APPLICATION NOTE

ACHIEVING HIGH AVAILABILITY USING POLYHEDRA
How Polyhedra's High-Availability mechanisms add resilience to your system and allow upgrades with zero downtime.

Abstract

The Polyhedra IMDB and Polyhedra FlashLite database systems both come with an inbuilt mechanism for setting up a hot standby configuration, with control over failover. This paper describes how this operates, and discusses how this can be used both to provide high availability and also a means to perform field upgrades with zero downtime.

Contents

ENEA

High Availability?

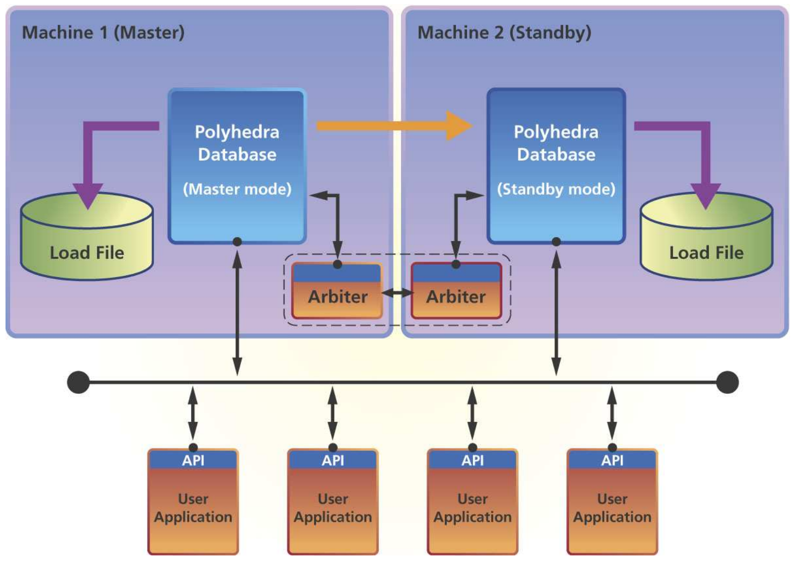
High Availability and Fault Tolerance are phrases that are often used rather loosely, especially when it comes to software. In practice, it is not software that is fault tolerant or highly-available, it is complete systems - but there are certain features or functions that the software components need to supply so as to make it possible to architect an HA solution. Embedded systems requiring High Availability - say, ‘five 9s’ (99.999%) or better for continuous running - are typically configured with redundant cards, power supplies, etc: rather than build systems to be fault-free, it is better to design them to be able to survive the failure of individual subcomponents. This can also give flexibility, as it can allow components to be swapped out and upgraded without upsetting the running system; while live upgradeability is not needed in some HA systems (for example, systems on board aircraft, that will be turned of when the flight is over or the aircraft is serviced), it is often crucial in more complex systems that need to operate continuously for years on end. For example, in telecoms infrastructure equipment a field upgrade should not disrupt calls in progress; in industrial process control systems, stopping a production line for an upgrade might cost tens or hundreds of thousands of dollars in terms of lost production and might have safety issues.

Looking at the data handling needs of HA embedded systems, they often have the following characteristics:

- Data changes fast, but needs to be quickly accessible.
- Data must be preserved in the face of a complete system failure, and must continue to be available in the face of a partial failure.
- The data structures, queries and types of alteration are relatively static in a deployed system…
- … though they will change frequently during application development, and will usually change when deploying new versions of the application.
- Field upgrades should not require scheduled downtime…
- … and the period of vulnerability to component failure should be minimised.
- The system should minimise the amount of application coding that is needed to handle the HA configuration.
- The application should be scalable, avoiding performance killers such as polling.

For transactional data stores, reference is often made to the ‘ACID’ conditions - a mnemonic for Atomicity, Consistency, Isolation and Durability. Atomicity means each transaction either fails (leaving the data in the pre-transactional state) or fully commits - no halfway houses. Consistency says that each transaction leaves the data store in a clean state, with all integrity conditions preserved (and if a transaction were to break this, it would be aborted, leaving the data in its pre-transactional state). Isolation says that transactions are independent, giving the appearance of complete serialisation - where each transaction is completed before the next one is started (though judicious use of locking by the system can avoid the need for the implementation to be quite so restrictive). Durability says that if the data store says the transaction is complete, then the data is already in a ‘safe’ state and will survive a subsequent system failure (within the capabilities of the underlying hardware and environment). In practice, the Durability requirement can significantly slow down a system (flushing files to disk can take some time, for example), so most data stores allow the level of durability to be tuned, balancing it against overall system responsiveness when operating normally. All true relational database systems are transactional, and offer a degree of ACID compliancy; most are SQL-based, and support on-the-fly schema changes, but may be too slow for use in embedded systems or not have suitable HA characteristics.

Polyhedra and HA



The Polyhedra family of relational database systems is designed for use in embedded systems, where state and configuration information has to be kept readily accessible, rapidly alterable… and safe. To ensure fast access, the Polyhedra32 IMDB and Polyhedra64 IMDB products keep the data in RAM, but backed up by a variety of configurable mechanisms such as snapshots, transaction journals, and even a hot standby where appropriate. Polyhedra FlashLite is somewhat different, as it is designed for systems where ultimate performance is less important than reducing the RAM footprint: it shares much of the same codebase as the other products, but uses a file to store the data (supplemented by a configurable RAM cache), with a technique known as shadow paging rather than journal logging to ensure resilience. Like the Polyhedra IMDB products, though, Polyhedra FlashLite also allows snapshots to be generated for offline backup, and supports the use of a hot standby server for high availability.

Looking in particular at the fault tolerance mechanisms in both Polyhedra IMDB and Polyhedra FlashLite, they include the following:

- The hot standby of a fault-tolerant pair of servers is given a complete copy of the database in the master (including the schema and any CL code attached to the database) on start-up, and once it is fully running it is immediately sent a copy of transaction records once each transaction has been successfully committed in the master, so that the standby is kept up to date and ready to take over at a moment's notice.
- The master-standby pair run under the control of an external arbitration mechanism, which controls when failover occurs (and decides which is master when both come up together after a cold boot). When only a few systems are being installed, a software-only system in possible, based on the use of a third machine for arbitration; where systems are deployed more generally, a 2-board solution is needed for cost reasons, with a hardware-based mechanism to assist in determining which board should be master. (Techniques for this vary, but the aim for all of them is to provide a reliable way for one board to know whether it should be master, and to avoid both boards being simultaneously in master mode even if the data connection between the two is not working.) Sample applications illustrating 2-board and 3-board solutions are provide in the standard Polyhedra release kits (and also in the evaluation kits), though for a 2-board solution the example would need to be tailored to interact with the customer-supplied hardware-based arbitrator.

- The Polyhedra client-server protocol includes a client-controlled heartbeat mechanism, so that a communication failure can be detected without too long a delay if the underlying transport is unable to give timely information about the failure of the server. (When TCP/IP is being used, for example, a process failure would normally be detected by the operating system which would then close any open ports - but in the case of a complete board failure it is impractical to wait for the TCP/IP stack at the other end to detect the problem by itself, as this could take up to half an hour.)
- The Polyhedra client libraries allow the user to set up a fault-tolerant connections to a list of servers; if the connection to the master server of an FT pair fails (whether reported by the transport or by the heartbeat mechanism described above, the library will automatically and repeatedly try all the servers in the list in turn (with a configurable delay between attempts, and a configurable maximum retry count). Once the connection is re-established to the now-current master, the library will also re-establish all open active queries, and work out the ‘delta’ between the previous state and the current one. Consequently, only minimal code changes are needed to covert a Polyhedra client application to one that will cope with an HA configuration of the Polyhedra database - basically, it could just be a change to the code that opens the initial connection, though of course (if needed) a client can monitor the changing status of the database connection if it is appropriate, and fine-tune the configuration parameters.
- Polyhedra allows the client applications to control transaction ‘durability’: normally the Polyhedra IMDB server acknowledges a successful transaction as soon as it has been fully committed in-store on the master, but clients can opt to have the response delayed until the transaction has been fully logged to disk, and also applied to - and acknowledged by - the standby, if one is running. (In the case of Polyhedra FlashLite, the transaction is only complete once the shadow pages have been updated on disk, so the durability flag on a transaction only affects whether the acknowledgement to the client is delayed until the standby has been updated.)

Field upgrades

In continuously-running systems, there is often the need to change the software or data structures on the fly, with no downtime. A simple case in point would be the addition of a new type of line card to a telecoms rack in a basestation, say; the new card may be very similar to an existing card, but might need additional configuration information or want to report additional status information. The simplest way of handling this would be for the software on the new line card to check the central database on startup, and if the columns it wants are not present in the tables it uses, it can just create them, using the ‘add’ form of the ‘alter table command’, for example

```
alter table linecard add ( config2 integer default 49, status2 integer)
```

(Alternatively, it could just add the columns one, without checking first if they exist; no harm will be done if they already exist, and the type can be checked when performing queries.) Once these changes have been made, the new line card can then start it application as normal. Provided other clients have not used the ‘select *’ form of query when inspecting the tables, their active queries and prepared statements will not be invalidated by this change, and they will see no interruption in the database service; all that would need to be done would be to update the software on management computers, to allow them to set the new configuration columns and monitor the new status columns.

Note that the heterogeneity built in to Polyhedra, plus a very high level of inter-version interoperability, means that the new line card does not have to be running on the same operating system or even processor type as the cards running the database server, and does not have to be using the same release version of the Polyhedra software.

More complex changes come in two categories: changes to the application software on the control cards or line cards, or changes to the underlying software used by the application software. Let us consider these two cases separately, starting with the easier case.

Upgrading the Polyhedra database software

From time to time, it may be necessary to upgrade the Polyhedra code on a system to a later version: the two reasons for doing this are because a new version of the application software wants to take advantage of a Polyhedra feature that was not present in the currently-used version, or because the new version incorporates a fix for a bug that was adversely affecting the application. In both cases, such upgrades are simplified by Polyhedra's adoption of a set of compatibility principles regarding version interoperability. These are covered in more detail in a separate document, but in summary:

- Old (already-built) clients can connect to the new server
- New clients can connect to older servers, and use those features they provide. (When using ODBC, client applications can interrogate the database to find out the release information and to determine which features are implemented.)
- New servers can read saved database files written be the previous release of Polyhedra
- New servers can act as standby to a master running the previous release of Polyhedra

Thus, to upgrade the control cards in a rack to use a new version of the Polyhedra server code, you would just:

- Instruct the database server to produce a snapshot to a named file; this will cause the current state of the database to be recorded on both systems, just in case of problems with the upgrade. In practice, this is unlikely to be needed, but the belt-and-braces approach should be ingrained in those developing and installing HA systems!
- Stop the standby card, install the new version of the Polyhedra server software, and restart the card. The database server would be told it was standby, and would connect to the master to obtain a new copy of the database. If the new software used a different format for the saved database, it would automatically create a local copy in the new format. The standby would then be able to receive transaction logs from the master, which would be applied to the local database to keep it in step with the master.
- The standby card would now be promoted to master, causing the server on the old master to relinquish control. The old master can then be upgraded in turn, and started up (as standby, naturally).

If it is necessary to upgrade the client software on the line cards, this can be done either before or after upgrading the server code, depending on which is more convenient. In many cases, though, the client software will only need upgrading in the rare event of the application being affected by a bug in the Polyhedra client libraries; there is no need for all the components of the system to be using the same version of Polyhedra. In fact, the client-server protocol is common to all members of the Polyhedra DBMS family, so they share the library code: thus, there would be no need to upgrade the code on the line cards if you were upgrading the database on the control cards from Polyhedra32 to Polyhedra64, say.

Upgrading the application code

If the new application code needs a new version of Polyhedra, it may be simplest to do this first, as a separate stage, using the procedure outlined above. Once the system is up and running the right version of Polyhedra both on the master and the standby, the database schema can be updated; the changes will automatically be applied on both the master and the standby. Polyhedra allows schema changes to be grouped together into a single transaction, by use of the ‘alter schema’ command; the changes are checked for correctness before execution, and if there any problems (such as an incompatibility of column names, say, or lack of room for temporary structure used when transforming the database) the database will revert to its earlier state

Summary

Both the Polyhedra IMDB and the Polyhedra FlashLite servers provide a read-only replication mechanism that can be deployed in a variety of configurations, ranging from the very simple set-up to a complex fault-tolerant fan-out configuration that can survive single-point failures if used on a resilient network. This flexibility does of course mean that some thought might need to be given to find the optimal configuration, but in general where there is a heavy query load (or where some client applications will be launching particularly complex queries) it is straightforward to deploy one or more replicas to improve the overall responsiveness of a Polyhedra-based system.

For more details of the Polyhedra® product, please visit our web site, www.polyhedra.com or www.enea.com/polyhedra, or email us at info@enea.com