Blocking commands in Redis modules

Redis has a few blocking commands among the built-in set of commands. One of the most used is BLPOP (or the symmetric BRPOP) which blocks waiting for elements arriving in a list.

The interesting fact about blocking commands is that they do not block the whole server, but just the client calling them. Usually the reason to block is that we expect some external event to happen: this can be some change in the Redis data structures like in the BLPOP case, a long computation happening in a thread, to receive some data from the network, and so forth.

Redis modules have the ability to implement blocking commands as well, this documentation shows how the API works and describes a few patterns that can be used in order to model blocking commands.

NOTE: This API is currently *experimental*, so it can only be used if the macro `REDISMODULE_EXPERIMENTAL_API` is defined. This is required because these calls are still not in their final stage of design, so may change in the future, certain parts may be deprecated and so forth.

To use this part of the modules API include the modules header like that:

```
#define REDISMODULE_EXPERIMENTAL_API
#include "redismodule.h"
```

How blocking and resuming works.

Note: You may want to check the `helloblock.c` example in the Redis source tree inside the `src/modules` directory, for a simple to understand example on how the blocking API is applied.

In Redis modules, commands are implemented by callback functions that are invoked by the Redis core when the specific command is called by the user. Normally the callback terminates its execution sending some reply to the client. Using the following function instead, the function implementing the module command may request that the client is put into the blocked state:

```
RedisModuleBlockedClient *RedisModule_BlockClient(RedisModuleCtx *ctx
```

The function returns a `RedisModuleBlockedClient` object, which is later used in order to unblock the client. The arguments have the following meaning:

- `ctx` is the command execution context as usually in the rest of the API.
- `reply_callback` is the callback, having the same prototype of a normal command function, that is called when the client is unblocked in order to return a reply to the client.
- `timeout_callback` is the callback, having the same prototype of a normal command function that is called when the client reached the `ms` timeout.
- `free_privdata` is the callback that is called in order to free the private data. Private data is a pointer to some data that is passed between the API used to unblock the client, to the callback that will send the reply to the client. We'll see how this mechanism works later in this document.
- `ms` is the timeout in milliseconds. When the timeout is reached, the timeout callback is called and the client is automatically aborted.

Once a client is blocked, it can be unblocked with the following API:

```
int RedisModule_UnblockClient(RedisModuleBlockedClient *bc, void *pri
```

The function takes as argument the blocked client object returned by the previous call to `RedisModule_BlockClient()`, and unblock the client. Immediately before the client gets unblocked, the `reply_callback` function specified when the client was blocked is called: this function will have access to the `privdata` pointer used here.

IMPORTANT: The above function is thread safe, and can be called from within a thread doing some work in order to implement the command that blocked the client.

The `privdata` data will be freed automatically using the `free_privdata` callback when the client is unblocked. This is useful **since the reply callback may never be called** in case the client timeouts or disconnects from the server, so it's important that it's up to an external function to have the responsibility to free the data passed if needed.

To better understand how the API works, we can imagine writing a command that blocks a client for one second, and then send as reply "Hello!".

Note: arity checks and other non important things are not implemented int his command, in order to take the example simple.

```
int Example_RedisCommand(RedisModuleCtx *ctx, RedisModuleString **arg
                         int argc)
{
    RedisModuleBlockedClient *bc =
        RedisModule_BlockClient(ctx,reply_func,timeout_func,NULL,0);

    pthread_t tid;
    pthread_create(&tid,NULL,threadmain,bc);

    return REDISMODULE_OK;
}

void *threadmain(void *arg) {
    RedisModuleBlockedClient *bc = arg;

    sleep(1); /* Wait one second and unblock. */
    RedisModule_UnblockClient(bc,NULL);
}
```

The above command blocks the client ASAP, spawning a thread that will wait a second and will unblock the client. Let's check the reply and timeout callbacks, which are in our case very similar, since they just reply the client with a different reply type.

```
int reply_func(RedisModuleCtx *ctx, RedisModuleString **argv,
               int argc)
{
    return RedisModule_ReplyWithSimpleString(ctx,"Hello!");
}

int timeout_func(RedisModuleCtx *ctx, RedisModuleString **argv,
               int argc)
{
    return RedisModule_ReplyWithNull(ctx);
}
```

The reply callback just sends the "Hello!" string to the client. The important bit here is that the reply callback is called when the client is unblocked from the thread.

The timeout command returns `NULL`, as it often happens with actual Redis blocking commands timing out.

Passing reply data when unblocking

The above example is simple to understand but lacks an important real world aspect of an actual blocking command implementation: often the reply function will need to know what to reply to the client, and this information is often provided as the client is unblocked.

We could modify the above example so that the thread generates a random number after waiting one second. You can think at it as an actually expansive operation of some kind. Then this random number can be passed to the reply function so that we return it to the command caller. In order to make this working, we modify the functions as follow:

```
void *threadmain(void *arg) {
    RedisModuleBlockedClient *bc = arg;

    sleep(1); /* Wait one second and unblock. */

    long *mynumber = RedisModule_Alloc(sizeof(long));
    *mynumber = rand();
    RedisModule_UnblockClient(bc,mynumber);
}
```

As you can see, now the unblocking call is passing some private data, that is the `mynumber` pointer, to the reply callback. In order to obtain this private data, the reply callback will use the following function:

```
void *RedisModule_GetBlockedClientPrivateData(RedisModuleCtx *ctx);
```

So our reply callback is modified like that:

```
int reply_func(RedisModuleCtx *ctx, RedisModuleString **argv,
               int argc)
{
    long *mynumber = RedisModule_GetBlockedClientPrivateData(ctx);
    /* IMPORTANT: don't free mynumber here, but in the
     * free privdata callback. */
    return RedisModule_ReplyWithLongLong(ctx,mynumber);
}
```

Note that we also need to pass a `free_privdata` function when blocking the client with `RedisModule_BlockClient()`, since the allocated long value must be freed. Our callback will look like the following:

```
void free_privdata(void *privdata) {
    RedisModule_Free(privdata);
}
```

NOTE: It is important to stress that the private data is best freed in the `free_privdata` callback because the reply function may not be called if the client disconnects or timeout.

Also note that the private data is also accessible from the timeout callback, always using the `GetBlockedClientPrivateData()` API.

Aborting the blocking of a client

One problem that sometimes arises is that we need to allocate resources in order to implement the non blocking command. So we block the client, then, for example, try to create a thread, but the thread creation function returns an error. What to do in such a condition in order to recover? We don't want to take the client blocked, nor we want to call `UnblockClient()` because this will trigger the reply callback to be called.

In this case the best thing to do is to use the following function:

```
int RedisModule_AbortBlock(RedisModuleBlockedClient *bc);
```

Practically this is how to use it:

```
int Example_RedisCommand(RedisModuleCtx *ctx, RedisModuleString **arg
                         int argc)
{
    RedisModuleBlockedClient *bc =
        RedisModule_BlockClient(ctx,reply_func,timeout_func,NULL,0);

    pthread_t tid;
    if (pthread_create(&tid,NULL,threadmain,bc) != 0) {
        RedisModule_AbortBlock(bc);
        RedisModule_ReplyWithError(ctx,"Sorry can't create a thread")
    }

    return REDISMODULE_OK;
}
```

The client will be unblocked but the reply callback will not be called.

Implementing the command, reply and timeout callback using a single function

The following functions can be used in order to implement the reply and callback with the same function that implements the primary command function:

```
int RedisModule_IsBlockedReplyRequest(RedisModuleCtx *ctx);
int RedisModule_IsBlockedTimeoutRequest(RedisModuleCtx *ctx);
```

So I could rewrite the example command without using a separated reply and timeout callback:

```
int Example_RedisCommand(RedisModuleCtx *ctx, RedisModuleString **arg
                         int argc)
{
    if (RedisModule_IsBlockedReplyRequest(ctx)) {
        long *mynumber = RedisModule_GetBlockedClientPrivateData(ctx)
        return RedisModule_ReplyWithLongLong(ctx,mynumber);
    } else if (RedisModule_IsBlockedTimeoutRequest) {
        return RedisModule_ReplyWithNull(ctx);
    }

    RedisModuleBlockedClient *bc =
        RedisModule_BlockClient(ctx,reply_func,timeout_func,NULL,0);

    pthread_t tid;
    if (pthread_create(&tid,NULL,threadmain,bc) != 0) {
        RedisModule_AbortBlock(bc);
        RedisModule_ReplyWithError(ctx,"Sorry can't create a thread")
    }

    return REDISMODULE_OK;
}
```

Functionally is the same but there are people that will prefer the less verbose implementation that concentrates most of the command logic in a single function.

Working on copies of data inside a thread

An interesting pattern in order to work with threads implementing the slow part of a command, is to work with a copy of the data, so that while some operation is performed in a key, the user continues to see the old version. However when the thread terminated its work, the representations are swapped and the new, processed version, is used.

An example of this approach is the Neural Redis module where neural networks are trained in different threads while the user can still execute and inspect their older versions.

Future work

An API is work in progress right now in order to allow Redis modules APIs to be called in a safe way from threads, so that the threaded command can access the data space and do incremental operations.

There is no ETA for this feature but it may appear in the course of the Redis 4.0 release at some point.