Spring 2014 CSCI 182E—Distributed Systems
Homework 5
Distributed Key-Value Storage

Value: 25% of Homework Assignments Grade

Daniel M. Zimmerman
Harvey Mudd College

Assigned 24 March 2014
Due 10 April 2014, 23:59:59

This assignment is to be completed in teams of 2 or 3

The main purpose of this assignment is to exercise various distributed algorithm techniques we have discussed in class by building a distributed system for key-value data storage based on a general design given here.

Unlike the previous assignment, this assignment *requires* you to consider issues of fault tolerance. It is possible both for nodes to join and leave the system at runtime and for nodes to "crash", meaning that they no longer generate or respond to messages. For the purposes of this assignment, nodes will crash by exiting (or by being forcefully exited with Ctrl-C or a kill signal) and Erlang's monitoring facility will be sufficient for determining whether a node has crashed.

You are strongly recommended to use Erlang and its distributed messaging for this assignment. You may choose to implement your programs in Java using Erlang messaging instead, but it's not a good idea.

System Overview

For large, important data sets, a storage system should fulfill three main requirements: it should be correct (that is, the value you store with a particular key should be retrievable using the same key, and should be the same value as was last stored for that key);[1] it should be highly available, exhibiting as little downtime as possible in the presence of failures; and it should be reasonably fast. The first requirement could easily be fulfilled, for data sets smaller than a few terabytes, by a server running on a single machine with ECC memory[2] and a large amount of reliable error-correcting storage (such as a redundant array of drives with ZFS or a similar filesystem) and using reliable communication protocols. The second and third requirements, however, are best fulfilled by a distributed system structured in a way such that (1) if one node in the system goes

[1] For some definition of "last"… more on that later.
[2] Error-correcting code memory, see http://en.wikipedia.org/wiki/ECC_memory.

down, the remaining nodes can still respond correctly to queries, and (2) the data is distributed across the system in a reasonable way such that accessing the value associated with a given key can be done quickly.

For this assignment, you will design and implement such a distributed system with these characteristics:

- A *key* is an Erlang string (a list of integers, not a binary string), and a *value* is an arbitrary Erlang object.
- There are 2^m (for some constant m) *storage processes*, numbered 0 to $2^m - 1$. Each storage process is an actual Erlang process, and knows both m and its own number when it starts up.
- Every storage process uses the same *hash function* $h(x)$, which takes a key and returns an integer from 0 to $2^m - 1$ inclusive. Each storage process i is responsible for storing (at least) the values for all keys k such that $h(k) = i$. You may write this function in any way you choose; ideally, it would uniformly distribute keys across the process IDs, but any reasonable function is acceptable.[3]
- Every storage process can respond to messages from the outside world requesting to store key-value pairs and retrieve values corresponding to keys. When a storage process receives such a request with key k, there are two possibilities:
  - If its ID matches $h(k)$, it performs the operation itself (in the case of store operations, this may involve communication with other processes) and responds to the request.
  - If its ID does not match $h(k)$ it forwards the request to the correct storage process, which performs the operation and responds to the request.

  Every storage process can also respond to messages from the outside world requesting information about the entire system that must be gathered with a distributed computation. These requests and responses are described in detail later.
- The communication channels among storage processes take the form of a *ring* with *chords*.[4] Storage process i has (unidirectional) channels to storage processes $i + 2^k$ for $0 \leq k < m$, with all addition done modulo 2^m; these are called storage process i's *neighbors*. For example, in a system with $m = 3$, storage process 5 would have channels to storage processes 6, 7, and 1; in a system with $m = 7$, storage process 0 would have channels to storage processes 1, 2, 4, 8, 16, 32, and 64. In this way sending a message between any two storage processes in a system with $m = K$ requires at most $O(K)$ message sends between storage processes,

[3] An extremely simple such function adds all the characters of the key string together and returns the modulus of that sum with respect to 2^m.

[4] This entire assignment is inspired by the *Chord* distributed hashtable algorithm; see http://~~pdos~~.csail.mit.edu/papers/chord:sigcomm01/.

while also only requiring each storage process to communicate with $O(K)$ other storage processes.

- A *node* is a single Erlang virtual machine that hosts at least one storage process. Each node also hosts some number of non-storage processes that handle management and fault tolerance with respect to the storage processes.

- There may be anywhere from 1 to 2^m nodes in the system, each with a unique ID from 0 to $2^m - 1$. Each node hosts the storage processes with consecutive numbers from its number (inclusive) to the number of the next highest numbered—modulo 2^m—node (exclusive). For example, in a two-node system with $m = 3$ where the nodes are numbered 3 and 6, node 3 would host storage processes 3, 4 and 5 while node 6 would host storage processes 6, 7, 0, 1, and 2.

- All storage processes are named in the "global" Erlang name registry[5] in a way such that they can determine each other's names (and therefore contact each other) using only their numbers. For example, you might register storage process 8 as `StorageProcess8`. Each node must also register at least one non-storage process in the registry to allow its node number to be discovered by other nodes (so that a node can learn about the other nodes that exist in the system). You may also register any other processes in the global name registry that you find necessary/useful.

- A new node can join the system whenever the system has fewer than 2^m nodes. When a new node joins the system, it chooses an available number (it can discover the registered node numbers by looking at the global name registry) and sends appropriate messages to other nodes in the system to force a *rebalance* of storage processes, such that it ends up hosting the storage processes it is supposed to (as described above). Rebalancing should be as localized as possible; adding one node to a system should generally not require communication with every other node in the system.

- A node can leave the system either in an orderly fashion (by notifying other nodes as appropriate) or by crashing (exiting suddenly either of its own volition or in response to a kill signal from the environment). When this happens, the other nodes must eventually[6] detect it and perform a rebalance. When this rebalance occurs, key-value pairs stored in the leaving node should be preserved in the system to the extent possible (i.e., they should not simply be lost). If the last node leaves the system, of course, all information is irretrievably lost.

  Note: You may find, when doing your design and implementation, that features of Erlang make it easier to assume that nodes always leave the system by crashing than to devise and implement a protocol that allows them to leave in an orderly fashion. An implementation that makes such an assumption is acceptable for the purposes of this assignment.

[5] It is "global" in the sense that all nodes connected with `connect_node/1` have access to the same registry; see the documentation for the `global` module for details.

[6] Erlang's features make "eventually", in this case, into something more like "very quickly".

Communication

Communication within your system (i.e., among your storage processes, and between your storage processes and any other processes you create) can be carried out in any way you choose, with these restrictions:

- Storage processes can only send messages to their neighbors (the processes 2^k hops away on the chords of the ring, described previously) and to non-storage processes on the same node.
- A non-storage process hosted on node i can send messages to storage processes on the same node, and to non-storage processes hosted on nodes containing neighbors of the storage processes hosted on node i. For example, consider a 3-node system with $m = 4$ and nodes numbered 0 (hosting storage processes 0, 1, 2, 3, 4), 5 (hosting storage processes 5, 6, 7, 8, 9, 10, 11, 12), and 13 (hosting storage processes 13, 14, 15). A non-storage process on node 0 can send messages to storage processes 0–4 and to non-storage processes on node 5, but not to storage processes on node 5 or to any processes on node 13 (because none of the storage processes on node 0 have neighbors on node 13).

Communication between the outside world and your system uses the following message formats. Every message from the outside world (except for `leave`) must be responded to except in situations where a crash failure prevents the action from completing. The response to the outside world need not come from the process to which the request was made; for example, a `stored` message can come from the storage process that actually stores the key-value pair rather than the storage process to which the original request was sent. All processes you create are allowed to communicate with the outside world, but only storage processes will receive requests from the outside world.

- {*pid*, *ref*, `store`, *key*, *value*} – sent by the outside world to store *value* for *key*.
- {*ref*, `stored`, *old-value*} – sent to the *pid* from a `store` request, with the *ref* from the request, to confirm that the store operation took place and overwrote the previously stored value *old-value*. If there was no previously stored value, *old-value* should be the atom `no_value`.
- {*pid*, *ref*, `retrieve`, *key*} – sent by the outside world to retrieve the value for a key.
- {*ref*, `retrieved`, *value*} – sent to the *pid* from a `retrieve` request, with the *ref* from the request, to indicate that *value* is stored for the request's *key*; if no value is stored for *key*, *value* should be the atom `no_value`.
- {*pid*, *ref*, `first_key`} – sent by the outside world to request the first key, in lexicographic order, stored in the entire system.

- {*pid*, *ref*, `last_key`} – sent by the outside world to request the last key, in lexicographic order, stored in the entire system.
- {*pid*, *ref*, `num_keys`} – sent by the outside world to request the number of keys for which values are stored in the entire system.
- {*pid*, *ref*, `node_list`} – sent by the outside world to request a list of node numbers (not storage process numbers, as there will always be 2^m storage processes) for the active nodes in the system.
- {*ref*, `result`, *result*} – sent to the *pid* from a `first_key`, `last_key`, `num_keys`, or `node_list` request, with the *ref* from the request, to communicate the *result*. In the case of `first_key`, `last_key` and `node_numbers` the result will be a list; in the case of `num_keys`, it will be an integer.
- {*ref*, `failure`} – sent to the *pid* from a request to indicate that the request failed. This is an optional, polite way to tell the outside world that a request failed rather than simply letting the outside world time out.
- {*pid*, *ref*, `leave`} – sent by the outside world to tell a storage process that its hosting node should leave the system.

Some of these requests from the outside world, such as `node_list` and `leave`, require storage processes to communicate with non-storage processes. Some requests, such as `first_key` and `num_keys`, necessarily cause distributed computations—effectively, snapshots (or approximate snapshots) of the system state—to be carried out. Such computations are best-effort, and the data returned by any of them might be obsolete by the time it is returned; however, the system should try to return results that are as accurate as possible. In particular, if no changes to the key-value pairs stored in the system and no node failures occur between such a request and the response to that request, the response must be accurate. If a node crashes while such a computation is being carried out, the computation need not be completed successfully (and the environment need not receive a response, though a `failure` response should be provided whenever possible).

Behavioral and Fault Tolerance Constraints

Making a system like this exhibit consistent behavior and be completely fault tolerant is provably impossible. The following constraints on the behavior expected from your system should make the task manageable.

- You need not preserve ordering when different values are stored with the same key. Clearly, if distinct storage processes receive requests to store different values with the same key, the one storage process responsible for that key will receive those requests in a nondeterministic order. As long as one of the values is correctly stored when multiple store requests are in progress simultaneously, your system will be considered to function correctly.
- You may assume that when a node joins or leaves the system (in either an orderly or disorderly fashion), the system always has sufficient time to complete a rebalance operation before another node joins or leaves the system. You may also assume that any node leaving the system in an orderly fashion stays in the system long enough for its data to be completely reallocated to other nodes; that is, it does not leave until it knows for certain that leaving will not destroy data.
- Your system must be fault tolerant in a limited sense: data that has been *completely stored* in storage process x must not be lost when the node containing x leaves the system (in either an orderly or disorderly fashion). A key-value pair (K, V) is completely stored in storage process x if a client asked some storage process in the ring to store (K, V), K hashed to x, and the client was notified that (K, V) was successfully stored. Any data that should have been stored in storage process x but was not completely stored before x left the system may be lost (and the client should, presumably, time out waiting for the acknowledgement that it was completely stored).

  Note: This implies that, in a system of at least two nodes, every key-value pair must be stored on at least two different nodes *before the client is notified that it was completely stored* to guard against a single crash failure. There are many strategies for accomplishing this; by giving careful thought to how you distribute the "redundant" stored information, you can make the process of rebalancing after a crash failure significantly easier.

Implementation Requirements

The requirements for your implementation are as follows:

- The module containing the main function (the one that starts a node) must be called `key_value_node` and must export at least the function `main/1`; it may also export other functions as you find necessary.
- It must take the following command line parameters (in this order):
  - m, the value that determines the number of storage processes in the system.
  - A name to register itself with. This should be a lowercase ASCII string with no periods or `@` signs in it.
  - An optional additional parameter, which is the Erlang node name of one other registered node. The first node to start will not take this parameter; each subsequent node will use it to get access to the global set of registered processes. Since you are using Erlang's `global` registry, only one such connection needs to be made for each new node to gain access to the entire global registry of processes.

  For example, to start a system with 1024 storage processes across 3 nodes, you might invoke the following commands (the different prompts indicate different machines—in this case `ash`, `elm`, and `oak`—on which you are running):

  ```
  {ash:1} erl -noshell -run key_value_node main 10 node1
  {elm:1} erl -noshell -run key_value_node main 10 node2 node1@ash
  {oak:1} erl -noshell -run key_value_node main 10 node3 node1@ash
  ```

- It must use the name provided on the command line as the node short name to start the network kernel, as on previous assignments (don't forget to start EPMD as well!). Once the network kernel has been started and the node has been connected to the existing node supplied on the command line (if any) with the `net_kernel` function `connect_node/1`, the node should use the results of the `global` function `registered_names/0` to find the other nodes that are already registered, pick an unused node number (from 0 to $2^m - 1$ inclusive), and negotiate with the existing nodes (most likely using the `global` function `send/2` for message sending) to rebalance the ring of storage processes.

  Effectively, this means that the first key-value node to start up will pick some number (most likely, 0), then start (with `spawn/3`) and globally register 2^m storage processes (and any necessary additional support processes) within the same Erlang virtual machine. The second node to start up will see that the first node exists, pick a different number, and (somehow) take responsibility for its share of the 2^m storage processes. Remember that you are guaranteed sufficient time for a node to completely start up and for the system to rebalance before any other nodes join or leave the system.

- It must generate sufficient console output to show exactly what is happening with respect to message sends/receives and state changes. Each message received or sent, and each state change, should be described in human-readable fashion on standard output (including information about the particular process that is sending/receiving the message or changing its state, since you now have more than one process per Erlang VM). In particular, it must be absolutely clear which storage processes are storing particular key-value pairs and which storage processes are communicating with each other. Each line of output must be timestamped to at least millisecond precision as required for previous assignments. In this way, since you are running on multiple machines on the same subnet (or multiple Erlang VMs on the same machine) and they are very closely synchronized in real time, you can use the outputs of multiple nodes to satisfy yourself that the system is working correctly.

  This will likely generate an extremely large amount of output for systems with large values of m.

- It *must not* use the global locking and transaction facilities provided by the Erlang `global` module. You will likely be tempted to use these facilities to guarantee that the same value is stored for a key across any redundant storage processes you have in your system; such usage is not allowed on this assignment.

In addition to writing the program, you must perform sufficient testing on it to convince yourself (and us!) that it works properly. There are many ways to do this:

- Create systems with various values of m and exercise them manually by sending messages to their storage processes from an Erlang shell, by killing nodes (forcefully or by sending `leave` messages), etc.

- Write test programs that automate the process of sending messages to your systems in particular patterns that allow you to evaluate their functionality; it is permissible for test controller programs to directly use the `global` registry to locate your storage processes in order to send them messages.

- Write shell scripts, invokable via `ssh` (having SSH keys and a working SSH agent will make this process much easier and prevent you from having to type your password many times), that allow you to automatically start multiple nodes on multiple lab Macs from a single terminal window and interact with them by sending messages.

The specific mechanism you choose for testing your system is up to you. Whatever you choose, you must provide us with sufficient information in your submission (descriptions of tests, logs, etc.) to convince us that your implementation works. Since the assignment has specific requirements for the nodes' command line parameters, as well as specific communication requirements between storage processes and the outside world, we will also be able to test your submissions in a semi-automated fashion.

Hints

The Erlang `global` module gives you global names, but you need not necessarily use global names for everything. For example, the Erlang `rpc` module has message broadcast functions that allow you to send identical messages to identically named processes on multiple nodes. One way to achieve redundant storage of data might be to duplicate processes across multiple nodes (remembering the communication restrictions of the ring and chords) while having only the "real" ones registered globally. Another way to achieve redundant storage of data might be to simply have the storage processes themselves redundantly store data for values of $h(x)$ other than their own IDs.

The built-in function `erlang:monitor/2` will likely be useful to you for detecting and responding to crash failures. You may also find `net_kernel:monitor_nodes` (two different variants) useful.

The `ets` module in Erlang has native implementations of tables, which you can use in your storage processes to store key-value pairs. You don't *need* to use `ets`—you can simply use Erlang tuples {*key*, *value*} stored in a list and appropriate lookup functions, because efficiency of lookups at individual storage processes is not the goal of this assignment. However, you might find them useful. Note that the number of `ets` tables you can store in a single Erlang VM is limited by default to about 1400, which may be problematic for large values of m. You can increase the limit as described in the `ets` documentation; if you do so, be sure to document it so that we know to do so when running your system.

Grading and Submission

40% of the grade for this assignment is based on a description of the algorithm(s) you use to set up and maintain your key-value store (similar to the description you provided for Assignment 4: the possible states, information stored by, message types handled, and actions taken by each type of process in your system, as well as an overall description of the set of processes in your system, how they interact, and how you deal with nodes joining and leaving the system) and brief correctness arguments. The remaining 60% is based on your implementation and testing. Your algorithms will be evaluated on their correctness and completeness (75%) and on how convincing your correctness arguments are (25%). Your implementation will be evaluated on its functional correctness (75%), determined from your testing strategy as well as from our testing, and on its code style/readability (25%).

To submit your assignment, upload 3 files to the Assignment 5 page on Moodle:

- one of `algorithm.pdf` or `algorithm.txt`, a PDF or text file (respectively) containing a writeup of your algorithm(s) and accompanying correctness argument(s). You can use any document preparation software you choose to generate a PDF or

you can submit a plain text document; no other document formats (LaTeX source, Word, Pages, RTF, etc.) will be accepted.

- `key_value.zip`, a Zip file containing your Erlang source files (you may only have one, but you may also choose to put support routines in another module or to include Erlang-based testing code), details about your testing strategy (in a text file), any test programs/shell scripts/etc. you have written, and any other files (such as logging output) that you wish to provide as evidence that your program works.
- `readme.txt`, a text file containing a list of your team members (this is *extremely important*), any information you think the graders need to know about your code, and a brief description of your experiences implementing the homework assignment.

Only one team member needs to upload the files to Moodle; be *absolutely sure* that `readme.txt` includes the names of all team members!

Evaluation

If you put no effort into your `readme.txt` file, you may be penalized up to 10% of the value of the assignment. As long as you write some genuine reflections about your experiences working on the assignment, you will not incur a penalty.

If the names of all team members (including the one who submitted the files) are not listed in `readme.txt`, all team members will receive a 0 on the assignment.