Kin Upgrade and DB Redefinitions Management

How Kin's hot-swap DB and Module replacement works.

Basic ideas

System is designed for real 24/7 operation. This means that it should be available in any system state, even when an upgrade is performed. To achieve this, we take advantage of the modularized, clustered architecture: while some modules and nodes are still used to attend currently running jobs, new modules and/or database definitions may be used with all the new jobs, until old ones finish and old code/data may be unloaded. Of course, this can't be achieved for core program, in one node, but may be used in a multinode system (replacing old nodes' core program when it may be idled and all the node emptied for a complete reload).

Module upgrade

Modules are pretty easy to upgrade, as execution control is often at the main core, from where old module code may be unloaded, and new one loaded to replace it. This works smooth if module has no local variables (only external ones, and global heap space), this is the usual case on user modules (task modules), which are changed/added quite often. For system modules, though, it may be necessary to request a data save performed by the module itself, then a data restore from the new loaded module (like a status save and restore), but system modules are seldom changed.

Database upgrade

Changes on database definitions are not rare, in live systems, as new requirements, or simply a design error, require a new field to be inserted, or an existing one to be expanded, or simply an old unused one to be removed (unusual, but may happen). Changing a database definition has several serious implications. In one hand, storage system must reformat existing data, to allow new definition to be managed and stored. This usually means to expand record size, so a complete regeneration is often needed. On the other hand, client applications may have to deal with more fields than expected, although this shouldn't be a problem in a well designed application (just ignoring unknown fields on read, and relying on default values for unspecified fields on write).

Strategy to replace an old database record definition by a new recent one has several stages:

1. Creating a new database object to hold new defined one. Object is still in init status, not available to application.
2. Filling new object with data ported from old, still working one. Old one is notified to keep a change log from this moment, just in case it is modified while this process runs. Porting is done in a field-by-field basis, assigning values to same-name fields. Conversions are performed as necessary. This process may fail, for several reasons, if “no data loss” mode is active (default): if a field can't be

reduced as required (doesn't fit), if a field is not present in new definition and has a valid value on the old one, if field type changes and cannot be converted...

3. Old object is placed in ReadOnly mode, to avoid further modifications (so now, system is not fully operative, but write operations are few, and still may be seeked, the most usual operation performed). Modification log is checked and, if not empty, processed like in step 2, to get a fresh, updated new-style object.
4. Storage is created for the new object definition, and filled with all the fresh content generated and updated in step 3. This is the most critical phase, as it may take some time, and there's no object (old nor new) available for writing (yet it is available for reading).
5. Till now, all the process is reversible.
6. New object is registered, so it becomes fully operative for every new query (both read and write). Definition changes become definitive.
7. Old object is marked as unusable, so no new operations will never be directed to it. Instead, they will go through the new object, fully operative by now.
8. Once old object becomes idle, it is unregistered and all its resources released.

Handicaps

As commented, when a record is redefined, there is not a true 24x7 operation, but usually won't be noticed. There's some short time where the changed object can't be written, usually in the order of a few seconds, while all the object data is stored at the *Backup Module*, but in a normal running system, write operations are little, short, and may wait for a short while to be executed. To reduce impact, and avoid client awareness on retry operations, *Query Manager* may catch that kind of queries for a while, and forward them once the object is completely up in its new form, so only noticing a slight delay on the process, and becoming a real 24x7 behavior.

Conclusions

Hot module replacing and database redefinitions on the fly allow to manage a live system easily, while keeping it working all of the time, with no down times at any moment, thus having real High Availability performance to be top 100%.