Data Management for Military and Aerospace Embedded Systems

A white paper
from McObject

22525 SE 64th Place
Issaquah, WA 98027
www.mcobject.com

Introduction

Modern cockpits bombard pilots with tremendous volumes of data, including tactical information, navigation data, system status, etc. Within modern weaponry, computers 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 and military embedded systems presents multifaceted data management requirements.

In other embedded systems fields – including industrial control, automotive systems, and communications infrastructure equipment – developers have cut programmer months from development and QA cycles by integrating commercial, off-the-shelf (COTS) databases within their applications. But Military - Aerospace systems' critical *performance, reliability and safety requirements* rightly breed caution. Most defense industry developers still laboriously craft their own data management code.

That status quo is changing, due to the emergence of embeddable database management systems (DBMSs) with features for unmatched performance, enhanced durability and High Availability. This white paper examines the data management needs of military and aerospace embedded systems, and focuses on existing and emerging data management technology and its suitability to meet these requirements. Topics covered include:

- **Application scenarios** of real-time, off-the-shelf data management in military and aerospace embedded systems;
- **Performance analysis:** how do different database system architectures, and processes such as disk I/O and caching, affect speed and predictability?
- **Concurrent access** – a database must coordinate two or more threads' access to data. In a defense system, can it distinguish the *mission critical* processes from the *background* jobs?
- **Database programming and reliability**. A database's approach to data typing, memory allocation, and error handling make a big contribution to reliability;
- **Durability**. How does data management survive hardware and software failure? Solutions include non-volatile RAM (NVRAM), transaction logging and recovery, and database replication;
- **High Availability**: When safety requires multiple, redundant databases, how is latency eliminated and synchronization ensured?

Application Scenarios

Combat helicopter

Manufacturers of a leading combat helicopter developed their own embedded data management to manage secure, digitized onboard battlefield data. In ongoing R&D, the company determined that replacing "homegrown" data management with a COTS database offering High Availability support would both reduce development time and improve reliability. As an additional benefit, the COTS database chosen for the task included an extensible markup language (XML) interface that facilitated communications, both internally and between the helicopter and external (ground and air) systems. Use of a COTS database rather than a self-developed solution was credited with cutting 18 programmer-months from the development cycle.

Unmanned reconnaissance aircraft

A high-altitude, long-endurance unmanned reconnaissance and surveillance aircraft was designed to fly at up to 60,000 feet, providing ground-based forces with high-resolution, near real-time imagery of wide geographic areas, and the ability to view and track critical mobile targets. On-board technology includes satellite and line-of-sight communication, electro-optical and infrared sensors, and synthetic aperture radar. However, the large volumes of complex information generated by these systems demanded improved data management for organization and fast storage and retrieval. Designers turned to off-the-shelf data management in order to reduce development time for this task of sensor and communications integration.

Homeland security radio

The developer of a two-way radio system used by first-responders and homeland security agencies sought to replicate central communications database information within field infrastructure units, giving each major system element its own copy of the data and decreasing reliance on links back to a single database server. Major requirements included small footprint, since embedded controllers and other remote equipment included relatively low RAM and CPU capacity. Reliability was also a critical requirement for the two-way radio system, and this need was met by deploying a High Availability COTS database. The chosen database featured an all-in-memory architecture, which enabled it to meet the stringent performance requirements of real-time radio communications.

Ground surveillance

Military planners sought improved "eyes in the sky" ground surveillance through a new, integrated system that mixes manned and unmanned aircraft equipped with radar sensors, and ground control stations. The new technology will provide situational awareness before and during operations, down to the level of detecting, identifying and tracking individual vehicles. Data management requirements ranged from real-time performance to meet the intense demands of war-fighting, to sophisticated data management programming tools—such as support for complex data structures and multiple querying methods—to meet a tight development schedule.

Multi-mission helicopter

A multi-mission helicopter required improved data management within the embedded software in its onboard computer systems. The manufacturer particularly needed to add support for Link 16, the new NATO-standard data link network, which promised to dramatically enhance information exchange among groups of aircraft. Faster, more reliable management and communication of this data would improve the helicopter's ability to communicate and exploit targeting, tracking, and command and control data. Minimal code size and efficient utilization of memory to store sensor, communication and radar data was required, as the on-board system memory allocated to the task was very limited.

Tactical data link integration

A military contractor was enlisted to design software that would act as a "universal translator," integrating radio data communication links (Tactical Data Links) and Internet connectivity on diverse military aircraft, ships and ground vehicles. The technology would enable ships, aircraft and other equipment lacking tactical data links, as well as those with different data links, to communicate with each other. The new integration technology aimed to add flexibility to existing war-fighting IT systems, and to improve the scalability of this infrastructure by enabling the military to insert the level of translation capability necessary for a specific mission. The translator was planned as a bridge between legacy military systems and the Global Information Grid, the Department of Defense's envisioned linkage of every individual in the military, defense and intelligence communities, worldwide. Within the software facilitating this integration, high-speed data management is needed to store and manage independent, synchronized copies of data on network participants and their communication requirements at different nodes.

Laser missile defense system

Computerized weapons systems understandably require a means of restricting access and of providing an audit trail of all user activities. For a laser missile defense system, the manufacturer considered a COTS database alternative to self-developed data management code for this purpose, largely to gain a more sophisticated and bulletproof user tracking system. A commercial database offered proven transaction logging, which records changes to the database. The developers also appreciated the database's 'history' mechanism that simplified detection of tampering by enhancing the ability to see when and how stored data objects have been changed.

Performance Analysis

The performance requirements and harsh operating environment of airborne and field-deployed systems dictate the use of embedded database systems that operate entirely in memory. The vibrations and high-gee conditions largely disqualify the use of conventional disks due to likely mechanical disruption. Some might ask, "Why not deploy a traditional on-disk 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 database system will outperform on-disk databases deployed on 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 common for an in-memory database versus 100% to 1000% for an on-disk database.

One of the most important differences between the databases used by real-time and non-real-time systems is that while the conventional DBMS aims to achieve good throughput or average response time, the real-time database must provide a *predictable* response time to guarantee the completion of time-critical transactions. Therefore predictability depends on avoiding the use of procedures that introduce unpredictable latencies, such as disk I/O operations, message passing or garbage collection. In-memory embedded databases forego disk I/O entirely, and their simplified design (compared to client/server databases) eliminates message passing in the main execution path and, where message passing is required for replication, delivers predictability through a time-cognizant transaction manager. The validity of the real-time data in the database may also be compromised if it cannot be updated fast enough to reflect real-world events. To address this, some means of transaction prioritization is required (see more about this in the following sections).

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 of other processes/threads. The database management system plays the role of a "traffic cop" and ensures 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.

Database Programming and Reliability

Airborne and military systems, like other systems on which human life depends, require the utmost reliability. Data management reliability is secured by a particular database's features for dealing with system failure, which is discussed in the Database Durability and High Availability sections below. Noted less often, though, is that the programming methods permitted by a database can greatly affect whether developers will create bullet-proof, or fallible, code. Likewise, the database's implementation of certain runtime functions affects the likelihood of bugs – and the advantage of software development tools that minimize this risk is self-evident. When evaluating a database system's potential contribution to reliability, consider three key programming attributes, for starters: Data typing, memory allocation, and error handling.

Data Typing

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 *eXtreme*DB -- is to build a type-safe programming interface that is generated for each particular data design, when the data definition language (DDL) for that design is compiled. This eliminates the general function (like WriteRecord()) and replaces it with functions to write the application's 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().

Memory Allocation

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 to be used for the database. This creates 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).

Error Handling

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 mission critical 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.

Database Durability

How does embedded data management survive hardware and/or software failure? In addition to providing predictability and high performance, databases for aerospace and military systems need to run without human intervention, and be able to recover from failure automatically and continue providing data access.

To achieve this durability, in-memory database systems can offer several solutions. An in-memory data store can maintain copies of data, so the loss of RAM and its content does not mean loss of data access and the data itself. In this solution – called database replication – fail-over procedures allow the system to continue using a standby database (see more on this in the next section, High Availability). Other solutions involve using a non-volatile media that helps recover the data in the case of system failure. These different approaches—including synchronous and asynchronous replication, transaction logging and the use of a non-volatile memory device—each have their own performance and resource utilization implications, which must be understood by the developer and taken into account when designing military and aerospace systems.

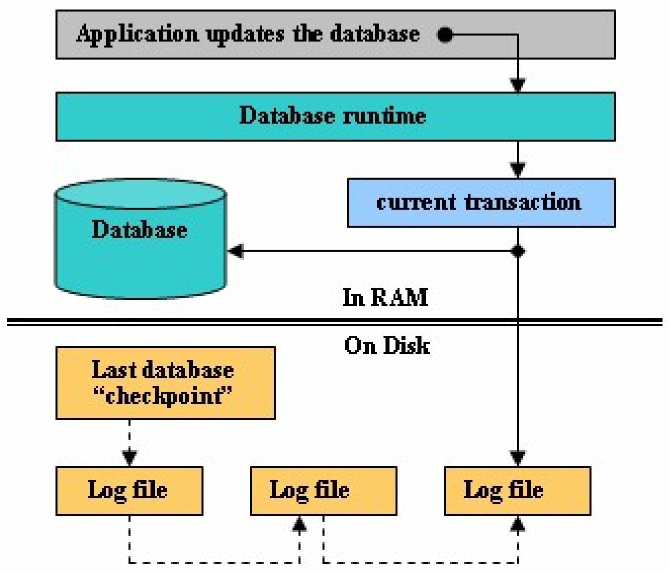
Transaction Logging

A fundamental concept is that within virtually any application, data management is carried out via basic units of processing called *transactions*. A transaction is a sequence of read and write operations on data that are treated together as a single unit of work and must offer the following: atomicity, consistency, isolation, and durability, called ACID properties. In particular, *transaction durability* means that once a transaction is committed successfully, all of the changes to the database are permanent, and must survive subsequent system failure or malfunctions. Thus, when discussing the durability of databases, we refer to transaction durability and the ways database management systems maintain it.

When non-volatile media such as a hard disk or a flash card is present in the embedded system, the in-memory database can be made persistent on this media. If the system fails and is restarted, the content of the database can be restored from the disk via the recovery procedures provided by the DBMS. There are basically two different strategies for accomplishing this: roll-back recovery and roll-forward recovery. With either approach, during normal operation, periodic snapshots of the in-memory database (also called "checkpoints") are taken and stored on non-volatile media. During the roll-back recovery, upon system restart, the content of the last checkpoint is simply loaded into memory. All database changes performed after the last checkpoint are therefore lost.

In roll-forward recovery, checkpoints are also stored regularly, but all intermediate transactions are written to a transaction log. For recovery, the log changes are applied chronologically from the last checkpoint, in the same order as they were entered into the log.

Doesn't this recording of database information to persistent storage introduce exactly the disk access overhead that in-memory databases seek to eliminate? With most commercially available main memory database products, such as McObject's *e*X*treme*DB, transaction logging may be set to different levels of transaction durability, allowing system designers to make intelligent trade offs between performance and risk for lost transactions. For example, the transaction log can be written to persistent storage either synchronously or asynchronously. With the synchronous approach, transaction data is written to disk within the transaction. In this case, all transactions are logged, but the database is locked during this process, which can cause unpredictable delays during the real-time process. With asynchronous logging, writing log data is scheduled to occur when the overall system load is low, or at some other time of the application's choosing, or even by a separate process, in order to reduce performance overhead that could affect the main application process. In asynchronous logging, transaction durability is relaxed, but the overall database responsiveness is higher.



With transaction logging enabled, the database runtime captures database changes and writes them into a "transaction log." Logging is performed through periodic checkpoints, where the image of the database is saved to persistent storage, and all intermediate changes to the database are written to the log files.

Transaction Logging

Generally, transaction logging does not alter the all-in-memory architecture of the in-memory database, and the database retains a performance advantage over on-disk databases. Read performance of in-memory databases is unaffected by transaction logging, and write performance will far exceed that of traditional on-disk databases. The reason is simple: transaction logging requires exactly one write to the file system for one database transaction, while an on-disk database will perform many writes per transaction for data pages, index pages, transaction log, etc. (For the on-disk database, the larger the transaction and the more indexes that are modified, the more writes that will be necessary).

NVRAM Storage

Today, many embedded devices feature NVRAM, a non-volatile memory whose content is saved when a computer is turned off or loses its external power source. NVRAM usually takes the form of static RAM with backup battery power, or an electrically erasable programmable ROM (EEPROM) used to save data. If NVRAM is used to store the database, the database management system can recover the data store from the last consistent state upon reboot. This approach presents a very attractive durability option for an in-memory database. In contrast to the transaction logging approach involving disk I/O overhead, or to the replication approach with its communication overhead, an NVRAM database incurs no disk or network overhead during operation. Until recently, commercial databases have rarely provided direct support for NVRAM data stores. This is primarily because of the cost of high-capacity NVRAM devices capable of holding a database. For some applications, and because an in-memory database minimizes the storage space requirements, the expense can be justified. In these systems, the NVRAM database can provide invaluable performance advantages over other database durability options and therefore justify the incremental expense of NVRAM.

High Availability

Mission critical 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.

Conclusion

Data management for military and aerospace embedded systems is outgrowing its self-developed roots. Managing *more* data demands that the data management software be able to scale. In other words, performance cannot noticeably degrade as data volume scales up from hundreds or thousands of objects to hundreds of thousands, millions, and more. *More complex* data means that data management software must also manage complex structures and relationships. As with real-time operating systems (RTOSes) before, increasingly demanding applications and operating environments are leading defense software developers to seek COTS data management solutions that can meet the requirements of cost-effectiveness, scalability, and the ability to tame complexity.

When considering data management for military and aerospace equipment, developers and system architects must inspect potential solutions at multiple levels. Database architecture must be streamlined and provide the performance needed for real-time systems. 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—and safe—systems.

In-memory database management systems are already widely used in embedded systems, providing the superior performance and predictability often required in real-time settings. Several strategies have emerged for these all-in-memory data stores to achieve high availability and durability of data, such as support for NVRAM databases, on-line backup, transaction logging and database replication. If implemented properly, these approaches enable the developers to adjust the level of durability to achieve the desired throughput and application performance. In-memory databases can achieve predictable response times in the microsecond range required to meet defense systems' performance needs. These databases are designed to operate in harsh environments, with strict requirements for resource utilization, and are ready to provide the performance and reliability required by real-life military and aerospace applications.