Distributed Systems Assignment 5: Distributed, Fault-Tolerant Key Value Store

Eleanor Cawthon & Claire Murphy

April 13, 2014

1 Algorithm

1.1 Overview

Our algorithm takes advantage of the connected chord system to store accessible backup processes while adhering to the requirements regarding node assignments for processes. Each globally registered storage process has an associated global process which stores the same map as the global process's last neighbor(the direct opposite on the ring). Due to this distribution of information, we are guaranteed that as long as we don't allow one node to hold more than half the processes(except in the case that there is only one node total), any node that dies will have a full copy of its information on a series of backup processes.

In order to maintain this invariant(that no node has more than half the processes), we have decided that any joining node should pick a name that divides the current largest node directly in half, tiebreaking towards smaller numbers. Upon the death of a node, its predecessor(the node with the next smallest number mod 2^M) takes over its processes. In the event that the expanded node now has more than half the processes, it changes its name so that it has becomes responsible for exactly half, and tells its predecessor to pick up the extras. Only once its predecessor has completed this can the system return to normal, prepared for the addition or subtraction of another node. The technical details are slightly more complex in order to be sure that no two processes on the same node share a name, (in this case a storage and its backup), but this is the resulting concept.

In our implementation, then, we have three types of processes: Node Processes, Storage Processes, and Backup processes.

- **Node Processes** do operations at the node level; they are named according to the hashable naming conventions defined in the assignment. There is only one per erlang node.
- **Storage Processes** are responsible for receiving and processing outside requests. They each keep a list of key, value pairs which they use to respond to lookup and store requests.
- **Backup Processes** have the same ID numbers as the Storage Processes they back up, but are given globally registered names that identify them as backups. They keep the same updated map as the Storage Process they back up, but are on a different node(unless there is only one node).

1.2 Message Passing

When sending messages between processes in a system with 2^M processes, we take advantage of the chord system's guarantee that any storage process can be reached from any other storage process in less than M jumps. We use a greedy but effective algorithm: whenever we want to reach a storage process from another storage process, we find the neighbor who is closest to our intended recipient while still being before it, and send the message there. That way we send the message at least halfway with each "hop" - it is essentially a binary search.

Similarly, when passing messages from one node to another, we greedily send the message as far as possible by finding the neighbors of our final storage process, and using the above algorithm to pick a good storage process(and therefore parent node) to forward the message to from among those neighbors. In some cases, this requires more total messages than choosing a "forwarding" node from a more complete list of the sender node's neighbors(such as one generated by polling all of the storage processes the sender node owns). However, is computationally much simpler to only consider the neighbors of one storage process.

1.3 Types of Processes

There are three types of processes in our system. The first are a number of node processes, which keep track of the node's name and act on behalf of the node. The second type of processes in the system are storage processes, which respond to store and lookup requests, and are associated with a node process. Finally, our system has backup processes, each of which contains identical information to one storage process, whose number it shares. Backup nodes are used for redistribution of information after a node dies or joins.

1.4 Node Process States

1.4.1 Joining

In the joining state, the first node process will initialize all 2^m storage processes and all 2^m backup processes, and will proceed to monitoring. Each subsequent node:

a. Connects its node to the specified Erlang cluster

b. Sends a message to a single, arbitrary, globally registered storage process requesting to join

c. Waits to receive a `join_ack` message from some existing storage process in the cluster, which contains the Node ID number for the joining node to use, the Node ID number of its successor node (some of whose processes it will take), and a list of storage process IDs for which it should become responsible. The node which sent the ack has already removed the corresponding processes in the successor node, so the system is ready for the new node to initialize its storage processes.

d. Globally registers itself

e. Spawns and globally registers all the processes it will assume responsibility for (storage processes and backups)

f. As it spawns each process, it registers it globally as either `StorageProcessN` or `BackupN`, where `N` is an integer between 0 and 2^M, as appropriate.

g. Waits to receive a `serving` message from each spawned storage process, indicating that it is ready to serve requests.

h. Transitions to monitoring.

1.4.2 Monitoring

In the Montoring state, a node process receives messages relating to the cluster's topography and handles them appropriately. Specifically:

a. In the event that its successor dies, begin the rebalancing process.

b. If it recieves a command to take over some number of storage processes from its successor, this means its successor's successor has died. It enters the Overflow state, below.

c. If it receives a `join_request`, it calculates the information it needs to send the joining node and either sends this directly to the joining node or forwards it to the node's new predecessor (using the chorded message passing). If it is the node's predecessor, it also kills the current versions of the processes that the new node will be responsible for, making room in the global registry for the new node to register its processes.

d. It forwards any messages it receives in transit to other nodes as part of the chord algorithm, unless it is the recipient specified in the message.

e. In the case of a `node_list` request, it adds its own name to the list in the message before forwarding it, and upon seeing the message a second time, it sends the response directly.

1.5 Overflow

This is the state a node enters when it learns it needs to take processes from its successor to allow its successor to take over a dead node. It receives a range of storage process IDs to take over, then it calls `take_responsibility`, which takes a `Range` and:

a. For any elements of `Range` whose backups are on this node, converts those backups to storage processes, and converts their corresponding storage processes (halfway across mod 2^n) to backup processes.

b. For the remaining elements of Range, requests their maps from their globally registered backups and then spawns and globally registers storage processes with those maps.

It then sends an acknowledgement to the node that triggered Overflow, and re-enters the monitoring state.

1.5.1 Rebalancing

In the rebalancing state, a node process will:

a. Determine its new successor by examining the global registry

b. Determines the number of processes from the dead node that it must now take over

c. Evaluate whether it may take all of the now-dead processes as well as its own, or whether that will overload it(give it more than half the processes in the case that it is not the only node).

d. If it can take all of its successor's processes safely, it will. Then it will change to Monitoring again.

e. If it cannot take all processes and still have half the processes or less, it:

   (a) Determines which of the deceased processes it already has the backups for (guaranteed if the new processes would push it over half of the total)

   (b) "switches" those backup processes to storage process, and their backups to storage processes. This is effectively removing the processes it doesn't have room for, while maintaining backups that its predecessor can ask for. The "switch" is done through sending a message to the appropriate processes and collecting all responses before proceeding.

   (c) Unregisters itself and re-registers itself with the Node ID that corresponds to its calculated new set of storage processes.

   (d) Routes a message to its predecessor node, telling it to transition to Overflow and sending it the numbers of the processes it must take (whichever are necessary for the successor node to have exactly half of the storage processes in the cluster–this is the same set of its own processes that it "switched").

   (e) Waits to receive an acknowledgement from the predecessor indicating that it has completed the Overflow state.

   (f) Spawns and initializes the remainder of the processes from the deceased node (Those whose backups are elsewhere)

f. It then re-enters the monitoring state.

1.5.2 Leaving

A leaving state is assumed to do so with no warning. Nothing is done in this state to ease the transition. If a node leaves or dies, all of its processes will be unregistered and halt as well.

1.6 Storage Process States

1.6.1 Joining

In the joining state, the first storage processes(initialized by the first node) will skip the joining phase and directly begin servicing requests. Otherwise, a storage process will remain in the joining state until it has completed the following actions: n Procedure for a process ID P responsible for map P and backup B, where $B = (P + 2^{M-1}) \mod 2^M$:

1. Send a message requesting P.map from P.backup (on the node that hosts process ID B.

2. Send a message requesting B.map from B.primary
3. Enter receive loop:
   (a) When we get a backup map, spawn and register the backup process with that map.
   (b) Once we've gotten both maps, alert P's host node's process with a `serving` message and transition to start serving.

1.6.2 Servicing Requests

In the standard, servicing requests state, a storage process will:

- receive requests for lookups and respond to them or forward them to the appropriate responder
- respond to requests for backup copies of its map
- When it receives a `store` message, it propagates the change to its backup and waits to receive acknowledgement before replying to the sender of the `store` message.
- respond to requests for replacement copies of the map by unregistering itself and then sending the map

1.6.3 Dying

A process can be told to die if it has been replaced. The process will then be killed.

1.7 Backup Process States

Backup processes have one state.

1.7.1 Backup

In its backup state, a backup process will:

- respond to requests for maps by sending its map to the globally registered process opposite it(the only one allowed to request maps from it.)
- Updates its map and sends an acknowledgement when it receives update instructions from its associated storage process
- Converts to a storage process when told to switch

1.8 Message Types

Messages that may be received by each sort of process

1.8.1 Node Process Messages

- {`nodedown`, *ErlangNode*} Sent by the system when a node you monitor goes down. In this case, each node monitors its successor(and unmonitors it if its successor changes due to a join and it's monitoring a new one), so it will receive this message only if its successor goes down.
- {`take_processes`, *Recipient*, *Range*} A message forwarded around the ring until it reaches the predecessor of the node that sent it. A recipient of this message will take responsibility for the storage processes whose numbers are listed in Range. It is assumed that by the time this message is sent, 1) The processes in Range now have no globally registered versions, so the recipient node is free to register them 2) The recipient node can request the maps of the Range nodes from its backups, which are fully functional by now.
- {`monitoring`} Sent by a node that has finished taking processes as instructed by a `take_process` message to the node that prompted it (which is now its successor) to indicate that it has finished taking over the processes and re-entered the monitoring state. The rebalancing process blocks while waiting for this, as described above.

- {`join_request`, *Pid*} Sent by an unnamed, uninitialized new node process. This causes the receiving node to forward a more informative version around the circle until it reaches the appropriate node.
- {`join_request`, *Pid*, *ItsPredNum*, *ItsNum*, *ItsSuccNum*}: Internal message sent by the nodes until it reaches the node that will be the predecessor of the joining node, at which point its predecessor sends it all the appropriate information in a `join_ack`.
- {`join_ack`, *Num*, *SucessorNum*, *Range*} The message received by a joining node when its request has reached the appropriate node.
- {*Pid*, *Ref*, `leave`}When a node process is told to leave, it calls `node_leave`, which kills the erlang node.
- {*Pid*, *Ref*, `node_list`, *Starter*, *AggregateList*} Initially sent to a node process from the storage process that received it, this message is passed around the nodes, and once every node has added its name to *AggregateList* the *Starter* node sends the list back to *Pid*.
- {*Ref*, `result`, *NodeList*} Sent by a `monitoring` node process to the *Pid* specified in the `node_list` request. Contains the current list of nodes, barring failures since the request was sent.

1.8.2 Storage Process Messages

A storage process may receive the following messages. Messages that can only be recieved in a specific state are specified.

- {*Pid*, *Ref*, `store`, *Key*, *Value*} Sent to a `serving` storage process by the outside world or another `serving` storage process. The storage process will then hash the key to a specific storage process, and forward the message as-is to the best neighbor to get it to that process as per section 1.2. If it is the right process, it will incorporate the included /*Key*, *Value*/ pair into its map. It will send {*Pid*, *Ref*, `write_backup`, *Key*, *Value*} and wait to receive an acknowledgement that the change has been committed before respond to *Pid* with the following:
- {*Ref*, `stored`, *OldValue*} Sent by a `serving` storage process to the *Pid* included in the `store` request, this message includes the value that was overwritten by the new store. If no value was overwritten, OldValue is`no_value`.
- {*Pid*, *Ref*, `retrieve`, `Key`} Sent to a `serving` storage process by the outside world or another `serving` storage process. The procedure repeats as outlined above if the recipient process is not the process that the *Key* hashes to. Otherwise, it will retrieve the associated *Value* from its map and respond directly to the *Pid* in the message with the following:
- {*Ref*, `retrieved`, *Value*} Sent by a `serving` storage process to the *Pid* included in the `retrieve` request, this message includes the value that was retrieved.
- {*Pid*, *Ref*, `first_key`} Sent to a `serving` storage process by the outside world. The storage process will then send the more verbose version below through the ring, starting by sending its own alphabetically first key to its first neighbor.
- {*Pid*, *Ref*, `first_key`, *Starter*, *First*} Sent to a `serving` storage process by another `serving` process to find the alphabetically first key. If the receiving process finds that *Starter* is its own process number, it knows the request has gone all the way around the ring, and it sends a response to the *Pid*. Otherwise, it replaces the *First* key with its own first key if applicable, and forwards the message to its first neighbor.
- {*Pid*, *Ref*, `last_key`} Sent to a `serving` storage process by the outside world. The storage process will then send the more verbose version below through the ring, starting by sending its own alphabetically last key to its first neighbor.
- {*Pid*, *Ref*, `last_key`, `Starter`, `Last`} Sent to a `serving` storage process by another `serving` process to find the alphabetically last key. If the receiving process finds that *Last* is its own process number, it knows the request has gone all the way around the ring, and it sends a response to the *Pid*. Otherwise, it replaces the *Last* key with its own last key if applicable, and forwards the message to its first neighbor.

- {*Pid*, *Ref*, `num_keys`} Sent to a `serving` storage process by the outside world. The storage process will then send the more verbose version below through the ring, starting by sending its total number of keys to its first neighbor.
- {*Pid*, *Ref*, `num_key`, `Starter`, `Total`} Sent to a `serving` storage process by another `serving` process to find the total number of keys in the system. If the receiving process finds that *Starter* is its own process number, it knows the request has gone all the way around the ring, and it sends a response to the *Pid*. Otherwise, it adds its number of keys to *Total*, and forwards the message to its first neighbor.
- {*Pid*, *Ref*, `node_list`} Sent to a `serving` storage process from the outside world. The storage node composes a more verbose version of this message, (seen in1.8.1,) starting with a singleton list of its own parent node. It then sends this verbose message to its parent node, so that the nodes may send the message around the ring and aggregate responses.
- {*Ref*, `Result`, *Result*} Sent by a `serving` storage process to the outside world in response to a request for information. *Result* will vary, as this same message format is used in response to multiple requests for system information.
- {*Pid*, *Ref*, `leave`} Sent to a `serving` storage process
- {*Pid*, `request_backup`} Sent to a `serving` storage process by a `joining` storage process which would like to make a backup process for the storage process's information. The storage process sends the map back to the *Pid* it includes, and continues `serving`.
- {`backup`, *Map*} Sent by a `serving` storage process to the Pid which sent the request.
- {*Pid*, `switch`} Sent to a `serving` storage process by its parent node, telling the storage process that it needs to become a backup process, because its parent node is taking over its natural opposite and giving it up. Its parent's predecessor is therefore going to need to be able to spawn a copy of it from a backup.

1.8.3 Backup Process Messages

- {*Pid*, `request_map`} Sent to a backup process by a `joining` storage process. The backup process will respond by sending its map back to the *Pid* in the request.
- {`map`, *Map*} Sent by a backup process to the *Pid* included in the original request.
- {*Pid*, `switch`} Sent to a backup process by its parent node, telling the backup process that it needs to become a storage process, because its parent node is taking over the storage process it backs up, and giving up the backup version. Its parent's predecessor is therefore going to need to be able to spawn a new backup from the storage process it will become.
- {`bACKup`, *Ref*} Sent by a backup process in response to `write_backup` after the backup process incorporates the change into its map. *Ref* is the unique identifier that came with the store request.

2 Correctness Argument

2.1 Some System Invariants

Our algorithm is centered around maintaining a few simple invariants. These are the invariants specified by the assignment:

1. After a process dies, all of the information which was stored in that process is still present in the system, unless the deceased process was the last one.
2. As long as all nodes have finished their last rebalancing process, they are able to serve all requests.
3. Each node is responsible for a consecutive list of all processes numbered from N to M, where N is the number of the responsible node, and M is the number of the next highest node (modulo the number of processes).

These are the invariants we have added for the purposes of our algorithm:

1. As long as all nodes have recovered from the latest rebalancing and are `monitoring`, every storage node will have one complete backup on the same node as its last neighbor. (For example, in a system with storage processes numbered 0 through 7, 6 will have a backup on the same node as 2(and vice versa), 7 with 3, 0 with 4, etc.)
2. As long as there is more than one node in the system and all nodes are in their standard `monitoring` state, no node may be responsible for more than half of the storage processes at once.

It is evident how the our two invariants allow us to maintain the first one: because each storage process is backed up on the node opposite it, no one `monitoring` node will ever contain both a node and its backup. Therefore no single fault can permanently remove a storage process's data from the system, (unless it is the last node, in which case all data will be lost).

2.2 Maintaining Invariants

Our algorithm for maintaining our invariants is as follows.

2.2.1 Nodes Joining

When a node joins, if it is the first node, it initializes all storage processes directly to the request-serving state and proceeds directly to monitoring. If it is not the first state, it finds the node with the longest sequence of storage processes, and takes half of the processes of that node. This way, the second node is guaranteed to take half of the processes from the first process, making sure that as soon as we have more than 1 node our invariant is maintained. Additionally, whenever a storage process is "taken over" or "restored", its associated backup process(meaning the backup the storage process opposite it) is as well.

Because joining nodes divide the longest list of processes in half, in a system where nodes only joined, it would be evident that a system with only joining nodes would maintain our invariant that no node may have more than half the processes. In fact, in the event that this is the case, a joining node would fix the issue. Therefore a malicious adversary would gain nothing by joining nodes.

2.2.2 Nodes Leaving

When a node leaves, we use a simple algorithm to rebalance: We tell the previous node to take responsibility for all the storage processes which were stored on the now-deceased node. This has the advantage that no node needs to change its name.

However, in the case where many consecutive nodes die before a new one can join, the predecessor to that sequence of nodes can easily contain a majority of processes, violating our invariant that no node may have more than half. To remedy this, every time a node dies, we check if we can safely overtake all of its nodes. If so, we do so. Otherwise, we take over all of the processes, but effectively discard the first k processes(where $total - 2^(M - 1) = k$). We also change our name such that we should have exactly half the total processes(which is the amount we keep). Once we have completed this, we send a message around the ring to our predecessor node, telling it to take responsibility for the extra processes we dropped. Therefore we have only the maximum allowable number of processes, no other node is in danger of having too many, and our invariant is maintained. Only once we have a message from our predecessor confirming that it is finished creating the new processes do we return to `monitoring`, ready for more joins or leaves.

This rebalancing method guarantees that our invariants are maintained in `monitoring` processes, no matter what order processes `join` or crash, as long as only one `join` or `leave` happens at a time, and only while the system is `monitoring`

2.3 Processing Requests for System Info

All requests for system information are processed in the same way–the first process sends an internal, more descriptive version of the request in order around the ring, aggregating the necessary information. Barring crashes which interfere with message passing, the first process eventually receives the message back and is able to reply. We use this method for the `node_list` as well, but we pass the message through node processes, not storage processes.

2.4 Message Passing

In general, we pass a message as far as we can without passing the recipient. Due to the chord system, we pass any message at least halfway to its recipient with each hop. Specifically, if Node A wants to send a message to Node B, `node_send_to_node_proc` computes Steps, the logarithm base two of the distance modulo 2^M between them, rounded down. It then determines the node to which storage process $A + Steps$ belongs. If $A + Steps$ is also on Node A, the function is called recursively with $A + Steps$ as A. If $A + Steps$ is on a node other than A, it is guaranteed to be a legal move because of how we calculate Steps, so the node passes the message on to the new target.

Storage processes only pass messages to their successors or to their backups. Both of these are legal: Their successor is their first neighbor, and we have defined backups to always be located on the same node as their last neighbor.