Notación: Al representar el nombre de un dato, las palabras en camelCase serán sustituidas por el nombre real de la variable y las palabras completamente en mayúscula se dejan igual

Ej: assetName:DEVICE:deviceId 🡪 BALEARES:DEVICE:SEM1

Las operaciones que pueden llegar a obtener una cantidad elevada de datos (como la lista de dispositivos) se hacen en múltiples llamadas scan.

Esto permite no bloquear la base durante estas operaciones, pero si otro cliente modifica la base a mitad de esas llamadas el resultado obtenido puede o no contener estas modificaciones (indeterminado)

Para solucionar esto habría que usar una llamada keys (o similar), pero esto bloquearía la base durante toda la operación, lo cual es muy poco recomendable.

Debido al buffer y la cache, en muchas ocasiones la libRTDB no es capaz de saber si una operación se ha completado correctamente o no.

Datos que se almacenan en la base:

Sets:

ASSETS -- Nombre de los assets almacenados

assetName:DEVICETYPES -- Tipos de dispositivo de un asset

assetName:DEVICETYPE:typeName -- Variables que contiene un tipo de dispositivo

assetName:DEVICETYPELIST:typeName -- Dispositivos reales que pertenecen a ese tipo

assetName:DEVICELIST -- Lista de dispositivos reales de un asset (sin cabecera, solo el nombre)

Hash:

assetName:TYPE:typeID:VARIABLE:variableName

sFormat

nChangeEvent

nAlarmEvent

nHistEvent

cType

assetName:DEVICE:deviceId -- Variables de un dispositivo real

tags : "variabeName:atribute"

Por ejemplo, el dispositivo SEM1 con una variable "color" quedaría así:

BALEARES:DEVICE:SEM1

sDeviceType:sValue = "SEM"

color:sValue = "red"

color:nDataTimestamp = "today"

color:nWriteTimestamp= "yesterday"

color:nQuality = "100"

Ciertos datos que antes se almacenaban ahora se obtienen a partir de los sets, en vez de estar directamente almacenados. Por ejemplo, el número de dispositivos de cada tipo o asset.

Si se modifican los structs de comunicación de la interfaz (Ej: ST\_DEVICE) hay que cambiar la libRTDB, dado que en redis no se almacena un struct si no campo a campo

IMPORTANTE: No eliminar la variable sDeviceType del struct ST\_DEVICE, dado que la librtdb la usa de forma interna para ciertos comandos.

Coste de las operaciones

Add\_asset: O(1)

Remove\_asset: 2\* O(NdevicesInAsset) + 2\*O(NdeviceTypesInAsset)

GetUsedEntries: O(1)

GetDeviceTypes: O(NdeviceTypesInAsset)

getVariables: O(nVariablesInType)

getDevice: O(1)

getDevices: O(nDevicesOfTheTypeInAsset)

getValue: 4\*O(1)

setValue: 4\*O(1)

AddDevice: 3\*O(1)

delDevice: 3\*O(1)

delVarToTypeOfDevice: 2\*O(nDevicesOfTheTypeInAsset)

addVarToTypeOfDevice: O(1)

modifyVarToTypeOfDevice: O(1)

AddTypeOfDevice: O(1)

AddTypeOfDevice (multipleDevices): O(nDevicesToAdd)

DelTypeOfDevice: 2\*O(nDevicesOfTheTypeInAsset)

GetAssets: O(nAssets)

Mejoras

La antigua librería RTDB estaba diseñada para una rápida comunicación cliente-servidor, dado que todo se ejecutaba en una misma máquina con memoria compartida. Por ello las únicas funciones de lectura/escritura que existen son variable a variable. Esto implica que para leer un asset de 10.000 dispositivos con 10 variables cada una, hay que hacer 100.000 llamadas a la libRTDB, lo cual tiene un rendimiento inasumible.

Por lo tanto, el principal cuello de botella serán las lecturas y escrituras seguidas que no estén en un pipeline.

A continuación se exploran varias mejoras a la librtdb que permitirían reducir este problema

Pipelines

La forma más sencilla de solucionar el problema es añadir un juego de funciones que permita al programador encapsular múltiples llamadas en un pipeline.

Sin embargo solo serviría de cara a implementaciones futuras dado que implicaría reescribir código. Actualmente, la libRTDB no dispone de estas operaciones.

Buffer

Todas las escrituras se pueden meter automáticamente en un buffer, para agruparlas y enviarlas a través de un pipeline.

Este buffer se vacía cada x ms, o de forma inmediata, y una vez se alcanza un número máximo de operaciones acumuladas.

El buffer dispone de dos colas internas, que van rotando dependiendo de su uso. Por defecto se encola en la primera, y cuando la primera está bloqueada se encola en la segunda. Así podemos minimizar los bloqueos por concurrencia. Esto permite también usar el buffer con latencia 0 con una pérdida mínima de rendimiento (un 3-4%), aunque el modo con retardo sigue siendo preferible ya que reducirá la carga de la red. Aun así, usar un buffer con retardo 0 es abismalmente superior a no usar buffer).

Además, para aprovechar el ancho de banda de la red hay dos hilos independientes: uno que ejecuta todas las escrituras en redis y otro que ejecuta las notificaciones pub/sub. Esto provoca que ambas cosas no estén sincronizadas, lo cual podría provocar que la notificación llegue antes de que el cambio se produzca. Dado que la notificación contiene el nuevo valor no parece un problema a priori, pero se puede cambiar a usar un hilo único con la configuración.

Cache

Con redis, leer un bloque de 10.000 elementos y 10 variables cada uno variable a variable frente a leerlas en bloque supone aumentar 15 veces el tiempo de acceso en local, y 160 veces en red (tiempos en el apartado de rendimiento). Por lo tanto, para lecturas intensivas usar Redis leyendo variable a variable supone un coste demasiado elevado.

Una cache permite aprovechar la ventaja de las pipelines de forma transparente a la interfaz.

Además también permitiría a un supercentro de control no cargar todos sus assets, si no que la cache los fuera cargando y descartando de forma dinámica conforme se fueran necesitando.

Llamaremos hit a las lecturas que estén ya en la cache y miss a las lecturas que requieran buscar el dato en Redis.

Escritura:

Todas las escrituras se envían directamente a Redis (o al buffer), manteniendo así la consistencia inmediata. Además, se envía una notificación al pub/sub de redis con la sintaxis:

channel: “nombreDispositivo;variable”

value: “valorVariable”

Por tanto, existe un canal para cada dispositivo (muchos canales no reducen el rendimiento si se usan pocos patrones de escucha que los engloben)

La propia cache recibirá también la notificación dado que está suscrita al canal, y en ese momento hará el cambio en local.

Hay un hilo que recibe las notificaciones y las añade a una cola, y otro que cada cierto tiempo procesa la cola y añade los cambios a la cache

Para mantener la consistencia siempre, en cada hget se actualizan también todos los cambios pendientes. Además, cada x ms un hilo independiente también actualiza todos los cambios pendientes.

Lectura:

Al hacer una lectura la cache trae todos los dispositivos del mismo tipo que el que se esté leyendo, así como la información general de las variables del tipo. También se puede traer el asset completo, dependiendo de la configuración. Esta carga se puede hacer de forma síncrona, en el hilo principal, o de forma asíncrona en un hilo aparte (configuración).

Tanto al traer un tipo como un bloque se empieza a escuchar cualquier cambio que haya en el asset. Por lo tanto, aunque solo se traiga un tipo, los dispositivos de otros tipos se irán rellenando conforme se vayan modificando. Otra opción sería que la cache escuche cada dispositivo que haya traído de forma individual, pero escuchar muchos canales puede lastrar el rendimiento así que habría que probar qué es más rápido.

En servicios con mayoría de operaciones de escritura podría desactivarse la caché, y si solo leen un número reducido de variables podría o desactivarse o hacer un modo reducido que solo mantenga, por ejemplo, las variables que se hayan leído hasta el momento.

Al trabajar con varios assets (en un supercentro) se podría almacenar solo n assets, y cuando se supere ese número eliminar el más antiguo.

Cambios en Sidera

Cambios necesarios en Sidera para compilarlo con la nueva LibRTDB. Estos cambios vienen de, entre otras cosas, el cambio a c++17. (Pueden no estar todos)

Añadir :: a los byte en secblock.h y el resto de ficheros de Cryptopp

acortar la declaración de los mapas en RoutingTable.h (no es necesario poner el comparator y allocator, y están mal puestos)

Corregir los punteros que devuelve la RTDB en MemoriaServidorNET.cpp