Preface

This document describes some ideas and hints of the Kin Distributed Database and Application system. It is not a well structured document, yet. It only holds ideas and tricks that I have, as a reminder of important things not to forget when development starts. It is not a complete reference, even the ideas are partial, with only some details like a draft for the whole painting.

Objectives

These are the main points that drag me to the project:

- High Performance (almost real time even for stats, and low latency)
- Compactness (resource optimization)
- High Availability (real 24/7)
- Low Cost (overall system)
- Easily expandable and customizable to every need
- Easy security definition and management (even to field and user level)

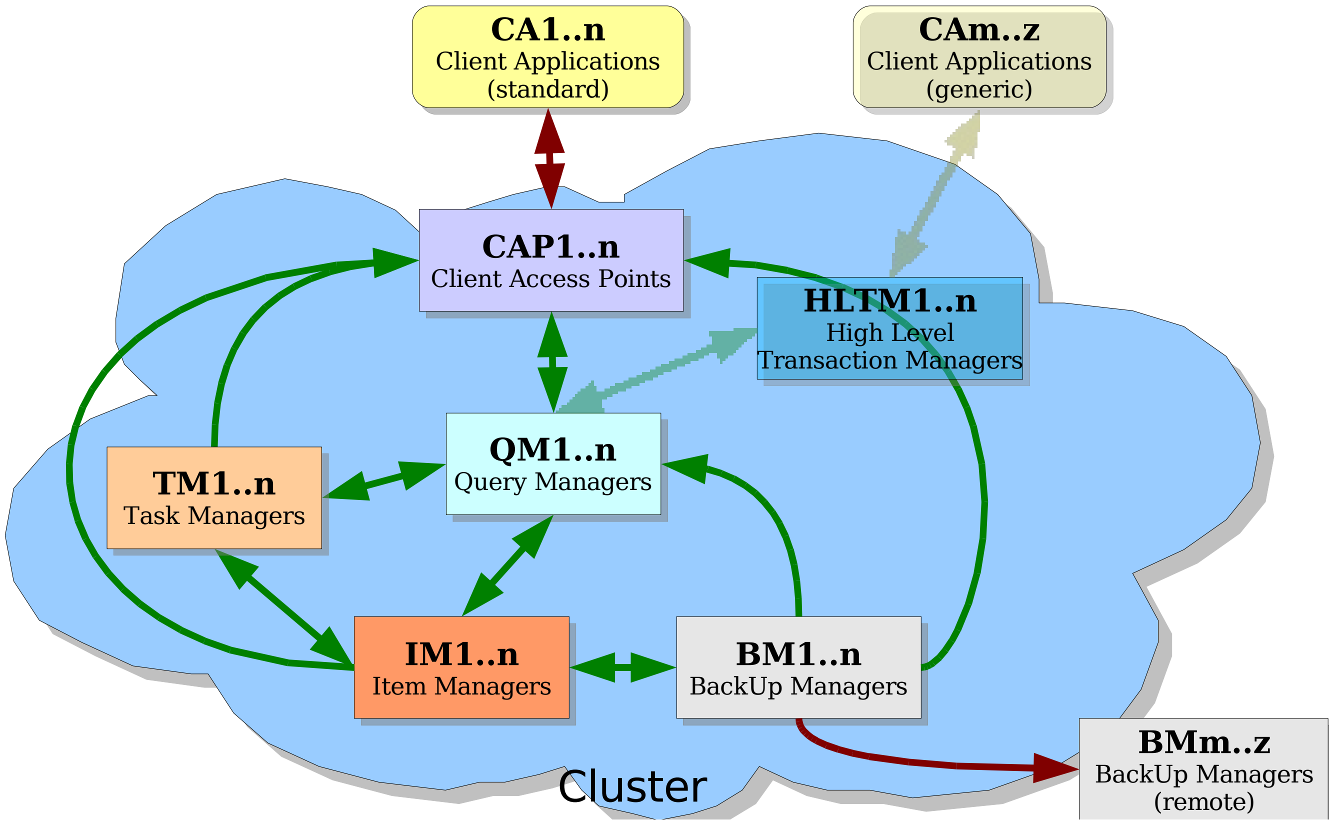
In a sentence: want to get an enterprise class database and application (process management, resource planning, decision making, etc.) system that is dirty cheap, easy to maintain and manage (easy to expand and adapt), failure proof, fast (as real time), and very, very scalable.
SQL compatibility is desirable, but not a priority, as it reduces some of the objectives (security, performance, and resource usage). It may be implemented as a simple task in the system.

Regrets

Nothing is perfect, and these are the things that may make working with this system a little harder:

- Need to write (code) Task Managers to get a good performance system. Anyway, you may use simple queries, to avoid this part (but then you'd better use a standard SQL DB server). So you need to know how to write programs and a fair knowledge of C language -or one that may interface to the API and create a dynamic object files-.
- Need to do a good design before starting development of a new application or database: definitions are at quite low level, that makes it simple and efficient, but maybe a little harder for somebody that thinks on SQL structures. On the other side, good design makes it work fast and spend little resources...

Kin Structure and Functions



CA - Client Application

This is usually a GUI application, integrated on the user system (for an instance, a Windows application), that manages all the user interaction (data entry, data formating and representation, etc.). This is what the users see, and is usually written in a high level language, with graphic capabilities (Visual C, Java...). It may also be a standard application connecting to a specialized HLTM (for an instance, a SQL client connected to a SQL HLTM)

CAP - Client Access Point

This is the basic system front-end, it manages all the connections from/to the clients. This includes socket management, security management (connection level - SSL, and application level - user+password validation), transport management (optional compression), and some global services (time, etc.). It assigns a unique ID to each connected client, and labels all the transactions and queries so their child processes' results may reach back the original requester, via any other CAP.
Physically, it works as a firewall to the cluster.

HLTM - High Level Transaction Manager

This optional modules give standards compliance to Kin system. One example is the SQL interface, that converts SQL every transaction to one or more Kin transactions, collects results, arranges them and sends that standard result to the client. Thus, some of these modules also may work as a simpler client access point, providing a standard interface to clients (even the listening port). Note that this usage leaks security to the level this module can handle.

QM - Query Manager

This blocks may simply forward a query to a Task Manager, or split that query into simpler sub-queries, send them to every requested module (TMs and IMs), and eventually collect results and post some other queries (second level), and so on till the transaction is finished or fails (even that kind of job is usually -natively- implemented on Task Modules). There's another important feature performed by these modules: object brokering. They keep track of all the TMs and IMs available into the cluster at any time. This information is essential at different levels: to the QMs and TMs, to find the node where an operation may be performed (by the corresponding Manager); to the Managers (TMs, IMs), to figure out how they should work (master, slave, clustered...) depending on how many copies (and their kind: complete or clustered) are active by now; and to the QMs to determine if a query will fail due to unavailability of one or more Managers needed for some sub-operation (as the system allows to load and unload TMs and IMs at the administrator's discretion, for maintenance or upgrading). This repository also helps to decide to which TM or IM are sent the operations when more than one is available, based on load, or even on version (you can upgrade loading a new version of a module, starting it up, and then shutting down the old ones, while the better available is being used without stopping the service at any time: QM will switch new queries to the new module version once it is up and running, the old one will shutdown once it finishes its pending jobs).

TM - Task Manager

This is the heart of intelligence of the system. It might be seen as a RPC (remote procedure call), but actually is a complete process program, that may analyze query, made several queries (to other TMs or IMs), create some temporary results, do a task -like, for an instance, sending an e-mail-, etc. It also may be used to create complex queries, collect the results, make some sort of processing on them, and assemble a complete answer to the client, simplifying logic at client side (and bandwidth overheat) and making part (or all) of the computing at the server side. Examples range from simple queries, to complex statistical processes, even like real time data mining with the original real data. Another example may be part of an optional SQL interpreter and processor.
This modules are packed into standard dynamic objects (like Linux kernel modules, apache DSOs, etc.) and usually written in C. There are some helper and tools libraries, available at the modules manager, so usually these modules only implement the 3-4 main methods (init, done, proc and event). You may write and install hundreds of these

modules, maybe one per each kind of query you have on your system, to get the better performance attainable.
You also may (and often should) declare a TM with the same name as an Item Manager (see below). This makes queries to divert to the TM instead of being sent to the IM, when they come from the clients, so you can do many things before query is actually processed (syntax checkings, creation of computed fields, check of complex conditions, creation of other objects based on received information...). You may apply that to Save Requests, or to Query Requests.
TM registers itself against the QM, and may also register the extensions it implements; extensions are false fields (this is a Task Manager, not an Item Manager, it can't have real fields), that can be accessed from applications like if they were there, but are actually generated on the fly by the TM. This allows to implement, for an instance, statistical fields, that actually fire a calculation at the TM and return the value, instead of existing as a constant into an IM. Another use is to receive data to save in a simple manner, on a simple field, and then translate to more complex data that is actually saved in real fields. Once extensions are registered at the QM, they look like simple fields, and are added to the possible record definition given from the IM.

IM - Item Manager

Items are the simplest elements stored at the data base. An item might be a record, but even a record field. There are two basic items: fixed size and variable size. Fixed size is, for an instance, a record (composed by fixed size fields: numbers, reference IDs, etc.). Variable size is always a field (never a record) like, for an instance, a text string. The Item Manager does the job of storing, arranging, saving, searching, and mirroring items. There are as many Item Managers in a system as records and variable field families (so usually tenths to hundreds), but there's only one instance that forks or multi-threads, for every kind of manager (a few units) at every node. So there are a lot of logical IMs, but just a few physical processes in each node.
This modules are, like TMs, dynamic objects, but usually you have enough with the system defaults (fixed size records, text strings). Sometimes, you have to write one for a specialized item, like telephone numbers, but that's not usual.
IMs manage data on their own criteria. For an instance, on fixed size records, they may detect a field that has not much values (some hundreds, maybe), that may be used to rearrange the object with a new dimension when it becomes too big to be managed flat. Then, the IM just breaks that object into multiple branches (threads of the same object), each with all the records for one value for that field, like you would do in a binary tree for sorting data. Having some few hundreds of objects representing one is not so expensive, and can improve throughput in several orders of magnitude, as it only has to work in very smaller sets of data. The decision to break the object in smaller parts is taken based on its size relative to the available resources (RAM), and the usual queries it has to process (massive searches don't take advantage of this partition, for an instance).

BM - Backup Module

Backup modules have two basic functions: storing and restoring items. Storage is asynchronous (background), restore is done, at least, when an Item Manager starts. To be really a backup concept, there should be at least two of these modules (main storage and a mirror). Storage might be encrypted, if necessary, and is always CRC protected. This modules also supply raw chunks of data when required by an IM for big items (like images or binary blocks, where the IM only manages the data, but has no real value/content in it).

Node

A node is a complete computing block. One node has network support, a task scheduler, and a manager for packets and processes. Nodes are allocated in a simple manner: one per each independent computing resource. So in a single processor computer, there's no sense to have more than one node. On a two processor computer, if you want to take advantage of both CPUs, you'd better have two nodes running. And so on.
A node can have any number of Kin blocks running in, from a single node (like a CAP), to the whole system. Blocks would be distributed based on resource availability and utilization: CAPs use pretty much CPU when sending and receiving (SSL encryption), and little memory (well, a little more if you manage hundreds of connections in each); IMs are memory and bus intensive, BMs use Hard Disk often... so try to make a balanced combination to get the best of your hardware. Remember that objects may move from one node to another, based on load balance, but a node never changes the kind of blocks it has, only the objects they manage (a CAP never moves to another node, objects into an IM may move to another node's IM).

Kin Concepts

1. Separate management and storage

Kin database makes a big difference between live database (data that you store and query) and stored database (simple files on disk or other storage). By definition, most of the data types are “live”, so they're living on RAM (or swap) all the time (of course, they're also stored on permanent media). This apply for smaller data types: bigger ones (images, binary blocks, etc.) work the usual way, with some caches and management info in RAM, and the actual data on disk (or other storage means). Working like this makes data caches unuseful, and throughput an easy thing to reach. Of course, it also has its own weakness: if you have a big database, you need a big RAM. But databases are not easy to grow so much in Kin structure (pretty optimized for size), RAM is cheap on cheap computers, and Kin is a cluster DB, so you can always escalate by adding more computers to the cluster (so growing RAM available for Kin). For seldom used data, some kind of swap to/from storage might be useful, but by now, this is managed by the OS through virtual memory (swap) as it is not considered (yet) to be a key point for Kin (intended for target systems with some ten million records or less, by now).

2. Transaction concept

Not the same as an SQL transaction, but a way to indicate an atomic operation. It is an internal process. When a request comes into the clustered system, a unique operation ID is included in it (at the CAP), so once it begins to split in several operations across the nodes, is still possible to keep it related to a given process (or "transaction"). Thus, if some part of the process fails, a transaction cancel can be sent/broadcast, so the other nodes implied can abort or undo the part of the process related to the failed request.

3. Undo concept

Objects (IMs) shall be designed to handle failed transactions. That means two cases have to be correctly supported: aborting process (when not yet finished), or deleting the stored value (when operation implied a store action, and was finished some time before the transaction failed elsewhere). The first is relatively easy to implement, as the process still lives and it has a status, that could be recovered. The second case implies a pretty higher degree of complexity: once something has been stored, an item ID generated, backups and mirrors refreshed (or posted to), and the process finished, all that job has to be undone. Into the object, that means storing item data has to be associated with a temporary storage of the operation unique ID. This reference has to be kept till the whole operation is acknowledged/validated (some node detects it's completely finished) or aborted (some node fails and recovery is not possible). Undoing means that some kind of journal should be implemented, so when a value is discarded (deletion is not always possible, as many other items may have been stored after), it can be reserved as room for a future new item storage. It also means that backup and mirror nodes should be aware of that temporary change in status for the item (now officially gone, but it's room still allocated somewhere), with some sort of journal for free data blocks into the main storage blocks, for an instance.

4. Tracking concept

Transactions should be traceable, i.e. every sub-operation should know the path till it was created. This allows to send acknowledgments, failure messages, etc. to the "parents" of the current process, so they can take some actions. Parents are other operations or the request itself (at the root of the transaction tree), and also working nodes (task routers, cluster access points, etc.)

5. Client concept

As requests are split in many parallel and sequential operations, answers shall find the way to the requester (client) in an easy way. If we add the redundancy in all the system levels, it may seem difficult to keep this requirement (multiple paths). The easiest way is to label the request (and thus all its child -operations-) with a client ID, so when an answer packet is ready to leave the cluster, any access point node can route that packet to the right external client. This means that, when a client connects to an access point, once validated, it gets an internal unique ID, and an association is made between that ID and its real address. This tables are shared by all the access points, so failure tolerance is achieved (any access point knows all the clients, although it only manages the connected ones). This also allows some kind of roaming between clients and access points (if connections are streamless, like UDP, any access point can send the packets to the client at any moment; if are streams, like TCP, when connection falls due to access point failure or shut down, client may seamlessly connect to another access point keeping its ID and so receiving pending packets, with no data lost at any time; even another access point could connect to the client instead, like a callback)

6. Security approach

System is designed for security from the bottom. Every client that connects to has to authenticate, at two levels: connection (public key), and application (user+password). Some services are available at the first level (e.g. time service doesn't require user +password), but database and applications are only reachable once a whole session is established (by means of a user+password). Connections between clients and the access points shall be encrypted, maybe using SSL or other secure systems, at any time. Access points manage client connections, among other services. They behave as a firewall, isolating (even physically) the internal cluster from the external world. So being isolated, communications into the cluster are not encrypted (but binary anyway) to improve bandwidth and performance. It is assumed that the network administrator builds the cluster on a private network.
Every user in system is expected to have a user ID and password, and a security profile that defines what can it do even at every field level. Of course, a developer may use a generic user ID, but this is not the intended use. Profiles (may) define to which database objects every user has access, the kind of access (ReadOnly, R/W), and even to which records (by the means of valid values for the dimensional field).