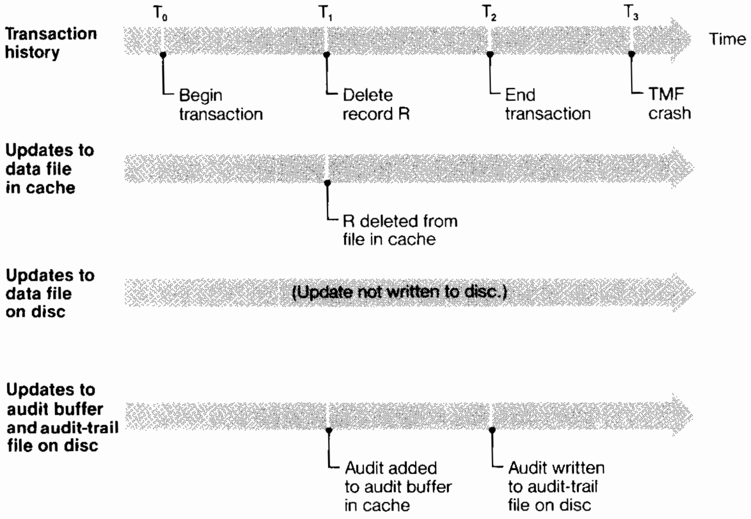

Transaction history
T_0
T_1
T_2
T_3
Time
Begin transaction
Delete record R
End transaction
TMF crash
Updates to data file in cache
R deleted from file in cache
Updates to data file on disc
(Update not written to disc.)
Updates to audit buffer and audit-trail file on disc
Audit added to audit buffer in cache
Audit written to audit-trail file on disc

Figure 2.
With the write-in-cache algorithm, an update to an audited file may not be reflected on disc even though the in-memory copy of the data page has been updated. The system determines when to update the data file on disc according to the write-ahead-log protocol and the utilization of its physical memory.

Figure 3.
If a TMF crash occurs and the update to an audited file for a committed transaction is not reflected on disc yet, the transaction can be redone because the log for the transaction is flushed to disc at ENDTRANSACTION.

The second major modification to the disc process for autorollback is the implementation of the *write-ahead-audit protocol.* Under this protocol, the disc process guarantees that the audit corresponding to an update is always written to disc before the updated data is reflected on disc. (Note that this protocol cannot be efficiently implemented with the write-through-cache algorithm.)

Figure 2 illustrates the processing involved with the write-in-cache algorithm and the write-ahead-audit protocol. At time T_1, record R is deleted from the file in cache in response to a WRITEUPATE request. The audit record for the delete is added to the audit buffer in cache.

Suppose, at T_2, the memory occupied by the page that used to contain record R were required by another process. The disc process would attempt to write the page to disc. Since the write-ahead-audit protocol dictates that the audit be written to disc before the data, the disc process would first write the audit buffer to disc. After the audit had been successfully written to disc, the disc process would then write the data to disc at T_3.

Notice that if a TMF crash occurred before T_2, the file would still be consistent, as the update would not be on disc. If a TMF crash occurred at T_4, TMF would have complete audit information for the file and would use this information to undo the incomplete transaction (in this example, to reinsert R).

Control Points

Suppose that the transaction described in Figure 2 had committed, but that the deletion of record R had not been written to disc when the TMF crash occurred (see Figure 3). Although the delete of record R would not be reflected on disc, the audit for the delete of record R would have been written to the audit-trail file on disc as part of ENDTRANSACTION processing. Since autorollback would have the audit information indicating that the transaction had committed and that record R should be deleted from the file, autorollback would redo the transaction to make the file consistent.

As illustrated in these examples, autorollback is a combination of a "mini-rollforward" and backout. Like backout, autorollback must completely undo all incomplete transactions. Like rollforward, autorollback must also redo all completed (both committed and aborted) transactions that have updates which may or may not be reflected on disc.

How does autorollback determine what transactions to redo? It limits the amount of "redoing" (and thus the time to recover logically inconsistent files) by requiring the disc process to periodically perform what is known as *control-point processing*. In a simple implementation of control-point processing, the disc process performs the following tasks every time a control point is required:

1. Suspends the processing of audited requests.
2. Flushes all the dirty audit buffers to disc (write-ahead-audit protocol).
3. Flushes all of its dirty data buffers in cache to disc.
4. Writes a control-point record into the data audit trail.
5. Resumes the processing of audited requests.

When it is time to perform rollback, autorollback begins the redo processing by reading the audit trails forward from the most recent control-point record. Autorollback does not need to reapply any after-images before this control point record since the disc process has already written them to disc during the control-point processing.

The algorithm described above works, but it has one major drawback. If the number of cache pages were very large and if they were all dirty, it would take a long time to write all of them to disc. When this happened, the system would periodically "hiccup", at which time the response time to an audited request would be quite long.

Figure 4

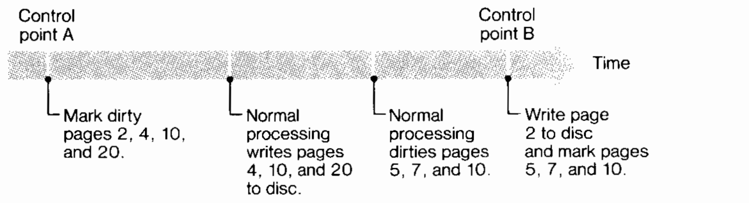

Control point A
Control point B
Time
Mark dirty pages 2, 4, 10, and 20.
Normal processing writes pages 4, 10, and 20 to disc.
Normal processing dirties pages 5, 7, and 10.
Write page 2 to disc and mark pages 5, 7, and 10.

Figure 4.
The two-phase control-point algorithm minimizes the number of dirty data pages that must be flushed to disc during each control point. For example, only one dirty data page has to be flushed at control point B, even though there are 4 dirty data pages in cache.

Autorollback solves this problem by implementing a *two-phase control-point algorithm*. This algorithm is as follows:

Phase 1

1. Look for dirty and "marked" data pages which have not been flushed since the last control point.
2. Flush all audit buffers to disc.
3. Flush all dirty and marked pages found in Step 1 to disc.

Phase 2

Mark all dirty data pages that are still in cache.

The purpose of the two-phase control-point algorithm is to minimize the number of marked and dirty data buffers that must be written to disc in Phase 1 of the algorithm (see Figure 4). In a balanced system, most of the dirty data buffers that are marked in Phase 2 should be written to disc as part of the normal processing between control points. In other words, the two-phase control-point algorithm guarantees that all dirty data buffers "older" than 2 control points have been written to disc.

Figure 5

File description

File name	Record size	Number of records	File type
ACCOUNT	100	1,000,000	Key-sequenced
BRANCH	100	18	Relative
TELLER	100	180	Relative
HISTORY	50	1 per transaction	Sequential

High-level description of the COBOL application program

```
Begin transaction.
   Read a 100-byte message from a teller terminal.
   Read the customer account from the Account File (random read).
   Update the customer account (random update).
   Write to the History File (sequential write).
   Read the teller record from the Teller File (random read).
   Update the Teller File (random update).
   Read the branch record from the Branch File (random read).
   Update the Branch File (random update).
   Write a 100-byte message to the teller terminal.
End transaction.
```

Figure 5.
A description of the files and application program used in the benchmark comparing TMF with autorollback against the A06 version of TMF.

Thus, autorollback begins the redo of committed transactions from the second-most-recent control point before the TMF crash. As in the simple implementation of control-point processing, the disc process for each audited volume periodically goes through the 2 phases of the algorithm.

The two-phase control-point algorithm in other data-base management systems has been observed to prevent periodic hiccups in the system.

The Benefits of Autorollback

Autorollback offers a number of benefits for a TMF installation. Its performance improvement over rollforward has already been discussed. Also, as autorollback does not require operator intervention, most human errors associated with rollforward are eliminated. It is now possible to have fast recovery at nodes that do not have a TMF operator.

Finally, it improves TMF performance. In a comparison of the performance of TMF with autorollback against the A06 verson of TMF, the following preliminary results were observed when an on-line transaction benchmark involving a bank-teller credit-debit application was run:

- The number of physical I/Os per transaction was reduced from 19 to 13.

- The transaction response time was reduced by 8.6% with a corresponding increase in transaction throughput.

The benchmark simulated the processing that occurs when a bank customer deposits or withdraws money from a teller. In Figure 5, the files and application program used in the benchmark are described. The performance gain for TMF with autorollback was the result of buffering updates to the Branch, Teller, and History files.[1]

The Cost of Autorollback

The conveniences and performance gains provided by autorollback are accompanied by some cost in TMF and disc-process resources. TMF may have to keep an audit-trail file on disc longer than it did in the A06 version. Previously, an audit-trail file was kept on disc until it had been dumped to tape and was no longer needed by the TMF backout process. With autorollback, an audit-trail file must be kept on disc as long as it is needed by autorollback.

[1] In the benchmark, all the files were kept on separate volumes. Since each current disc-process volume that participates in a transaction causes an audit flush at ENDTRANSACTION, the performance could have been better if the Teller and Branch files had been kept on the same volume. Furthermore, no attempt was made to balance the system when running the benchmark.

Also, audit-trail files may become full more quickly with autorollback than with A06 TMF for 2 reasons. First, each disc process periodically generates a control-point record of (12 + 4 * the number of active transactions) words. Second, autorollback audits backout. It does this because:

1. Autorollback is faster.

2. Potentially fewer audit-trail files have to be kept on disc.

To understand the value of auditing backout, consider the example shown in Figure 6. A transaction starts when the audit-trail file is AT000010. Sometime afterward, an ABORTTRANSACTION is issued, and the abort does not complete until the audit-trail file is AT000012. After 2 control point records have been written to the audit-trail file AT000012, TMF crashes.

When backout is audited, autorollback processing for the transaction shown in Figure 6 consists of reapplying its after-images found between control point 1 and the point at which the abort completed. Otherwise, autorollback would have to undo the transaction by reapplying all its before-images found in audit-trail files AT000010 through AT000012.

Furthermore, when backout is audited, audit-trail files AT000010 and AT000011 do not have to be kept on disc for autorollback. Only audit-trail file AT000012, containing the last two control points, need be kept on disc (assuming, of course, that all other active transactions at the time of the crash began in audit-trail file AT000012).

Even if the requirement for all audit-trail files needed by autorollback to reside on disc were relaxed, auditing backout would still allow faster recovery and less operator interaction (to mount tapes). The latter alone makes auditing backout worthwhile. Costs in disc-process resources result from the following:

- More memory is locked down for a longer period of time because of the write-in-cache algorithm.

- The disc process must periodically perform control-point processing.

- The backup disc process performs more work because of the increased amount of data checkpointed.

Figure 6

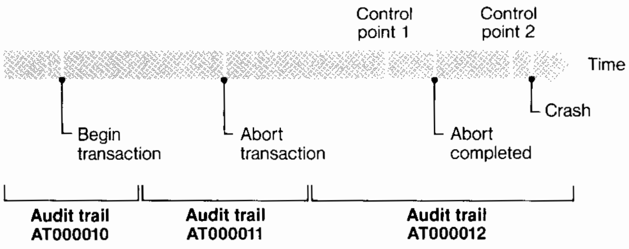


Figure 6.
Auditing backout eliminates the need to keep audit trails AT000010 and AT000011 on disc, as the effect of aborting the transaction can be achieved by redoing the transaction after the crash from control point 1.

The Limitations of Autorollback

Currently autorollback cannot recover key-sequenced and/or relative files that have become physically inconsistent as a result of a system crash in the middle of an index block split or delete. Autorollback detects the presence of such a file and flags it with "rollforward needed." The user must then use rollforward to recover the file. Tandem Software Development is working to eliminate this limitation in a future release.

References

Crus, D. 1984. Data Recovery in IBM Database 2. *IBM System Journal.* vol. 23, no. 2.

Gray, J. 1978. Notes on Data Base Operating Systems. In *Operating Systems, an Advanced Course*, Lecture Notes in Computer Science 60, eds. Goos and Hartmanis, pp. 393-481. Springer-Verlag.

SQL/Data Systems Planning and Administration - VSE. SH24- 5014-2. IBM Corporation.

Transaction Monitoring Facility (TMF) Reference Manual. Part No. 82341 B00. Tandem Computers Incorporated.

Acknowledgments
The author would like to thank Phil Garrett, Keith Hospers, Gary Tom, and Mike Treese, all of whom contributed their time and ideas to this article.

Michael Pong is currently a member of the High-level Data-base Group in Software Development. He joined Tandem in December 1981 to work on the TMF Recovery Project. Before joining Tandem, Michael was involved in the implementation of IBM's Database 2. He received a B.S. and M.S. in Electrical Engineering from Stanford University.

The ENABLE Program Generator for Multifile Applications

The ENABLE™ program generator can be used to create application programs from a high-level, nonprocedural specification. Applications generated by ENABLE can do the following:

1. Display data to the user on customized screens.
2. Perform record additions, deletions, and modifications.
3. Allow the user to browse through the data by generic and approximate key positioning.

This article highlights the features in the most recent release of ENABLE. It then summarizes how one interacts with a generated application and how one uses ENABLE to generate an application. A working example illustrates the generation process.

What Does ENABLE Do?

ENABLE generates applications designed for the PATHWAY™ transaction processing system. The user interacts with the generated application by entering data on the screen and pressing the terminal's function keys. Figure 1 illustrates a sample application screen.

The generated Screen COBOL application is run in the PATHWAY environment established by PATHCOM instructions that are also generated by ENABLE. Transactions requested by the user are passed to the ENABLE General Server.

In its recently released version, ENABLE is an order of magnitude more sophisticated than its earlier, single-file version.[1] The earlier version is upwardly compatible with the latest one: instructions from the earlier version generate the same application (with minor cosmetic differences) when given to the latest version.

The major enhancements in the latest version (as illustrated in Figures 1 and 3) include:

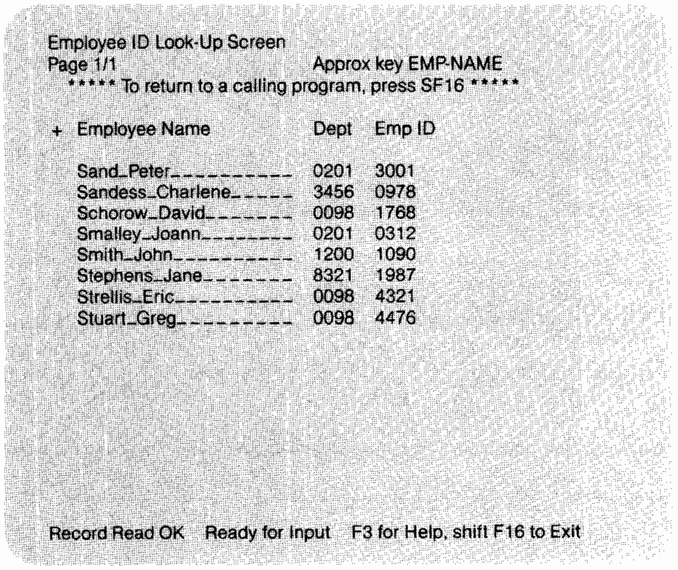
- Display and update of multiple records within a file.
- Columnar display format for multiple records within the same file.
- Selection of chosen fields and rearrangement of their layout on the screen.
- Access to multiple files per application.
- Links between records from different files.

Interacting with an Application Generated by ENABLE

To see records from an existing data base, one presses the key for READ FIRST, and the first 8 records from the data file are then displayed on the screen (as in Figure 1). To see the next 8 records, one presses the READ NEXT key. (These keys are documented in the *ENABLE Reference Manual* and also on the HELP screen, displayed when the HELP function key is pressed.)

[1] The latest version of the ENABLE Program Generator is product T9155. The earlier version is product T9105.

Figure 1



```
Employee ID Look-Up Screen
Page 1/1                         Approx key EMP-NAME
  ***** To return to a calling program, press SF16 *****

+  Employee Name               Dept   Emp ID

   Sand_Peter___________       0201   3001
   Sandess_Charlene_____       3456   0978
   Schorow_David________       0098   1768
   Smalley_Joann________       0201   0312
   Smith_John___________       1200   1090
   Stephens_Jane________       8321   1987
   Strellis_Eric________       0098   4321
   Stuart_Greg__________       0098   4476









Record Read OK   Ready for Input   F3 for Help, shift F16 to Exit
```

Figure 1.

A screen generated by ENABLE for the inspection, addition, modification, and deletion of up to 8 records from an employee file. New features include the display of more than one record from any file, columnar data display, and the selection of chosen fields and rearrangement of their layout on the screen.

After entering a value into a key field (e.g., an employee field), one can:

- Press a key to READ EXACT. This displays the record(s) in which the corresponding field has exactly the value entered.

- Press another key to READ APPROXIMATE. This displays records starting with the first one in which the corresponding field begins with the value entered.

- Press another key to READ GENERIC, and enter a number specifying how many characters at the start of the first value entered must be matched by the value in the corresponding field. This displays those records in which the value in the corresponding field exactly equals the entered value for the specified number of characters.

To add new records to the data base, one enters them individually on the screen, then adds them all in a single keystroke with INSERT BOX. (The term *box* is defined in a later section.) Similarly, one displays a screenful of records and deletes them all with DELETE BOX or modifies them and then changes the data base with UPDATE BOX. Keys to perform single-record INSERT, DELETE, and UPDATE are also available.

Generating an Application with ENABLE

Input to ENABLE is a high-level, nonprocedural specification of the capabilities desired for the application. The ENABLE instructions specify:

1. The screens to be displayed to the application user.
2. The transactions (inspections, additions, modifications, and deletions) to be performed on data-base records.

ENABLE starts with a Data Definition Language (DDL™) dictionary containing a description of each record type. From the dictionary, it generates:

- A Screen COBOL program that describes the screen display and the transactions that can be performed on records in the various files the generated application is to access. Typically, from ENABLE instructions of a dozen lines, a Screen COBOL program containing thousands of lines is generated.

- A file of PATHCOM instructions for configuring a PATHWAY environment in which the application is to run.

Thus, an ENABLE user can avoid writing any Screen COBOL code or PATHCOM instructions for those applications whose requirements can be met by ENABLE.

ENABLE comprises not only the generator of the above application components, but also the ENABLE General Server, a program that performs the selected operations. It is context-free, requiring each transaction request to include the name of the logical record against which the transaction is to be performed.

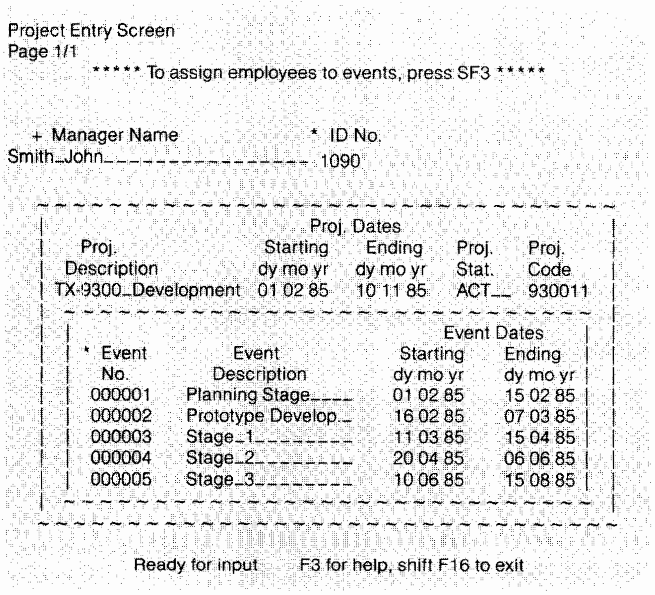
Figure 2

```
--   Identify the DDL record description
SET  RECORD dept-employees
--   Provide user information
SET  BOXTITLE 1 " ***** To return to a calling program, press SF16 *****
SET  BOXTITLE 2 " "
--   Provide a tabular format for the screen, including appropriate
--   screen field names
SET  SIZE 8
SET  SCREENFORMAT COMPRESSED
SET  HEADINGS NULL
SET  BOXTITLE 3 " +  Employee Name             Dept Emp ID"
--   Identify the order in which the fields are to appear on the screen
SET  INCLUDE (emp-name, emp-dept, emp-no)
ADD BOX employees

--   Provide a screen title
SET  TITLE "Employee ID Look-Up Screen"
--   Identify the file for the PATHCOM commands
SET  PATHCOMFILE prfile3 !
ADD APPL look-up

GENERATE APPL look-up
```

Figure 3



```
Project Entry Screen
Page 1/1
          ***** To assign employees to events, press SF3 *****

   + Manager Name                     *  ID No.
Smith_John________________  1090

   ~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~
   |                             Proj. Dates                   |
   |   Proj.                 Starting    Ending    Proj.   Proj. |
   |  Description            dy mo yr    dy mo yr  Stat.   Code  |
   | TX-9300_Development     01 02 85    10 11 85  ACT__   930011|
   |  ~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~   |
   |  |                                   Event Dates        |   |
   |  | * Event        Event          Starting    Ending     |   |
   |  |   No.       Description       dy mo yr    dy mo yr   |   |
   |  |  000001   Planning Stage____  01 02 85    15 02 85   |   |
   |  |  000002   Prototype Develop_  16 02 85    07 03 85   |   |
   |  |  000003   Stage_1_________    11 03 85    15 04 85   |   |
   |  |  000004   Stage_2_________    20 04 85    06 06 85   |   |
   |  |  000005   Stage_3_________    10 06 85    15 08 85   |   |
   |  ~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~   |
   ~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~

            Ready for input      F3 for help, shift F16 to exit
```

Figure 2.
Sample ENABLE commands for the screen in Figure 1. Note the commands BOXTITLE, SIZE, INCLUDE, and TITLE, which are included in the most recent release of ENABLE.

Figure 3.
Project Entry Screen, displaying data from the employee file, the project file, and the events file. A box outline distinguishes the data from each file. To establish such a multi-file application, the user defines a tree-like relationship between the data files.

There are 3 steps in generating an ENABLE application. The first is to define the record descriptions using DDL and to compile them into a dictionary. The second is to define the ENABLE commands for each application and compile them. The third is to create the supportive code necessary to integrate the individual applications. The code for the third step has 3 components:

1. The instructions for starting PATHMON or augmenting an existing PATHWAY environment.
2. A Screen COBOL program from which to branch to the individual programs that have been generated. (This avoids the need to return to PATHCOM to select a different program.)
3. Code for the generated Screen COBOL program that allows chaining between it and the generated programs. (This avoids the need to return to the menu or PATHCOM to change programs.)

A Sample Application Generated by ENABLE

The following is a description of a sample project-management application built with ENABLE. This application was developed to allow managers to track the projects under their control.

The program accesses an employee file and displays several employee records at one time. Figure 1 shows a screen displayed by one of the programs in this application. Figure 2 shows the ENABLE commands used to generate the program. These commands are defined below:

SET RECORD tells ENABLE the name of the record description for the employee file.

SET SIZE specifies the number of employee records to be displayed by the program.

SET SCREENFORMAT specifies the type of screen format desired.

SET INCLUDE specifies which fields from an employee record are to be displayed by the program.

SET PATHCOMFILE specifies the name of the file to which ENABLE is to direct PATHCOM instructions.

The program that produces the screen shown in Figure 1 can display several records in a single data-base file at the same time. ENABLE can also generate programs that access records from several data-base files. Figure 3 shows the screen displayed by a program that can be used to access information about a manager, the projects under that manager, and the events associated with those projects.

Notice the boxes displayed on the screen in Figure 3. Each box contains fields and/or records from a single data-base file. In ENABLE, the term *box* means a "window" on a data-base file. An ENABLE program can open the same file several times, with each box presenting a different perspective of the records in the file. For example, a program could open an employee file 3 times to access information about an employee, his or her manager, and the manager's manager.

Note also that in Figure 3, one box appears to be nested within the other. This reflects the hierarchical manner in which the ENABLE program accesses the data-base files. The user defines the hierarchical relationship by identifying the level of each box within the hierarchy and by identifying a field from each box that connects it to another box. The user describes the relationship by supplying a value with the TREE attribute of the SET command. For example, the screen in Figure 3 was generated as follows:

```
SET TREE (01 manager
   02 projects LINK emp-no TO OPTIONAL
      proj-mgr
   03 events LINK projects TO OPTIONAL
      events VIA proj-code)
```

Note that the LINK OPTIONAL clause in ENABLE serves the same purpose as the LINK OPTIONAL statement in ENFORM.

Chaining Between Programs

Users who have generated several related programs may want to connect or *chain* between these programs to enhance their usefulness. ENABLE facilitates this by providing a special area of code in a generated program. To chain between programs, one requests Screen COBOL source code when generating an ENABLE program, makes simple modifications to the source code, and recompiles the program.

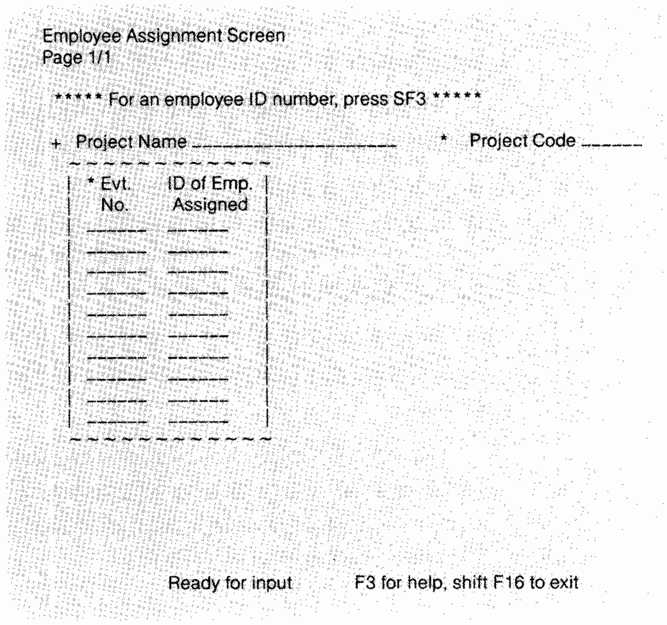
For example, modifications have been made to the source code of several of the programs in the sample project-management application. These modifications allow the user to call the Employee Assignment Program (see Figure 4) from the Project Entry Program (Figure 3).

The Complete Application

The complete project-entry application consists of several programs generated by ENABLE that are integrated into a single application via a user-written menu program.

Figure 4.
The Employee Assignment Screen, to which the user can chain from the Project Entry Screen shown in Figure 3.

Figure 4



```
Employee Assignment Screen
Page 1/1

 ***** For an employee ID number, press SF3 *****

 +  Project Name ____________________     *  Project Code ______
   ~~~~~~~~~~~~
   | * Evt.    ID of Emp. |
   |   No.     Assigned   |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   |  ______   ______     |
   ~~~~~~~~~~~~

               Ready for input        F3 for help, shift F16 to exit
```

Figure 5.

The structure of the sample project management application, showing the 6 application programs generated by ENABLE and the user-written program (the Project Tracking Menu). The screens from Figures 1, 3, and 4 correspond to the Look-up, Project Entry, and Employee Assign screens.

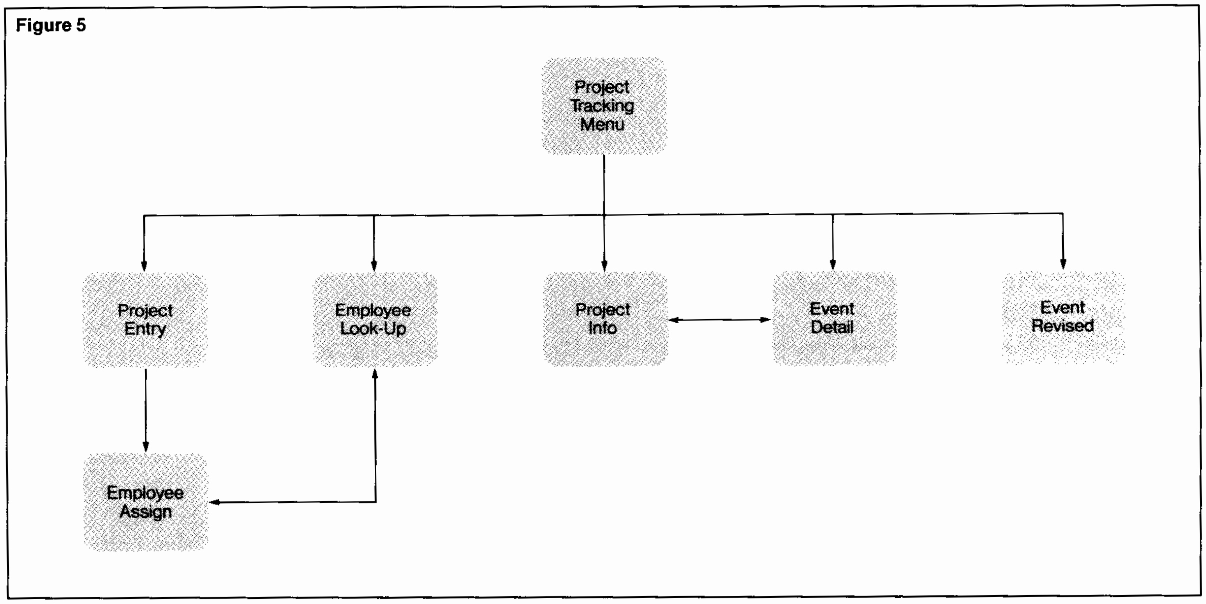


Figure 5 diagrams the structure of this application. The application consists of 6 programs:

▪ *Project Entry* allows users to enter, maintain, and display information about projects and their events.

▪ *Employee Assign* allows users to assign an employee to a particular event within a project.

▪ *Employee Look-up* allows users to obtain the identification number of each employee.

▪ *Project Info* allows users to obtain detailed information about each project.

▪ *Event Detail* allows users to obtain detailed information about the milestones associated with each event and the employees assigned to each event.

▪ *Event Revised* allows employees to revise milestones associated with their assigned events.

The *ENABLE User's Guide* contains more information about this application. The sample program is also available on tape through Tandem analysts.

The Advantages of Using ENABLE

In addition to generating data-base editing applications such as that described above, ENABLE is excellent for generating the following:

▪ *Prototypes of production applications.* User requirements can be transformed into a working prototype that can be used to further define the problem an application is to solve.

▪ *Test-data generators.* ENABLE can generate a data-entry application for entering test data into a sample data base. The data can then be used in testing other developing applications.

▪ *Reference-file maintenance.* Such files contain relatively static but widely used information.

▪ *Multifile data-base maintenance.* ENABLE can generate applications for correcting errors in production files, freeing programming resources for more sophisticated tasks.

The first use listed above, application prototyping, is becoming increasingly popular, for the following reasons:

- Users become involved in the application design from the earliest stages of the project, and therefore identify with the project. Their enthusiasm contributes significantly to the success of the implementation.

- Users of a new software application are often unable to picture how it will work and what its interface to other applications will be like. A prototype is a tangible, working application for users to test.

- Prototypes are quickly developed and easily modified to conform to users' requests and system changes. As a result, users regard the prototype, and hence the application, as friendly and flexible.

- Users may be completely satisfied with the prototype, so that no further development is required.

Enhancing Programs Generated by ENABLE

With ENABLE, basic programs can be generated by users inexperienced in application design and development. Sometimes, however, an application must accomplish more than those generated by ENABLE, and the Screen COBOL code generated by ENABLE can be modified to accommodate this. For example, an application prototype generated by ENABLE could be modified to satisfy the following application requirements:

- One application program must call another.

- A menu must relate a set of ENABLE applications.

- Integrity constraints must be followed, e.g., the insertion, deletion, or modification of a record in one file must trigger associated changes to the data base.

- The user must be able to alter the placement of displayed text, box borders, or data fields.

Modifying one generated application to call another is straightforward. The changes for the other needs require more substantial effort, but they can be accomplished far more readily by modifying the generated Screen COBOL source than creating the program afresh.

Conclusion

The ENABLE program generator is a powerful productivity tool within the ENCOMPASS™ distributed data-base system. It produces application programs that can access many related data files simultaneously, and it provides users with considerable control in specifying screen formats.

References

Boar, B.H. 1984. *Application Prototyping*. New York: John Wiley and Sons.

ENABLE Reference Manual. Part No. 82360 B00. Tandem Computers Incorporated.

ENABLE User's Guide. Part No. 82361 B00. Tandem Computers Incorporated.

Acknowledgments

The authors are grateful to Kay Carlyle, Ernie Chilberg, Deborah Evelyn, Jim Gray, Rob Holbrook, and Howard Moehrke for reviewing versions of this article. Without their time and energy, this article would have been written sooner, but would probably not have been published.

Bobbie Chapman joined Tandem as a technical writer, documenting high-level data-base products. In 1983, she wrote an update to the *ENABLE User's Guide* for the T9105 version of the product. Early in 1984, she began work on the *ENABLE Reference Manual* and the *ENABLE User's Guide* for the most recent version of the product. In April, she transferred to Software Quality Assurance where she tests ENABLE, DDL, and portions of the PATHWAY transaction processing system.

Joan Zimmerman joined Tandem's Software Education Group in May 1978. She wrote a course on MUMPS and enhanced a course on the ENFORM relational query language, adding sections on unusual ENFORM reports and ways to enhance query performance. In January 1982, she joined Software Quality Assurance, where she developed test libraries for ENFORM, ENABLE (including the new extended version), the TRANSFER delivery system, and the Data Definition Language (DDL). Before joining Tandem, Joan developed computer-aided instruction and medical-record applications at Washington University, St. Louis, MO.

The PATHWAY TCP: Performance and Tuning

In the E07 version of the PATHWAY transaction processing system, a new version of the Terminal Control Process (referred to as TCP2) was introduced. Before TCP2, the PATHWAY TCP performed a number of disc I/O operations for checkpointing, terminal-context swapping, and fetching Screen COBOL pseudocode. TCP2 reduces disc I/O operations in the TCP by using an extended data segment to store terminal context and Screen COBOL pseudocode. This reduction has resulted in significant performance improvement for applications based on the PATHWAY system.

In the Spring 1984 issue of the *Tandem Journal*, the article "A New Design for the PATHWAY TCP" described TCP2's design and general performance implications. This article provides the following additional information about the performance of the TCP:

1. Benchmark comparisons of TCP1 and TCP2.
2. Discussion of the components that contribute to the performance improvements.
3. Guidelines for tuning applications that use TCP2.
4. Discussion of the costs of TCP functions.

The Performance Comparison

Benchmarks were performed to compare the performance of TCP1 and TCP2, to further understand the performance characteristics of TCP2, and to identify improvements to be made in the future. The objectives of the benchmarks were as follows:

- To measure the performance of the TCP within a complete application, not as a stand-alone program.
- To measure and compare performance in "bottom-line" terms of response time and throughput, not in terms of CPU milliseconds or physical disc I/O operations.

Tests were performed using both TCPs, at several transaction throughput rates for each. Each version of the TCP was tuned to favor its characteristics. For example, more TCPs were configured for TCP1 than for TCP2, so that no context swapping occurred in TCP1.

The application, software environment, and hardware environment for the TCP benchmarks are summarized in Figure 1. As benchmark results are dependent on the characteristics of the application used, the TCP benchmarks should be evaluated within their application context.

Benchmark Results

The percentage of improvement in throughput for TCP2 over TCP1 at various response times is shown in Table 1. Figure 2 shows the same comparison graphically.

While average response time is an important indicator of performance, it does not reflect the range of response times in a benchmark. For example, if the average response time is 2 seconds but a number of transactions took more than 30 seconds, the performance is not acceptable.

Table 1.
A throughput comparison of TCP1 and TCP2 for average response times.

Average response time (seconds)	Percentage of improvement in throughput for TCP2 over TCP1
1.5	85%
2.0	30%
2.5	19%
3.0	20%
3.5	20%

Figure 2

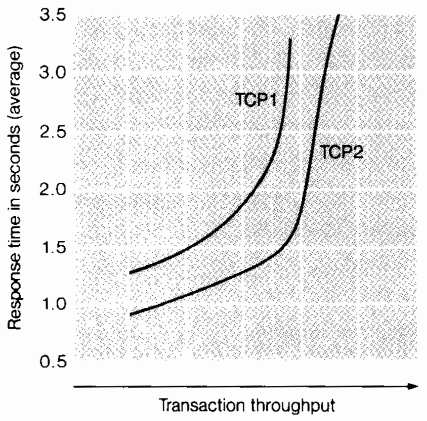


Figure 1.
The benchmark environment used to compare TCP1 with TCP2.

Figure 2.
A throughput comparison of TCP1 and TCP2 for average response times.

Figure 1

Hardware environment

6 NonStop II CPUs
2Mb of memory in each CPU
6 mirrored-disc volumes
180 terminals (simulated with ENCORE™)

Software environment

TCP1	TCP2
A05 version of the GUARDIAN™ operating system	A06 version of the GUARDIAN operating system
24 TCPs (4 per CPU)	6 TCPs (1 per CPU)
24 to 48 servers	6 to 30 servers

The number of server processes varied with throughput. More TCP1s were configured to avoid context swapping.

Software used for both TCP1 and TCP2 included:
NonStop TCPs
The Transaction Monitoring Facility (TMF™)
The ENCORE stress test generator (for terminal simulation)
COBOL servers

Application data base

	Number of records	File type	Record size
FILEA	1,000,000	Key-sequenced	100
FILEB	180	Relative	100
FILEC	18	Relative	100
LOG	1 per transaction	Entry-sequenced	50

Assignments of files to disc volumes:
FILEA was partitioned on 2 volumes.
FILEB and FILEC shared a volume.
LOG was dedicated to a volume.
The other two volumes contained the TMF monitor and data audit trails.

Application server transaction flow

7 logical I/O operations (3 reads and 4 updates or writes audited by TMF), as follows:

READ 100-byte message
READ and UPDATE FILEA
READ and UPDATE FILEB
READ and UPDATE FILEC
WRITE LOG
REPLY with 100-byte message

Because of the large size of FILEA, each read required 2 physical reads of an index and data block.

Because of the small sizes of FILEB and FILEC, reads to these files were always satisfied from disc cache.

Application requester (Screen COBOL) transaction flow

ACCEPT ten 10-byte fields
PERFORM DEPENDING ON function-key (always F1)
BEGIN TRANSACTION
SEND 100 bytes with 100-byte reply
10 MOVE statements
5 ADD statements
5 SUBTRACT statements
10 IF statements
END TRANSACTION
DISPLAY ten 10-byte fields

The terminal type was T16-6530.
The terminal context size was 6000 bytes.

A second, more demanding performance criterion is one referred to as *90th-percentile response time.* In this criterion, 90% of the transactions for each response time represented were completed in that amount of time or less. Table 2 compares the 90th-percentile response-time results for TCP1 and TCP2. Figure 3 shows the same comparison graphically.

Analyzing the Results

The percentage of performance improvement for TCP2 over TCP1 is very high at low response times, decreasing as the response time is increased. There are 2 reasons for this.

Table 2.
A throughput comparison of TCP1 and TCP2 for 90th-percentile response times.

90th-percentile response time (seconds)	Percentage of improvement in throughput for TCP2 over TCP1
1.5	184%
2.0	75%
2.5	30%
3.0	18%
3.5	17%

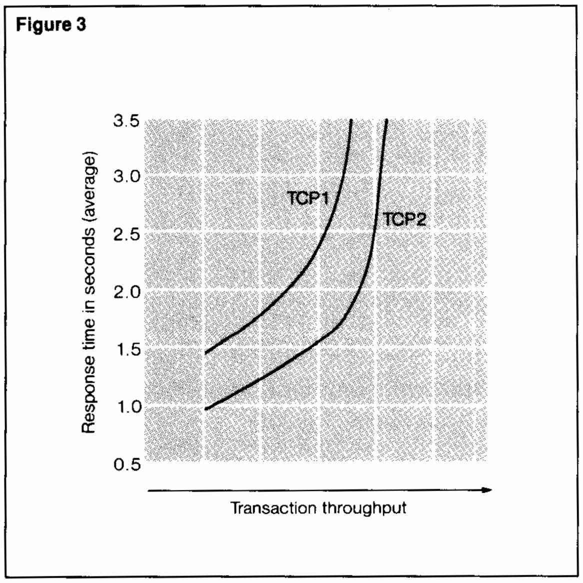


Figure 3.
A throughput comparison of TCP1 and TCP2 for 90th-percentile response times.

First, at lower response times, TCP1 approaches its minimum possible response time. Because TCP1 consumes more CPU cycles and disc-device time than TCP2, its minimum response time is higher. To achieve lower response times for TCP1, throughput must be decreased to reduce contention for resources among concurrent transactions.

Second, for higher response times, both TCPs approach a throughput close to the maximum for this application, in which the maximum throughput is determined not by TCP demand but by disc demand. For these benchmarks, the disc volume containing FILEB and FILEC was the bottleneck.

In any case, TCP2 performs noticeably better at any throughput or response time. The difference is more dramatic when one considers that the majority of the CPU and disc demand for this application was independent of the TCP used for the comparison. For the TCP2 tests, the TCP consumed only about 27% of the CPU demand and none of the disc demand. For the TCP1 tests, the TCP consumed about 35% of the CPU demand and 48% of the disc demand.

TCP2 reduced CPU demand by 170 ms per transaction, and it reduced disc-device demand by 280 ms per transaction. These reductions were primarily due to the elimination of disc I/O operations in TCP2 and resulted in its increased throughput and lower response times.

How Much Will an Application's Performance Improve with TCP2?

Several major factors, discussed below, affect an application's performance with TCP2.

- *Context size.* The larger the context size, the greater the improvement with TCP2. TCP2 is relatively insensitive to context size when performing checkpoints. For TCP1, every 4K bytes of context produces an additional disc I/O operation for each checkpoint.

- *Amount of TCP data and code swapping.* The more TCP data and code swapping, the greater the performance improvement with TCP2. TCP2 does no data swapping, and code swapping can be eliminated if a large enough pseudocode area is configured.

- *Complexity of the data base.* The simpler the data-base portion of the application, the bigger the increase for TCP2. If an application does 50 disc-file updates on every transaction, the relative cost of the TCP is insignificant in the whole transaction, so a 50% improvement in the TCP portion may only create a 5% improvement in total throughput.

- *Percentage of updates to inquiries.* The greater the percentage of updates to inquiries, the greater the improvement for TCP2. The TCP2 improvements reduce dramatically the cost of checkpointing. If an application performs relatively few checkpoints or little context swapping, the performance of TCP1 and TCP2 is similar. For those applications based on the PATHWAY system that use TMF, inquiry transactions perform no checkpointing in the TCP, whereas update transactions perform 2 terminal context checkpoints, at BEGINTRANSACTION and ENDTRANSACTION.

- *Amount of memory versus number of disc devices.* TCP2 uses more physical memory than TCP1, while TCP1 creates more disc demand than TCP2. The cost of the additional memory for TCP2 is less than the cost of the additional disc devices for TCP1; however, if the hardware configuration is not changed, the performance difference depends on how "tight" the configuration is on memory versus disc devices. The reduced sensitivity to context size and the reduced data and code swapping of TCP2 mentioned above rely on sufficient physical memory.

Performance improvements will continue to be made to TCP2, increasing the relative improvement of its performance over that of TCP1.

Tuning Applications that Use TCP2

Number of Terminals per TCP

An important decision to be made whenever TCP1 was used concerned how many terminals to configure per TCP and, thus, how many TCPs to configure. Because TCP1 had a limited area for terminal context, if too many terminals per TCP were configured, significant TCP context swapping would occur as the terminals contended for context space. TCP2 does no context swapping, as each terminal has its own context area in extended memory.

In the benchmarks, 24 TCPs were configured with TCP1, each with 7 or 8 terminals. This was necessary to avoid context swapping. For TCP2, 6 TCPs were configured, each with 30 terminals, so that each CPU had one primary TCP process.

New users of TCP2 often ask the following questions about the number of terminals to configure:

1. Now that TCP2 has removed the context area limitation, are there other limitations on the number of terminals per TCP?
2. How many terminals per TCP should be configured?
3. Is it best to run only one TCP per CPU?

The answers can be found in the considerations discussed below.

- *Control-block Memory Limitations.* TCP2 is theoretically designed for up to 255 terminals per TCP. A limiting factor on the number of terminals per TCP for TCP2, however, is control-data space for terminals, server classes, server processes, and external PATHMONs. A second limiting factor is the size of the TCP's SERVERPOOL. The actual limit is very dependent on the application configuration. Memory size limits the number of terminals per TCP2 to no more than 100; however, other factors such as manageability, discussed below, make one-third to one-half that number desirable.

- *TERMPOOL and SERVERPOOL Contention.* Since terminal context and the code area have been moved to an extended segment, more space is available in the user data segment for TERMPOOL and SERVERPOOL. With a large number of terminals per TCP, however, contention for these areas is still a limiting factor.

- *System Management.* With TCP1, the primary consideration was improving transaction performance by reducing context swapping. With TCP2, other considerations, such as system management and recovery, become important. If a TCP has 100 terminals, and it fails or it must be stopped to be reconfigured and then restarted, 100 terminals must be stopped and restarted. Therefore, even though 100 terminals could be configured, the recommended maximum number of terminals per TCP is between 30 and 50.

- *CPU Contention.* A common misconception is that multiple TCPs in the same CPU increase CPU demand due to additional process dispatching. In fact, measurements have shown a slight decrease in CPU demand with multiple TCPs as compared to that of a single TCP per CPU, when the total number of terminals remains constant. The decrease is due to the efficiencies in the file I/O achieved by the GUARDIAN operating system when less terminals are open in each TCP.

- *Memory Consumption.* The memory required for terminal context is dependent only on the number of TCP terminals in each CPU. Each additional TCP has its own copy of some global data, however, and each TCP has its own copy of the Screen COBOL pseudocode area. The code area required varies for each application.

In summary, TCP2 allows the user to configure many more terminals per TCP than TCP1 did, resulting in less TCPs running in the system. Instead of configuring the most terminals per TCP possible, however, the user should heed the considerations discussed above. A key consideration is that of system management: picking a manageable number of terminals per TCP and a manageable number of TCPs. For the benchmarks, one TCP per CPU was configured, each with 30 terminals. If the benchmarks had required twice the number of terminals, two TCPs per CPU would have been configured, still with 30 terminals per TCP.

Memory Balancing

Since TCP2 uses more memory than TCP1, memory balancing is important with TCP2. When NonStop TCPs are run, the primary TCP and backup TCP both have an extended segment, and both contain copies of the terminal context for each terminal. Since the backup TCPs have a copy of terminal context, these processes should be evenly spread across available CPUs to balance memory utilization, even though the CPU utilization for backup TCPs is relatively small. Although both TCP processes have a copy of terminal context, the primary TCP uses a code segment for Screen COBOL pseudocode, while the backup TCP does not.

TCP2 statistics no longer indicate the memory pressure for the terminal context area because the memory is managed by the GUARDIAN operating system. For monitoring memory pressure, XRAY must be used.

Ideal performance is attained with TCP2 when enough physical memory exists for all terminal context and Screen COBOL pseudocode. With some memory pressure, TCP2 still outperforms TCP1; however, as is true of most performance behavior, response time with TCP2 follows an exponential curve as utilization increases. Thus, with too much memory pressure, response time degrades significantly.

In the benchmarks, there was effectively no memory swapping for the TCP2 tests. Memory was balanced by evenly spreading all processes across the available CPUs, and the system was configured with discs and other devices also balanced across the CPUs. When a system is being balanced, memory allocation in each CPU for configured devices should be checked on the SYSGEN output.

TCP configuration parameters that affect memory balancing include the TCP SWAP parameter and the TCP CODEAREALEN parameter. The TCP SWAP parameter for TCP2 defines the disc location of the GUARDIAN swap file for extended segments. These swap files should be evenly spread across discs, just as the swap-file locations for TCP1 were balanced.

The TCP CODEAREALEN is a new parameter that defines the amount of space within the extended segment used for Screen COBOL pseudocode storage. Making the code area larger reduces the number of disc I/O operations needed to load Screen COBOL programs for repeated calls of the same program unit. Making it very large does not hurt, since GUARDIAN tends to swap out pages containing seldomly used Screen COBOL programs before swapping out memory pages referenced more often.

The Cost of TCP Functions

Performance questions often asked about TCP2 include:

1. Should NonStop TCPs be run?
2. How important is reducing the size of terminal context in designing Screen COBOL requesters?
3. Does running the TCPs with the STATISTICS option on affect performance?
4. When should tests or calculations be performed in the TCP or in a server?
5. What changes can be made to Screen COBOL requesters to improve performance?

The following is a discussion of each topic.

Running NonStop TCPs

With TCP1, a NonStop TCP could be expensive. Typically, each transaction requires two checkpoints at BEGINTRANSACTION and ENDTRANSACTION, or without TMF, before and after each SEND. With TCP1, for every 4K bytes of context, a disc I/O operation to the TCP swap file was required. For example, with the 6K context used in the benchmarks, 4 TCP swap disc I/O operations per transaction were required. At each of two checkpoints, a 4K and a 2K block were written.

The cost of NonStop TCP operation is significantly reduced with TCP2. Only a single interprocess message per checkpoint from the primary TCP to the backup TCP is required. For typical transactions the cost is easily less than 10% of the total CPU demand.

> *Select a manageable number of terminals per TCP and a manageable number of TCPs.*

Reducing the Size of Terminal Context

With TCP1, large context degraded performance in two ways. First, if the context of all terminals did not fit within the TCP's context area, context was swapped to the TCP swap file. Users could avoid the swapping by running more TCPs; for example, in the benchmarks, 24 TCP1s were configured, each with 7 or 8 terminals. Second, during checkpointing, each 4K of terminal context required a disc I/O operation to the TCP swap file.

TCP2 is much less sensitive to context size since a single interprocess message is sent, regardless of the size of the checkpoint. For example, an increase from 2K bytes to 6K bytes of terminal context with TCP2 increased CPU demand by less than 3% for the benchmark application.

Note, however, that terminal context does consume memory. Since TCP2 maintains multiple copies of the context, with a large number of terminals, a significant increase in terminal context could affect physical memory requirements. In any case, the cost of the increased physical memory for TCP2 would be less than the cost of additional disc drives required to support the TCP swap file I/O with TCP1.

Performance with the TCP STATISTICS Option On

Measurements of the TCP with and without statistics show a cost of about 3 ms per transaction to run with statistics, or less than 0.5% of the transaction CPU demand.

Performing Tests or Calculations in the TCP or Server

There are many reasons independent of performance for selecting the best place to perform tests or calculations. These reasons are based on ease of application design and maintenance. In some cases, however, performance considerations may affect the decision.

When the application is already sending to a server, calculations by the server in COBOL are much faster than those executed in Screen COBOL. Relative speed needs to be kept in perspective, however. For example, in the benchmarks, the 30 Screen COBOL statements (10 MOVEs, 10 IFs, 5 ADDs, and 5 SUBTRACTs) consumed about 1% of the total CPU demand of the transaction. A common mistake in measuring the performance of Screen COBOL, COBOL, and TAL™ is to compare their raw calculation speeds. In most applications, the total cost of disc-file I/O, interprocess messages, and data communications far outweighs the cost of the calculations in any language.

In some instances, applications make additional SENDs to servers solely for calculations or data manipulation. With TCP2, about 60 simple operations can be performed in Screen COBOL for the same cost as a SEND to a server. A simple operation is defined as a MOVE, IF, ADD, SUBTRACT, MULTIPLY, or DIVIDE. If indexing or subscripting is involved, each indexed reference can be considered an additional operation. If the SEND requires checkpointing, reformatting of data, or other pre- and postprocessing, the ratio increases.

In future releases, performance enhancements are expected for both SEND and simple operations. More improvement should be seen in simple operations, causing the ratio of simple operations to a SEND to increase.

Changes to Screen COBOL Requesters to Improve Performance

The objective of Tandem's PATHWAY development group is to make performance a secondary consideration in the design of applications based on the PATHWAY system. The primary considerations should be clean design and ease of maintenance. If performance is an issue, however, the following information about Screen COBOL requesters should be helpful to designers:

1. TCP2 is less sensitive to context size, as was discussed earlier.
2. A few simple MOVEs, IFs, or arithmetic calculations do not noticeably affect total throughput or response time; however, massive table searches will.
3. For DISPLAY, ACCEPT, and SEND statements, the factor that affects performance most is not the number of bytes transferred, but the number of fields transferred. For each field in a block-mode terminal DISPLAY or ACCEPT, the TCP must reformat and move data from working storage to the screen, and decode or format attribute bytes and terminal-buffer addresses.

For SENDs, a typical technique is to build the server message directly from the screen description using the TO clause. Then the server message is sent as a single Level 01 message rather than as a list of 10 to 20 fields. This technique reduces the field-level moves and data reformatting.

4. Simple Screen COBOL PERFORM statements are very inexpensive, about the cost of a Screen COBOL MOVE statement. Screen COBOL CALLs, assuming that the program called has already been referenced once and is in memory, are about one-fifth the cost of a single SEND, ACCEPT, or DISPLAY. With these ratios in mind, users should structure their Screen COBOL programs for ease of maintenance and design.

PATHWAY Performance in Perspective

Significant improvements in the performance of the PATHWAY transaction processing system have been made, and more are in progress.

In the E06 version of the PATHWAY system, the CHECK-DIRECTORY and REFRESH-CODE options were added to eliminate reads of the Screen COBOL pseudocode directory file during Screen COBOL program CALLs. For production environments, this eliminated one logical disc I/O for every Screen COBOL CALL. Also in the E06 version, the TCP swap file block size was increased from 2K to 4K bytes.

In the E07 version, TCP2 with extended memory was introduced, eliminating all I/O to the TCP swap file. Enhancements to the TCP's dispatching algorithm and enhancements to the GUARDIAN operating system for NOWAITed I/O further reduced CPU consumption by the TCP.

Most of the past improvements were aimed at reducing disc I/O in the TCP, which was the major cost of TCP functions.

Future performance enhancements will focus on 3 areas:

1. Streamlining basic TCP functions common to all applications, i.e., SEND, terminal I/O, TMF BEGINTRANSACTION and ENDTRANSACTION, and multithreaded operation.
2. Speeding up simple Screen COBOL operations such as MOVE, IF, and arithmetic calculations.
3. Reducing the cost of screen formatting.

References

Wong, R. 1984. A New Design for the PATHWAY TCP. *Tandem Journal.* vol. 2, no. 2.

Joel Vatz is a software developer for the PATHWAY Transaction Processing System, working primarily on the Terminal Control Process (TCP). Since joining Tandem in 1981, he has also been a systems analyst and analyst manager, involved in benchmarks and performance studies for several major customer projects.

The GUARDIAN Message System and How to Design for It

In order to guarantee that all processors in a Tandem system have the same degree of accessibility to resources, regardless of their location, the GUARDIAN operating system defines and implements a strict message-exchange protocol with its Message System. Almost all information transfer, even within a single processor, is via messages rather than shared data structures. The Message System plays a key role in providing Tandem users with the ability to incorporate features such as communications homogeneity, location transparency, geographic independence, and modular expandability into their applications.

This article describes the user interface to the Message System, the message-exchange protocol itself, and the inherent advantages of its implementation. It concludes with suggestions for designing applications that take advantage of the features in the Tandem architecture. Systems analysts who design applications for the Tandem system and those who tune and balance Tandem systems should find this information helpful.

An Introduction to the GUARDIAN Operating System

The GUARDIAN operating system (for the Tandem NonStop™, NonStop II, and NonStop TXP systems) provides all the standard services available with modern operating systems: virtual memory management, resource allocation, process scheduling and control, I/O and data communications support, and a comprehensive set of file-management functions. In addition, GUARDIAN plays a key role in providing a fault-tolerant operating environment for applications running on Tandem systems.

Tandem hardware consists of multiple processors and I/O controllers connected via dual, high-speed, parallel interprocessor buses. GUARDIAN functions are distributed over all the processors in the system, certain components existing in every processor, others existing only in those processors where they are necessary. For example, a monitor process runs in every CPU to handle process starts and stops, maintain the system time of day, and return information about resources attached to its processor. I/O processes, however, are configured only in those processors to which the devices they control are attached. Thus, a separate configuration of the operating system exists in every processor. The operating system processes communicate with each other via fault-tolerant messages.

The User Interface to the Message System

The GUARDIAN operating system links together multiple discrete processors to form a system, and extensions to GUARDIAN link multiple systems to form a network. (These extensions, the EXPAND™ networking software and the FOX™ fiber optic extension, are described later.) Users need not be aware of the physical boundaries between processors within a system or between systems in a network, and are able to access a resource anywhere in the network without complicated programming. GUARDIAN makes this possible by supporting a requester-server approach to performing operations. In this approach, user processes performing I/O operations are, for example, the requesters, and I/O processes are the servers.

GUARDIAN can be viewed as having the following components:

- Various system processes.
- The File System.
- The Message System.

The *system processes* provide process control, virtual-memory management, and peripheral-device control functions. They are configured by the SYSGEN program and are started during system cold load.

The *File System* is a set of privileged GUARDIAN procedures that provide users with a uniform mechanism for performing I/O operations. Some of the File System procedures are callable by nonprivileged user code. For example, the callable File System procedure READ enables a user process to obtain data from a disc file by requesting the I/O system process responsible for controlling that disc to perform the necessary input logic.

The request is delivered to the I/O process by the *Message System*, which is a set of privileged GUARDIAN procedures and interrupt handlers. If the disc process is executing in a different processor, the Message System transfers the request over one of the dual interprocessor buses that connect all CPUs in the system. When it receives the request, the disc process performs the physical I/O, if necessary, and formats a reply containing the required data. The Message System then transports this reply back to the user process.

The File System provides an interface to the Message System for nonprivileged users. None of the Message System procedures are callable by nonprivileged user code. Privileged system processes such as I/O processes can interface directly to the Message System.

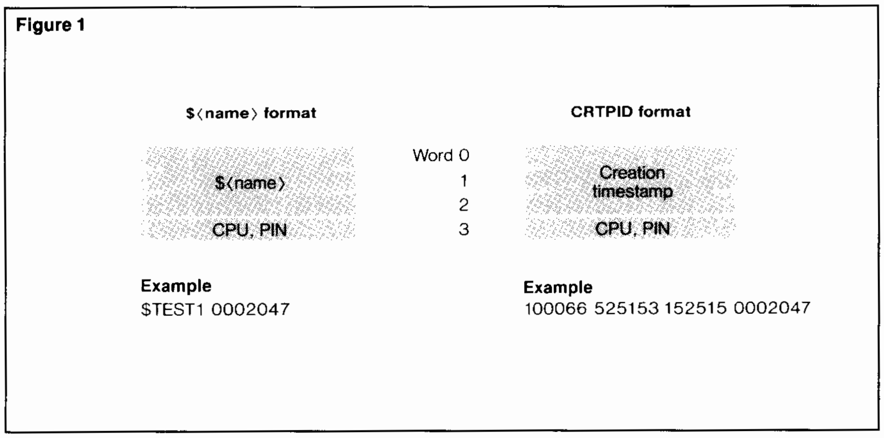
Process Identification

Sending a message and requesting a reply implies that both the sender (the requester) and the receiver (the server) must be readily identifiable. All processes, regardless of whether they are application processes, I/O processes, or other system processes, are distinguished by logical names, either of the $⟨name⟩ format or the CRTPID format.[1] Figure 1 depicts the formats of these logical names.

All systems in a network are distinguished by a logical network node name or identifier. The GUARDIAN operating system uses the local process name to determine its physical location in the system. It uses the local process name along with its system's network node name to determine the process' physical location in the network.

Figure 1.
Logical name formats.

Figure 1



[1] Uppercase characters represent keywords and reserved words. Lowercase characters enclosed in angle brackets (⟨ ⟩) represent variable entries supplied by the user.

Physical Resource Location

The resolution of the logical name into a physical path to a destination is performed by the File System when a user process opens a resource (such as a file, peripheral device, or another process) by calling the File System procedure OPEN. Information regarding the location of a resource identified by the $⟨name⟩ format is obtained by the File System from the Destination Control Table (DCT), a copy of which exists in every CPU in the system. For a resource identified by the CRTPID format, a message is sent to the monitor process in the destination CPU to verify its existence.

Interprocess Communication

If the resource being opened is a "logical device" (an I/O process), an OPEN request is sent to it, whereas if the resource being opened is another user process willing to participate in interprocess communications, an OPEN message is sent to it. Included as part of the OPEN request and OPEN message is information such as the identity and security clearance of the opener and the desired access mode. The receiving process (server) has the option of either accepting and acting on the request or rejecting it based on conditions such as security requirements.

A process can initiate a message exchange with any known process anywhere in the network. It is up to the receiver of the message to decide whether or not to participate in the exchange. Thus, access security is the responsibility of the receiving process. For example, a request to open a disc file is denied by the disc processes as a security violation (Error 48) if the requester does not have adequate security clearance.

If a server accepts the OPEN request and returns a positive response to it, the File System assigns a file number to it. This file number is unique to the opening process (requester) and can be viewed as a virtual connection (file) through which all subsequent communications with that server are performed. All resources being accessed via the File System must first be opened before they can be used.

The Message Queue

Requesters and servers execute asynchronously. They can execute in the same CPU, in different CPUs in the same system, or even in different systems. The GUARDIAN operating system enables processes to synchronize their activities so that they can send and receive messages by providing each process with event flags that signal the arrival of a message or the completion of a previously initiated message.

In addition, each process has an associated incoming message queue, which has the system-specified name of $RECEIVE. A process can check for the completion of a previously initiated message by calling the procedure WAIT with a parameter of LDONE (link done), and it can await notification of an incoming message by calling WAIT with a parameter of LREQ (link request). The File System performs this logic for nonprivileged user processes when procedures such as READ, WRITE, READUPDATE, and AWAITIO are called.

When a requester initiates a message, the address of the Message System data structure used to control that message exchange (see the section "Message Control") is inserted into the server's $RECEIVE queue, and its LREQ event flag is posted. The requester is suspended from the time it initiates a message until the time the server notification is posted. The requester can then continue executing and check for the receipt of a reply at a later time.

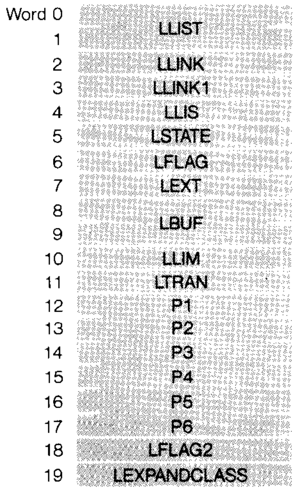
The requester can have several outstanding requests at the same time. The ability to initiate several requests and check for their replies at a later time is used by the File System to allow user processes to perform NOWAIT I/O operations.

Although the arrival of the message is posted as soon as the message is initiated, the server can check for and process incoming messages at its convenience. File System procedures enable servers to access $RECEIVE by treating it like a file. In order to invoke File System support for $RECEIVE, server processes participating in message exchanges must OPEN it. Once $RECEIVE has been opened, the server can receive and respond to requests via $RECEIVE by calling the File System procedures READUPDATE and REPLY respectively.

To participate successfully in a message exchange, both the requester and server must understand what constitutes a message and agree upon an exchange protocol. The GUARDIAN operating system defines the structure of a message and imposes an explicit message-exchange protocol. The File System procedures such as READ, WRITEREAD, and WRITE, which are called by the user to perform I/O operations, understand the structure of a message and comply with the message-exchange protocol by invoking the appropriate Message System procedures in turn. The message-exchange protocol is described in detail below.

User processes exchanging messages must define data-exchange sequences or dialogues meaningful to their tasks. These data sequences or *messages* are transferred between the two participating processes in accordance with the message structure and exchange protocol defined by GUARDIAN.

Figure 2



Word	
0	LLIST
1	
2	LLINK
3	LLINK1
4	LLIS
5	LSTATE
6	LFLAG
7	LEXT
8	LBUF
9	
10	LLIM
11	LTRAN
12	P1
13	P2
14	P3
15	P4
16	P5
17	P6
18	LFLAG2
19	LEXPANDCLASS

Figure 2.
The format of a Link Control Block (LCB).

Message Control

A message consists of a special data structure called a Link Control Block (LCB) and optional user data. LCBs contain information about the message, such as the identities of the requester (Linker) and server (Listener), the address and size of an optional user-data buffer, and information as to whether the message requires special security checking.

The LCB also contains 6 parameter words which may be sufficient to contain the text of a short message. The optional user-data portion of the message can be used to contain additional information. A process sending a WRITE request to a disc process would include the data to be written to disc in the user-data portion of the message. The File System may include a message header containing control information meaningful to the disc process as part of the user-data portion of the message. At a minimum, a message consists of an LCB.

In addition to containing information about a message, LCBs are used by the Message System in implementing the message-exchange protocol. Figure 2 represents the format of an LCB, and Table 1 describes the data contained in it.

Table 1.
The data contained in a Link Control Block (LCB).

Words	Data
0:1	**LLIST.** Linkage used by the Message System to attach the LCB to various lists, such as $RECEIVE.
2	**LLINK.** Identity of the process initiating the message (Linker).
3	**LLINK1.** Address of the sender's (Linker's) LCB in the sender's CPU.
4	**LLIS.** Identity of the destination process (Listener).
5:6	**LSTATE, LFLAG.** State of the LCB in the message-transfer protocol and bit fields used by the Message System to keep control information.
7	**LEXT.** Address of the LCB extension, if any. (Used for transfers to another system, for example.)
8:9	**LBUF.** Absolute extended address of the sender's data buffer.
10	**LLIM.** Size, in bytes, of the sender's data buffer.
11	**LTRAN.** Transfer count in bytes. This may be different from LLIM in word 10. It is used to prevent buffers from being over-written.
12:17	**P1-P6.** User parameter words that have special significance to the two processes participating in a message exchange.
18	**LFLAG2.** Control information maintained by the Message System.
19	**LEXPANDLCASS.** For future enhancement.

A complete message exchange consists of a request sent to the receiver (Listener) and, optionally, a reply returned to the sender (Linker). The successful transfer of a message requires that one LCB be allocated for the Linker and one for the Listener, in their respective processors. The LCBs are allocated out of a pool in system data. The number of LCBs available in a processor is configured by the user during SYSGEN. LCB allocation is controlled by providing both "pool" LCBs and "reserved" LCBs.

In order to ensure that message transfers are not hampered due to the Message System's inability to allocate LCBs, a process can reserve LCBs for queuing incoming requests and initiating requests by calling the procedure RESERVELCBS. If a process has all of its reserved LCBs (possibly none) in use, pool LCBs are allocated when they are available. If an LCB cannot be allocated within 10 seconds, the message initiation fails. System server processes reserve one or more LCBs for incoming messages and a sufficient number, dependent solely on the server's needs during request processing, for outgoing messages.

The Message System allocates LCBs and initializes the various fields according to the data supplied by the caller. For user processes, the caller is a File System procedure. Once the LCBs are successfully allocated, the Message System uses the information contained in them to transport the message to its destination. If unable to deliver a message either because the Listener did not exist or a Listener's LCB could not be allocated, the Message System returns appropriate error indicators to the caller. The caller can then take any recovery action desired.

The Message-exchange Protocol

The message-exchange protocol is implemented as the following sequence of events:

1. The initiator of the message (Linker) informs the receiver (Listener) that it has a message to send. It does this by inserting an LCB into the Listener's incoming message queue.

2. The Listener, at some point in its processing sequence, determines that it has an incoming message by examining its message queue. If it decides to accept the message, the Listener requests the Message System to transfer any associated user data.

3. When the complete request (LCB and user data) is received, the Listener processes it and may return a reply.

4. When the Linker receives the optional reply, the message-exchange protocol is implicitly complete. If the Listener does not reply within the time expected by the Linker, the Linker may choose to complete the message explicitly.

The Message System uses the LSTATE word in the LCB to keep track of the message as it moves through the various stages in the protocol. Figure 3 depicts a message exchange between two processes executing in different CPUs, which requires the involvement of the interprocessor hardware and software. Refer to it as the message protocol is described.

LINK, LISTEN, READLINK, WRITELINK, and BREAKLINK are Message System procedures called by the File System on behalf of the user process. They can be called directly by privileged system processes.

LINK initiates the message exchange. It causes the allocation of an LCB for the Linker and initializes the various fields according to the information passed to it by the caller. It calls other Message System procedures that cause the message to be queued for transmission to the destination processor over one of the 2 interprocessor buses.

The Linker is then suspended. The Dispatcher interrupt handler subsequently issues the SEND instruction to transfer the LCB (in PMSG state) to the receiver's processor.

The BUSRECEIVE interrupt handler in the Listener's processor stores the incoming message temporarily in a special control package area while it determines whether the Listener exists. If it does, a Listener's LCB is allocated. If an LCB is successfully allocated, BUSRECEIVE copies the contents of the incoming message from the temporary storage area into the Listener's LCB and inserts it into the Listener's message queue. BUSRECEIVE calls the procedure AWAKE to post an LREQ (link request) condition to the Listener. It then acknowledges the transfer.

If the Listener has been WAITing on LREQ, i.e., if it has called the procedure WAIT to monitor link-request notification, it is allowed to resume execution. When BUSRECEIVE in the Linker's CPU receives the acknowledgment, it allows the Linker to resume processing. If unable to allocate a Listener's LCB, or if the Listener does not exist, BUSRECEIVE in the receiving CPU notifies the Linker that it was unable to successfully initiate the message, by returning a special acknowledgment word.

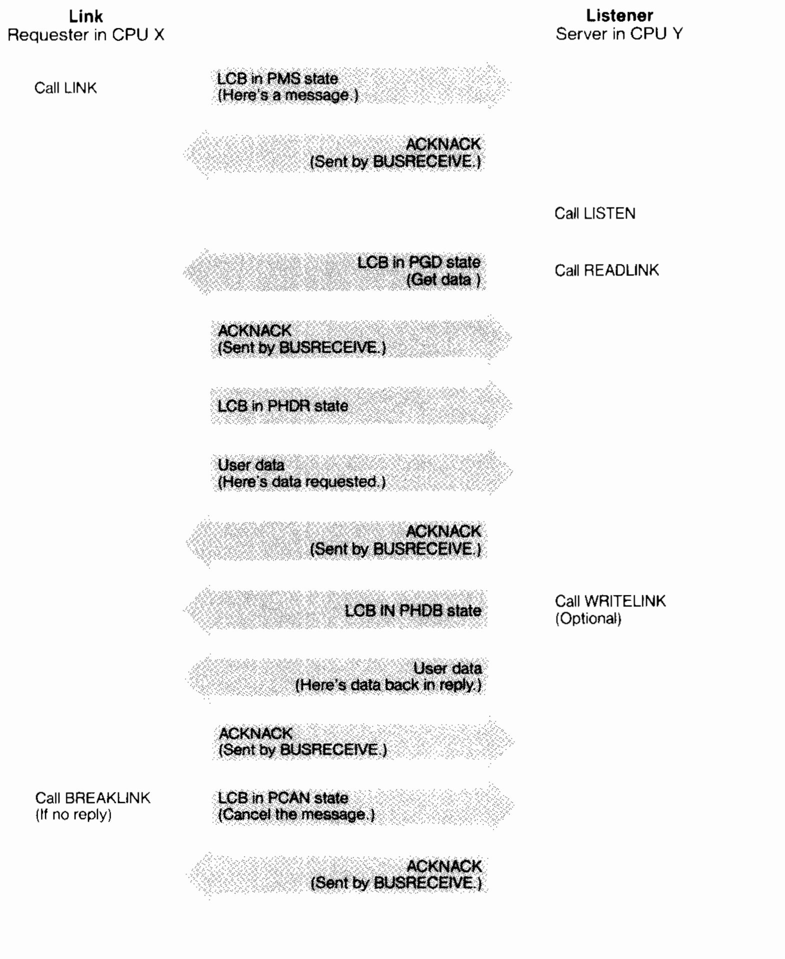
To check for incoming messages, the Listener calls the procedure LISTEN, which returns the address of the first element in the caller's message queue. The Listener now has the option of specifying how much user data it is willing to accept from the Linker and the location of the buffer where it is to receive the data. The Listener then calls READLINK, which causes the LCB in the PGD state to be sent to the Linker's CPU. The Listener is suspended at this point.

The BUSRECEIVE interrupt handler in the Linker's CPU acknowledges receipt of this LCB. Recognizing the PGD state, it queues the LCB for transmission after setting the state to PHDR. The Dispatcher then issues SEND instructions to transfer this LCB and any associated user data to the receiver's processor.

The BUSRECEIVE interrupt handler in the Listener's CPU stores the incoming LCB in a special control-packet area. Recognizing the LCB in PHDR state, BUSRECEIVE uses the LLIS field of the incoming LCB to determine the identity of the Listener. With this information, BUSRECEIVE is able to locate the corresponding Listener's LCB, which specifies the address of a buffer where the incoming user data is to be stored. When the entire transmission is complete, BUSRECEIVE returns an acknowledgment.

Figure 3.
The protocol for a message transfer.

Figure 3



The Listener is now allowed to process and may choose to return a reply by calling the WRITELINK procedure. WRITELINK causes an LCB in the PHDB state to be queued for transmission to the Linker's CPU. The Dispatcher then issues SEND instructions to transfer the LCB and its associated data to the Linker's CPU.

Recognizing the LCB in PHDB state, BUSRECEIVE in the Linker's CPU stores the incoming reply data into the buffer specified by the corresponding Linker's LCB and sets the LDONEB bit in the LSTATE word of that LCB to indicate message completion. It then acknowledges this receipt and calls AWAKE to post LDONE notification to the Linker. If it has called WAIT to monitor LDONE, AWAKE allows the Linker to resume execution. If the Linker has not been waiting on LDONE, a subsequent call to WAIT on LDONE returns completion notification to it.

For transmitting data over the interprocessor buses, a protocol similar to ISO's HDLC is used.

Since the reply is optional, the Linker may choose to complete the message by calling either BREAKLINK or REVOKELINK. BREAKLINK cancels the message by sending an LCB in PCAN state to the Listener. If REVOKELINK is called instead, it causes the message to be deleted from the Listener's message queue if it has not yet been listened to. The Listener is thus unaware of that message and does not have to spend time in processing a cancelled message.

By convention, the Linker may not modify the contents of its LCB once the message has been successfully initiated. Since it cannot alter the LBUF field in the LCB during the message exchange, the same buffer must be used for sending request data and receiving reply data. In order to prevent its buffer from being overlaid by the reply data, the Linker can request the Message System to inhibit transfer of the user-data portion of the reply.

The Listener is expected to initialize the LBUF, LLIM, and LTRAN fields in its LCB prior to calling the procedure READLINK. This ensures that the user-data portion of the message is transferred into a buffer that belongs to the Listener. While the Listener can use the same buffer for returning the reply data, it can also use a different buffer by modifying the LBUF, LLIM, and LTRAN fields of the LCB before calling WRITELINK. Both the Linker and the Listener can set LTRAN to 0, indicating that no user data is associated with the message.

The Messenger Process

Certain privileged processes that are sending noncritical, information-only messages (such as the Command Interpreter sending a CLOSE message to a process it has created) may not want a reply or want to spend the time to participate in the various stages of the message-exchange protocol. These processes can request the Message System to notify the messenger process when errors occur or when the message completes successfully. The messenger process then retries, if necessary, and handles message completion. There is no way for a nonprivileged user process to use this service of the messenger.

The messenger process is also responsible for notifying interested processes of the arrival of system status messages, such as those messages stating that a CPU or network line is up or down.

Interprocessor Bus Control

Message transfers between cooperating processes executing within the same processor are handled entirely by the Message System procedures, which use the standard mutual-exclusion primitives and perform the necessary MOVEs within memory. Messages between processes executing on different processors in the system are transferred over one of the dual, high-speed, parallel interprocessor buses (IPBs), the X or the Y bus. Since bus transfer is much faster than memory transfer, a hardware-buffering scheme has been implemented to ensure efficient use of the buses.

One set of registers for buffering incoming data (INQ) and another for buffering outgoing data (OUTQ) exists for each bus in every processor. The INQ and OUTQ are capable of buffering up to 13 words of data and 3 words of control information required for managing the interprocessor transfer. Therefore, transfers across the bus occur in "packets" of 16 words. If the size of the message to be transferred is larger than 13 words, it is divided into as many packets as necessary by the microcode controlling the bus hardware. Since the size of a packet is set at 16 words, messages with less than 13 words of data are padded out with zeroes. Additional detail on the IPB is contained in the *System Description Manual.*

For transmitting data over the IPB, a protocol similar to the International Standards Organization's High-level Data Link Control (HDLC) is used. This protocol uses sequenced packets and returns only positive acknowledgments. Following transmission by the sending processor, the message waits on the wait acknowledgment (WACK) list. If the message is still on the WACK list after one second, it is re-sent over the other bus. This cycle continues until either the transfer is acknowledged or the receiving processor is considered as having failed. Repeated failures to acknowledge transfer over the bus cause the sending processor to mark the receiving processor as having failed. Each processor can have up to 4 transfers awaiting acknowledgment from each processor in the system. Subsequent transfers are queued until they have been acknowledged.

Each processor maintains a Bus Receive Table (BRT) entry to control incoming data from each processor. The BRT entry includes a buffer address, a transfer count, and the next-expected sequence number. When a packet arrives, it is checked for correct routing (the receiver's CPU number in the incoming packet is the receiving CPU), the sequence number is verified, and the checksum is computed by the IPB microcode. If the packet is without error, the BRT entry is updated. When the transfer count becomes zero, or a packet error occurs, a software BUSRECEIVE interrupt is posted. When a packet error is detected, the receiving processor merely notes the type of error that occurred, discards the packet, and flushes the rest of the message. Error recovery is the responsibility of the sending processor.

Since the IPB is implemented as a closed environment, fewer errors occur within it than occur within conventional data-communications environments. Observations have shown that the rate of IPB error occurrence is very low; for example, the error log for the system on which this article was prepared contained no IPB error entries for the entire month the article was written. Therefore, any time lost due to time-out processing or packet flushing is not the primary concern in the error-recovery mechanism; correctness of error detection and recovery are.

Approximately every second, each processor sends an unsequenced packet over each bus to each processor, including itself. This packet serves 2 purposes: to recover from lost acknowledgments and to inform the other processors that it is up. Approximately every 2.4 seconds, each processor checks to see whether these packets have arrived from all processors in the system. If a packet from a particular processor has not arrived, that processor is designated as down.

Message System Features

The message-exchange protocol described above is used both for intraprocessor and interprocessor message exchanges. Although the latter cause additional LCB state changes, no additional user programming is required. Since all message exchange is done by moving data rather than with shared data structures, the Message System appears to function in the same way regardless of the locations of the requester and the server. The Message System also allows message cancellation by both the requester and the server.

The requester can cancel the message by calling the procedures BREAKLINK or REVOKELINK. If the message has not been completed, BREAKLINK sets the cancelled flag in the Listener's LCB and awakens it on the LCAN (link cancelled) event. Calling the procedure REVOKELINK causes the Listener's LCB to be deleted from its $RECEIVE queue if it has not yet been been removed by LISTEN. This saves the Listener the time that would have been spent in processing a cancelled message.

The server can also cancel the message by setting the LCAN flag in the LCB LSTATE word and calling the procedure WRITELINK without specifying a buffer address in the LBUF field. This results in the requester being AWAKEned on LDONE with both the LDONE and LCAN flags set in its LCB.

The cancel flag provides a uniform mechanism for signalling failures. It enables outstanding messages to be completed upon process or processor failure by simulating message cancellation on the part of the failed end of the message transfer.

Once a message has been successfully initiated, no further participation is required by the Linker in the message exchange. This allows the Linker to continue with other processing and check for message completion later. It also allows the Linker to have several requests outstanding at any given time. The File System utilizes this ability in supporting NOWAIT I/O operations for user processes.

Incoming messages are inserted into the Listener's $RECEIVE queue in first-in-first-out (FIFO) order. The Listener can optionally request the Message System to insert incoming messages into its $RECEIVE queue in the order of the priority of the Linker. The Listener can pick up multiple requests by calling LISTEN successively, queuing them internally, and not READLINKing or WRITELINKing a request until ready to process it.

When OPENing $RECEIVE, a nonprivileged server can specify the RECEIVEDEPTH parameter with a value greater than 1. This causes the File System to enable the server to process that number of requests concurrently.

If users can identify a resource by name, the File System can determine the location of that resource and handle the Message System interface functions on their behalf. It complies with the message-exchange protocol described above by calling the various Message System procedures in turn. The File System provides users with a uniform callable interface for accessing other resources on the system. Regardless of whether the other resource is a user process, a disc file, or other peripheral device, users are able to call the File System procedures, such as OPEN, READ, WRITE, and CLOSE, to perform I/O operations to it.

The File System, along with the Message System, provides user processes with communications homogeneity and location transparency. It allows systems analysts to optimize system performance by distributing the processing load among the available processors.

Isolating user processes from configuration details by forcing them to communicate with other entities via the Message System facilitates on-line repair of failed components, an important feature of fault-tolerant system availability. The message-based operating system also allows the addition of hardware components such as CPUs, memory boards, and peripheral devices, as needed, to meet increased processing requirements and obviates the painful upgrades necessary with conventional computer architectures.

If the application software has been designed to allow it, additional copies of software modules (requesters and servers) can be implemented to meet increased transaction volumes. Benchmark results have shown that in a well-balanced system, the incremental increase in throughput resulting from the addition of processing modules is linear. Thus, a well-balanced 16-processor system with an adequate complement of peripheral devices supports twice the transaction throughput of a well-balanced 8-processor system.

Message System Extensions

The Tandem hardware architecture and the GUARDIAN operating system provide a local network of CPUs. Processes executing in one CPU have transparent access to processes and devices in other CPUs. The only limitation of distance between processors is that imposed by the interprocessor bus, which can be no longer than approximately 20 feet.

In order to support the need for distributed processing, or to meet processing requirements that cannot be handled by one 16-processor system, additional systems can be installed and connected together in a network. EXPAND networking software extends the GUARDIAN operating system beyond the boundaries of a single system. GUARDIAN/EXPAND allows up to 255 geographically dispersed systems to be interlinked via telephone lines and/or satellites. EXPAND maintains the geographic independence of resources provided by GUARDIAN, so that any resource in the network can be addressed by its logical file name, without regard to its physical location. Figure 4 represents the formats of the network name.

The major components of the EXPAND networking software are the end-to-end protocol, the Network Control Process (NCP), the line handlers, the Network Routing Table (NRT), and various network utilities. For additional information about the GUARDIAN/EXPAND network, see Katzman and Taylor.

Figure 4

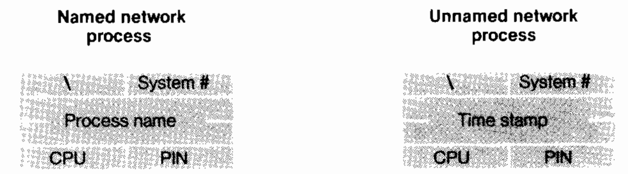


Figure 4.
Network name formats.

FOX, Tandem's fiber optic extension to the bus architecture, can also be used to connect systems located within 1 km of each other into a network. Up to 14 systems of up to 16 processors each can be connected with FOX. Geographically dispersed FOX rings can then be connected with data communications lines supported by EXPAND.

FOX software includes enhancements to the Message System and system processes, such as the FOX line handler and FOX IPB monitor. These perform start-up handshaking logic, maintain the Message System data structures, and process "secured" (described below) intersystem message traffic. The EXPAND and FOX extensions to GUARDIAN enable users of Tandem systems to use simple read/write communications with other processes anywhere in the network.

Messages bound for remote systems in an EXPAND network require the participation of the communications line handlers to transport the message over the communications line. Messages sent between systems within a FOX network are transferred directly over the extended bus hardware. The majority of messages between systems in a FOX ring do not require the service of the FOX line handlers; they are handled directly by the Message System procedures. FOX line handlers are only required for the establishment and management of node connection and the processing of secured messages.

In both EXPAND and FOX networks, if a direct connection does not exist between the sending and receiving nodes, the line handlers take care of forwarding the message to its ultimate destination. Message System operation at the source and destination nodes is the same as described above for intrasystem messages.

GUARDIAN security includes features designed specifically for the network environment. A password validation is performed when a remote file is opened or a remote process is created. When a remote OPEN or NEWPROCESS occurs, it is directed to the EXPAND or FOX line handler in the remote system rather than to the disc process or monitor process. For messages bound for a remote destination, the network software in the sending node sends the local remote password along with the message. The network software on the receiving node checks the USERID file for a match on that password. If the check fails, the server is notified of the security failure via the LSECUREB bit in the LCB.

An application should be composed of a set of modular, tunable entities.

The LSECUREB bit in the LFLAG word of the LCB must, by convention, be set to 1 for secured messages, such as an OPEN message, to indicate to the receiving process that the sender has cleared security in the receiver's system. If this bit is not on, the receiver should reject the request. The Discprocess enforces this rule; for example, OPEN requests arriving on its message queue without LSECUREB being true are rejected as security violations (Error 48).

The network extensions to the GUARDIAN operating system provide user processes with almost complete location transparency in accessing remote resources. For the application programmer, accessing a remote file, device, or process is the same as accessing a local file, device, or process, with the following exceptions:

1. Names of devices to be accessed remotely are limited to 6 characters instead of 7.
2. Names of processes to be accessed remotely are limited to 4 characters instead of 5.
3. The backup process of a process pair must execute in the same system as the primary.
4. A program to be run remotely must exist on a disc that is physically connected to the remote system.

Application Design

Although GUARDIAN and its network extensions facilitate easy access to remote resources, developing a distributed application requires careful analysis and the selection of an application structure that optimizes system performance and reliability.

Requester-server Design Philosophy

The requester-server design philosophy is recommended for applications. This approach assigns separate application system modules to handle the user or terminal interface (*requesters*) and the data-base interface and management functions (*servers*). It allows an application to take full advantage of the inherent features of the Tandem architecture.

In this approach, the interface between requesters and servers is restricted to a well-defined and limited set of message formats and request codes. The modular structure of well-designed requester-server software greatly facilitates the installation, maintenance, modification, and expansion of a system. In general, the following ideas should be kept in mind when designing an application for the Tandem system.

The application should be composed of a set of modular, tunable entities, such that additional copies of the modules can be readily configured to meet increased demand, new modules can be easily incorporated into the system to implement new functions, and the modules can be distributed across all the processors in order to tune and balance system performance.

The major components of an application are shown in Figure 5. The communications I/O process performs processing that is dependent on devices and communications lines. It supports specific line protocols and device types; handles error recognition, reporting, and recovery; and performs functions such as software downloading to intelligent terminal devices.

The requester receives requests from the user at the terminal by calling the File System procedure WRITEREAD. It formats requests from the user into messages, ensures that all the information required to process the request is present, and initiates a message to the appropriate server by calling the File System procedure WRITEREAD.

The server processes the functional requests from the requester and performs all the data-base I/O necessary. It obtains the request by calling the File System procedure READUPDATE, processes the request, and returns a reply by calling the procedure REPLY.

Design Method

In this section, one method of achieving a request-server design is described. As it is beyond the scope of this article to discuss the need for careful data-base design and the problems of data-base integrity, recoverability, system management, and operation, refer to the Tandem Application Monograph Series and applicable Tandem product manuals for a discussion of these critical issues.

With careful analysis, the requirements of an application system can be grouped into functional subsystems. For example, the requirements for a banking money-transfer system might be grouped into two subsystems: wire-transfer line management and wire entry and repair service (WERS).

Within each subsystem, sets of related functions can then be grouped to form transactions. Transactions in this context define units of work as perceived by the end user. In the example above, the WERS subsystem might consist of the following transactions:

1. A wire-entry transaction, in which a wire to be sent over one of the wire services (e.g., TWX, SWIFT, or FEDWIRE) is entered.

2. A wire-review transaction, in which incoming and outgoing wires are reviewed and approved.

3. An inquiry transaction, in which, for example, a customer's balance is queried or a wire is queried by account number.

Figure 5

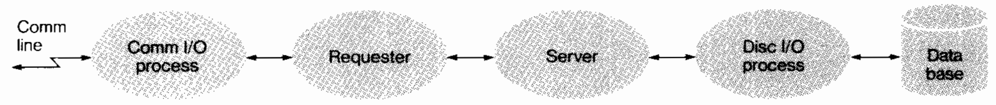


Figure 5.
The components of an application.

After the various transactions are defined, the series of requests necessary to execute each transaction can be defined. The inquiry transaction might consist of the following requests:

- Perform access security clearance.
- Format an inquiry request, and send it to the inquiry server.
- Receive the reply from the inquiry server, and inform the user.

These requests can then be named by single imperative statements as follows:

- Accept customer account number and password.
- Send authorization request to security server.
- Send inquiry request to inquiry server.
- Display customer account information.

Each request identified in this manner can then be broken into a set of elementary services that must be performed by the server. For example:

- Read customer master record.
- Authorize or deny access based on a comparison of the password supplied by the requester and that stored in the file.
- Read customer detail record.
- Format and return a reply to the requester.

Each of these services can then be described by single imperative statements, as follows:

- Read customer master file.
- Authorize access.
- Read detail record.
- Reply to requester.

Once each transaction is analyzed in this manner, the list of requests and services should be examined to eliminate duplicates. The result is a set of transactions that cause specific requests, resulting in specific services.

The next step is to define the relationship of each transaction with others in the same subsystem so that duplicate or similar requests and services within each subsystem can be isolated. Once this is done, it is possible to map requests and services into requesters and servers.

As a rule, those processes that initiate a request for service are requesters. They call the File System procedure WRITEREAD. Servers are those processes that process a request and return a reply. They call the File System procedure READUPDATE to receive an incoming message from $RECEIVE and the procedure REPLY to respond to the requester.

Request and Reply Formats. Once the requesters and servers have been identified, formats for a meaningful dialogue can be defined. A successful message exchange requires both the coordination of message delivery and receipt, and cooperation in formatting meaningful requests and replies.

The Message System provides the coordination mechanism. For processes performing I/O operations with peripheral devices, the File System provides all the cooperation logic because it understands and complies with the format of the requests expected by the I/O processes and the replies returned by them. User processes communicating with each other must similarly agree upon message formats. It is recommended that the format of requests and replies be defined using the Data Dictionary Language (DDL). A central source will then exist which, when copied by both the requester and server programs, will ensure a meaningful dialogue between them.

Subsystem Interdependencies. Once the processing requirements within each subsystem have been defined, the interface requirements between various subsystems must be identified. Methods for detecting and handling errors, and requirements such as availability and recoverability must also be examined. Completion of the steps outlined above will result in a top-level design which can be used as a basis for developing a detailed design and implementation.

Benefits of Requester-server Modularity

When the application is divided into modules of requesters and servers, each of them can be configured in the processor or system that is closest to the resource they control. For example, in a distributed on-line transaction processing system, the requesters can be configured in the system to which the terminals are attached and the servers can be configured in the system on which the data resides. Within one system, requesters and servers can be distributed among the available processors based on system-load balancing and performance-tuning requirements. In either case the GUARDIAN operating system, along with the EXPAND and FOX extensions, ensures that no special programming effort is required.

Conclusion

The Tandem architecture enables a system to expand from a single 2-CPU system to a network of 255 systems of 16 CPUs each, for a total of 4080 CPUs (excluding FOX considerations). The CPUs in a system and the systems in a network are connected by powerful, fault-tolerant software that enables applications to access files, processes, and devices anywhere in a network, using simple read-write logic.

The GUARDIAN Message System (along with EXPAND and FOX network extensions) guarantees that all processors in a system and in the network have access to all available resources, no matter what their locations. If designed to take full advantage of the Message System's features, an application running on the Tandem system will, in turn, use the architecture to its maximum advantage, readily responding to changes in transaction volumes and business requirements.

References

Collins, J. 1982. *Transaction Processing on the Tandem NonStop Computer: Requestor/Server Structures.* Tandem Application Monograph Series. Tandem Computers Incorporated.

Katzman. J. A. and Taylor, H. R. 1978. *GUARDIAN/ EXPAND: A NonStop Network.* Tandem Computers Incorporated.

Smith, L. 1982. *Designing a Network-Based Transaction-Processing System.* Tandem Application Monograph Series. Tandem Computers Incorporated.

System Description Manual: NonStop II System. Part No. 82077 D00. Tandem Computers Incorporated.

Acknowledgments

The author would like to thank Richard Carr, Kevin Coughlin, Mike Lisenbee, Dick Thomas, and Denis Winn for providing valuable technical reviews.

Mala Chandra joined Tandem in San Francisco as a Senior Systems Analyst in August 1982. She transferred to the Customer Application Support Group (CASG) in March 1984, where she taught GUARDIAN Internals courses to Tandem software developers. Currently she is with the Installability and QA group, where she is working to ensure the supportability and quality of Tandem products.

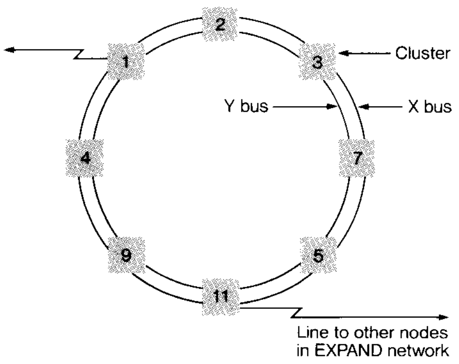
Using FOX to Move a Fault-tolerant Application

The 6700 Fiber Optic Extension (FOX) was designed to connect NonStop II or NonStop TXP systems into a high-speed fault-tolerant local ring network. FOX's fiber-optic technology allows information to be sent and received concurrently at almost 300 times the rate provided by dual 56K-bit lines. This speed and the amount of processing power in the ring can create a "virtual system" that provides more on-site transaction processing power than that of most large mainframes. Thus, FOX allows the extensive sharing of resources between multiple systems that large applications require without the performance penalties or CPU overhead associated with lower-speed telecommunications.

Figure 1.

A FOX ring. Each cluster (system) is connected to its neighbors via two FOX links, which function as X and Y extensions to the DYNABUS. A ring can exist alone or as part of an EXPAND network. Here, Clusters 1 and 11 connect to nodes outside the ring via EXPAND lines.

Figure 1



FOX is also useful for other applications for which a high-speed link between local systems is required. This article describes how Mellon Bank, in Pittsburgh, used FOX to relocate their 8-CPU Tandem production system with no interruption or slow down of their 24-hour, 7-day-a-week automated-teller processing. Indeed, FOX was essential to Mellon's success in avoiding more conventional (and costly) moving alternatives.

FOX Facts

FOX can be used to connect 2 to 14 NonStop II or NonStop TXP systems in a local ring network. Each system in a FOX network is called a *cluster*. These clusters must be within 1 km of one another but may contain up to 16 CPUs each, for a maximum of 224 in a ring.

Each cluster in a ring connects to each of its neighbors via two FOX links. These intersystem links function as X and Y extensions to the interprocessor DYNABUS™. A single connecting cable contains 5 optical fibers: X-send, X-receive, Y-send, Y-receive, and a spare. In this way the cable provides four 1-Mb/second links from each cluster (two in each direction) for a theoretical aggregate data-transfer rate of 4 Mb. A ring can exist on its own, or it can be part of an EXPAND network, as shown in Figure 1. EXPAND lines connect the ring to other rings or EXPAND nodes. (The maximum number of nodes in an EXPAND network is still 255.)

Figure 2

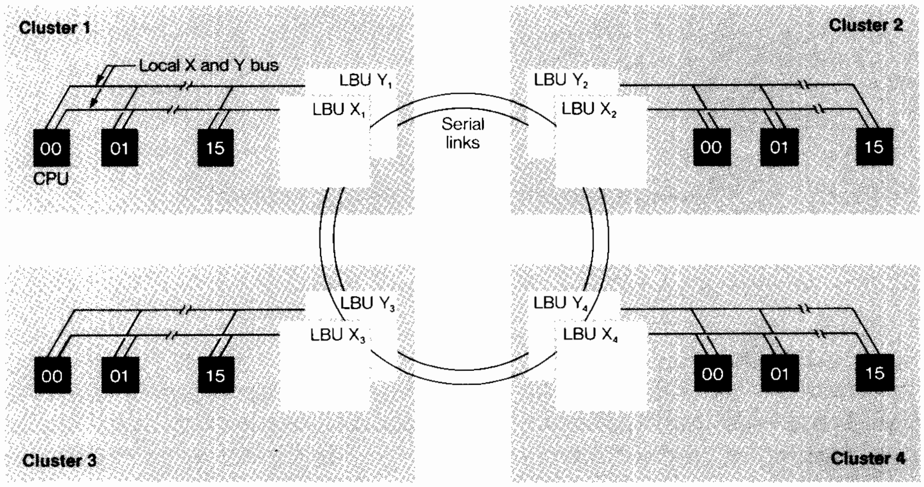


Figure 2.
Every cluster in a FOX ring has two Local Bus Units (LBUs), which control the cluster's interprocessor bus and send and receive packets over the ring network.

Data can move "left" or "right" around the ring. The user can specify which link to try first. If that link fails, the system automatically attempts to transfer data over the next preferred link, and so on. Interestingly, the ring topology creates an added measure of redundancy, providing for continuous operation in the unlikely event that both links between adjacent clusters should fail.

Every cluster has a Local Bus Unit (LBU) for each bus. A FOX network consists of interconnected LBUs, as illustrated in Figure 2. The LBU controls the interprocessor bus and sends and receives packets over the ring network. It consists of a set of 3 FOX circuit boards for each bus: the Serial Link Board (SLB), the Control Processor Board (CPB), and the Interprocessor Bus Control board (IPC).

As long as a cluster's SLB is powered on, data can pass through it (even if the cluster is no longer in service). If the SLB should fail or be powered off, data can still move around the ring in the opposite direction.

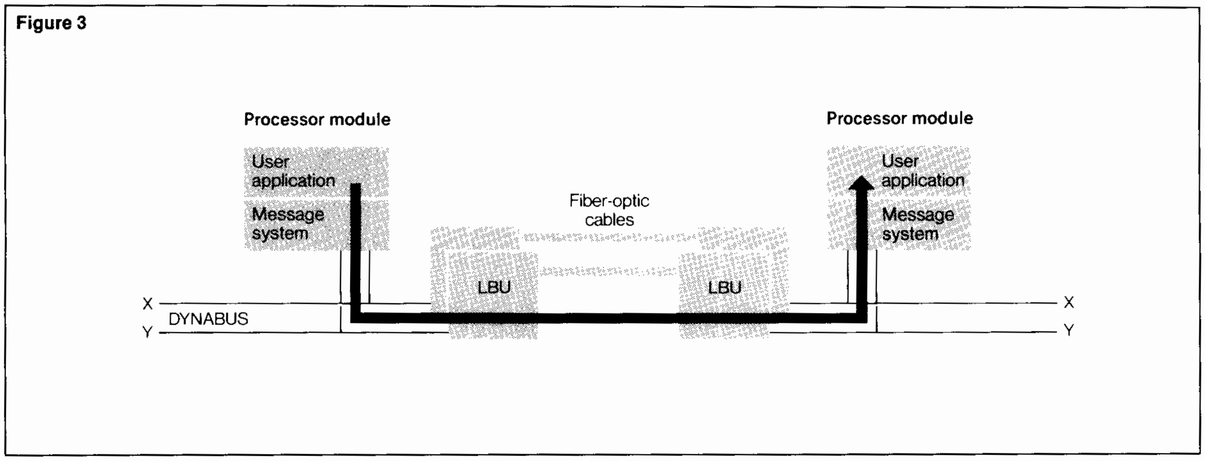
The GUARDIAN interprocessor-bus monitor process controls the state of the LBU. It also controls the downloading of the Loadable Control Store (LCS) with the LBU microcode.

FOX provides a true system-to-system link, not a device-to-CPU interface. Each cluster in a FOX network appears to the user to operate just as a system in an EXPAND network.

Since most data transfers do not involve communications line-handler processes, the user should notice a reduction in CPU-busy rates attributable to network functions. Specifically, requests to OPEN or CLOSE files require line handler intervention; file access requests (such as READs and WRITEs) do not. As illustrated in Figure 3, data for file-access requests is passed in the following manner:

1. Directly from the application process/GUARDIAN message system over the DYNABUS to the LBU.
2. Over the optical fibers to one or more intervening LBUs.
3. From the destination cluster's LBU to its DYNABUS and on to the destination application (or system) process.

Figure 3.
FOX provides a true system-to-system interface (not a device-to-CPU interface).



EXPAND line handlers assist in checking remote passwords and in recovering from certain kinds of errors (e.g., CPU failures and controller-ownership switches).

Several techniques work to ensure data integrity in a FOX network. FOX provides parity checks on all data paths, the control store, and control-store address. It provides cyclical redundancy checks on all packets sent over the fibers. The GUARDIAN operating system provides CHECKSUM and sequence-number checks via the message system.

FOX Applications

FOX is primarily used to expand Tandem systems beyond 16 CPUs or to configure high-performance local ring networks. Users are also discovering other, less permanent applications, such as the system move discussed in this article.

A fiber-optic link such as FOX transports information at a much higher speed than wire cable can. This is because, in parallel or coaxial cable, the bandwidth is inversely proportional to the *square* of the cable length; in fiber-optic cables, it is inversely proportional to the cable length only.

FOX also provides several other advantages inherent in fiber-optic technology. It is impervious to electromagnetic interference, since data transmission is optical rather than electrical. This means that FOX cables can run through environments containing rotating machinery, transformers, relay panels, high-voltage power supplies, arc welding equipment, or industrial lighting fixtures without distortion or significant loss of signal.

FOX provides total electrical isolation between systems. This helps to ensure that large spurious signals, unexpected power surges, or unwanted ground loops do not damage hardware across network nodes.

The Mellon Bank Application Environment

Mellon Bank operates 4 Tandem systems: two each at Mellon Bank, Pittsburgh, and Girard Bank, Philadelphia (Figure 4). These systems constitute production and development environments for the CASHSTREAM automated-teller (ATM) network. Mellon has built its application software around a multinode version of the Advanced Communications, Incorporated (ACI) BASE24 product.

With more than 1100 ATMs, CASHSTREAM is among the largest ATM networks in the United States. On a given day, Mellon might service over 50,000 CASHSTREAM customers in Pittsburgh alone (more than 8% of the city's population).

Many of the ATMs reside in Giant Eagle grocery stores. These stores provide 24-hour service and, since many customers use cash from the ATMs to pay for their purchases, demand continuous availability from the Mellon network.

Much of Mellon's success derives from its aggressive marketing organization, which has capitalized on the reliability and data integrity intrinsic to the Tandem architecture. In its two-year association with Tandem, Mellon has evolved into a highly sophisticated user willing to evaluate innovative technological solutions to their data-processing needs.

The Decision to Move

Last year, Mellon Bank faced several data-processing challenges. The unexpectedly rapid growth of CASHSTREAM had produced an I/O-intensive hardware configuration with a preponderance of 6202 byte-synchronous controllers. This resulted in a skewing of I/O versus system cabinetry, which would not allow the MELLON1 system to expand beyond 12 CPUs without incurring a lengthy network outage to reconfigure cabinetry and power supplies.

Also, the initial Tandem system had been installed so close to other existing hardware that there was no room to expand, even within the 12-CPU limit mentioned above.

Finally, Mellon management found the original installation site too close to the main banking area of the building. To upgrade security, they decided to move especially critical hardware components to an area where they would be inaccessible to customers and unauthorized personnel. These components included the MELLON1 system (where the ATM software resided) and the MELLON2 system (an EXPAND node which Mellon uses for development and testing of new application and system-software releases).

Traditional Approaches to the Move

In examining conventional ways to make the move, Mellon came up with two unacceptable alternatives:

1. They could take an extended outage (24 to 48 hours) to relocate the entire hardware and software environment.
2. They could replicate the existing environment in preparation for an almost instantaneous application cutover.

The first alternative would not be very expensive: while it included customer engineering fees for moving the hardware, it required no additional hardware or software. Nevertheless, because the application had to be accessible to ATM users continuously, Mellon could not accept an outage of sufficient duration to both move and reconfigure their hardware in a single step.

The second alternative would eliminate the prolonged outage; however, it would be quite expensive, since it would include the cost of an entirely redundant hardware system and the customer engineering fees to install it.

Figure 4

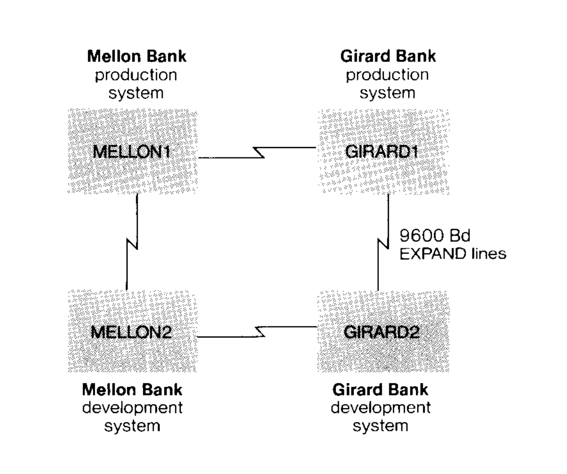

Mellon Bank production system
Girard Bank production system
MELLON1
GIRARD1
9600 Bd EXPAND lines
MELLON2
GIRARD2
Mellon Bank development system
Girard Bank development system

Figure 4.
Mellon Bank's CASHSTREAM ATM Network before the MELLON1 system was moved.

An Intermediate Solution with FOX

Since neither of the above alternatives seemed viable, Mellon asked Tandem to help them find a solution that would balance the high cost of system replication against the negative exposure that might result from an extended outage. Tandem suggested using FOX to make a phased move of system components while minimizing the frequency and duration of application outages. FOX would act as the bridge between the existing production system (MELLON1) and an additional "skeleton" system (MELLON0), which would become the new production system as components were grafted onto it.

In this situation, FOX was to provide geographic independence in the form of increased bandwidth between nodes. While conventional EXPAND lines could support efficient internodal access to their data base, Mellon felt that only FOX would provide the throughput necessary to communicate with communications protocol processes and device drivers on another node. In fact, it was reasoned, to use a conventional communications line handler to reach communications line handlers on another node would, at best, result in double-handling of any communications I/O spanning the nodes.

The skeleton system was to consist of the additional cabinetry required to solve the skewing problem limiting MELLON1's expansion. Also, to allow for a more orderly move, Tandem allowed 4 NonStop II CPUs to remain in place on the original system for the duration of the move. (Mellon traded these in under the upgrade program for NonStop TXP systems.) Parts could be borrowed from the MELLON2 development system, but such borrowing was to be limited to:

1. Brief periods, to avoid delaying testing and user certification on the development system.
2. Only those components that would significantly increase the cost of the move if purchased.

As this system was a skeleton, it contained fewer peripherals than even the smallest NonStop node. It did not begin to approach the size (or cost) of the system replication considered earlier as an alternative.

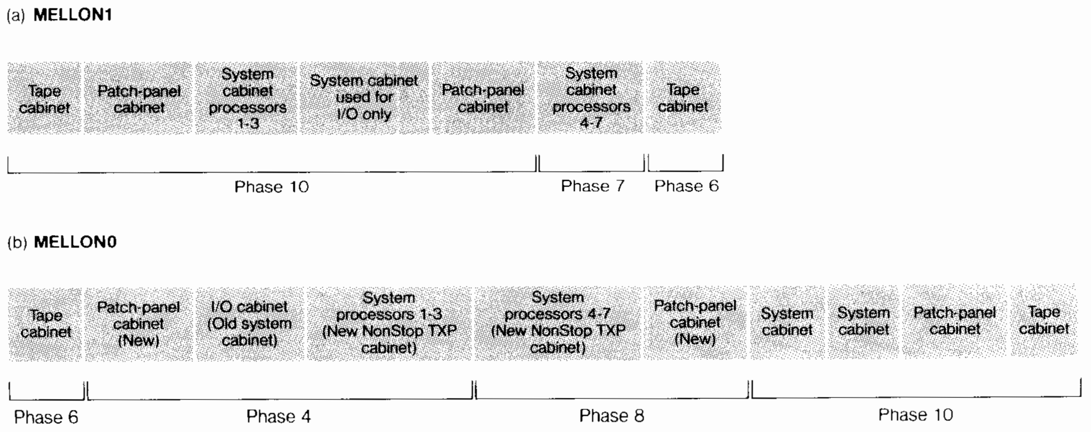
The Move: A Closer Look

On the page opposite is a phase-by-phase description of the move. With FOX, the move became a fluid transposition of hardware modules from one cluster to another. Although the system components described here are unique to Mellon Bank, many of the operations (i. e., the clearing of 6 I/O slots for FOX) would apply in almost any situation in which a hardware and software environment were to be relocated with this approach. Figure 5 shows the physical results of the move.

Figure 5.

(a) The components of MELLON1 and the phases in which they were removed. (b) The components installed as MELLON0 and the phases in which they were installed.

Figure 5


(a) MELLON1
Tape cabinet
Patch-panel cabinet
System cabinet processors 1-3
System cabinet used for I/O only
Patch-panel cabinet
System cabinet processors 4-7
Tape cabinet
Phase 10
Phase 7
Phase 6
(b) MELLON0
Tape cabinet
Patch-panel cabinet (New)
I/O cabinet (Old system cabinet)
System processors 1-3 (New NonStop TXP cabinet)
System processors 4-7 (New NonStop TXP cabinet)
Patch-panel cabinet (New)
System cabinet
System cabinet
Patch-panel cabinet
Tape cabinet
Phase 6
Phase 4
Phase 8
Phase 10

Phases of the Move

Phase 1

NonStop II processors 6 and 7 on the MELLON1 production system were replaced by 2 NonStop TXP processors from the MELLON2 development system. This freed a TXP system cabinet in the MELLON2 system for use in building the new MELLON0 production system.

Phase 2

FOX requires 6 consecutive I/O slots under CPU 0. In this phase, 2 of MELLON1's 6202 byte-synchronous controllers and one 6303/6304 asynchronous controller and extension were moved from CPU 0 to other processors within MELLON1 to make room for FOX.

Phase 3

The MELLON2 development system was moved to its new location beside the planned MELLON0 site. Since MELLON0 would not have an Operations Services Processor (OSP) until the original was disconnected from MELLON1, this allowed Tandem customer engineers to use the MELLON2 OSP for diagnostic testing.

Phase 4

Phase 4 included several critical subphases. First MELLON0, the core of the new production system, was installed. It included a patch cabinet, an I/O-only cabinet, and a NonStop TXP system cabinet. Along with its 4 NonStop TXP processors, MELLON0 required the purchase of the following additional hardware: one pair of 3106 disc controllers, a 6303 asynchronous controller, a disc patch panel, an asynchronous patch panel, a 4114/4115 mirrored system volume, two 6530 terminals, and an OSP cable to be connected to the MELLON2 OSP.

MELLON0 did not contain a tape drive. Rather, a disc drive containing an appropriate system image was installed, and the system was COLDLOADed from disc.

After MELLON0 was thoroughly tested, the LBU boards were installed in I/O slots 1 through 6 under CPU 0. This provided a dry run for the FOX installation on MELLON1.

Phase 5

Next, LBU boards were installed on MELLON1. This required only a single processor outage (CPU 0) and took down one IPB at a time, allowing processing to continue as normal. The connection between MELLON1 and MELLON0 was then tested. Figure 6 shows the Mellon network at the end of this phase.

Phase 6

All I/O controllers and associated devices attached to CPUs 4 and 5 were moved from MELLON1 to MELLON0.

Phase 7

The MELLON1 NonStop II system cabinet, originally containing CPUs 4 through 7, was moved to MELLON2, replacing the NonStop TXP cabinet freed up in Phase 1.

Phase 8

The NonStop TXP cabinet was then moved to MELLON0 to comply with FCC requirements for pure NonStop TXP systems. (This was done to prepare for a possible future upgrade of Mellon's remaining NonStop II processors to NonStop TXP processors.)

At the end of this phase, there were 8 CPUs on MELLON0 and 4 CPUs on MELLON1.

Phase 9

All I/O controllers and devices remaining in MELLON1 were moved to MELLON0.

Phase 10

The 2 remaining MELLON1 system cabinets were installed as a patch-panel cabinet on MELLON0 and an I/O-only cabinet on MELLON2.

Figure 6

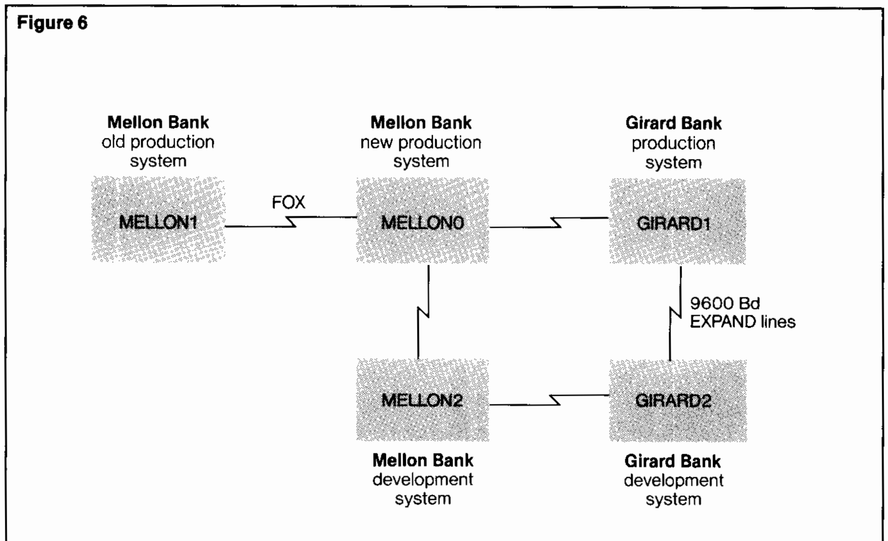

Mellon Bank old production system
MELLON1
FOX
Mellon Bank new production system
MELLON0
Girard Bank production system
GIRARD1
9600 Bd EXPAND lines
MELLON2
GIRARD2
Mellon Bank development system
Girard Bank development system

Figure 6.
Mellon Bank's CASH-STREAM Network at the end of Phase 5 of the move.

Conclusion

The success of FOX in the Mellon Bank system move demonstrates that it is more than a high-speed hardware extension to the DYNABUS for permanent configurations: FOX can also be used to move high-volume, critical on-line transaction processing systems with no noticeable down time. FOX adds a new level of meaning to the Tandem architectural features of geographic independence and modular expandability.

Significantly, Mellon Bank noticed no degradation in performance while operating during the move. This allowed the move to be performed at a "comfortable" pace, instead of being rushed through to regain an acceptable level of performance. (As mentioned above, at the end of Phase 8, Mellon temporarily applied the CPU power of 4 TXPs and 8 NonStop IIs to an application that previously ran on 8 NonStop II processors. Along with their CPU cycles, the 4 residual NonStop IIs added 20 Mb/second of aggregate I/O channel capacity and actually enhanced the system's ability to respond to peak loads and survive the loss of peripheral devices.)

Of course, the relocation of hardware described above required corresponding changes in the configuration of both system and application software. In most instances, the software changes reflected the hardware moves, but were much more easily accomplished (e.g., with a COLDLOAD or PUP UP/DOWN) than their hardware counterparts.

A small part of the cost savings in this move lay in the availability of the MELLON2 development system. This allowed the customer engineers to borrow its components, eliminating the cost of purchasing redundant parts. Since users running critical on-line applications like Mellon's often use development and test systems in addition to their production systems, they should find an approach similar to this one useful when planning to move their systems without an interruption of service.

References

Communications Management Interface (CMI) Operator's Guide. Part No. 82344 B00. Tandem Computers Incorporated.

GUARDIAN Operating System Tables Handbook. Tandem Computers Incorporated.

System Management Manual: NonStop II System. Part No. 82069 B00. Tandem Computers Incorporated.

Acknowlegments

The writer acknowledges Fernando Alarcon (District Hardware Specialist), Jim Hilinski (Senior Customer Engineer), and Mike Krygowski (Senior Account Analyst), all of the Pittsburgh District, for their contributions to the planning, execution, and documentation of this project.

Craig Breighner is a Senior Account Analyst in Tandem's Pittsburgh Commercial Branch. After joining Tandem in March of 1983, one of Craig's first assignments was as account analyst for Mellon Bank. The strategy for moving Mellon Bank's Tandem systems was first championed by Craig, although the project's success reflects the combined efforts of the entire Tandem account-support team.

TANDEM PUBLICATIONS ORDER FORM

The *Tandem Systems Review* and the Tandem Application Monograph Series are combined in one subscription. Use this form to subscribe, change a subscription, and order back copies.

For requests *within the U.S.*, send this form to:

Tandem Computers Incorporated
Sales Administration
19191 Vallco Parkway
Cupertino, CA 95014-2599

For requests *outside the U.S.*, send this form to your local Tandem sales office.

Check the appropriate box(es):

□ New subscription (# of copies desired ________)
□ Subscription change (# of copies desired ________)
□ Request for back copies

Print your current address here:

____
ADDRESS

____

____

____

____

____
ATTENTION

____
PHONE NUMBER (U.S.)

If your address has changed, print the old one here:

____
ADDRESS

____

____

____

____

____
ATTENTION

____
PHONE NUMBER (U.S.)

To order back copies, write the number of copies next to the title(s) below.

NUMBER OF COPIES

Tandem Journal

________ Vol. 1, No. 1, Fall 1983, Part No. 83930

________ Vol. 2, No. 1, Winter 1984, Part No. 83931

________ Vol. 2, No. 2, Spring 1984, Part No. 83932

________ Vol. 2, No. 3, Summer 1984, Part No. 83933

Tandem Systems Review

________ Vol. 1, No. 1, February 1985, Part No. 83934

Tandem Application Monograph Series

________ *Transaction Processing on the Tandem NonStop Computer: Requester/Server Structures,* January 1982, SEDS-001

________ *Designing a Network-Based Transaction-Processing System,* April 1982, SEDS-002

________ *A Close Look at PATHWAY,* June 1982, SEDS-003

________ *A Multi-Function Network for Business Automation,* May 1982, SEDS-004

________ *Developing TMF-Protected Application Software,* March 1983, AM-006

________ *Integrating Corporate Information Systems: The Intelligent-Network Strategy,* March 1983, AM-007

________ *Application Data Base Design in a Tandem Environment,* August 1983, Part No. 83903

________ *Capacity Planning for Tandem Computer Systems,* October 1984, Part No. 83904

TANDEM
COMPUTERS

Part No. 83934

400095 02/85 Printed in USA