Distributed Key-Value Storage Algorithm
CS181E — Distributed Systems
Assignment 5

Alejandro Frias   Ravi Kumar   David Scott

April 13, 2014

Algorithm Description

Table 1: Description of messages in the system

Message Description	**Erlang Pattern**
Store Request. Sent by either OW or a `storage_process`to a `storage_process`to store `value` for `key`. These are only ever received by storage processes in the network. When an `storage_process`receives such a message, it first checks if the hashed key is equal to its own ID. If it is, then it stores the value for the specified key, and sends a `backup_store` message to its `store_handler`. If not, it forwards it to the closest storage process to the destination (closest here meaning nearest process *before* the destination, since processes can only send messages forward in the ring).	{Pid, Ref, store, Key, Value}
Stored Confirmation. Sent by an `store_handler`to OW after the corresponding request has been stored in the proper storage process and the data has been backed up in the `store_handler`sending the message.	{Ref, stored, Old_Value}

This table continues on the next page...

Table 1: Description of messages in the system

Message Description	**Erlang Pattern**
Retrieve Request. Sent by OW and storage processes; received by storage processes. When a storage processes receives such a message, it checks if the hash of the key is equal to its own process ID. If it is, the storage process has the value for that key, and replies with a retrieve response. If not, it forwards it to the closest storage process to the destination (closest as defined above).	{Pid, Ref, retrieve, Key}
Retrieve Response. Sent by storage processes to the OW. After a storage process receives a retrieve request meant for it, it looks up the relevant value and reports it to the requesting process in the OW.	{Ref, retrieved, Value}
First Key Request. Sent by the OW to storage processes, and by storage processes to storage handlers. When a storage process receieves this, it will forward the message up to its storage handler. When such a message is received by a storage handler, it will start a first key computation by adding the ref to its list of ongoing computations and sending a First Key Computation message to the next node in the ring.	{Pid, Ref, first_key}
Last Key Request. Sent by the OW to storage processes, and by storage processes to storage handlers. When a storage process receieves this, it will forward the message up to its storage handler. When such a message is received by a storage handler, it will start a last key computation by adding the ref to its list of ongoing computations and sending a Last Key Computation message to the next node in the ring.	{Pid, Ref, last_key}
Num Keys Request. Sent by the OW to storage processes, and by storage processes to storage handlers. When a storage process receieves this, it will forward the message up to its storage handler. When such a message is received by a storage handler, it will start a num keys computation by adding the ref to its list of ongoing computations and sending a Num Keys Computation message to the next node in the ring.	{Pid, Ref, num_keys}

This table continues on the next page...

Table 1: Description of messages in the system

Message Description	Erlang Pattern
Node List Request. Sent by the OW to storage processes, and by storage processes to storage handlers. When a storage_processreceieves this, it will forward the message up to its store_handlerWhen such a message is received by a store_handler, it will query the global registry for the list of nodes and report that data to the requester.	{Pid, Ref, node_list}
Request Result. Sent to the OW by a storage handler. This reports the result of a First Key, Last Key, Num Keys, or Node List Request to the original requester after the storage handlers have finished computing the result.	{Ref, result, Result}
Failure Notification. Sent to the OW by storage handlers or storage processes to notify the OW that a particular computation has failed.	{Ref, failure}
Leave Request. Sent by the OW to storage processes and by storage processes to storage handlers. When received by a storage_process, it forwards the message to its store_handler. When received by an store_handler, it immediately kills all storage processes on the node it is running on, and kills itself.	{Pid, Ref, leave}
Backup Store Request. Sent by store_handlerand storage_process, and received by store_handler. If a store_handlerreceives this message from a storage_process, it forwards the message to the next store_handler. If an store_handlerreceives this message from another store_handler, it will back up the data in the message, then notify the OW of the store's success and the old value.	{Pid, Ref, backup_store, Key, Value, ProcessID}

This table continues on the next page...

Table 1: Description of messages in the system

Message Description	**Erlang Pattern**
Messages About Keys. Sent and received by `store_handler`. If `Ref` is in the list of the receiver's in-progress computations, the computation is over and the received message contains the result. The receiver will then send the result `ComputationSoFar` back to the OW. Otherwise, it will update `ComputationSoFar` with its relevant value and forward the message to the next node's `store_handler`.	{Pid, Ref, *_key, ComputationSoFar}
Joining Behind. Received and sent by `store_handler`. A `store_handler`will send this when it is joining to the next node's `store_handler`to indicate that it is joining behind the recipient in the ring. When received, send all stored backup data to the sender, then delete all backup data for processes numbered less than `NodeID`.	{Pid, joining_behind, NodeID}
Joining in Front. Received and sent by `store_handler`. A `store_handler`will send this when it is joining to the previous node's `store_handler`to indicate that it is joining in front of the recipient in the ring. When receiving such a message, kill the data storage processes that the new node is now running (i.e. the ones numbered from `NodeID` to the ID of the node after the new one.	{joining_front, NodeID, DestID}
Node with `NodeID` Died. Received by `store_handler`, sent by a `gen_server` listener started by that particular `store_handler`. This is a notification that the node behind the receiving node has stopped running. When such a message is received, the `store_handler` changes the node's ID to `NodeID`, then uses all of the backup data it's holding to start up new data storage processes. Then it deletes the backup data, and sends a `backup_request` message around the ring, to get the data it should be backing up from the node behind it.	{died, NodeID}
Backup Node Data. Received and sent by `store_handler`. When received, add all the data to existing backup data. Send by a node *A*'s predecessor when node *A* died and *A*'s successor is taking over for it.	{backup_node, Data}

This table continues on the next page...

Table 1: Description of messages in the system

Message Description	**Erlang Pattern**
Backup Request. Received and sent by `store_handler`. If it is received on the node with `DestID`, send each of this node's `storage_process`es an `all_data` message. After compiling all of the results from those requests, send all of this node's stored data to this node's successor node in a `backup_node` message. If this node is not `DestID`, just forward the request message to the next node's `store_handler`. Initially sent by a node which stepped into the void left by a node that died.	{backup_request, DestID}
All data request message. Received by `storage_process` and sent by `store_handler`. When received by a `storage_process`, respond with an `all_data_send` message containing all this `storage_process`'s data.	{all_data}
All data send message. Received by `store_handler` and sent by `storage_process`. The `store_handler` adds the received data to an ongoing list of data and removes the sender from the list of processes it is waiting for. If it's the last response that was being waited for, send the `backup_node` message to the next node.	{all_data, Data}

Algorithm Setup

Types of Processes

Our system involves two types of processes: storage processes and handlers. The storage nodes are responsible for the primary data storage in the system (both storing and retrieving), and the handlers are reponsible for meta-storage stuff, such as backing up of data and handling snapshot-related functionality. On a given Erlang node, there will always be exactly one handler running, and some number of storage processes. In our system, the data on a storage process is backed up by the handler on the node in front of the processes node.

Storage Processes

Storage processes in our system are responsible for receiving all messages from the outside world. When a `storage_process`gets a store message, they forward it to the closest `storage_process`to the destination. When the correct `storage_process`gets the store request, it stores the data and notifies its `store_handler`. When an `storage_process`gets a retrieve message, it forwards it along until it gets to the correct `storage_process`, which reports to the outside world. Finally, storage processes can easily handle `node_list` messages, as they have access to the global registry. All other messages received from the outside world are simply forwarded to the storage process's handler. A `storage_process`has only one possible state, that of waiting for messages. For a more detailed description of what messages a `storage_process`sends and receives and what actions it takes upon receiving these messages, refer to the message description table.

In our system, a `storage_process`stores the following information:

m The size of the system, so that it knows what other processes it can message.

myID The ID of the storage process.

myDict A dictionary of data stored by the process.

myHandlerID The ID of the node's handler, so that it can message it.

Handler

Handlers are responsible for everything else in the system. They handle all requests related to system snapshots, manage when `storage_process`s should be started or stopped due to rebalancing when nodes join and leave, and keep backups of data. Specifically, the storage process for a given node backs up all data for the node behind it. If the node behind it dies, it will use this backup data to start all storage processes that died, ensuring that no data is lost. For a more detailed description of what messages a `store_handler`sends and receives and what actions it takes upon receiving these messages, refer to the message description table.

In our system, a `store_handler`stores the following information:

m The size of the system.

myID The ID of the handler process.

nextNodeID The ID of the next handler in the system. The handler uses this to send messages around the ring and to send data to the next node to be backed up.

prevNodeID The ID of the previous handler in the system, which this node is monitoring for failure.

myBackup A backup of the data held by all storage processes on the previous node.

minKey The minimum key in the backup data.

maxKey The maximum key in the backup data.

myBackupSize The size of the backup data.

myInProgressRefs A list of snapshot computations we have started by sending them around the ring. If we get a snapshot message with a ref in this list, we know the computation is finished and can send the result to the outside world.

myMonitoredNode The node object we are monitoring for failure.

myHandlerID The ID of the node's handler, so that it can message it.

Algorithm Description & Correctness

The setup: Each node has a backup of the node preceding it, held by a non-storage process for that node that we call the handler, since it also handles much of the inter-node communication. Each node is listening to the preceding node for crashes. In this way each node has one node that it is responsible for and one that is responsible for it.

When a node enters the system, it does so in the middle of the node that is responsible for the most storage processes (breaking ties arbitrarily by process ID number). The new node grabs the entire backup data of the node in front of it (since this is backing up the data that the new node's `storage_process`s will be responsible for); it also tells that next node to delete the first half of its backup data since the new node will be backing it up. The new node uses the first half of that data (that is, data corresponding to processes behind it) to create its backup and the other half to start the storage processes it will be responsible for. Additionally, it messages the node behind it to stop running the relevant processes so the new node can run them. In this way, a joining node only needs to talk to two other nodes, the ones that it will be adjacent to. Thus, our system maintains its functionality and correctness when new node's join, as no information is lost and everything stays backed up.

When a node crashes or leaves, the next node finds out, as the next node is the only one listening to it. Since it has the entire backup data of that node, it can immediately startup all of the processes that just died. So it immediately takes over for the dead node, changing its own node number to the one of the deceased. Then, since we lost the backup data of the dead node, it sends a request around the ring to the node just previous to it to get all the data on that node's processes. This takes a maximum of m messages, where there are 2^m processes. In this way we can get the system back up and running immediately and then, while still accepting requests from the outside world, start rebuilding the back up data. When a node dies, the node in front of it takes over for it. This method ensures that our system is correct when nodes die, as the node in front of it has the full backup data for the node that died. Thus, our system works correctly even when nodes die, and doesn't lose any data.

To ensure redundancy, a storage process never communicates back to the outside world directly when a store request comes in. Instead, after the store request is forwarded to the correct storage process, that process stores the new key-value pair and tells its handler process that it did so. The handler then sends this to the next node to be backed up. That next node will then notify the outside world that the value has been stored after it stores the backup. This ensures every store request has been backed up before letting the outside world know that the store is completed. Also, very few messages need to be passed. It takes a maximum of m messages to get the store request to the right storage process and then a couple more after that to back the data up and respond to the outside world, since the next node is guaranteed to be visible to the node that has that storage process. This is part of why we chose to have nodes store the back up of the previous node.

Messages about the system, like first-key and last-key, are forwarded directly to the handler process to take care of. The process will start a message that will go around the ring of nodes and will store that node's contribution to calculating the requested computation. Since each handler has a backup of the previous node, it doesn't need to communicate directly with each of the processes, but instead just sees if the first-key that has been found is better or worse than it's first key, or similar comparison for the other key requests. Once a handler gets its message back, it can tell the outside person the result of the snapshot. The runtime of these system processes scales with the number of nodes. They are guaranteed to work if the a node doesn't crash mid computation. Each node's computation will be constant since it can keep track of the first key, the last key, and the number of keys in it's back up as store requests come in. And since a completely stored value is only considered complete once the handler has backed up one up, we're guaranteed to be accurate.

Retrieve requests are the only messages that don't involve a handler at all. They simply forward the message along the closest chord or if they have the value (or should have the value), return the result to the outside world.

Overall, our system's correctness argument is fairly basic. The system works correctly when the system remains static, as store requests work and ensure that the data is both in the correct storage node and backed up before they finish. When nodes join, we ensure that all data remains backed up, so that the state of our systems stays correct. Finally, when nodes die, other nodes immediately step in and replace the dead storage processes using the backup data, and quickly work to replace the backup that was lost, so that all data in the system is backed up. Thus, our distributed hashtable algorithm is correct even in the face of node's joining and leaving.