



Developing Effective, Reliable Aerospace Equipment: The Data Management Challenge

<Written by> Steven T. Graves, McObject LLC </W>

The torrent of data in aerospace applications presents a range of problems. How best can the data be managed?

MODERN COCKPITS bombard pilots with tremendous volumes of data, including tactical information, navigation data, system status, etc. Computerized systems are used extensively to process, prioritize, and present critical data. Necessarily, on-board systems have evolved into substantial computing platforms that are tightly integrated and continuously share information, both internally and with ground-based sources. This torrent of data within aerospace embedded systems presents multifaceted data management requirements, including high performance, concurrent access, high availability, complex searching, and reliability.

One class of commercial, off-the-shelf (COTS) software plays a growing role in helping Military-Aerospace (MilAero) developers meet these challenges: real-time in-memory databases with high availability (HA) capability. While coding custom data management used to be—and for many developers, still is—the industry norm, many firms have found that the performance, reliability and time-to-market benefits of proven databases often justify their cost.

Technological advances have made the use of “real” databases an option in embedded MilAero systems. An in-memory embedded database operates near the speed of RAM access, and eliminates the unpredictable latency accompanying file I/O and inter-process communication. In addition, with “eager, 2-safe” replication implemented via a time-cognizant protocol, in-memory data management offers the unsurpassed reliability of a high availability system with complete redundancy and failover capability, which can be further enhanced by the use of non-volatile RAM (NVRAM).

Performance

The performance requirements and harsh operating environment of airborne systems dictate the use of embedded database systems that operate entirely in memory. The vibrations and high-g conditions largely disqualify the use of

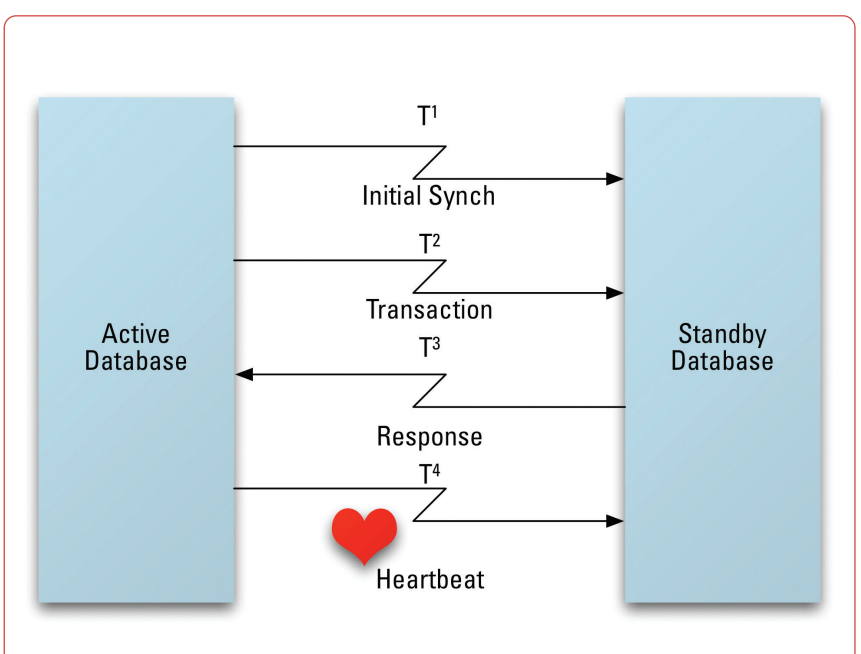
conventional disks due to likely mechanical disruption. Some might ask, “Why not deploy a traditional disk-based database entirely in RAM, to eliminate the hard disk’s role?” The answer is that while RAM-disk deployment speeds up traditional databases’ performance somewhat, such databases’ fundamental reliance on a file system, as well as assumptions built into their optimization strategies, result in much lower performance and less efficient use of memory than databases that are designed to operate in main memory.

Consequently, an embedded in-memory data-

base system will outperform disk-based databases in a RAM-disk by an order of magnitude or more, and will require much less memory to store the same amount of data – 15% to 35% overhead is typical for an in-memory database versus 100% to 1000% for a disk-based database.

Concurrent access

Every successful data management solution should be able to coordinate concurrent access to the data. In other words, two or more processes or threads should be able to read and/or write to the database without concern for the actions >>



T ¹	Time within which initial synchronization must complete
T ²	Time within which Standby must receive entire transaction, once communication is initiated by Active
T ³	Time within which Standby must acknowledge receipt of the transaction
T ⁴	“Keep alive” signal to distinguish quiescent Active from failed Active database

Figure 1: Time Cognizance in High Availability Inter-Process Communication

<< of other processes/threads. The database management system must ensure this isolation.

Even better, the database management system should provide a means for prioritizing access to the database. In real-time systems, certain activities often have higher importance than others. For example, a navigation process would have higher priority than a background process that receives updates from ground-based systems. A database that uses a simple FIFO technique for granting access to database elements will be unaffected by prioritization of processes and threads applied at the operating system level. Developers of aerospace systems invest heavily in real-time operating systems (RTOSs) that support prioritization, but the wrong database can squander this advantage. In contrast, a database that is designed for embedded systems and is “aware” of priorities can complement prioritization at the RTOS level.

High availability

Airborne embedded hardware and software must be able to provide for redundant systems. The benefits of such redundancy range from preventing aborted missions and loss of costly equipment, to saving lives. For data management, redundancy means maintaining two or more database instances in synchronization in an active/standby configuration, with the standby database ready to take over instantly in the event the system hosting the primary database fails.

Ordinarily, the primary and standby database instances will be on separate systems connected only by a communication channel. This necessitates frequent inter-process communication to replicate changes from the primary database to the standby(s). A major challenge becomes ensuring that performance will not be unduly affected by the latency entailed by this inter-process communications. A solution is to introduce time-cognizance into the communication, so that if the active database instance does not receive acknowledgement of its communication by a pre-set deadline, it assumes the uncommunicative standby database has failed, decommissions it, and continues with normal processing. A watchdog process can then recognize the failure and optionally re-boot the system, giving the decommissioned standby database the chance to re-attach and re-synchronize. Time-cognizance in the inter-process communication channel implementation provides predictability in spite of inter-process communication latency.

Reliability

Airborne systems, like other systems on which human life depends, require the utmost reliability. When evaluating a database system’s potential contribution to reliability, consider three key attributes, for starters: Data typing, memory

allocation, and error handling.

C is the dominant programming language of embedded systems. It derives much of its power through the use of pointers. Pointers are a double-edged sword, though, allowing programs to self-destruct if used improperly. One type of pointer, the void pointer, is particularly widely used in database systems. It allows database vendors to create a common programming interface that can be used for any database design (e.g. `WriteRecord((void *) &data)`).

A major drawback of void pointers, however, is that neither the C/C++ compiler nor the database runtime can validate them (that is, confirm they are used correctly). Instead, the “one size fits all” type of interface based on void pointers relies on the programmer to ensure that valid pointers are passed to the database run-time. Unfortunately, there is no guarantee a bug resulting from improperly used void pointers will emerge during testing. It could crop up after deployment, with results ranging from inconvenient to disastrous.

A safer approach – and one that we’ve implemented in McObject’s eXtremeDB – is to build a type-safe programming interface that is generated for each particular data design, when the database definition language (DDL) for that design is compiled. This eliminates the general function (like `WriteRecord()`) and replaces it with functions to write specific types of data, such as `Position_new(&PositionRecord)` or `IFFIdentity_new(&IFFIdentityRecord)` and so on. With a type-safe programming interface, the C/C++ compiler can and will detect invalid arguments and refuse to compile the code until the error is fixed. An entire class of common database programming mistakes is eliminated. A welcome by-product of this approach is more readable and maintainable code: `Position_new()` conveys more information than the general `WriteRecord()`.

Another C programming language feature used liberally by databases is dynamic memory allocation, or allocating system memory to processes on an as-needed basis at run-time. This, too, is powerful but potentially risky. Failure to diligently free (release) dynamically allocated memory when it is no longer needed or out of scope leads to memory leaks that will eventually exhaust available memory, resulting in system malfunction or failure.

Dynamic memory allocation can serve many data management purposes: database dictionaries, cache, transaction buffers, and more. While it seems logical that in-memory databases, in particular, would rely on dynamic memory (and some do), this isn’t a requirement. For greater safety, the in-memory database run-time can delegate memory assignment to the calling application, so that the application must assign a given amount of memory at a given starting address to be used for the database. This cre-

ates maximum flexibility. The application can still allocate the memory dynamically in non-critical situations (entertainment systems in a commercial jet, for example), provided a memory leak will not affect more important systems.

For mission-critical systems, database memory can be treated like video memory in a flat memory model system (think old PC-DOS). A buffer is simply set aside that is dedicated to that particular task. The database run-time can use that memory to store data and all its run-time data structures (transaction buffers, connection handles, etc.) and never have to allocate memory dynamically itself. If the memory given to the database run-time is exhausted, the database run-time can inform the application and let it determine how to resolve the situation (prune the database, or find more memory to dedicate to the database).

Finally, database error handling also contributes to reliability. The database development kit should provide diagnostic capability to help ensure that the software is being used properly. Generally speaking, in a software library such as a database, errors are handled by returning the error to the application through the call stack, or by calling an error handler. For embedded systems, an error handler is the preferred technique. First, the error handler is certain to be invoked, whereas the program may not check function return codes. Second, the error handler can present a default action such as an infinite loop that would automatically cause a watchdog to restart the system.

In critical airborne systems, error handling becomes more than a development and testing concern. In a scenario where the errors are not addressed centrally by an error handler and the programmer has failed to check a return code, a program may ignore a database error code. It could continue operating with false data, with serious unwanted results when the data is relied upon for navigation, targeting or a host of other critical functions.

Conclusion

When considering data management for MilAero equipment, developers and engineering managers must inspect potential solutions at multiple levels. Database architecture must be streamlined and provide the performance needed for real-time systems. Maintaining data availability in the face of hardware or software failure must be addressed, usually with a redundant solution. Finally, developers must understand their database at the programmatic level. This is where details such as error and memory management present hidden challenges to developing and deploying highly effective aerospace applications. <Ends>