Software Considerations in the Control of Digital Communications Switching Systems

Ronald P. Ward
Communications Systems Technology, Incorporated
(CSTI)
9740 Patuxent Woods Drive
Columbia, Maryland 21046-1526
(410) 381-5080

KEY WORDS

Communications Software, Concurrency Control, Commercial Databases

ABSTRACT

Today's complex implementations of integrated packet and circuit switched digital communications networks demand that the software used for controlling these systems be robust, fault tolerant, and capable of runtime recovery from all but the most severe of operational errors.

The typical modern switched communications system includes the use of multiple circuit switches, each with potentially thousands of end-user interfaces. Further, these switches are often inter-connected to each other via high-capacity trunks. A single connection between two end-user interfaces often traverses a number of intermediate circuit switches in order to effect the end-to-end communications desired.

In this complex, distributed environment, the establishment and dissolution of end-to-end user connections involve far more than simple binary connection states indicating the existence, or non-existence, of a link. More commonly, a single end-to-end connection requires multiple node links across multiple, heterogeneous interfaces.

The command and control software used to establish, monitor, and dissolve these connections must be capable of dealing with errors which arise at any node along the way in a consistent and reliable manner. Most critically, the system software must be capable of maintaining an accurate, multi-level mapping of distributed resources' availability, allocation, and status. Further, the software must have the capability of "healing itself"

during operational run-time when it can, and of accurately reporting the nature of inconsistencies caused by anomalous events that cannot be fixed on the fly.

The Edwards Digital Switch (EDS), developed by CSTI, provides a case study of possible solutions, and potential pitfalls, that can arise in the design, development, and implementation of the controlling software in today's dynamic, distributed communications' system architectures.

BACKGROUND

When the Control Display Subsystem (CDS) Software was designed for the EDS System, a number of design decisions were made regarding how to keep track of the connection state of the system. Several alternatives were explored, including using global programming data structures, using a commercial database package, and interrogating the actual hardware. Using a commercial database package was eventually settled on for the CDS applications.

Selection of the commercial database approach was based on several factors: 1) a commercial package would minimize the amount of custom code that would need to be written to manipulate the database of connection information; the package selected contained built in support for embedded SQL statements that could be called directly from the source code. 2) the commercial database packages contained built in transaction processing capabilities that were thought to be useful for error recovery situations. 3) the commercial database packages supported concurrency and integrity interlocks which would allow multiple operators to issue connection requests and commands simultaneously.

As we shall see in the remainder of this paper, some of the reasons cited above for selecting the commercial database approach wound up presenting some problems of their own. These problems and some of the solutions to them will be discussed below.

EDS CONNECTION COMMANDS

Most of the operation of the EDS System is predicated upon the ability to establish and dissolve communications connections between a number of differing user interfaces in a wide range of connection modes. The commands for controlling connections within the EDS are usually issued by an operator at a control console. During normal operations, there may be several operators issuing connections commands simultaneously from several different control terminals. When any operator issues a command, the command is preprocessed at the control console, then forwarded to the Network Controller (Netcon)

computer for further processing and execution. Thus, the Netcon computers can often be working on several simultaneous connection commands

The specification for the EDS System directed that there be two different ways of issuing the same commands: real-time and command file (batch) commands. These two major subdivisions of commands can include many types of commands, but I will limit my discussion here to connection commands within these two categories.

Real-time commands are foreground commands where the operator (and the software) waits for the response to the command (success/failure) before being permitted to issue another command. The name real-time is somewhat of a misnomer because not all real-time commands are necessarily of short durations. The real-time nomenclature is used primarily to indicate those commands that are not contained in command files.

Command file commands are background commands that actually consist of a number of connection (and some other types of commands) in a “batch” file. When the operator creates a command file and submits it for execution, the Netcon computer immediately acknowledges the receipt of the command file, and the software implementation permits the operator to move on to another request, perhaps a real-time command, which would then be executed in parallel with the command file.

One of the key differences between real-time commands and batch commands is the approach to error handling and recovery. For nearly every real-time command, the entire command is treated as a “transaction” that can be “rolled-back” if an error occurs. In a command file, each command within the batch job is treated as a separate “transaction”, and while an individual command within a command file may be “rolled-back” if errors occur, the rest of the command file continues its normal execution.

The transaction processing capabilities and the concurrency control aspects of the commercial database package selected for EDS were initially thought to be a perfect match for the database operations required in the above described connection command environment. If an error occurred during one link in a complex connection path, the application software only had to undo the physical link connections made up to the point of the error. The logical connection state of the system could be automatically undone using the transaction processing and roll-back features built into the database.

INITIAL SOFTWARE IMPLEMENTATION

The method for implementing simultaneous connection command requests within the Netcon software consisted of spawning (creating) a new connection command process for each real-time or command file request that was received from an operator terminal. Once

spawned, these processes effectively became independent of each other, and attempted to execute the real-time or command file commands requested.

The individual connection processes perform some validation of the commands, read the database for required information, issue actual device commands to communications switches and user interface devices, process device and switch responses, log errors or special conditions, update the database when successful, and finally notify the operator terminal applications upon completion. When concurrent requests are executing, several connection processes attempt to perform all of these functions simultaneously.

When the initial implementation went into the system integration testing phase, a number of problems began to surface. Many of the connection processes were prematurely failing and issuing what were generically referred to as Database Access Errors. Further investigation into these database access errors uncovered the fact that the competing simultaneous processes were contending for common database records needed to proceed with the connection commands issued. This contention resulted in two specific database access errors: record locking and process deadlocks. Both of these access errors are related to the implementation of transaction processing and concurrency control features of the commercial database package.

The transaction processing features of the database require that certain parts of the database be “locked out” from other processes when one process is using that part of the database. These locks are essential to maintaining the logical integrity of the information in the database. The level of locking granularity is controllable to the database, table, and record locking levels. The CDS software implementation selected record level locking, the least restrictive level of granularity, in an attempt to minimize the impacts on concurrency. In a transaction based application, only the records used by one process acquire locks for the duration of the transaction.

It is worth noting that the competing connection processes actually acquire locks for every record accessed by the process, even if the record is ultimately unchanged by the process. Effectively, this means that every record used by one process, even if only for reading purposes, is unavailable to other competing connection processes until the process possessing the record locks completes the transaction in progress.

The initial database application specific portions of the CDS software recognized this contention problem, and attempted to address the concurrency implications of this problem through the use of record waiting. The commercial database package advertised that a locked record could be waited for by the application process by implementing lock wait statements. The wait statements include a parameter for specifying how long the process should wait for a locked database resource before giving up. The CDS software

implemented this lock-wait strategy in order to avoid the concurrency conflicts described. Unfortunately, after extending the lock-wait time-out values to exceedingly long values, we determined that the lock-wait mechanism advertised by the database package did not in fact work. To the contrary, a connection process requesting a database record that was locked by another competing process did not wait for the locked record at all, regardless of the coded wait-lock time-out value. Instead, a failure value was returned to the locked-out requesting process immediately.

The process deadlock condition mentioned earlier is a further manifestation of the transaction processing and concurrency implementations of the database package. The deadlock situation occurs when two or more competing processes are each making requests for database resources that are locked by another process. In the simplest case, one process has locked record A and needs record B while another process has locked record B and needs record A. It is easy to see that this situation results in both processes being unable to proceed.

The database package was clever enough to detect when the above described deadlock condition arose, and it returned a deadlock error time-out to one of the processes causing the deadlock in order to continue with the other process requests. The recommended solution for dealing with deadlock errors from the database package was to have the process receiving the deadlock time-out error completely rollback its work, and to retry the transaction. Again, this mechanism was put into place in the CDS application software, and again, the mechanism failed to solve the problem.

Many discussions with the developer of the database package were held in attempts to solve the record locking and process deadlock problems cited herein. Ultimately, the vendor told us that the problems we described did indeed exist, and that they would look into addressing these “bugs” in some future release of their package.

The two problems described above combined to cause the failure of simultaneous connection commands and batch commands a large percentage of the time. Since the EDS system was required to provide multiple operator terminals concurrent access to issue connection commands and command files, these database access errors were deemed unacceptable in an operational communications environment. Additionally, relying on the ability of the commercial database developer to supply an updated release of their package with the problems fixed was not a viable alternative in view of the EDS delivery schedules that were in place.

IMPLEMENTED SOFTWARE SOLUTION

The CDS software team analyzed the problems described above and decided that the concurrency control and process synchronization problems would have to be addresses within the application software on the Netcon computers as a "wrapper" to the commercial database interface used within the code. The best mechanism available for ensuring synchronization of the connection command processes was to implement the semaphore facilities of the Inter-Process Communications packages supplied with the UNIX operating system. Semaphores provide globally available flags that allow concurrent processes to wait or proceed, based on the semaphore values. The operating system provided semaphore implementation guarantees the integrity of the semaphores such that two competing processes won't erroneously be allowed to proceed at the same time into an exclusive portion of code.

The semaphore implementation presented certain difficulties that needed to be addressed. Specifically, real-time command requests needed to be able to proceed as quickly as possible without be blocked by a command file process that might be also running. The method employed to implement the semaphore interlocks involved using two sets of semaphores, one for real-time commands and the other for command file commands. The semaphore implementation was "wrapped" around the exclusive database access portions of the software for both real-time command and command file commands.

Whenever a new real-time command is received by the Netcon computers, the command proceeds until it reaches the exclusive database access portion of the code. At that point, the software checks the set of real-time semaphores to determine if the process can continue. If the real-time command semaphores are available, the process acquires the semaphores, effectively locking out other real-time commands that might arrive. When the real-time command finishes the exclusive (transaction) database portion of the code, the semaphores are released to the next real-time process that might be waiting on the semaphore.

When a command file request is received, a second set of semaphores is used. Recalling from the above description of command file command processing that each command within a command file represents an independent transaction, the command file processes check, acquire, and release semaphores for each command within a batch file. Thus, any pending real-time command that arrives after a command file process has begun need only wait the duration of a single command file command before the real-time process can acquire the semaphore locks and proceed.

The semaphore method of connection process interlocking and synchronization provided an alternative to using the transaction processing and concurrency control features of the

commercial database package that proved to be inadequate to the EDS processing requirements. The semaphore mechanism does provide the required level of process isolation, while still presenting the operator terminals with the illusion of complete concurrency.

While the implementation of semaphore interlocking for concurrency control does provide a viable solution to the inadequacies of the commercial database package, there are some drawbacks. First, there is some overhead associated with the implementation of the semaphore wrapper around the exclusive database access portions of the software. This overhead consists of checking, setting and releasing the semaphore locks as needed.

A further drawback of the semaphore implementation is that, for those exclusive database access portions of the software, the connections commands issued by the control consoles become partially serialized. In most cases, this partial serialization of connection commands is not a major problem in that the duration of the semaphore interlocks are relatively short for most real-time commands and for most commands within a command file. Additionally, had the interlocking mechanisms of the commercial database package worked as advertised (using the lock waits and deadlock retries), the net effect on the duration of commands would have been nearly the same.

A final problem associated with the semaphore interlocking strategy employed arises as the result of certain real-time commands that take considerably longer to execute than the others. These real-time command requests are limited to only a few of the possible commands that can be issued at the operator terminals, but they can sometimes involve up to several hundred connection links within the EDS System. These few commands can acquire the semaphore locks for quite a long period of time, and effectively “starve” other connection command processes, most of which are of the much simpler and shorter duration variety.

This problem with the so-called aggregate real-time commands is currently being investigated by the CDS software staff for the most optimal solution. One of the approaches being investigated is to have the aggregate commands acquire and release the semaphores on a sub-command basis, that is, at suitable times during the execution of multiple connections. Another alternative that is being explored is the conversion of the few aggregate real-time commands into command file requests. The ultimate solution to this problem will, in one way or another, partition these aggregate real-time commands such that apparent concurrency is maintained for control console requests.

LESSONS LEARNED

A couple of significant lessons have been learned from the software implementation problems and approaches to those problems presented above. First, the non-critical acceptance of the usefulness of Commercial-Off-The-Shelf (COTS) Software without considerable prototyping of the application environment early in the software design phases of a program can be dangerous. COTS products are often selected and accepted at face value in development programs because of the savings represented by not re-inventing the wheel and using a "mature" commercial package. As was the case with the commercial database package selected for the CDS software application, a COTS product with advertised features, and unadvertised deficiencies, can ultimately have a considerable impact on the desired execution of project software. Further, many COTS providers, upon notification of certain bugs within their product, may not be able to provide fixes or corrected releases in a timely manner.

Secondly, the problems associated with the so-called aggregate real-time commands presented above could have probably been avoided if a more thorough timing analysis of each type of command issued by an operator were done very early in the design phases of the program. A thorough timing analysis of these commands would have highlighted the duration of these commands early on, when addressing the partitioning of these commands would have been considerably easier to undertake.

In summary, this paper has attempted to address a very specific range of software considerations concerning concurrency control and commercial database implementations that can be major issues in a multi-tasking environment. Some of the problems encountered were described, the approaches to solving these problems presented, and lessons learned were discussed.

ACKNOWLEDGEMENTS

Steven M. Bugher

REFERENCES

1. "Statement of Work Edwards Communications Switching System," Attachment 1 to Contract F004611-89-D-0033, AFFTC/PK, Edwards AFB, CA, April 1989.