Section 8: Key-Value Stores

Frank Austin Nothaft

July 22, 2015

Contents

1 Warmup 2
  1.1 ACID Semantics .................................................. 2

2 Problems 3
  2.1 Practical ACID ..................................................... 3
  2.2 Simple Key-Value Store .......................................... 3
    2.2.1 Sketch Design ................................................. 3
    2.2.2 Code Design .................................................. 4
1 Warmup

1.1 ACID Semantics

In class on Monday, we introduced the concept of ACID semantics. What do each of the letters in the ACID acronym stand for? Can you describe the semantics each letter provides?

A: Atomicity. Either all actions in the transaction occur, or none of the actions in the transaction occur.
C: Consistency. Transactions maintain invariants on the data (e.g., a bank balance cannot go negative, or you cannot schedule two meetings at once).
I: Isolation. Transactions are isolated from each other → no problems occur from overlapping two independent transactions in time.
D: Durability. If a transaction commits successfully, the changes made by this transaction will persist despite any crashes that may occur after the transaction.

Why do we care about ACID semantics? Why would it be difficult to use a database that did not provide ACID semantics?

ACID semantics allow us to reason well about how concurrent transactions are processed. If a database provides “full” ACID semantics, then we know that concurrent transactions will appear to have been serial updates, and we know that we’ll never lose a prior update.
2 Problems

2.1 Practical ACID

Do you think that all databases provide full ACID semantics? If you think they don’t, why don’t they? What might be the hardest part of ACID semantics to provide?

Most deployed parallel DBMS provide so-called “weak” ACID semantics. They do this because providing full ACID semantics (and thus full serializability) is very expensive to implement in a parallel/distributed system. Generally, these systems will weaken isolation.

2.2 Simple Key-Value Store

2.2.1 Sketch Design

We want to implement a simple iterative key-value store that runs on a single, multithreaded machine. In our simple KVS, our keys and values are restricted to be integers. Sketch out what your implementation looks like. In your sketch, show where synchronization is necessary between the threads, and show what synchronization primitives you are using. Use the simple message passing API we introduced last section, and assume that there is one directory thread, and \( n \) key/value storage threads. For this problem, assume that key/value pairs are \textit{not} replicated.

As a reminder, here is our message passing API:

\[
\begin{align*}
\text{int } &\text{ msg_send(inbox_t * inbox, void * msg, size_t msg_size);} \\
\text{int } &\text{ msg_recv(inbox_t * inbox, void ** msg);} \\
\end{align*}
\]

TBA
2.2.2 Code Design

Let's implement the design you just sketched out. Assume that the following functions are defined:

```c
// Returns the number (between 0 and num_nodes) of the subpart of the KVS that
// owns a key.
int get_key_owner(int key, int num_nodes);
```

```c
// A thread-unsafe map-like data structure that stores integer key-value pairs.
typedef struct int_map;
```

```c
// Getter/putter functions for the map.
int get(int_map * map, int key);
int set(int_map * map, int key, int value);
```

New structs that you’ve added, if any:

```c
typedef struct {
    int key;
    int value;
    bool is_get;
    inbox_t * sender_mailbox;
} request;
```

Global variable definitions, if any:

```c
#define NODES 8
```

```c
// mailbox for the directory
inbox_t directory_inbox;
```

```c
// mailboxes for each storage thread
inbox_t node_inbox[NODES];
```

```c
// map per thread for storing key/value pairs
int_map key_map[NODES];
```
Key getter function:

```c
int get(int key, mailbox_t * my_mailbox) {
    // make request struct
    request * req = malloc(sizeof(request));
    *req = {key, 0, true, my_mailbox};

    // ask directory where key is.
    msg_send(&directory_inbox, (void *) &req, sizeof(request));
    int * node;
    msg_recv(my_mailbox, (void **) &node);

    // send a message to the relevant node asking for the value
    msg_send(&node_inbox[*node], (void *) &req, sizeof(request));
    int * value;
    msg_recv(my_mailbox, (void **) &value);

    // processing node malloc'ed memory; by convention, we must free
    int return_value = *value;
    free(value);
    free(node);

    return return_value;
}
```

Key setter function:

```c
int set(int key, int value, mailbox_t * my_mailbox) {
    // make request struct
    request * req = malloc(sizeof(request));
    *req = {key, value, false, my_mailbox};

    // ask directory where key is.
    msg_send(&directory_inbox, (void *) req, sizeof(request));
    int * node;
    msg_recv(my_mailbox, (void **) &node);

    // send a message to the relevant node telling it to update the value
    msg_send(&node_inbox[*node], (void *) req, sizeof(request));

    // this always succeeds, so return 0
    return 0;
}
Directory message processing function:

```c
void directory_thread() {
    request * req;
    int * node;
    while (1) {
        // malloc an integer
        node = (int *) malloc(sizeof(int));

        // wait for a request
        msg_recv(&directory_inbox, (void **) &req);

        // get node id
        *node = get_key_owner(req->key, NODES);

        // send reply back
        msg_send(req->sender_mailbox, (void *) node, sizeof(int));
    }
}
```

Key-value storage thread processing function:

```c
void kvnode_thread(int node) {
    request * req;
    int * value;
    while (1) {
        // wait for a request
        msg_recv(&directory_inbox, (void **) &req);

        // is this a get or a put?
        if (msg->is_get) {
            // malloc an integer
            value = (int *) malloc(sizeof(int));

            // get the value for this key
            *value = get(&key_map[node], msg->key);

            // send reply back
            msg_send(req->sender_mailbox, (void *) value, sizeof(int));
        } else {
            // set the value for this key
            set(&key_map[node], msg->key, msg->value);
        }

        // deallocate the memory from the request
        free(req);
    }
}```