The Byzantine-Fault tolerant Distributed Hash Table

Tim Krentz and Charlie Hartsell

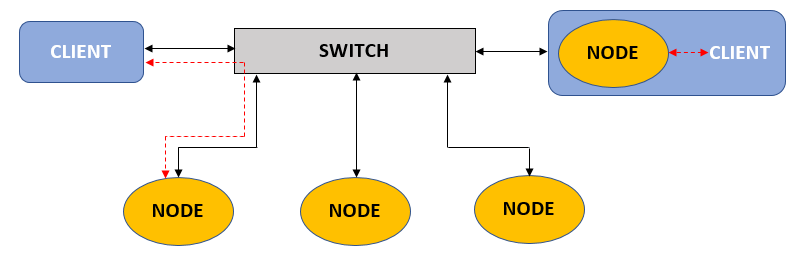
### **Purpose**

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 will test our system.

### **Design**

###### System Architecture

The BFDHT is implemented as a C++ library which provides functions for an external program to access the table. Additionally, the library allows for initialization and hosting of new nodes in the table. 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. 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.

****

**Figure 1.** System Layout

###### Software Design

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

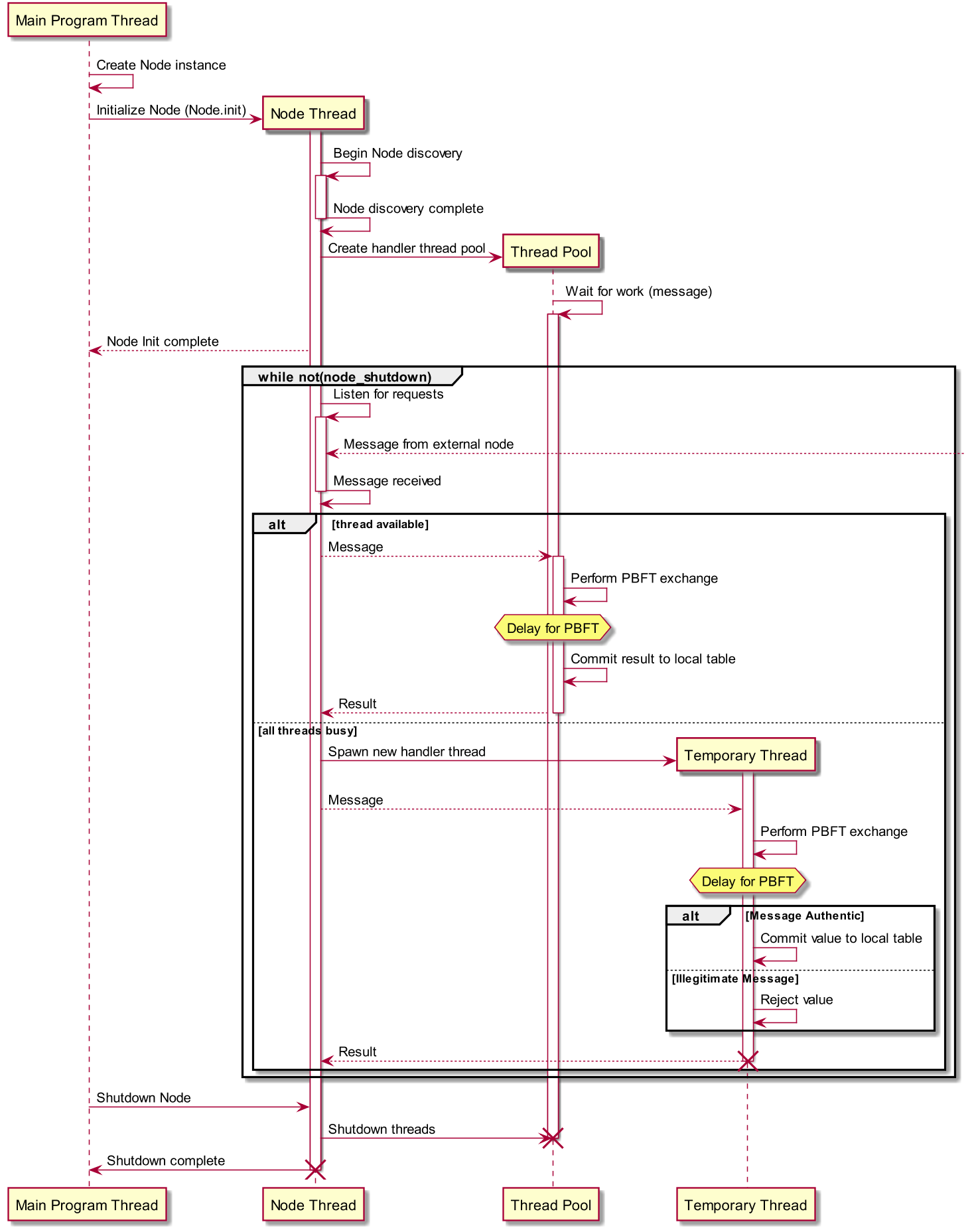
1. Node Initialization
2. Access Point Initialization
3. Put/Get
4. Shutdown

Node initialization consists of creating an instance of the Node class, then calling the Node initialization function within this class. The init function first spawns a dedicated thread, then attempts to discover any other available nodes in the BFDHT. Once a sufficient number of nodes have been discovered or a timeout period has elapsed, several worker threads are created in a thread pool. The node then begins listening for incoming messages from other nodes in the BFDHT. When a message is received, it is dispatched to one of the available worker threads. If all worker threads are busy, a new temporary thread is spawned to handle the message. 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 a notification is sent back to the primary thread. Once the message has been processed, the worker thread returns to waiting for another message. 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 initialization sequence is shown in Figure 2.

Clients which do not wish to host a node may still access the BFDHT using other nodes as access points. Access point initialization is similar to a truncated full-node initialization, but it does not spawn a pool of persistent handler threads. Instead, initialization only performs node discovery to identify a minimum of 2 nodes to use as access points then begins listening for messages from any access point. One of the access points discovered is designated as primary, with the remainder being saved as backups. All client requests are initially sent only to the primary node. If an acceptable response is not received within a set timeout, the primary node will be assumed faulty and a backup node will be appointed as the new primary.

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. Results from the put and get functions are not immediately available after the function returns due to the distributed nature of the system. Instead, a pointer to a suitable results data structure is returned with a flag indicating if the results are valid. When the client receives a response from the BFDHT, the corresponding result is updated and marked as valid.

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.



**Figure 2.** Node Initialization Sequence

###### PBFT Transaction Description

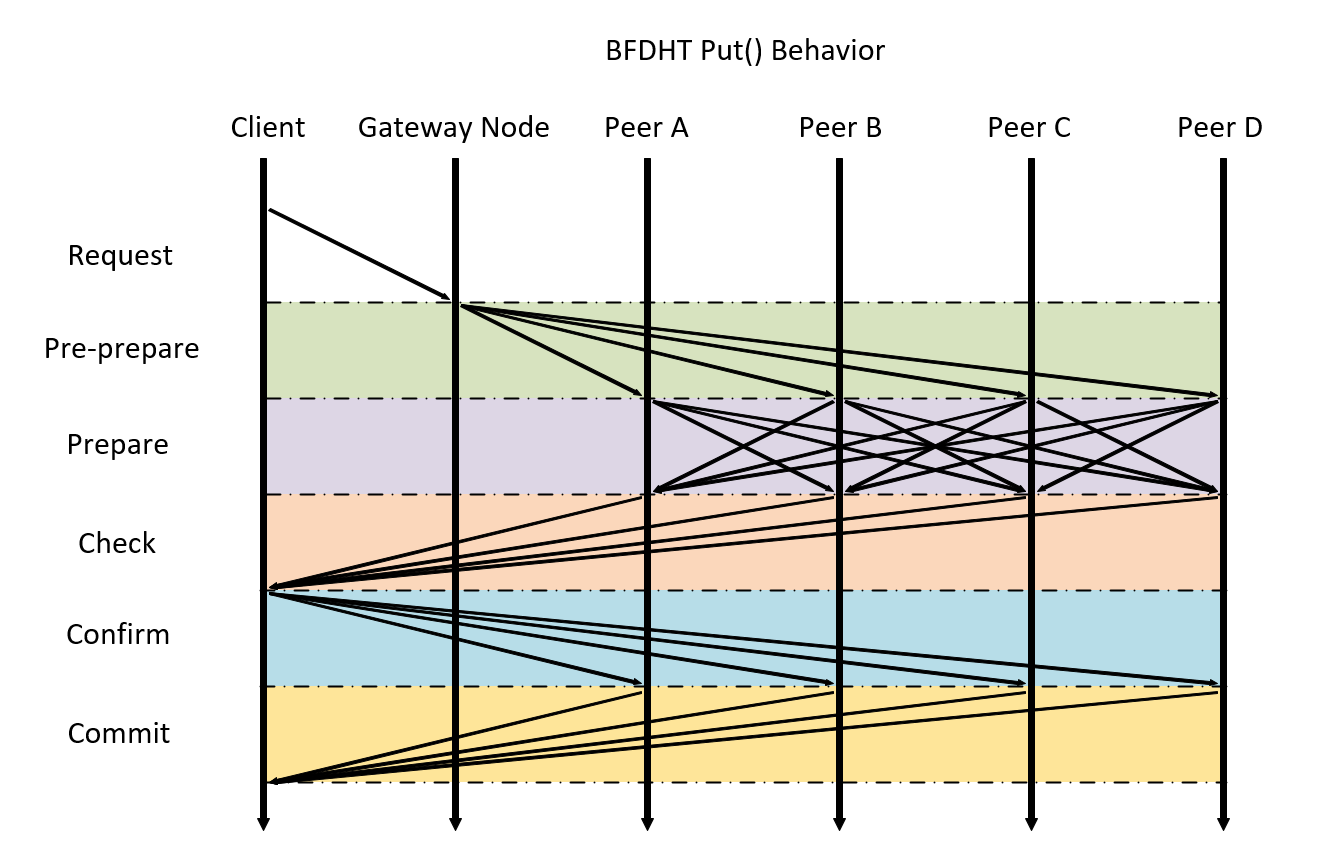
BFDHT provides *put* and *get* calls that are implemented as proposed in Practical Byzantine Fault Tolerance. These calls are implemented as follows:

***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, Client 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, Client 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, Client IP}. As soon as a Peer has collected 3f + 1 (in our implementation, 4) Prepare message (including its own), it proceeds to the next stage.
* ***Check*:** The peer sends a *Check* message to the client, formatted as {CHECK, key, value, Peer IP}, then proceeds to the next stage.
* ***Confirm*:** The client collects Check messages from the Check stage. As soon as the client receives 3f + 1 (4) correct Check messages, it responds to each Peer it received a Check message from with a *Confirm* message, formatted as {CONFIRM, key, value, Client IP}. As soon as a Peer has received a Confirm message it moves onto the next stage.
* ***Commit*:** The peer stores the key and value in its portion of the hash table, then sends a *Commit* message to the client, formatted as {COMMIT, key, value, Peer IP}. When the client has received 3f + 1 (4) Commit messages, it sets the validity flag for the response.

***Get****(key)* is much simpler because we assume any value is stored by 3f +1 peers in BFDHT. Thus, we don’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 can provide *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 performs peer lookup using the key’s hash to find the responsible Peers, here called *Peer A, Peer B, Peer C*, and *Peer D.* The Gateway Node then forwards the Request message to each responsible Peer.
3. When a Peer received a Request message, it sends a *Reply* message directly to the client, formatted as {REPLY, key, value, Peer IP}.
4. The client collects 3f + 1 Reply messages, then assumes the median value is the correct value.



**Figure 3.** The BFDHT put() process

### Threat Models

Due to the time available for implementation of the system, we place 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 may 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 3*f* + 1 nodes. However, we consider cases with only a single faulty node.

Faulty nodes can redirect messages from their intended destination, change the contents of a message, or drop messages entirely. Any single node within the BFDHT may 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 extensively throughout the system to address cases of dropped messages or faulty gateways. If a particular gateway or other node is non-responsive for an extended period of time, the node will be assumed faulty.

A malicious gateway node can change the value or key the client is attempting to store, a threat that is mitigated by requiring the storing peers to verify the key and value with the client in the Check and Confirm phases. If the Check phase does not produce the results the client is expecting, the client deduces that the gateway node is faulty and retries using a backup gateway node. 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. Every message described herein will have an accompanying timeout, so lost messages are handled by a new attempt for that request. An active node going offline can be handled by the key-based routing protocol we implement.

### **Testing**

To test BFDHT, we will create a network of BFDHT nodes on a single computer using Mininet, then create 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: The malicious gateway node will be used for three tests, one with each of the following behaviors
  + Change the requested value and key for ½ of the pre-prepare messages
  + Change the requested value and key for all of the pre-prepare messages
  + Send zero Pre-Prepare messages
* Malicious Peer Node: The malicious peer node will be used for three tests, one with each of the following behaviors:
  + Report the wrong value in the Prepare message to half of the other nodes
  + Send zero Prepare messages
  + Report Commit, but fail to store the key-value pair requested
  + Send zero Check messages