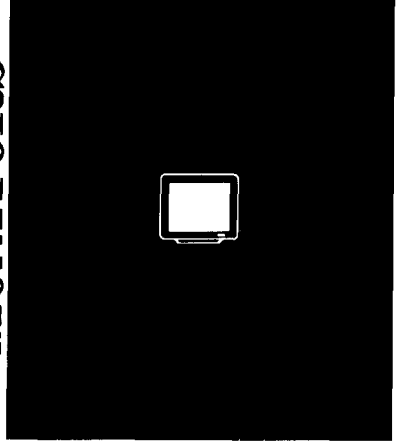
Transaction Processing MONITORS

Philip A. Bernstein

A *transaction processing (TP) application* is a program that performs an administrative function by accessing a shared database on behalf of an on-line user. A *TP system* is an integrated set of products that supports TP applications. These products include both hardware, such as processors, memories, disks, and communications controllers; and software, such as operating systems (OSs), database management systems[1] (DBMSs), computer networks, and TP monitors. Much of the integration of these products is provided by TP monitors, which coordinate the flow of transaction requests between terminals that issue requests and TP applications that can process them.

Today, TP represents over 25 percent of the computer systems market, and is one of the fastest growing segments of the computer business. TP applications appear in most sectors of large-scale enterprises, such as airline reservation, electronic banking, securities trading, inventory and production control, communications switching, videotex, sales management, military command and control, and government services.



Transactions

A *transaction* is a unit of work that executes exactly once and produces permanent results. That is, transactions should be

- *serializable*—the system should appear to process transactions serially;
- *all-or-nothing*—each transaction should either execute in its entirety or have no effect; and
- *persistent*—the effects of a transaction should be resistant to failures.

A "user-oriented request" may require executing several transactions. For example, to process an order, a user may enter the order, request a shipment, and issue a bill, each of which may execute as a transaction. To simplify the discussion, this article assumes that a user-oriented request executes as one transaction.

To ensure serializability, all-or-nothing, and persistence, the system requires application programmers to delimit the boundary of each transaction, (e.g., by bracketing the corresponding application program with the Start-transaction and End-transaction operations). The programmer can cause the system to obliterate an active transaction by issuing the Abort-transaction operation. At execution time, each transaction either *commits* (it executes in its entirety and its results persist) or *aborts* (its effects are undone).

Most of the system's support for serializability, all-or-nothing, and persistence is within DBMSs. Each DBMS uses concurrency control (usually locking) to make its execution serializable. It uses recovery mechanisms (usually logging) to make transactions all-or-nothing and persistent, by undoing the effects of transactions that do not finish, and by writing all of a transaction's updates to the log before the transaction commits.

If multiple DBMSs are accessed by a single transaction, then additional DBMS coordination is needed, often with TP monitor or OS support, via the two-phase commit protocol: To ensure all-or-nothingness despite failures, the first phase of two-phase commit requires every DBMS accessed by a transaction to put that transaction's updates on stable storage (e.g., disk). After all DBMSs acknowledge phase one, each DBMS is told to commit the transaction.

Concurrency control, recovery, and two-phase commit mechanisms are well-documented in the literature, and are not discussed further in this article (see [4]).

A typical TP application contains relatively few transaction types—sometimes less than ten, usually less than a few hundred. The resources required by a transaction can cover a broad range. Typically, it uses up to 30 disk accesses, up to a few million machine language instructions, and up to a few dozen network messages. Today's largest TP systems have about 100,000 terminals and 1000 disks, and can sustain several thousand transactions per second at peak load. Many TP systems, large and small, are distributed, consisting of multiple nodes that can process transactions.

The TP Computing Style

Most of the attention on TP technology in the technical literature has focused on database aspects of TP. Although databases are quite important to TP, the database view of TP is incomplete, because performance and ease-of-use are also much affected by the OS and its integration with communications.

Unless a computing platform is specifically designed for TP (and few of them are), there are likely to be many inefficiencies and inconveniences in implementing TP applications. The reason is that TP is a style of computing different from other standard computing models: batch processing, time-sharing, and real-time. Most importantly, TP systems support the transaction abstraction, which is absent from the other three models. The transaction is the basic unit of computation, different from the "process" or "task" abstraction supported by the underlying OS. TP differs in other ways too.

In classical batch processing, a batch of sorted transaction requests is merged with an input master file to produce a new master file. TP differs from classical batch processing in its need to support a large number of terminals and active users, associative and random access to files, and fine-grain failure handling.

In classical time-sharing, a user logs in from a terminal and executes a sequence of requests that is unpredictable from the system's viewpoint. TP differs from time sharing in the regularity of its application load, the intensity of database management and communications over computation, and its requirement for very high availability.

In real-time systems, fast response time is the main goal. Due to its on-line response time requirements, TP is essentially a database-intensive real-time system that supports the transaction abstraction.

TP Monitors

Through the early 1970s, building a TP system was a roll-your-own activity. Computer vendors supplied hardware, an OS, and sometimes a DBMS, although often the latter could not meet the response time requirements of TP. The vendor's system was usually designed for batch, time sharing, or real-time. The customer had to modify it into a platform suitable for TP.

Today's customers expect far

[1]For the purposes of this article, "database management system" is used in its broadest sense, covering any manager of shared data, including block- and record-oriented file systems.

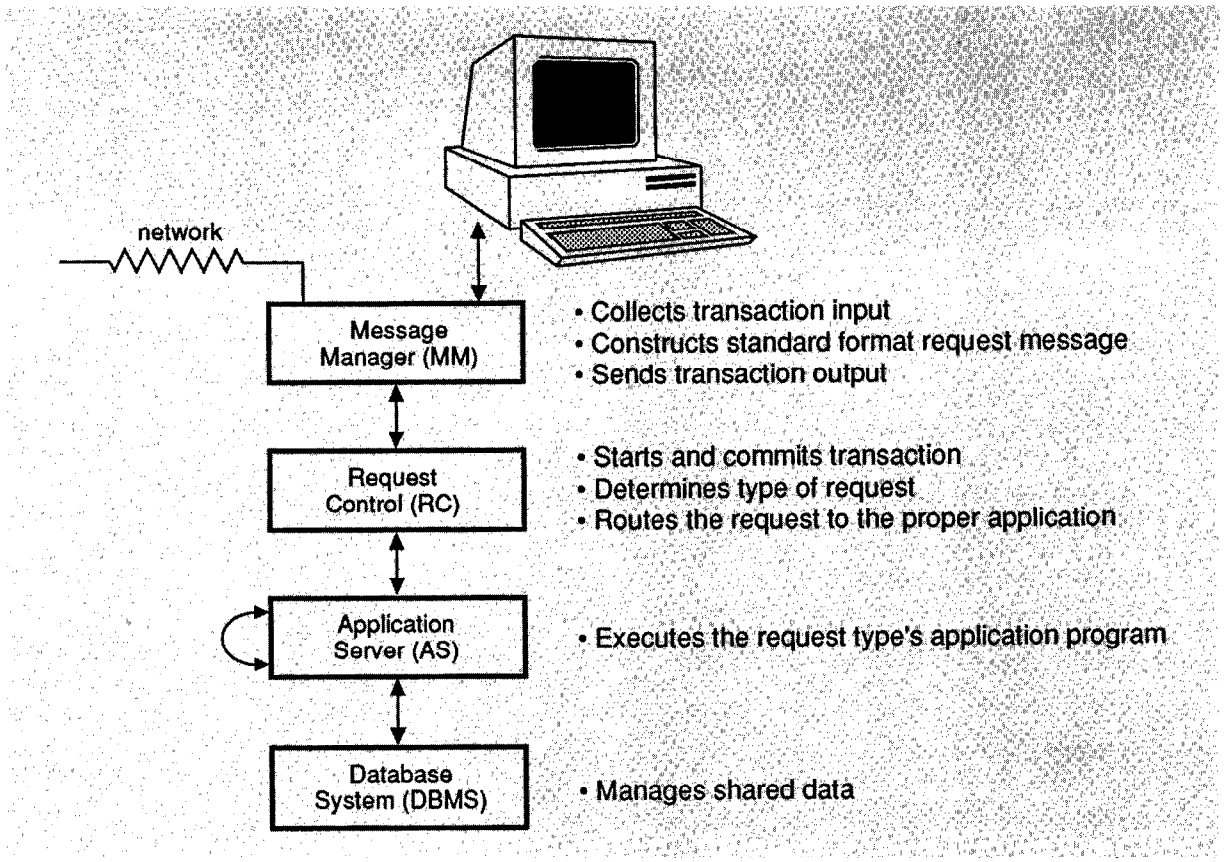


FIGURE 1. A Model for TP Monitors.

more. The vendor must supply a full complement of integrated basic software, including a high-performance DBMS, communications system, and TP monitor. Application builders expect the TP monitor to offer a seamless integration of the basic software. The TP monitor should manage processes, should provide an interprocess communication abstraction that hides networking details, and should help system managers efficiently and easily control large networks of processors and terminals. In this sense, a TP monitor is a combination of "glue" and "veneer"—glue that ties together independent components and a veneer that provides a single, integrated interface to those components.

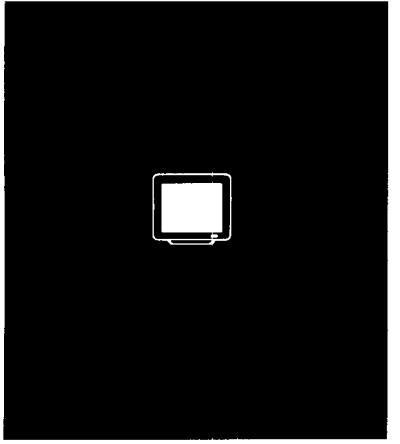
There are several ways to structure a TP monitor to provide these functions [2]. In the following section, a model for these structures is presented. All TP monitors of which the author is aware conform to this model. The functions of each component are discussed in the section entitled "TP Monitor Functions." A major aspect of TP monitor functionality is the way it maps the model's components into OS structures. This is described in the section entitled "Process Management." Queuing functions that provide fault-tolerant message passing are described in the "Queuing" section, followed by a discussion of system management and application recovery in the section entitled "System Management and Recovery." The article concludes by showing how distributed computing features of new OSs are subsuming traditional TP monitor functions, and how TP monitors are likely to evolve as a result.

Throughout the article, it is explained why vendors choose one structure over another, using popular commercial products as examples, such as Digital's ACMS [9] and DECintact [8], IBM's CICS [11, 16, 24] and IMS/DC [15], and Tandem's Pathway [21] TP monitors. This is not an exhaustive list; most commercial computer manufacturers offer a TP monitor product.

TP Monitor Architecture

The main function of a TP monitor is to coordinate the flow of transaction requests between terminals or other devices and application programs that can process these requests. To accomplish this, the TP monitor imposes a certain structure on the software components of a TP system and offers functions to support the activities of each component. In this section, TP monitor structure is described. Later sections examine the support functions more deeply.

The vast majority of TP applications are structured to perform the following steps for each terminal:



(1) Interact with the terminal user to collect the transaction's input, usually through forms and menus.
(2) Translate the transaction input into a standard-format request message.
(3) Start the transaction.
(4) Examine the request's header to determine its type.
(5) Execute the request type's application program, which may in turn invoke DBMSs and other application programs.
(6) Commit the transaction after the application has finished.
(7) Send the transaction's output to the terminal.

All TP applications can be structured to follow this control flow. A TP monitor divides an application into components that perform the above steps (see Figure 1):

- Message Manager (MM)—performs steps (1), (2), and (7).
- Request Control (RC)—performs steps (3), (4), and (6).
- Application Server (AS)—performs step (5), in collaboration with DBMSs.

A particular system typically has many instances of MMs, RCs, ASs, and DBMSs. These instances follow a communication paradigm imposed by the TP monitor: MMs communicate with RCs, which communicate with ASs, which communicate with DBMSs and with each other. This communication paradigm is consistent with the flow of events in the seven-step procedure of the previous paragraph. By decomposing the application in this manner, the TP monitor can simplify application programming by mapping these components into OS processes and by providing communication support between components. It also provides system management operations to monitor and control performance, faults, and security.

Despite the importance of these system management operations, this article contains relatively little about system management, in the section entitled "System Management and Recovery." Instead, this article will focus on the components that directly affect the execution of each transaction—message management, request control, and application servers.

TP Monitor Functions

Message Manager

A Message Manager (MM) must interact with a variety of terminal types to collect input and display output. It performs four main functions: it formats requests, manages forms, validates input, displays output, and performs user-oriented security checking. Each function is described in the following paragraphs.

Since terminal technology changes frequently, today's TP systems have a mix of terminal types, such as character-at-a-time terminals, screen-at-a-time (i.e., block mode) terminals, and personal computers. There are many industry-specific variations of these devices, such as bar-code readers, automatic teller machines, and point-of-sale terminals. For the purposes of this article, "terminal" is used to describe all these devices.

To isolate Request Control (RC) from the diverse interfaces provided by these devices, an MM translates each input that asks to run a transaction into a standard-format *request message* or, simply, a *request*. RC can count on receiving its input in standard format. This makes RC programs independent of terminal types: such *terminal independence* provided by MMs is akin to the data independence provided by DBMSs, which insulate applications from the variety of physical database formats through a standard database format.

The format of requests is defined by the TP monitor. It includes a standard header, which is the same for all applications that use the TP monitor, and a request body, which is defined by the application. The header may include the terminal's address, the name of the user at that terminal, and the name of the request type. This header format varies from one TP monitor to the next. The request body includes the parameters to the transaction.

The *forms manager* is the component of an MM that is responsible for translating between terminal-specific format and standard request format. Each form consists of a set of fields, each field has a set of characteristics, such as a label, a data type, and a representation on the physical screen or window. The forms manager provides an editor for the application programmer to define and modify forms. The forms manager also provides a compiler, which translates a form definition into a translation table and a record definition (see Figure 2). The translation table is the compiled version of the form that is used by the forms manager's run-time system to translate between terminal-specific format and standard request format. The record definition is a high-level language declaration for the form's standard request format, suitable for including in an application program that uses the form.

Often the execution of a request produces output. The output may be displayed by the forms manager, or may be interpreted as a special device command, for example, ask a teller machine to dispense cash.

An MM is also responsible for validating input. It can check that each input is of the proper type (e.g., no alphabetic characters in a numeric field) and that it is in the allowable range of values (e.g., by a table lookup). While performing data validation, an MM ordinarily may not read shared databases that are updated by transactions. This allows it to efficiently execute close

to the terminal and far from the shared database. However, it may use a local snapshot copy of such databases.

In an MM, the application programmer writes forms definitions and data validation routines. The TP monitor does the rest: it compiles forms definitions and does run-time translation of each form into a request.

In addition to request preprocessing, an MM usually performs some security functions. It authenticates each user, by checking a password, and puts the user's identifier in each request the user submits. It may also perform access control, by only displaying menu entries that correspond to request types that the user is authorized to access from the terminal he or she is using. (Certain requests can only be issued from terminals in a guarded area, such as a money transfer room.) Since the OS does not know about request types, and DBMSs typically do not know the terminal from which requests originate, this function must be performed in the TP monitor.

Following is a list defining acronyms used throughout this article (excluding commercial products).

AS: Application Server
DBMS: Database Management System
MM: Message Manager
OS: Operating System
RC: Request Control
RPC: Remote Procedure Call
TP: Transaction Processing

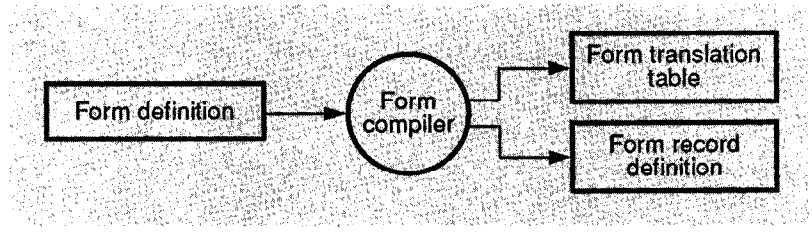


FIGURE 2. A Forms Manager's Compilation Process.

Request Control

Each request constructed by an MM is passed to an RC for processing. The RC is responsible for sending the request to an application server (AS) that can process the request. The application programmer only has to provide a table that relates each request type to the identifier of the AS that can process that request type. The RC does the rest.

The RC looks in the request's header for the symbolic request type, and maps it into the identifier of the appropriate AS. The RC then calls the AS that has that identifier, passing parameters that it extracted from the request. Usually, the RC is responsible for bracketing the transaction, by calling Start-Transaction before invoking any AS, and calling End-Transaction or Abort-Transaction when the transaction is completed.

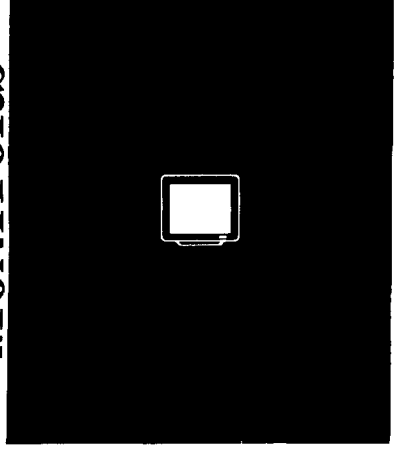
RC-to-AS Mapping. The mapping from symbolic request type to AS identifier should be dynamic. This is helpful for fault tolerance, since it allows the system to quickly remap a request type to a different AS identifier if the original AS fails. It also allows the system manager to redistribute request types to different ASs, e.g., for performance reasons.

A table that is local to the RC provides an easy way to implement this dynamic mapping. If there is more than one copy of the RC, then each copy may have a different copy of this table, leading to a problem: If different RCs control different request types, then a request might arrive at an RC that cannot handle that request's type. There are two standard solutions:

- Every MM knows which RCs can handle each request type. Each MM is designed to send each request, R, to an RC that can handle R's request type.
- Every RC knows which RCs can handle each request type. An MM sends R to any RC, and that RC forwards it, if necessary, to another RC that handles R's request type.

Some systems support a *global name service* [12]. This service maps names into attribute-value pairs, and is accessible from any node. One can use a global name service to map request types into RC identifiers. Since the name service is globally accessible, either MMs or RCs can take responsibility for forwarding each request to the proper RC.

RC-to-AS Binding. To call an AS, an RC must use the AS's identifier to *bind* to that AS. The nature of this binding is determined by the OS and communications architecture. For example, if an RC and AS are bound in the same address space, then an AS identifier could be an



external reference that is resolved by the linker. Alternatively, if an RC and AS execute in different address spaces that are bound through a communications session, then the AS identifier could be a network address that is used to create a session binding.

A global name service may be used to store the mapping from request types into AS identifiers. Since the mapping is globally accessible, ASs can access it directly, without using RCs as intermediaries. In this case, the TP monitor need not distinguish RCs from ASs. That is, the notion of RC disappears.

However, even if the TP monitor does not distinguish RCs from ASs, applications usually retain the distinction. That is, some ASs map request types into AS identifiers (i.e., they perform the RC function), and others execute a request type's application program. This structure tends to minimize the number of bindings, which is important for performance in a distributed system. Typically, each MM binds to one—or at most a few—ASs that perform the RC function. And each AS that does not perform RC functions only binds to those that do. In the absence of this structure, all MMs would have to bind to all ASs, and/or all ASs would have to bind to each other. To minimize rebinding cost, these bindings are usually long-lived.

After an AS completes a call, it may return a result to the RC. The RC typically returns this result to the appropriate MM. Therefore, the RC must be able to map each request back to the MM that sent it. For this reason, each request's header usually contains the identifier of the request's originating MM.

Application Servers

Each AS consists of one or more application program, which typically access a shared database. In addition to linking ASs with RCs, the TP monitor usually provides ASs with access to the terminal that supplied each request. It may also compensate for certain OS limitations, in the areas of process management and communication, which are discussed in the next section.

Process Management

One function of a TP monitor is to define a *process management strategy* for creating and managing processes for MMs, RCs, and ASs. By a process, we mean the OS abstraction consisting of an address space, processor state, and set of resources (e.g., a *task* in IBM MVS, or *process* in UNIX or VAX/VMS OSs). There are several popular process management strategies, which depend on

(1) whether MMs, RCs, and ASs execute together in a single address space or separately in different address spaces (i.e., client-server), and
(2) whether a process has one or more than one thread of control (i.e., single-threaded vs. multithreaded).

We treat each strategy in turn.

Single Address Space

Single-Threading. A simple process management strategy is to create one process for each terminal. Each process executes an image (i.e., load module) consisting of its MM, RCs, and ASs linked together. Thus, a standard intraprocess procedure call can be used by its MM to call an RC, and by an RC to call an AS. That is, the process executes a sequential program of the form shown in Figure 3. This process-per-terminal structure is commonly used in time-sharing systems, where each terminal is assigned a unique process when a user logs into that terminal. It is used in small TP systems too. Unfortunately, it does not scale well; when a system has a large number of terminals, it is inefficient to have a process for each terminal. The inefficiency arises from OS overhead: lengthy searches of the OS's table of process descriptors; too much processor context switching between processes (i.e., swapping the contents of registers, address translation tables, and processor cache memory); too much fixed memory per process; and the potential for too much paging I/O if processes are not fixed in memory. The problem may be compounded in a distributed system, since a terminal may require a

```
do forever

    /* message manager */
    get input from terminal and translate it into a standard
       format request;

    /* request control */
    call the appropriate server program to execute the request;

    /* transaction servers */
    server1: . . . go to last;
    server2: . . . go to last;
                . . . ;

last: end do
```

FIGURE 3. A Program Implementing MM, RC, and AS Functions for a Terminal.

process on each system on which it executes transactions.

Another problem with the process-per-terminal approach is the inconvenience in controlling load. To reduce load, the TP monitor can deactivate terminals by stopping the processes connected to those terminals. However, it cannot easily lower the priority of specific request types, since the set of all processes would need to cooperate to achieve this effect.

Multithreading. One can avoid the overhead of the process-per-terminal approach by having a single process manage all terminals that are connected to a node. Conceptually, each terminal has a private thread of control, but shares its address space with all other threads in that process. These private threads can be implemented by the application. More often, they are implemented by the TP monitor or OS. Each thread in a process must have a private data area for local variables. If implemented by the OS, this data area normally consists of a private stack and register save area; if implemented by the TP monitor, it consists of a local process memory region, indexed by thread. Thus, the system switches its attention between terminals by switching threads. By contrast, in the process-per-terminal model, the system switches between terminals by switching processes. Switching threads is more efficient than switching processes, because switching threads does not involve replacing address translation tables or processor cache memory.

The term *service call* is used to denote a call to an OS service, including communications and database functions implemented in the OS. Two aspects of service calls require special treatment when implementing threads: context switching and synchrony.

First, if a service call cannot complete its work immediately, it will change the processor context, so that another program can execute until the service call completes. For example, it may change the program counter, a register pointing to a stack, or a pointer to an address translation table. In this case, the processor context of the calling program must be restored on the return from the call.

Due to the possibility of context switching, each service call must identify the thread that issued the call, so the call can return to the proper thread. If threads are implemented by the OS, then the OS can implicitly attach a thread identifier. If threads are implemented by the TP monitor, the OS regards the service call as a call from a process, not from a thread. So the thread identifier must be an explicit parameter to the service call. When the OS returns to the process after a call, the TP monitor code in the process passes the return to the proper thread.

Second, a service call may be *synchronous*, meaning that the caller stops executing until the callee finishes and returns to the caller. A synchronous service call may become blocked, because it is dealing with an external agent that will not immediately perform the operation (for example, a call to a DBMS, which can block while waiting for a disk I/O, or a call to receive a message, which can block until the message arrives). A synchronous call by a thread may cause the thread's whole process (including other threads) to block, even though only one thread is waiting for the result of the call. There are two solutions to this problem:

(1) Implement every service with an asynchronous interface, so the caller is not blocked. The caller receives the return as a message or software interrupt (e.g., a *signal* in UNIX OSs, or *asynchronous system trap* in the VMS OS).

(2) Implement multithreaded processes in the OS. The OS knows the identity of the thread that makes each synchronous call, so it can block that thread without blocking other threads in the same process.

In the above, (2) is a more general solution than (1), because it solves the problem for all services. In (1), a special asynchronous interface for each service type is required.

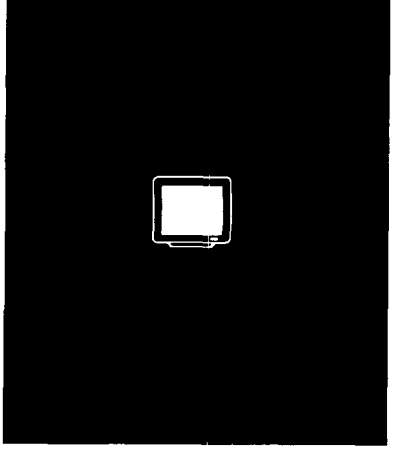
On shared-memory multiprocessors, OS-based multithreading has another advantage: different threads of a process can execute concurrently on different processors. If multithreading is implemented by the TP monitor, one must create multiple (perhaps multithreaded) processes to get this physical concurrency.

Multithreaded processes have two main disadvantages: First, they have weaker protection than single-threaded processes, in that all threads can access the processes' memory. This problem can be mitigated by using a stack-based machine architecture (where each thread has a private stack for local data) and by using a strongly typed programming language (to ensure that programs do not make stray memory references). Second, the system now has two levels of scheduling—processes and threads within processes. This makes it difficult to adjust scheduling parameters to obtain the desired relative priority of threads in different processes.

On balance, the benefits of threads outweigh the disadvantages in most systems. Nearly all TP monitors that use a single address space also use multithreading (e.g., IBM's CICS and Digital's DECintact TP monitors).

Inter-Process Communication

For efficiency reasons, TP systems are often configured as distributed systems. For example, a TP system may have a large, geographically-distributed terminal network. Since there is generally more communication between a terminal and its MM than between an MM and RCs, it is efficient to put each MM in a computer near the terminals it serves; if the "terminal" is a workstation, it probably has its own MM. However, since RCs and ASs are



shared across the entire network, these functions may be remote from the MMs.

A TP system may also be distributed for manageability reasons. For example, a TP system may include several subsystems, each dedicated to request types that are relevant to one division of an enterprise. A large bank may have separate TP systems to process checking accounts, credit cards, loans, and trust accounts. In such a system, a request that originates in one division may require running ASs in another division. For example, a request to pay a credit card bill from a checking account may require running an AS on the credit card division's system and an AS on the retail banking division's system.

These examples—and distributed computing environments in general—pose a problem of interprocess communication: How does a program in one address space (i.e., process) call a program in another address space (i.e., process)?

Message Passing. One popular approach is connection-oriented message passing. A process establishes a connection (i.e., a session or virtual circuit) with another process, after which the processes can exchange messages.

This approach is used in IBM's CICS TP monitor, using SNA LU6.2 [10]. A process establishes a half-duplex connection, called a *conversation*, with a process on another system. Each process can send and receive messages over the conversation. To control the half-duplex connection, when a process is finished sending, it explicitly tells the other process that the latter may now send.

Conversations are intended to be long-lived, spanning many transactions. Consider a set of processes where there is a path of conversations connecting every pair of processes in the set. All processes in the set are implicitly executing within the same transaction. Each process independently tells when it is finished with its part of the transaction; at this point, the process is blocked. When all of the processes say that they are done with the transaction, the transaction commits. Then all of the processes concurrently begin executing the next transaction. This programming model is sometimes called *chained transactions*, because each process begins executing a new transaction when the previous transaction commits.

The main benefit of this approach is that it imposes little structure on message exchanges. For example, programs can communicate using a request-reply paradigm, or they can pass long data streams. The LU6.2 version of this approach has another benefit; it exploits the half-duplex communication style to minimize the number of messages required to control this distributed execution.

Using connection-oriented message passing, programs in different processes communicate using a different mechanism (message passing) than within a single process (local procedure calls). There are two main problems with this approach. First, it complicates the application-programming interface, since the application programmer uses different syntax and semantics for calling local procedures and remote processes. Second, it makes application programs dependent on the assignment of functions to nodes of the distributed system. For example, if an RC was programmed to call an AS in the same address space, and the AS is moved to a different node, then the RC must be modified to call the remote AS.

Remote Procedure Call. The disparity between intraprocess and interprocess communication can be hidden by making interprocess message passing look like procedure calls to the application programmer. This avoids modifying programs whenever a process is moved from one machine to another. It also avoids certain common programming errors. For example, suppose a client sends a message to a server, but the client forgets to wait for the reply. Replies accumulate until an overflow condition arises. Or, suppose a client gives up waiting for a reply from a server, deciding that the server must be dead. If the server is merely very slow and ultimately does reply, the client may no longer be able to cope with that reply and may malfunction [13].

Remote procedure call (RPC) is a mechanism, implemented by the OS or TP monitor, that makes message passing look like procedure calls [7]. In an RPC, a *client* process issues what looks like a local (synchronous) procedure call to a *server* process. The RPC mechanism translates this synchronous call into an asynchronous message from the client to the server, and then waits for the reply. The client cannot forget to wait for the reply, because the RPC mechanism is guaranteed to do so. The client's RPC mechanism can give up waiting and return with an error message to the client. In this case, it will throw away any replies that arrive late.

When an RPC message arrives at a server, the server allocates a thread for this call, either by creating a new thread or by reusing an idle one. Or, if no threads are available (e.g., the server is single-threaded and is executing another call), the message waits. After the server executes the call, a return message is sent to the client and the thread either becomes idle or is destroyed.

Some RPC designs hide some differences between the programming languages of the client and server. The client and server each have a *stub* program for the server. The client's stub translates the pa-

rameters into a standard, machine-independent format. The server's stub translates the parameters from the standard format into the server's language-specific format.

The request-reply nature of RPC communication can be inconvenient if a server has a lot of data to send back to its client. It could send it back in one big package, but this prevents the client from working on the result until the whole result is available. It could require the client to ask for the data a chunk-at-a-time, but this requires a round-trip pair of request-reply messages from client to server for each chunk. Or, a special mechanism can be designed to stream data back to the client a chunk-at-a-time, without an acknowledgment message for each chunk [14].

Client-Server in TP Monitors

An RPC system manages the problem of locating and invoking remote servers. To fully exploit this capability, one should separate different functions into different processes. In a TP monitor, this suggests that

- MMs, RCs, and ASs execute in different processes;
- Each MM process (a client) calls RC processes (acting as servers); and
- Each RC or AS process (acting as a client) calls AS processes (acting as servers).

Some processes act as both client and server—an RC process is a client with respect to AS processes, and a server with respect to MM processes.

The client-server model is used in Digital's ACMS and Tandem's Pathway TP monitors. In the ACMS monitor, MMs, RCs, and ASs execute in separate processes. In the Pathway monitor, there are two types of processes: *requesters*, which execute MM and RC functions, and *servers*, which are ASs. Both systems support RPC for interprocess communication.

Performance. The main benefits of the client-server model are ease of reconfiguration and ease of programming. The main disadvantage of the client-server model is the expense of message-based communication. In the single-address-space model, MMs, RCs, and ASs call one another using a local procedure call—typically costing under 50 instructions. In the client-server model, these calls are implemented by messages—typically costing 1000 to 10,000 instructions. Recent research has shown that this performance penalty can be greatly reduced [6, 18]. Another overhead in the client-server model is its generous use of processes, which leads to more context-switching overhead than a single-address-space model. This overhead can be minimized using multithreading.

Multithreading. To limit the number of processes, MMs, RCs, and/or ASs may be multithreaded. Multithreading may be implemented by the TP monitor or the OS. If the TP monitor implements multithreading, then the issue of synchronous service calls must be handled. In the single-address-space model, this problem is usually solved by intercepting synchronous service calls in the TP monitor. In the client-server model, the problem is often solved by restricting the use of multithreading and synchronous calls, as follows.

First, the TP monitor implements multithreading for MM and RC processes, but does not allow MM and RC processes to call DBMSs. Thus, the TP monitor does not need to intercept DBMS calls in MMs and RCs. But it still has to intercept receive-message calls by MMs and RCs, to make them asynchronous.

Second, the TP monitor requires AS processes to be single-threaded. Thus, an AS process can make a synchronous service call that blocks. The process is put to sleep, but since there are no other threads in the process, this is acceptable. This avoids having to implement either an asynchronous interface to all service calls or multithreading in the OS.

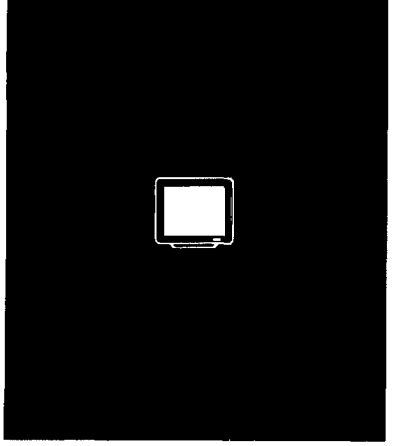
If an AS can only be single-threaded, then it may become a bottleneck. The obvious solution is to have many processes running the same AS program. But now there is a communications problem. When an RC wants to call an AS, to which AS process should it direct the call? What if it sends the call to an AS that is busy with another request? Since the AS is single-threaded, the request will wait until the callee finishes and asks for another input message. This is undesirable if other AS processes are idle at that time.

To avoid this problem, some TP monitors support an abstraction called AS classes. An *AS class* is a set of AS processes that execute the same AS program. A process can send a message to an AS class, instead of directing it to a particular AS process.

The input message queue for an AS class is shared by all AS processes. If a process sends a message to an AS class, that message will be processed immediately if *any* AS process is idle. AS classes are supported by Digital's ACMS and Tandem's Pathway TP monitors.

The issues of synchronous service calls and AS classes arise because multithreading is implemented by the TP monitor, not the OS. If the OS implements multithreaded processes, then the problems disappear. When a thread makes such a synchronous call, the calling thread can block, without affecting other threads in the process. In addition, there is no need for AS classes. Since a thread can send a message to a process, not just to another thread, and since all threads in the process can share the same input message queue, an AS process functions just like an AS class. However, a multithreaded AS process does have weaker protection between threads than AS classes, where the AS processes have independent address spaces.

A system management benefit of



the client-server model arises from the use of AS classes or multithreaded ASs: A system manager can easily control the relative speeds of different request types. When AS classes are used, the speed of that AS is controlled by the number of processes in the AS class. Allocating more processes in an AS class increases the fraction of the processor that is dedicated to that class' AS type. Multithreaded AS processes achieve the same effect, whether threads are implemented by the OS or TP monitor.

Queuing

Another communication problem arises from the fact that clients and servers can fail independently. If possible, the failure of a server should not prevent its clients from making progress. TP monitors help clients cope with server failures by providing queued communications.

It is sometimes impossible to run a transaction as soon as a user enters a request. For example, consider a distributed TP system in which an MM sends messages to a remote RC. If the RC's process is unavailable, due to a failure or overload, then the MM cannot forward the requests that it receives. The MM can either block until the RC is again available, or it can save the requests and forward them when the RC is available.

In many applications, it is unnecessary to run a transaction as soon as a user enters a request. For example, a request by a clerk to ship an order can be buffered for several hours, with negligible loss of service to the customer. As long as the request is not lost, and the transaction eventually runs, the customer is satisfied.

In some applications, it is convenient and cost effective to buffer requests for long periods, and then process the requests as a batch. For example, a retail system can gather information about sales from electronic cash registers during the day, and then run a batch that updates its inventory totals overnight. Batch processing can often be made more efficient than on-line TP, and is therefore preferable when fast response time is unimportant.

In each of these cases, the request produced by an MM may be held for awhile before it is sent to the appropriate RC. Since these cases arise frequently in TP, most TP monitors offer special facilities to manage queues of requests. Each queue has a name and is accessible to MMs and RCs. MMs enqueue requests. RCs dequeue requests and process transactions on their behalf.

Although a user may not need fast response time, he or she may want the system to guarantee that a request will not be lost (e.g., the shipping example above). For this reason, it is important that there be an option to store requests in stable storage, such as a disk, before acknowledging receipt of the request to the user. In this case, the MM's processing of a request is essentially a transaction, which must be committed before acknowledging that it is done.

Additional reliability is attained if each transaction that executes a request dequeues the request within its transaction. If the transaction aborts, the dequeue operation is undone. Thus, the request is automatically restarted by the next RC that dequeues the request. If the queue is in main memory only, then this approach guards against losing the request due to a transaction abort. If the queue is stable, it also guards the request against losing the contents of main memory (e.g., if the OS crashes). This style of operation is typical in the IMS/DC and DECintact TP monitors.[2]

A similar effect for guarding against system failures can be obtained if the TP monitor logs all messages from an MM to an RC (an option in CICS). In a stable database, the application squirrels away a description of each transaction it executes. If the system fails and subsequently recovers, the application's recovery procedures can compare the message logs to information about committed transactions that it saved before the failure, so it knows which requests were submitted before the failure but did not execute.

A transaction can enqueue output that is destined for a user. But to guard against losing the output in the event of failure, the MM transaction that dequeues the output must not commit until it is sure that the user actually saw it. Again, a message log can substitute for a queue.

Some requests require the execution of more than one transaction. To avoid losing information if the system fails after some but not all of a request's transactions have executed, each transaction can pass its results to the next transaction via a queue. Technical details of this approach appear in [5].

Queuing systems usually incorporate scheduling features. For example, each request may be assigned a priority by the MM that enqueues it. An RC can then dequeue requests based on that priority, or perhaps based on other fields in the request. An application can explicitly scan the contents of request queues, to find especially important requests that should be expedited. Or the TP monitor may offer a *scheduler* that sits between a queue and RCs and explicitly assigns requests to RCs based on the scheduling criteria.

Given that requests are buffered in queues, the length of queues is a natural measure of system backlog, which can be made available to the system manager.

The main disadvantage of re-

[2]Notice that this approach usually requires two-phase commit, since the transaction accesses two DBMSs—the queue manager and ordinary DBMSs. Two-phase commit is avoidable if the queue manager and DBMSs share a common recovery log.

TP monitors provide the "glue" that binds the many software components of a TP system through their support of multithreaded processes, interprocess communication, queue management, and system management.

quest queues is performance. It generally is more expensive for an MM to enqueue a request and subsequently for an RC to dequeue it, than simply to send the request directly from MM to RC.

Most TP monitors offer queuing services. In some TP monitors it is an optional feature, as in IBM's CICS and Digital's ACMS TP monitors. In other TP monitors, it is the main communication technique, as in IBM's IMS/DC and Digital's DECintact TP monitors.

System Management and Recovery

System managers require on-line tools to monitor and control all aspects of a running TP system, including performance, failures and security. These tools gather data and adjust parameters in many component subsystems. For ease of use, the monitor and control functions of subsystems should be well-integrated into a seamless interface. This is especially important for a large distributed system, in which complexity and distributed control make it quite difficult to manage. Personal computers have compounded this problem enormously, since each desktop machine is now an independent node, with a user who wants to treat it as an appliance. System managers also need off-line tools to test early versions of applications, and to analyze data produced by monitoring tools (e.g., for capacity planning and to analyze failures and security breaches).

A TP monitor provides system management operations to manage the set of MM, RC, and AS processes. To do this, the TP monitor maintains a description of the *configuration* of processes in the system. This description includes the terminals and forms attached to each MM, the security characteristics of users, the set of request types routed by each RC, the set of programs managed by each AS, etc. In a distributed system, it also includes the names of the nodes on which each process executes. A system manager can create and destroy processes, move them between nodes, and alter the set of forms and programs used by each process.

The TP monitor can measure the performance of the running system, and offer this information to the system manager in application-oriented terms—transaction rates, response times, etc. The system manager can use this information to adjust the configuration, to improve response time and throughput.

The TP monitor's system management knowledge of the MM-RC-AS configuration is useful for managing failures. If a node fails, the TP monitor can re-create that node's MMs on another node that has access to the same set of terminals (e.g., one that is connected to the same local area network), and can create sessions between those terminals and the new MMs. It can also re-create the failed node's RCs and ASs on another node that can load the appropriate programs and has spare capacity to run the processes. Using its configuration description, the TP monitor can perform these actions without human intervention—either by using predefined backup configurations or by redesigning a feasible configuration at recovery time.

The transaction abstraction and queued requests help make recovery transparent. When a process fails, transactions that were executing in that process abort. After the TP monitor recovers the failed process (possibly on another node), requests that correspond to the aborted transactions automatically restart, as described in the section entitled "Queuing." If this recovery is fast enough, the terminal user sees this failure merely as slow response time. Moreover, this recovery is accomplished almost entirely by the TP monitor and transaction mechanisms, with little or no application programming.

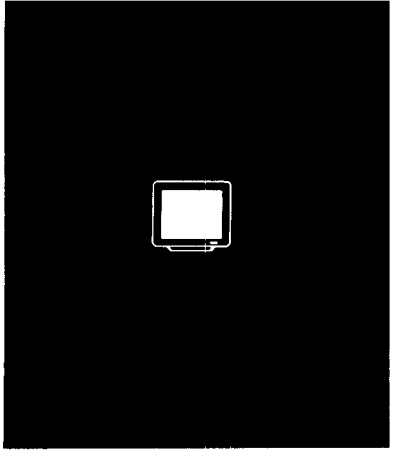
The TP monitor can also perform system management functions related to accounting, security, and capacity planning.

Future of TP Monitors

In this article, it was shown that TP monitors have evolved to solve distributed computing problems that are not solved by the underlying OS, DBMS, and network. In particular, they support multithreaded processes, message routing, queuing, and system management and recovery. Sometimes, they support the transaction abstraction (e.g., the CICS monitor supports two-phase commit).

Many TP Monitor functions are starting to be found in OSs and DBMSs, via name servers and data dictionaries, remote procedure call systems, and OS- or DBMS-supported transactions (i.e., two-phase commit). As such OS and DBMS facilities become popular, the need for these TP monitor functions may diminish. In response to this trend, we can expect TP monitor vendors to offer higher functionality versions of these facilities, to maintain demand.

One positive effect of putting TP monitor facilities into the OS is that all programmers will be able to program using transactions—not just those working in an environment controlled by a TP monitor. Just as today's programmer assumes that the computing environment includes processes, memory management, and files, tomorrow's programmer will assume it includes transaction management and queue management. This will simplify the development of many reli-



able, distributed applications—not just those that fit the traditional TP mold.

The need for TP monitors is likely to increase in the area of system management. With the proliferation of powerful workstations and servers, the complexity of the computing environment is quickly outstripping the ability of system managers to control it. Since TP monitors are already providing many of these management functions today, they are well positioned to fill this rapidly increasing requirement.

Acknowledgments.
This article grew from discussions with many colleagues, I thank them all for their help, especially, Andrew Black, Umesh Dayal, Bill Emberton, Jim Gray, Rivka Ladin, Dave Lomet, Bruce Mann, Murray Mazer, Mike Stonebraker, and Diogenes Torres.

References

1. Acker, R.D., and P.H. Seaman. Modeling distributed processing across multiple CICS/VS sites. *IBM Syst. J. 21*, 4 (1982), 471–489.
2. Anderson, K.J. Bucket brigade computing. *UNIX Rev 8*, 3 (Aug. 1985), 58–64.
3. Anon. et al. A measure of transaction processing power. *Datamation 31*, 7 (Apr. 1985), 113–118.
4. Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.
5. Bernstein, P.A., Hsu, M., and Mann, B. Implementing recoverable requests using queues. In *Proceedings of the 1990 ACM SIGMOD Conference on Management of Data* (May 1990), pp. 112–122.
6. Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight remote procedure call. In *Proceedings of the Twelfth ACM Symposium on OS Principles.* (December, 1989), pp. 102–113.
7. Birrell, A.D., and Nelson, B.J. Implementing remote procedure calls. *ACM Trans. Comput. Sys 2*, 1 (Feb. 1984), 39–59.
8. Digital Equipment Corp. *DECIntact Transaction Processing System: Application Programming Guide.* Order number AA-KZ03B-TE, Maynard, Mass., 1989.
9. Digital Equipment Corp. *VAX ACMS Guide to Transaction Processing Programming.* Order number AA-N691E-TE, Maynard, Mass., 1989.
10. Duquesne, W. LU6.2 as a network standard for transaction processing. In *High Performance Transaction Systems,* D. Gawlick, M. Haynie, A. Reuter Eds., Springer-Verlag, New York, 1989, 20–37.
11. International Business Machines. *CICS/OS/VS Intercommunications Facilities Guide,* Form SC33-0230, White Plains, New York, 1986.
12. Lampson, B. Designing a global name service. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing.* (Aug., 1986), ACM, NY, pp. 1–10.
13. Liskov, B.H., and Herlihy, M.P. Issues in process and communication structure for distributed programs. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems* (October, 1983) IEEE Computer Society Press, pp. 123–132.
14. Liskov, B., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (June, 1988).
15. McGee, W.C. The information management system IMS/VS, Part V: Transaction processing facilities, *IBM Syst. J. 16*, 2 (1977), 148–168.
16. Pacifico, A. *CICS/VS Command Level with ANS Cobol Examples.* Van Nostrand Reinhold Co., New York, 1982.
17. Serlin, O. Fault-tolerant systems in commercial applications. *IEEE Comput. 15*, 8 (Aug. 1984), 19–30.
18. Schroeder, M. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on OS Principles,* (December, 1989), pp. 83–90.
19. Siewiorek, D.P. Architecture of fault-tolerant computers. *IEEE Comput. 15*, 8 (Aug. 1984), 9–18.
20. Siwiec, J.E. A High-Performance DB/DC System. *IBM Syst. J. 16*, 2 (1977), 169–195.
21. Tandem Computers Inc. *An Introduction to Pathway,* Part: T82339, 1985, Cupertino, CA.
22. Wallace, J.J., and Barnes, W.W. Designing for Ultrahigh Availability: The Unix RTR operating system. *IEEE Comput. 15*, 8 (Aug. 1984), 31–39.
23. Wipfler, A.J. *CICS Application Development and Programming.* Macmillan, New York, 1987.
24. Wipfler, A.J. *Distributed Processing in the CICS Environment.* McGraw-Hill, New York, 1989.

CR Categories and Subject Descriptors: C.2.4 **[Computer-Communication Networks]:** Distributed Systems—*Distributed applications;* D.4.4 **[Operating Systems]:** Communications Management—*Message sending, Network communication;* H.2.4 **[Database Management]:** Systems—*Transaction processing*

General Terms: Design

Additional Key Words and Phrases: Client-server, forms management, interprocess communication, multithreading, processes, queuing, remote procedure call, system management, transaction processing, TP monitor

About the Author:

PHILIP A. BERNSTEIN is a member of the technical staff at Digital Equipment Corporation's Cambridge Research Laboratory, and is an architect of Digital's transaction processing products. **Author's Present Address:** Digital Equipment Corporation, One Kendall Square—Building 700, Cambridge, MA 02139. Internet: pbernstein@crl.dec.com.