The Byzantine-Fault tolerant Distributed Hash Table

Tim Krentz and Charlie Hartsell

Purpose

The purpose of this document is to describe the implementation of a Byzantine-Fault tolerant Distributed Hash Table (BFDHT) based on the Practical Byzantine Fault Tolerance (PBFT) protocol. We will describe the system architecture, software design and functionality, how our algorithm works, the types of threats our system can handle, and how we tested our system. In particular, we will compare the final implementation to the originally proposed design, as detailed in the design document, and provide justification for any design changes.

Design

System Architecture

We implemented BFDHT as a C++ program which owns a thread who has access to he table. We wanted the library to allow for initialization and hosting of new nodes in the table at runtime. This proved to add too much work to fit within our deadline, so we assumed specific nodes would be available at all times. An external program is not required to host a node in order to access the table, but instead may act as an external client if desired. Each client must know of at least 1 non-faulty node within the BFDHT to serve as a table access point. In general, this requires the client to be aware of $f + 1$ nodes, where f is the maximum number of faulty nodes, to ensure at least 1 access point is non-faulty at all times. In practice, we just assumed the access point node wouldn't cause any faults. Clients which opt to host a node within the table may use that node as their access point. We assume a flat network topology where all entities in the network are connected by a single switch. For the purposes of this project, all nodes will be run on a single machine and interconnected via a virtual network provided by Mininet. Figure 1 shows the system layout with standard network connections shown as black arrows and client access paths shown as dashed red arrows. We ran our BFDHT on 4 and 8 mininet node clusters.

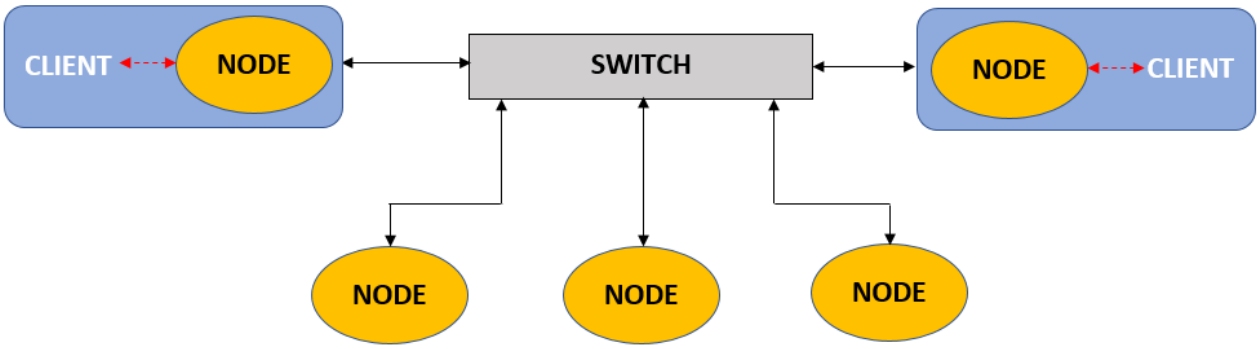

Figure 1. Implemented System Layout.

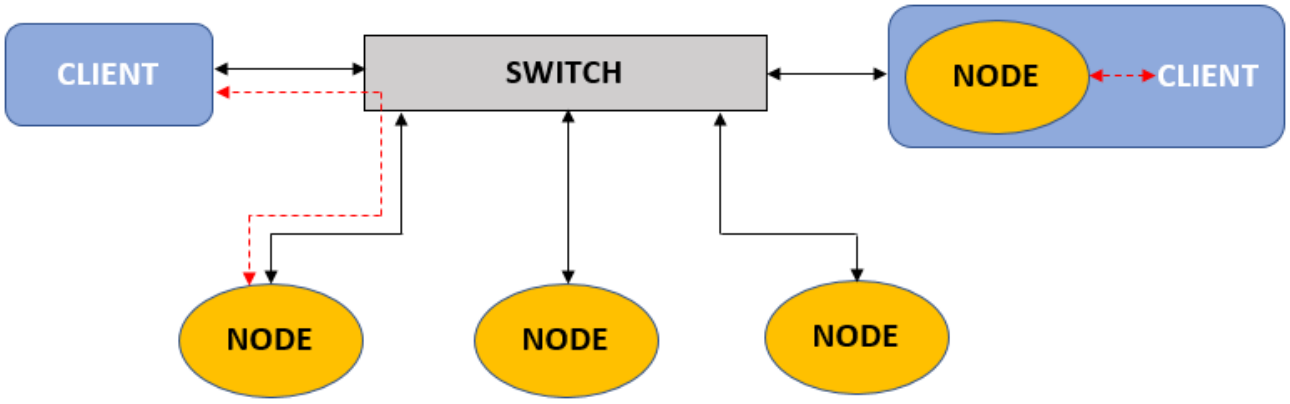

Figure 2. Original System Layout.

Software Design

The BFDHT library is constructed as a C++ class which provides interface functions to the external program including:

1. Node Initialization
2. Put/Get
3. Shutdown

Node initialization consists of creating an instance of the Node class, then calling the Node initialization function within this class. All nodes connect to a single broadcast group, and filter incoming messages with their own ID to act as though they received a point-to-point message. The init function first spawns a dedicated thread, then asserts its own topic to listen on. Once the main thread has started, 10 worker threads are created in a thread pool. The node then begins listening for incoming messages from other BFDHT nodes on its ZMQ_PGM subscribe port. When a message is received, it is dispatched to one of the available worker threads. The worker threads are responsible for executing the PBFT-based exchange with other nodes in the BFDHT. If a message is determined to be valid, the corresponding value is committed to the portion of the hash table stored locally on the node. Otherwise, the message is rejected and dropped. We originally planned to send an error message back in this case, but it wasn't high enough on the priority list to be achieved on time. Once a message has been processed, the worker thread returns to waiting for another task. At any time, the main program thread may call the Node shutdown function which will terminate all threads and return to the main program. A diagram of the Node execution structure is shown in Figure 3. This execution structure had only minor deviations from the originally proposed structure. In particular, node discovery was removed in favor of hard coding the expected node addresses due to the time available. Also, in the case that all 10 worker threads are busy when a request arrives, the message is simply dropped instead of spawning an additional temporary worker.

Clients which do not not wish to host a node may be able to access the BFDHT using an API in the future. Access point initialization would have been similar to a truncated full-node initialization, but it would not spawn a pool of persistent handler threads. Instead, initialization would only perform node discovery to identify a minimum of 2 nodes to use as access points, then begin listening for messages from any access point. For our product, we implemented a simple put()-get() interactive window that

was connected directly to a BFDHT node, in the same process. We wanted to implement functionality where if an acceptable response is not received within a set timeout, the primary node would be assumed faulty and a backup node would be appointed as the new primary, but this was limited by our schedule.

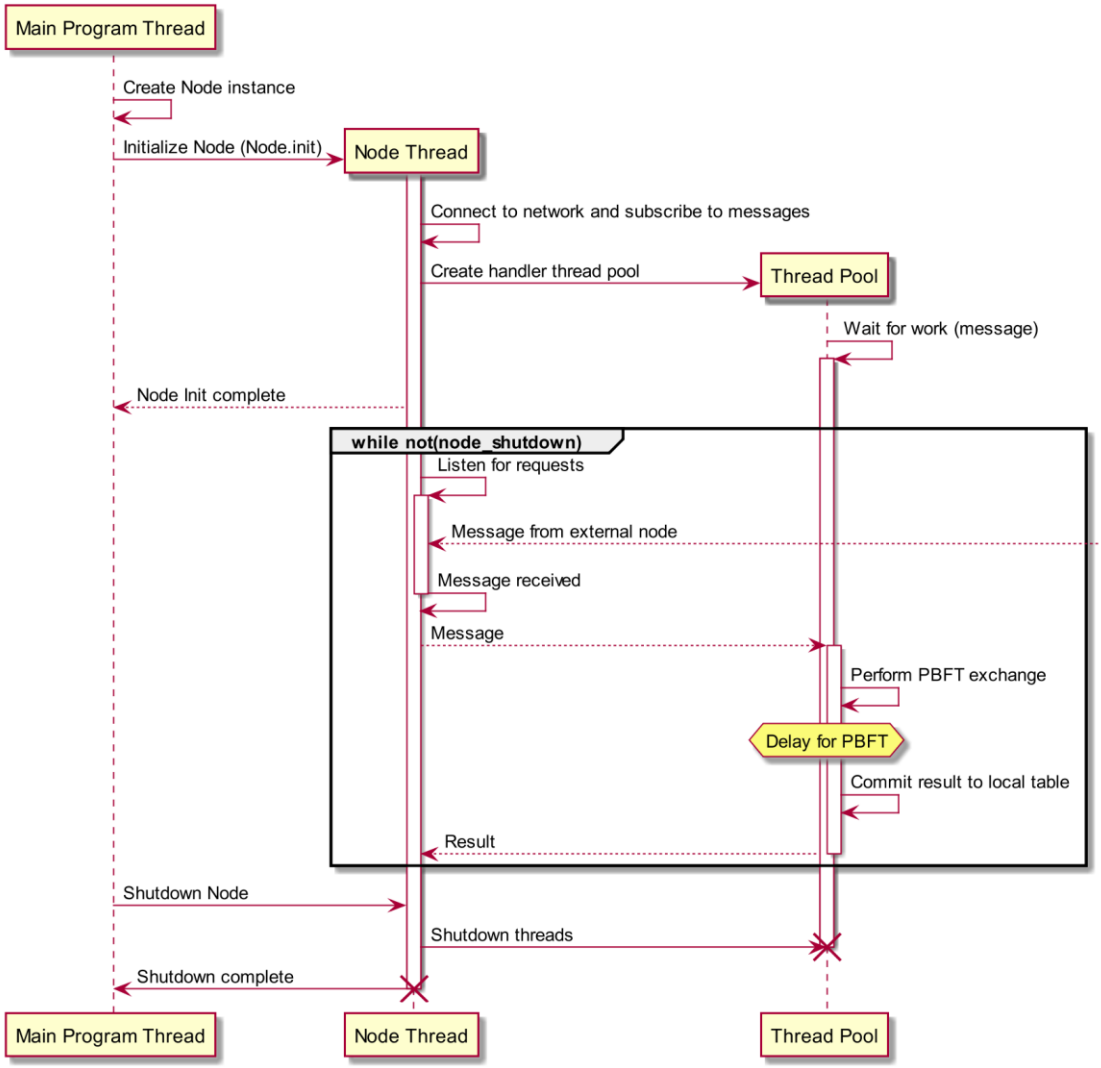

Figure 3. Node Initialization Sequence

Clients may access the BFDHT by making requests using the put and get functions. These functions require that either the node or access point initialization function has already completed successfully, and will otherwise return an appropriate error code. When a client makes a request, an appropriate message is constructed and the request is sent to the client's access point(s). Clients currently hosting a node use their own node as the access point to the rest of the BFDHT. The put and get functions are currently implemented in a blocking manner, and must wait on any necessary message exchanges to complete before returning. Non-blocking versions of these functions are entirely feasible, but were not implemented due to the limited time available.

At any time, a client may call the shutdown function to cleanly terminate any threads which have been spawned and close all communication sockets. The shutdown function will return 0 if successful or an appropriate error code otherwise.

PBFT Transaction Description

BFDHT provides *put* and *get* calls that are implemented as proposed in Practical Byzantine Fault Tolerance. The implemented *put* procedure is a significant change from the originally proposed design. Since we were not able to implement standalone access points to the DHT, clients were required to run a full node instance if they wished to access the DHT. Every client accessing the network used its own gateway node, which was part of the same program as the client itself. Therefore, we determined that allowing for a faulty gateway node would not be significantly different from assuming the client itself was faulty. The differences in the two PBFT-based message exchange protocols are shown in Figures 4 and 5 below. The *put* and *get* calls are implemented as follows:

Put*(key,value)* is requested by a 'client' thread, running on some computing node inside the network. *Put(key,value)* is a request to store some *value* behind the provided *key* in the DHT. BFDHT implements this request in the following stages, visualized in Figure 3:

- ***Request***: The client must have knowledge of more than one *Gateway Node* participating in BFDHT. The Gateway Node runs the same code as every other node in BFDHT. In Request, the client sends a put() request to the Gateway Node, formatted as {PUT, key, value, Requestor IP}.
- ***Pre-prepare***: The Gateway Node looks up the key's hash in the DHT to find the nodes responsible for storing said key, here called *Peer A, Peer B, Peer C*, and *Peer D.* The Gateway Node then sends a *Pre-Prepare* message to the responsible peers, formatted as {PRE-PREPARE, key, value, Requestor IP, [List of Peers]}
- ***Prepare***: The peers send a *Prepare* message to the [List of Peers] from the Pre-Prepare message, and collect Prepare messages from other nodes. These messages are formatted as {PREPARE, key, value, Requestor IP, [List of Peers]}. As soon as a Peer has collected 3f + 1 (in our implementation, 4) Prepare message (including its own), it proceeds to the next stage.
- ***Commit***: Each peer checks for consensus among at least *3f* of the the received *Prepare* messages. If this condition is met, each peer sends a message to the [List of Peers] with the agreed upon value and key. These messages are formatted as {COMMIT, key, value, Requestor IP}. Once a peer has received 3*f* + 1 *Commit* messages (including its own), it proceeds to the next stage.
- ***Response****:* Each peer again checks for consensus among at least 3*f* of the received *Commit* messages. If this condition is met, the key-value pair is stored into the local hash table. Each peer then sends a response back to the gateway node (Requestor IP). These messages are formatted as {RESPONSE, key, value}.
- ***Result****:* The gateway node receives a *response* message from each peer node, then checks for consensus among at least 3*f* of the received messages. If a consensus is reached, the *put* is considered successful.

Get*(key)* is much simpler because we assumed any value is stored by 3f +1 peers in BFDHT. Thus, we didn't need the sections of BFDHT that peers use to confirm they are doing the same thing as other peers, nor do we assume the client is asking for the wrong key. We provided *Get(key)* using the following steps:

1. The client sends a *Request* message to the Gateway Node, formatted as {GET, key, Client IP}.
2. The Gateway Node forwards the message to the responsible Peers, here called *Peer A, Peer B, Peer C*, and *Peer D.*
3. When a Peer received a Request message, it sends a *Reply* message directly to the Gateway Node, formatted as {REPLY, key, value, Peer IP}.
4. The Gateway Node passes the messages to the client, who shares the process. The client collects 3f + 1 Reply messages. From here, the client could choose to check for consensus on what it has received, but this is not yet implemented.

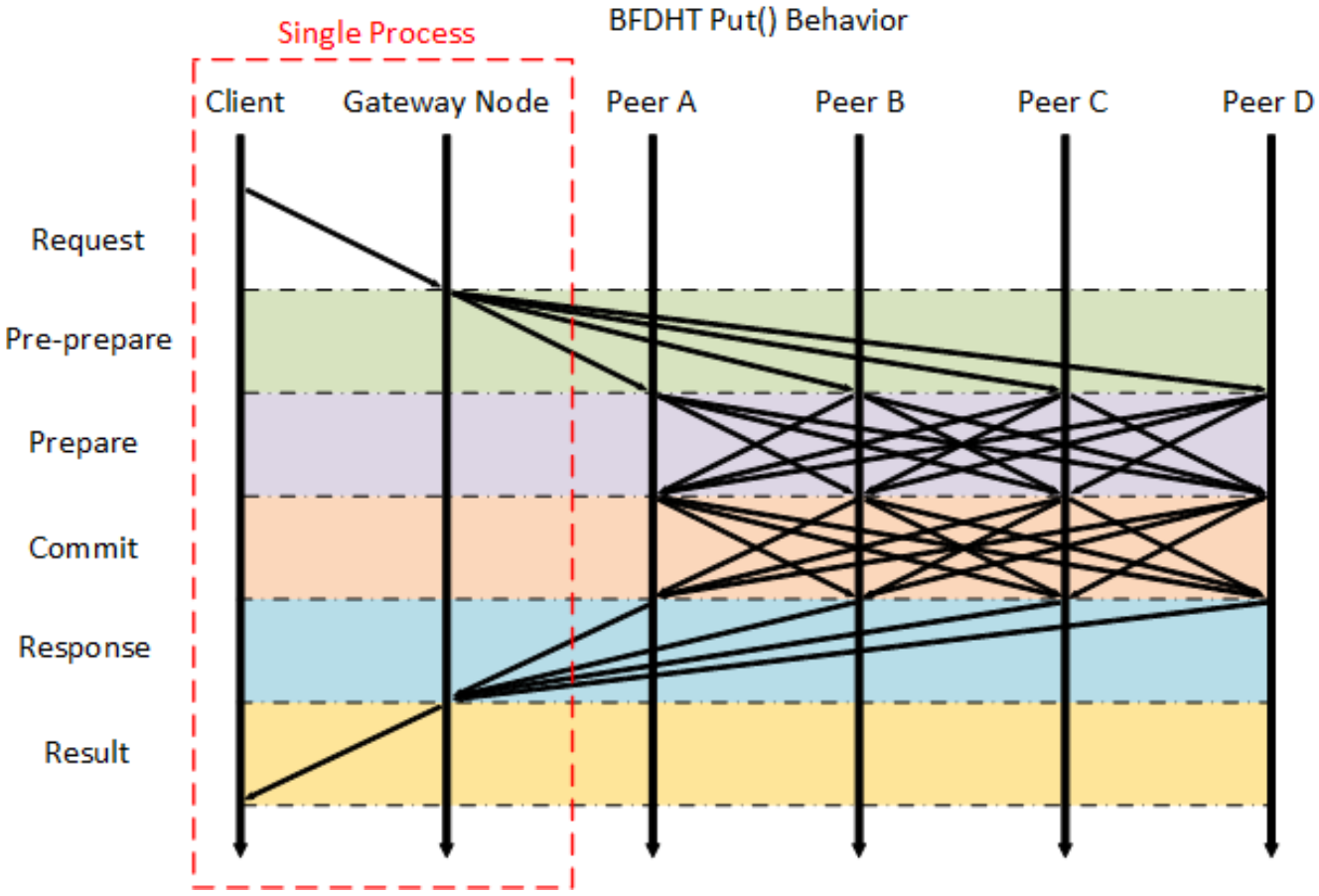

Figure 4. BFDHT put() process.

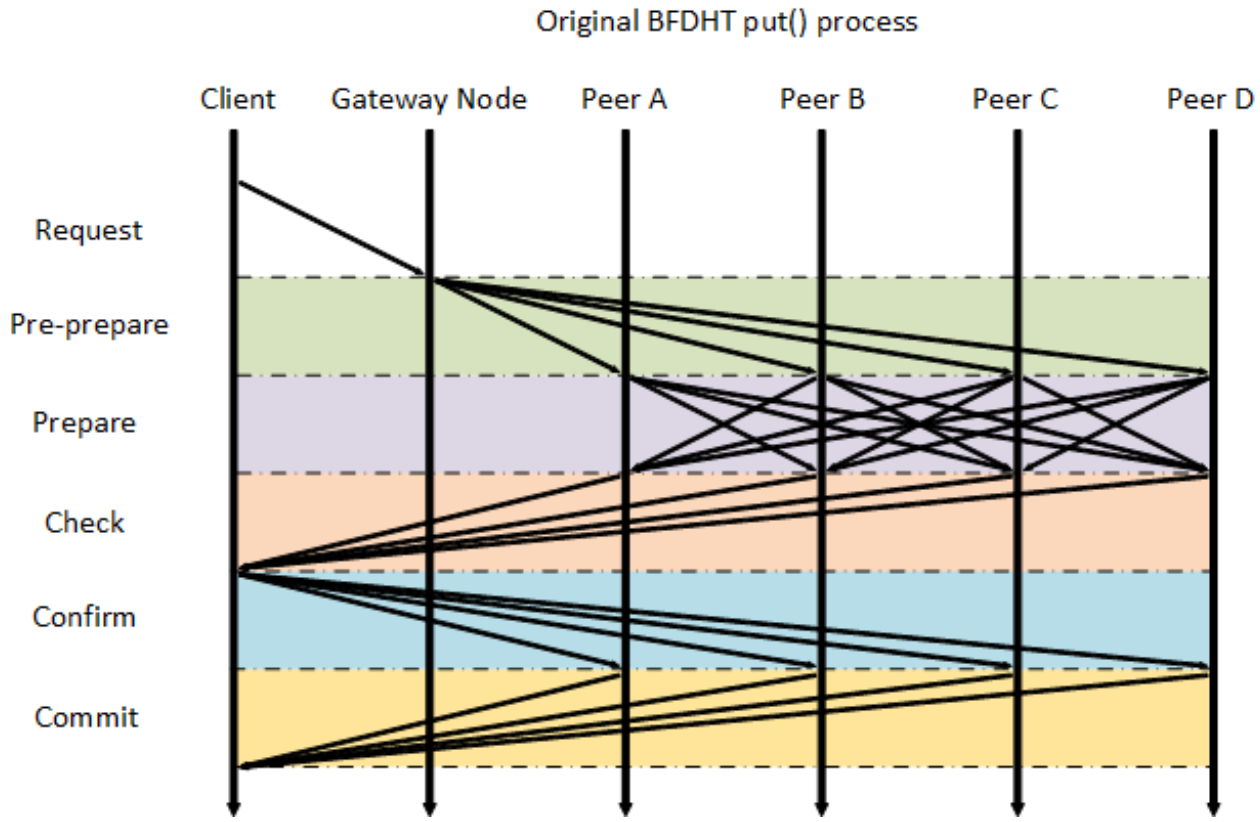

Figure 5. Original put() process.

Threat Models

Due to the time available for implementation of the system, we placed two primary restrictions on the possible failure modes of faulty entities. First, faulty clients accessing the BFDHT are assumed to only fail in a fail-stop manner, and therefore all client requests are authentic and valid. Tolerating faulty clients is outside the scope of this project, but is a possible area for future work. Second, nodes could not impersonate other nodes by altering the source address of a message. Restricting identity spoofing allows messages to be sent without the use of cryptographic methods, reducing the time necessary for implementation. Additionally, the technique is based on PBFT and is scalable to *f* number of faulty nodes if data is replicated on at least $3f + 1$ nodes. However, we considered cases with only a single faulty node.

Faulty nodes are allowed to change the contents of a message, or drop messages entirely. Any single peer within the BFDHT is allowed to present such faulty behavior. Non-faulty nodes only shut down by following the proper procedure for exiting the BFDHT, but faulty nodes may shut down abruptly with no notice to the rest of the system. Additionally, messages may occasionally be lost due to faults in the network itself. A client's request may be ignored entirely if the chosen gateway node is faulty. Timeouts are used in parts of the system to address cases of dropped messages or faulty gateways. Future revisions will track when a particular gateway or other node is non-responsive for an extended period of time, so the node may be assumed faulty.

A responsible DHT peer can be malicious by lying about the value it will store, or by not responding at all. These threats are handled in the Prepare phase. Most of the messages described herein have an accompanying timeout, so lost messages are handled by a new attempt for that request. Right now, we cannot handle an active node going offline, because our key-based routing is fixed. Improving on this is high-priority.

Testing

To test BFDHT, we created a network of 8 BFDHT nodes on a single computer using Mininet, then created a malicious version of BFDHT with each of the following behaviors for a write request. The test is considered passed if the write request is eventually stored.

- Malicious Gateway Node: We did not handle malicious gateway nodes in this version, to a lack of time.
- Malicious Peer Node: The malicious peer node was used for tw0 tests, one with each of the following behaviors:
    - Report the wrong value in the Prepare message to all other nodes
        - Other peers achieved consensus. Test Passed.
    - Report Commit with the wrong value to all other nodes
        - Other peers achieved consensus. Test Passed.