A Minimal Bare-Metal DHCP Server for Ultra-Low-Memory Devices

Jihan Xia*, Linfan Qian*, Ruiting Chen*, Di Bao*
Stanford University
{zxia2172, linfanq, ruitingc, dbao3}@stanford.edu

Abstract

Dynamic Host Configuration Protocol (DHCP) is responsible for assigning IP addresses to devices joining a network. Although the protocol is simple, common implementations maintain per-client lease records whose memory footprint grows linearly with the number of addresses managed by the server. This design can become problematic for low-cost routers and embedded networking devices with limited memory.

This paper investigates how DHCP servers can be redesigned for resource-constrained environments. We implement a minimal DHCP server running directly on bare-metal hardware and evaluate five different lease allocation strategies, including traditional table-based lease tracking (e.g., arrays or hash tables) and three memory-efficient bitmap-based schemes. Our most compact design, inspired by the New Presumed-Commit (NPrC) in two-phase commit (2PC), tracks only a small window of outstanding allocations and requires just 16 bytes of memory independent of the address pool size.

We implement all schemes using the Circle bare-metal framework on a Raspberry Pi and evaluate them using a synthetic workload that generates bursty DHCP requests. Our results show that bitmap-based designs reduce memory usage by orders of magnitude compared with traditional lease tables while maintaining comparable allocation latency and stable performance under high request concurrency.

These results demonstrate that DHCP functionality can be implemented with extremely small memory footprints without sacrificing performance, making it possible to provide reliable DHCP server even on very low-cost networking devices.

1 Introduction

Wi-Fi connection failures are surprisingly common in crowded environments such as cafés or shopping malls. A large-scale study shows that a major source of these failures is the Dynamic Host Configuration Protocol (DHCP), which assigns IP addresses to newly connected devices [1].

Although DHCP itself is a simple protocol, it is commonly implemented as part of a full router software stack. Commodity firmware such as OpenWrt [2] integrates DHCP together with DNS, NAT, and firewalling on top of the Linux system, increasing the overall resource requirements for the system.

These requirements can be significant for low-cost routers. Modern OpenWrt releases (v18.6+) require at least 64 MB of RAM [3]. By comparison, some cheap routers still on market provide much less memory, such as the TP-Link TL-WR841N series with 32 MB RAM [4], and routers like PIX-LINK LV-WR09 [5] that are built on the MediaTek MT7628KN chip (8 MB memory) [6]. While large venues may deploy enterprise networking equipment, dense public Wi-Fi networks are sometimes operated using inexpensive routers with limited memory, particularly in small businesses. These observations motivate us to explore how networking services can be implemented efficiently under constraint resources.

Among all router services, DHCP is a natural target for exploration: not only because obtaining an IP is the very first step of network access, but also protocol logic is simple and self-contained, making it a good starting point for developing low-cost, resource-constrained router implementations.

In this paper, we studied how DHCP can be implemented under tight resource constraints. We implemented five DHCP designs in Section 3. The system is prototyped on Raspberry Pi 3 Model B as a proof of concept, chosen for its well-supported development ecosystem and tooling.

We analyzed the memory footprint for different approaches in Section 4 and in Section 5 measured the average time required to obtain an IP in the Ethernet-based experiments.

Our optimizations are able to provide DHCP functionality with much smaller memory footprint while maintaining competitive IP assignment latency, showing that better service can be achieved even under resource-constrained conditions. This can benefit both router vendors and users, allowing them to consider using less expensive hardware and reduce costs.

2 Related Work

DHCP assigns IP addresses using a four-step protocol (DISCOVER, OFFER, REQUEST, ACK) [7]. After assigning an address, the server creates a *lease*, which records the mapping between a client identifier (typically a MAC address) and an IP address for a limited period of time.

Maintaining lease state ensures that clients can renew their assigned addresses and prevents the server from reassigning addresses that are already in use.

Many embedded routers run lightweight DHCP servers such as `dnsmasq` [8] or `Kea` [9]. These implementations maintain lease records containing information such as the client MAC address, assigned IP address, hostname, and lease expiration time.

For example, the `dnsmasq` lease file stores entries in the form

```
<expiry> <mac> <ip> <hostname> <c-id>
```

which typically requires around 50–80 bytes per lease in practice, depending on the hostname and client identifier length.

Similarly, `Kea` DHCP stores lease records containing the IP address, hardware address, lease lifetime, expiration timestamp, and optional client metadata, resulting in typical lease sizes of roughly 60–100 bytes.

Both systems allocate lease entries dynamically when new clients appear. That is, when a DHCP DISCOVER or REQUEST message arrives from a previously unseen client, the server creates a new lease structure at runtime rather than preallocating entries for the entire address pool.

In `dnsmasq`, the DHCP server typically searches for available addresses by scanning the lease table and checking candidate IP addresses sequentially. This process can involve a linear scan through existing leases or portions of the address pool, leading to worst-case lookup time of *O(N)*, where N is the number of active leases or candidate addresses.

The `Kea` DHCP server improves lookup performance by storing leases in hash tables indexed by identifiers such as MAC address, client identifier, and IP address. These hash tables allow faster lookup of existing leases and typically provide near constant-time access for renewal requests.

In both implementations, the memory required to store lease state grows linearly with the number of active clients.

For a large number of client connections to the address pool this memory requirement can become significant. For example, consider a private network with the address pool `10.0.0.0/8`, which contains 16,777,214 usable addresses. If each lease record requires approximately 80 bytes, the total memory needed to store lease information could possibly reach

$$16{,}777{,}214 \times 80 \approx 1.34 \text{ GB}.$$

Even if only a fraction of the address pool is actively used, the memory overhead of maintaining large lease tables can still be substantial for resource-constrained networking devices such as low-cost routers or embedded access points.

This motivates exploring alternative DHCP lease management designs that reduce memory usage while maintaining high allocation performance.

3 DHCP Algorithm

Our memory-optimized designs build on the Circle [10] bare-metal framework, making them substantially leaner than production DHCP servers that usually runs `dnsmasq` or `Kea` on Linux. Three deliberate simplifications are shared across all our designs.

Bare-metal kernel instead of Linux. Production servers run atop a general-purpose OS, paying the cost of system calls, scheduler preemption, virtual memory, and kernel networking stacks on every packet. Our server runs directly on the Circle bare-metal framework for the Raspberry Pi, eliminating OS overhead: packet reception and transmission are single-function calls directly into the Ethernet driver, and there is no context switching or memory protection boundary to cross on the critical path.

Pre-allocated static memory instead of dynamic allocation. Production servers typically store leases in a linked list or doubly-linked hash table with heap-allocated nodes, incurring a per-lease `malloc` and pointer-chasing on every lookup. Our implementations allocate all lease storage statically in the BSS segment, avoiding dependence on a general-purpose heap allocator and keeping the DHCP library self-contained, portable to other bare-metal environments.

Reduced lease record size. Production servers maintain lease records of 50–80 bytes per client, storing fields such as client hostname, FQDN, client identifier options, lease state machine flags, and filesystem-backed timestamps for persistence across restarts [9]. Our lease record stores only the fields necessary for correct address assignment.

Beyond the structural simplifications, our memory-optimized designs in Sections 3.2 and 3.3 rest on one key assumption grounded in the characteristics of our target environment (dense public venues such as cafés, shopping malls, and stadiums, where client sessions are typically short-lived). We assume that most clients will disconnect before their lease expires, making renewal uncommon in practice. Consequently, some designs han-

dle renewal by requiring the client to repeat the DISCOVER–OFFER–REQUEST–ACK sequence instead of supporting renewal natively, which is acceptable since the additional latency is small relative to the overall connection time.

3.1 Minimized bare-metal Baseline

We implemented two baseline variants following the allocation strategies used in `dnsmasq` and `Kea`, but replacing dynamic heap allocation with statically preallocated structures.

- **Array.** Leases are stored in a flat array of up to `DHCP_MAX_LEASES` entries. Each entry stores a 4-byte IP address, 6-byte MAC address, 4-byte expiry timestamp, and a 1-byte in-use flag (15 bytes total). MAC lookup, IP conflict checks, and lease allocation require a linear scan through the array, giving $O(N)$ complexity where N is `DHCP_MAX_LEASES`.

- **Hashmap.** We used two open-addressing hash tables to store the leases: a MAC-keyed table (storing MAC, IP, and expiry per entry) and an IP-keyed set (storing IP only). Both tables are statically allocated with a fixed number of slots sized 20% larger than `DHCP_MAX_LEASES` to keep the load factor low. This reduces MAC lookup, IP conflict checks, new lease insertion to $O(1)$ average time.

Both variants explicitly track the MAC-to-IP binding for every active lease. This enables lease renewal with a consistent IP address and allows the server to detect duplicate requests and NAK clients that attempt to obtain multiple leases, providing robustness against misconfigured or malicious clients. The tradeoff is the per-entry metadata overhead causing the memory footprint to become quite large when scaling to support a high volume of concurrent leases.

3.2 Bitmap-Based Lease Allocation

The MAC-to-IP binding supports advanced features, but we may not need to worry about several duplicate requests from the same client if we have a large enough IP pool. Non-malicious clients may obtain multiple IP addresses but use only one. Therefore, MAC-to-IP binding is beneficial, but is not strictly required.

The critical information needed for a lease is the allocated IP and its expiration time. Following this, we implemented a bitmap-based IP allocation scheme: the server maintains a bitmap together with the starting address of the IP pool, where each bit corresponds to one IP address. The expiration time is tracked per IP range instead of per IP to reduce the per-entry metadata overhead. This approach reduces the memory required to track each IP from tens of bytes to a single bit.

The DHCP IP allocation process requires the server to propose an IP during the DISCOVER-OFFER round but does not commit it until the REQUEST-ACK round. We assign an increasing counter that maps to a candidate IP for DISCOVER and set the corresponding bitmap bit only during the REQUEST-ACK round. The counter advances monotonically within an IP range until a free IP is found, giving amortized $O(1)$ candidate generation, while committing the allocation strictly takes $O(1)$ time.

We implemented two variants of this bitmap-based allocation. Their data structures and IP recycling schemes are discussed below.

- **Variable-Time Allocation:** The entire IP pool has a single lease period, and all IPs are recycled together when it expires. Each client receives a lease with the current period's remaining time. For late requests in a period, the lease time may be very short and require immediate renewal.

- **Unified-Time Allocation:** The IP pool is divided into multiple ranges, each having its own expiration time, equal to the time when the final IP in the range is offered plus the fixed lease duration. All IPs in one range are recycled together. This method provides more consistent lease durations at the cost of storing additional expiration metadata per range.

As shown in Section 4, this bitmap-based algorithm significantly reduces memory usage for IP pool sizes $\geq$ 4Ki by approximately 140×–160× compared with the array baseline, and by approximately 210×–240× compared with the hashmap baseline.

3.3 NPrC-Inspired Bitmap Optimization

We observe that DHCP lease management bears structural similarities to two-phase commit protocols [11]. Both systems share two key characteristics:

- a two-phase, bidirectional communication protocol

- a continuous range of identifiers (transaction id and IP address) that clients use to track their state with the server

The structural parallels between the two problem domains motivate a minimal-footprint DHCP server design inspired by Lampson et al.'s in-memory data structure from the New Presumed-Commit (NPrC) protocol [11].

The centerpiece of this design is the **ack-bitmap**, a 64-bit bitmap directly inspired by NPrC's sliding window over non-committed transactions. It tracks IP addresses that have been OFFERed but not yet ACKed, alongside ACKed addresses that fall within a window of still-pending assignments. The window is fixed at 64 entries, meaning the server tolerates at most 64 concurrent out-of-order DISCOVER-REQUEST sequences. This is a concurrency level we consider reasonable for commercial environments. When this capacity is exceeded, the oldest unresponsive clients are evicted, and any subsequent REQUEST from an evicted client on a prior OFFER will receive a NAK.

Besides the ack-bitmap, we also employ two auxiliary variables: cache-base and offset-next. **cache-base** is a 32-bit unsigned integer marking the lower bound of the active window. **offset-next**, meanwhile, denotes an 8-bit unsigned integer indicating the next un-OFFERed IP address as an offset from cache-base. These two variables together bound the window and provide mappings to actual IP addresses.

The result is a constant-memory data structure whose footprint is entirely independent of the IP address pool size, occupying just 13 bytes in total (or 16 bytes with alignment).

4 Memory Footprint

Algorithm	IP Pool Size (IPv4 Prefix Length)				
	254 (/24)	4Ki (/20)	64Ki (/16)	1Mi (/12)	16Mi (/8)
Baseline (Array)	4.97 KiB	80.0 KiB	1.25 MiB	20.0 MiB	320 MiB
Baseline (Hashmap)	7.53 KiB	120 KiB	1.88 MiB	30.0 MiB	480 MiB
Bitmap (Variable-Time)	60 B	540 B	8.03 KiB	128 KiB	2.00 MiB
Bitmap (Unified-Time)	88 B	568 B	8.05 KiB	128 KiB	2.00 MiB
Bitmap (NPrC)	**16 B**	**16 B**	**16 B**	**16 B**	**16 B**

Table 1: Memory usage of different DHCP allocation implementations across address pool sizes (Ki- and Mi- denote kibibytes and mebibytes).

Table 1 compares the memory footprint across all five designs.

Both baseline schemes, Array and Hashmap, scale linearly with pool size but with a considerably larger per-entry constant than the bitmap schemes, meaning memory consumption scales up quickly as the address space expands. At a pool size of 64Ki addresses, both schemes exceed 1 MB, and their footprint continues to grow rapidly as the pool increases. The two non-NPrC bitmap schemes also scale linearly, but with a smaller constant factor of $\frac{1}{8}$, which is 1 bit per address, resulting in substantially lower footprints even for large pools.

The NPrC-inspired design, by contrast, maintains a completely pool-agnostic footprint of **16 B** regardless of address space size.

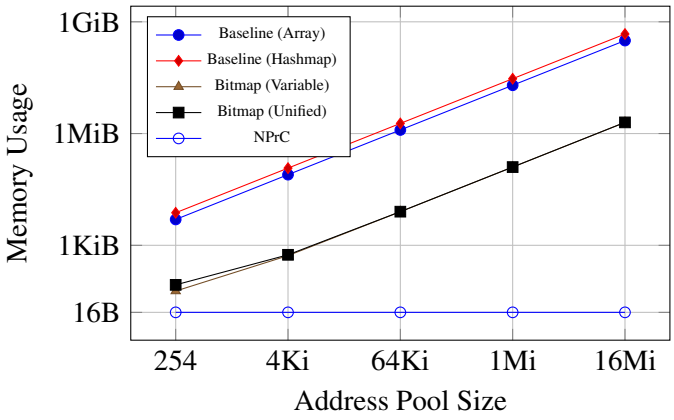

Figure 1: Memory usage across DHCP allocation algorithms (log scale).

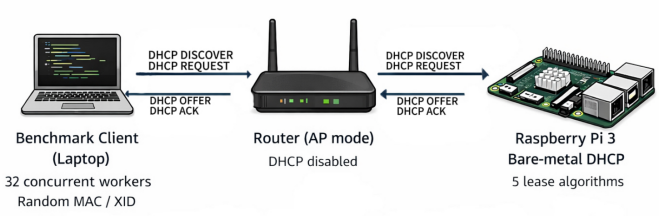

Figure 2: Experimental Setup

In our implementations, the runtime static memory footprint (`.text`, `.rodata`, `.data`, and `.bss`) excluding DHCP lease management structures is at most 255 KB. The total memory usage can therefore be estimated as 255 KB + our DHCP structures + runtime kernel overhead. The runtime kernel overhead is hard to estimate precisely, but the Circle kernel is expected to introduce a relatively small overhead.

5 Implementation and Evaluation

We evaluate our implementations using a synthetic benchmark that simulates bursts of DHCP DISCOVER and REQUEST messages.

The benchmark generates multiple clients simultaneously requesting addresses from a large pool.

5.1 Experimental Setup

We evaluate our DHCP server implementations on real hardware using a minimal bare-metal environment.

Hardware Platform Our DHCP server runs on a Raspberry Pi 3 Model B equipped with a 1.2 GHz ARM

	1 worker	16 workers	32 workers				
# clients / Method	16, 32, 64, 256	16	32	256	512	1024	2048
TP-Link	8ms	113ms [14/16]	224ms [30/32]	57ms [66/256]	–	–	–
Array	4ms	93ms	178ms	39ms	33ms	28ms	26ms
Hashmap	4ms	93ms	178ms	36ms	25ms	20ms	20ms
Variable-Time	4ms	92ms	178ms	35ms	25ms	20ms	20ms
Unified-Time	4ms	94ms	174ms	35ms	25ms	21ms	20ms
NPrC	4ms	92ms	176ms	35ms	25ms	20ms	18ms

Table 2: Benchmarking results for average DHCP handshake completion time per client. The 1 worker column aggregates 4 sequential runs of different number of clients. The TP-Link baseline exhibits failures under concurrent load (indicated [# success / # total] in the bracket). All five bare-metal schemes complete every request with no timeouts or NAK responses across all configurations.

Cortex-A53 CPU. The server is connected via Ethernet to a consumer router (TP-Link Archer AX55), which is configured to operate only as a switch and wireless access point. The router's built-in DHCP functionality is disabled when testing our implementations.

Software-simulated clients connect from a Linux laptop to the router over Ethernet and obtain IP addresses from the Raspberry Pi DHCP server.

Software Environment The DHCP server is implemented as part of a bare-metal networking stack running directly on the Raspberry Pi without a full operating system. We use the lightweight Circle kernel [10], which provides the necessary support for Ethernet and the UDP/IP stack.

The five lease management schemes are all implemented using a unified two-function interface:

- *dhcp-init-server(server, config)* initializes scheme specific data structures
- *dhcp-process-message(server, request, &response, cur-time)* processes incoming client requests and writes response packets

The complete source code is publicly available at: `https://github.com/linfanqian/baremetal-dhcp`

5.2 Real-time Performance

To simulate dense IP acquisition patterns, a single laptop sends raw Ethernet frames with randomized MAC addresses and transaction IDs in the DHCP message headers, mimicking concurrent requests from a large number of distinct clients. We use a single raw socket to issue all requests, capping concurrency at 32 workers to minimize overhead introduced by the benchmark harness itself and ensure measurements reflect DHCP processing rather than client-side concurrency management.

Since our bare-metal, single-threaded implementation provides efficiency advantages even for the baseline schemes, we supplement the internal comparison with an external baseline: the TP-Link access point operating in its default router mode. The TP-Link router mode [12] runs `dnsmasq` 2.62 on the Linux kernel via OpenWrt, a lightweight Linux distribution tailored for embedded router hardware. Results are shown in Table 2.

Our implementations perform significantly better than the TP-Link baseline across every configuration. In the synchronous single-worker trial, TP-Link exhibits an average latency of 8 ms, which is twice that of all our schemes at 4 ms. Under concurrent load, the router degrades further: requests with 16, 32, and 256 asynchronous clients show increasing failed requests despite sufficient available IP space, as confirmed by the synchronous trial. As the router is configured with a /24 pool ($\approx$254 addresses), we did not evaluate beyond 256 clients, but the trend suggests further degradation with more concurrent connections.

Among our five schemes, all optimized designs (Variable-Time, Unified-Time, and NPrC) achieve latency comparable to the Hashmap baseline and consistently better than the Array baseline. Given the numeric closeness of most measurements, we refrain from making strong claims about the relative superiority of any individual scheme. We do note, however, that the NPrC-inspired method consistently hits or approaches the lower-bound latency across configurations. This is a somewhat surprising result, given that NPrC is theoretically more sensitive to out-of-order arrivals due to the overhead of maintaining its 64-bit sliding window. Our

results suggest that this overhead is negligible in practice and is potentially outweighed by NPrC's inherent efficiency advantage in packet processing.

Our 32-worker results in Table 2 also show a consistent pattern in which the average handshake completion time decreases as the number of requests per worker increases. Inspecting per-request timings shows that the first request issued by each worker is significantly slower than all subsequent ones. We attribute this to a cold-start effect, where the Raspberry Pi's USB Ethernet interface, CPU caches, and DMA state are not yet warmed up. Because this one-time cost is amortized over all requests in a worker's run, averages computed over more requests are less distorted by it. So, we consider the high-client-count measurements (e.g., 2048 clients across 32 workers) to more closely reflect the steady-state bursty DHCP handshake latency of our server.

6 Limitations

Our memory-optimized bitmap designs rely on the assumption introduced in Section 3. However, if lease renewals are frequent, clients would repeatedly undergo the full DISCOVER–OFFER–REQUEST–ACK sequence rather than the lighter REQUEST–ACK renewal path, increasing per-client latency.

Furthermore, all bitmap-based designs are vulnerable to Denial-of-Service (DoS) attacks: a single MAC address sending overwhelming DISCOVER or REQUEST messages can drain the entire IP pool, since these designs do not track MAC-to-IP bindings and therefore cannot detect or reject duplicate allocation requests from the same client.

NPrC-inspired design carries two additional limitations. First, managing out-of-order OFFER-REQUEST sequences under high concurrency introduces extra overhead in sliding window maintenance. Our live benchmark confims that this does not meaningfully degrade server performance in practice, as NPrC method consistently exhibits lowest latencies across different configurations. Second, the current implementation does not support IP address reuse. In practice, this is unlikely to be an issue on a daily operational basis: NPrC's pool-agnostic memory footprint accommodates very large address spaces, such as a /8 prefix with up to 16Mi addresses, making pool exhaustion within a single service period unlikely under normal conditions. A straightforward extension would introduce four additional timestamp variables to track consumption across four equal quarters of the address space, recycling each quarter independently once exhausted. This scheme preserves the pool-agnostic property of the design at a cost of just 16 additional bytes, assuming 32-bit timestamps.

7 Future Work

While this work focuses on reducing the memory footprint of DHCP lease management, a practical Internet gateway requires additional network-layer functionality such as packet forwarding, Network Address Translation (NAT), and firewalling. Future work could extend our approach toward a minimal router stack for resource-constrained devices. In particular, NAT introduces additional connection-tracking state that can grow with the number of active flows, making it another candidate for memory-efficient data structures. Another promising direction is exploring minimal IPv6 router architectures, where DHCPv6 prefix delegation and Stateless Address Autoconfiguration (SLAAC) can reduce the need for per-client state. Together, these directions could enable fully functional Internet gateways running on extremely low-cost hardware with only a few megabytes of memory.

8 Conclusion

We presented a minimal bare-metal DHCP server designed for resource-constrained networking devices. By replacing traditional lease tables with bitmap-based allocation schemes, our designs reduce memory consumption by more than two orders of magnitude while maintaining comparable allocation latency. Our NPrC-inspired variant further eliminates dependence on the address pool size entirely, requiring only 16 bytes of memory while sustaining stable performance under concurrent request bursts.

More broadly, our work demonstrates that DHCP lease management can be significantly simplified when the design is tailored to realistic deployment environments, where client arrivals may occur in bursts and lease renewals are relatively infrequent. Many public Wi-Fi networks already provide sufficient radio coverage but still suffer from unreliable connectivity due to DHCP failures on resource-constrained routers. Our design shows that reliable IP address allocation can be achieved even on very low-cost hardware, suggesting that improving DHCP server design can be an effective path to more reliable connectivity in dense public networking environments.

9 AI Disclosure

We used AI to debug our code implementation, generate an experimental setup figure, fix LaTeX syntax, help install the libraries required for the bare-metal experimental setup, and refine the grammar of paper.

References

[1] C. Pei, Z. Wang, Y. Zhao, Z. Wang, Y. Meng, D. Pei, Y. Peng, W. Tang, and X. Qu, "Why it takes so long to connect to a wifi access point," in *IEEE INFOCOM 2017*, 2017.

[2] OpenWrt Project, "OpenWrt: A Linux Operating System for Embedded Devices." `https://openwrt.org/`. Accessed: 2026-03-12.

[3] OpenWrt Project, "Openwrt on 8/64 devices." `https://openwrt.org/supported_devices/864_warning`. Accessed: 2026-03-12.

[4] TechInfoDepot, "TP-LINK TL-WR841N series." `https://techinfodepot.shoutwiki.com/wiki/TP-LINK_TL-WR841N_series`. Accessed: 2026-03-12.

[5] TechInfoDepot, "Pix-link lv-wr09 v1 hardware specifications." `https://techinfodepot.shoutwiki.com/wiki/PIX-LINK_LV-WR09_v1`, 2025. Accessed: 2026-03-12.

[6] MediaTek, "Mt7628k/n/a router and repeater platform." `https://www.mediatek.com/products/home-networking/mt7628k-n-a`. Accessed: 2026-03-12.

[7] R. Droms, "Dynamic host configuration protocol." RFC 2131, 1997.

[8] S. Kelley, "Dnsmasq: A lightweight dns and dhcp server." `https://thekelleys.org.uk/dnsmasq/`, 2025. Accessed: 2026-03-12.

[9] I. S. Consortium, "Kea dhcp server." `https://kea.isc.org/`, 2025. Accessed: 2026-03-12.

[10] R. Stange, "Circle: A c++ bare-metal programming environment for the raspberry pi." `https://github.com/rsta2/circle`, 2026. Accessed: 2026-03-12.

[11] B. W. Lampson and D. B. Lomet, "A new presumed commit optimization for two phase commit," in *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, (San Francisco, CA, USA), p. 630–640, Morgan Kaufmann Publishers Inc., 1993.

[12] TP-Link, "Gpl code center - archer ax55." `https://www.tp-link.com/us/support/gpl-code/?model=Archer`. Accessed: 2026-03-15.