Embedded database design for HA

Andrei Gorine, principal architect for McObject, describes the eXtremeDB real time database system.

Database management systems, long a staple of desktop and enterprise applications, are playing increasingly prominent roles in real time embedded systems, in fields as diverse as industrial process control, telecom and network infrastructure, and medical devices. In this new terrain, predictability and performance are two critical requirements.

In contrast to office applications, data processed too late in a real time setting often becomes incorrect or even dangerous, so it is vital for the actual state of the external world and the state represented in the database to be close enough to remain within the limits of the application. In-memory databases have emerged as a popular solution, offering superior predictability and performance through a streamlined design that eliminates disk I/O, caching and related overhead.

By its nature, a database that operates entirely in RAM is vulnerable, in the sense that if the RAM content is destroyed, the database is destroyed with it. In real-time embedded systems, such failure can spell disaster or loss of money. With the increasing number of embedded devices serving real-life processes, there is strong demand for highly available, database-equipped systems that provide service with absolutely no down time. Consider a few real-life examples.

An industrial controller at a paper mill incorporates an in-memory database that holds thousands of measurements made throughout the factory by vibration, temperature, pressure and other types of sensors attached to machinery. Based on these data the controller makes control decisions delivered to the paper-making equipment.

If the controller goes out or malfunctions, at best the paper machine is stopped, at worst it begins to make bad paper or even breaks. With an average paper machine making between $500K to $1million worth of newsprint per day, any interruption is disastrous.

Internet IP routers use in-memory databases to maintain their internal storage subsystems – the routing tables. Carrier-class IP routers often demand high availability for both internal configuration and internal state data.

Router architecture is engineered to meet 'five-nines' availability requirements (99.999% up-time, which equals 5 minutes down time per year) and often provides hardware and software fault-tolerance to support this requirement. The storage subsystem used for building the routing table management is a critical component. IP router failure means interruption of service for millions of customers.

Medical device failure could have even more calamitous effects on those who depend on such systems. Fire or intruder-alarms are yet another example of embedded systems that must be on-line 24/7.

To achieve high availability, an in-memory data store often must offer a means of maintaining copies of data, so the loss of RAM and its content for the database or one of the replicas 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 database. This article examines approaches to database replication in embedded systems, with a focus on introducing replication without downgrading performance.

Within applications, data management is carried out via basic units of processing called *transactions*. This is true whether the database is incorporated in real-time embedded software, or a more traditional office system, because both kinds of application must safeguard data consistency.

A transaction is a collection of operations such as reads, writes, inserts and deletes that together perform a single logical function in the database. Transactions are characterized by ACID properties:

- Atomicity: either all the effects of a transaction appear or none of them do. A transaction is performed in its entirety or not at all.
- Consistency: a transaction takes a database from one consistent system state to another.
- Isolation: effects of a transaction are hidden from other concurrently executing transactions – a transaction is isolated from other ongoing transactions. Only updates of committed transactions are visible.
- Durability: once a transaction is completed successfully, all of the changes it made to the system are permanent, and must survive all subsequent malfunctions.

Database replication is the traditional mechanism for increasing the availability of databases. A replicated database is one in which data items are replicated at different failure-independent nodes or sites. The node at which a transaction was initiated is referred to as a master site. The database system manages the data distributed over multiple nodes, making sure that queries and data updates are executed transparently for the application, even in the case of a node failure.

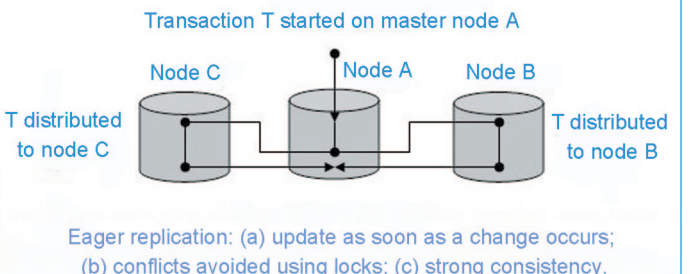
Replica control mechanisms facilitate data propagation between copies. These mechanisms can be categorized according to when updates – changes introduced by transactions – are propagated to all database copies.

Update propagation can be done within or outside transaction boundaries. In the first case, replication is termed *eager* or active. If it occurs outside the boundaries of a transaction, replication is *lazy* or passive.

Eager replication allows the detection of conflicts within a transaction, before the transaction commits. This approach provides data consistency in a straightforward way, and the quickest application recovery time when a fault occurs – see figure 1. No transactions are ever lost in the active replication scheme, there is no overhead associated with node synchronization during fail-over, and the replica database is available immediately. At the same time, active replication has higher processing costs and communications overheads that can increase the response times during normal use.

In contrast to eager replication, lazy replication schemes propagate updates to replica nodes asynchronously and after the transaction commits on the master node – see figure 2. These updates are applied to replica nodes as separate transactions. Compared to eager propagation, lazy update propagation can improve transaction responsiveness by saving on the message round-trip within the transaction. However, since updates are applied to replica nodes asynchronously, replica transactions run the risk of operating with stale data or falsely

Fig1: Eager (synchronous) replication



Eager replication: (a) update as soon as a change occurs; (b) conflicts avoided using locks; (c) strong consistency.

reporting certain data as unavailable if a certain sequence of updates and lookups occurs. For example, a replica application can read a data element that has been removed by the master node transaction, if the read occurs before this transaction is committed at the replica node.

Another major drawback of the lazy replication is a potential risk of losing committed transactions in case of a network or a node failure. Additional algorithms are necessary to regulate replica updates and address these issues. These mechanisms fall into two categories: *lazy group* and *lazy master* schemes.

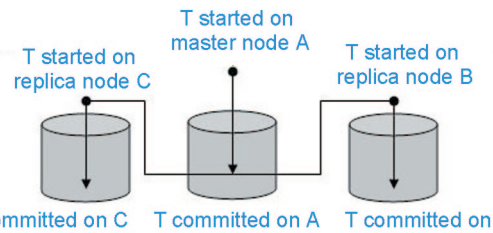
Lazy group algorithms allow for two nodes to update the same data and race each other to install the updates on other nodes. All the conflicting transactions have already been committed, and cannot be rolled back. Therefore, the replication mechanism must be able to detect conflicts and resolve them by executing compensating transactions or update reconciliation.

In lazy-master algorithms, each data element is assigned a master node – the 'owner'. Only the master can update the primary copies of data elements. All other copies of the data element are read-only. The master node stores the correct value of the data element and updates are first done at the master node and then propagated to replicas. Thus the transactions that would have been reconciled in the group replication are forced to wait and can even deadlock.

Due to the lazy updates, the algorithm may cause the update transaction to read stale data, which results in an inconsistent database. To ensure consistency, lazy-master algorithm must employ additional high-overhead concurrency control mechanisms.

Both types of mechanism are often complicated and require additional system resources, which could easily become prohibitive in the harsh environment of an embedded system. Furthermore, in the fail-over situations, lazy replication algorithms must recover lost propagations after node failures. This

Fig2: Lazy replication



Lazy replication: (a) update asynchronously; (b) resolved by 'reconciliation'; (c) weak consistency; (d) complicated algorithms

processing may add substantial delays and decrease overall system availability.

A better choice for an application requiring predictable response times may be data management using *time-cognizant* eager replication protocols. Embedded systems frequently impose strict processing deadlines, and such an approach ensures on-time delivery of the transaction data from master to replica sites. As with any active replication scheme, fail-over procedures are extremely short.

The simplified execution of a transaction using time-cognizant eager replication is summarized as follows. A transaction 'T' is submitted to a master node and the transaction manager assigns a timestamp to it. The transaction performs all reads and writes locally and combines the update into a single message called the 'write set'. When a transaction is ready to commit, the master node sends a message to all nodes that a write set is available, and then the write set is sent to all available nodes. Upon receiving the 'availability' message, each node opens a transaction and assigns a timestamp to it. If the entire write set is not received within a specified timeout, the replica node considers the communication failed, and attempts sending the notification back to the master node.

When the entire write set is received in time, the transaction manager on each node performs updates and tests the set for conflicts. It then sends the result of the commit back to the master node. The transaction T only waits for the commit acknowledgments for a specified time period. Any result received after the timeout is not considered, and the node responsible for this late response is excluded from further processing. With time-cognizant eager replication, transaction predictability is enforced.

Maintaining a replicated database is often a key to achieving high-availability in data-centric embedded applications. However, database replication comes at a price, providing fault-toler-

ance at the cost of storage duplication and increased system resource usage. In addition, adding replicas involves messaging that boosts communications cost and the likelihood of network congestion. Duplicating transactions at each node also increases CPU utilization. In situations where operating resources are extremely scarce, the replication strategies outlined above, using multiple replicas, may be impractical.

Developers can scale down resource demands via a simplified 'hot backup' form of replication. In this approach, the backup copy is located on a different hardware node, connected to the primary node via a common bus, Ethernet or other mechanism. Transaction processing takes place at the primary node and the log of changes made to the primary database is propagated to the hot backup, which reconstructs the state of the primary data store. In the event of the primary node failure, the backup node takes over the transaction processing transparently to the application. Algorithms for maintaining the backup copies can be classified by the strategy used to propagate changes from primary to the backup.

In the 1-safe design, the transactions commit at the primary node and the updates are propagated to the backup at a later time. The system throughput is therefore essentially the same as in the single node system, without a backup. However, the transactions may not be durable: if a transaction did not propagate to a backup before primary failure, it will be lost when the backup takes over the processing. This design corresponds to a lazy replication schema in its reliance on propagating changes to the backup node only after they have been committed on the primary node

In the 2-safe design, the backup system is involved in the commit, and the primary node will not commit the transaction until the backup notifies the primary of the commit via a 2-phase commit protocol. This approach guarantees the durability of transactions and avoids a potential loss of committed data due to primary failure. This design corresponds to the active replication schema.

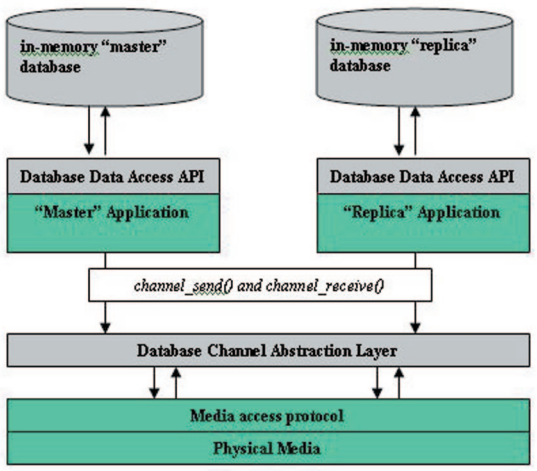
Embedded databases using the primary-backup replication scheme are often available commercially. The approach to choose depends heavily on the application's requirements, operating environment and communication channels. Like other lazy designs, 1-safe can provide better performance but risks a loss of data during failover procedures. While ensuring transaction durability, the 2-safe approach may impose performance degradation due to communication delays. In the real-time environment of embedded systems, it is often helpful to add time awareness to the 2-safe approach, utilizing a time-cognizant eager replication method.

In database replication as elsewhere in embedded systems, communication is a major performance bottleneck. The real-time nature of embedded networks demands highly efficient, deterministic, robust and configurable communications. To answer to these demands, embedded systems use a great variety of media access protocols and transports. While some high-end embedded systems communicate over a VME backplane or similar architecture, there are many that use multiple physical CPUs and require a LAN-based comms bus.

A variety of media access protocols serve as foundations for LANs: traditional connection-oriented protocols; polling (highly popular in the embedded world for its simplicity and determinism); Token Ring networks and Token Bus protocol (which are well suited for manufacturing plants and are adopted by the Manufacturing Automation Protocol, or MAP); a Binary Countdown protocol, used as a foundation for the Control Area Network (CAN) , a high integrity serial data communications bus (originally introduced for vehicle communications); and the TDMA protocol, which is often used in satellite communication, but is also suitable for LANs. Many combinations of the above, as well as proprietary solutions, are also available.

As a practical matter, a database replication mechanism should be able to adopt the communication protocol used for any given embedded application. In addition to having certain properties required for the replication itself, such as time-awareness, and compatibility with diverse media/transport combinations, the database should also provide a way to 'plug-in' various network implementations. Embedded databases are often implemented as libraries of functions with interfaces that can be used to integrate data replication processing into applications. These libraries often introduce the communication channel abstractions that prototype *channel_send ()* and *channel_receive ()* functions.

Fig 3: Communication channel abstraction layer



Communication Channel Abstraction Layer: HA applications communicate data between replicas using a thin abstraction layer over the actual embedded communications protocol

These or similar abstractions allow configuration of the communication channel used by the database engine to fit the embedded application's existing underling transport protocol. The communication channel abstraction is usually a thin layer within a database engine, but is one of the key components since the efficiency can only be achieved when the communication overhead is small.

Many modern embedded real-time systems must remain operational even in the face of failed hardware or software components. For an embedded database, this means surviving the failure of the hardware device on which the database resides or the software environment in which the database operates. Replication of persistent data is crucial to achieving high-availability of data. For embedded systems replication, fundamental requirements include low processing overhead, predictability, efficient fail-over procedures and overall transaction efficiency. Data consistency and performance are often competing goals in the management of replicated data. The correct replication schemes for high availability may suffer performance penalties. Developers need to choose whether to use synchronous or asynchronous approaches to data replication based not only on an application's performance goals during normal operation, but its requirements for data durability, data availability during device failures, and CPU and bandwidth consumption.

References:

1. Bernstein, P.A., V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley, Reading MA., 1987.
2. Gray, J., Reuter, A., Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Francisco, CA. 1993.
3. Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha. The Dangers of Replication and a Solution. SIGMOD, June 1996
4. "eXtremeDB High Availability Addendum", McObject, Issaquah, WA 2003
5. Upender, B., and P. Koopman, "Communication Protocols For Embedded Systems," Embedded Systems Programming, November 1994